GPU Parallelization of Cryptographic Primitives using Multivariate Quadratic Polynomials and its Security Evaluation

田中, 哲士

https://doi.org/10.15017/1500750

出版情報：九州大学, 2014, 博士（工学）, 課程博士
バージョン：
権利関係：全文ファイル公表済

GPU Parallelization of Cryptographic Primitives using Multivariate Quadratic Polynomials and its Security Evaluation

Satoshi Tanaka

January 2015

Department of Informatics,
Graduate School of Information Science and
Electrical Engineering, Kyushu University

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

$\mathcal{MP}$	The Multivariate Polynomial Problem
$\mathcal{MQ}$	The Multivariate Quadratic Problem
$\mathrm{Gal}(\mathrm{GF}(q)/\mathrm{GF}(p))$	A Galois group from $\mathrm{GF}(p)$ to $\mathrm{GF}(q)$
$\mathrm{GF}(q)$	The Galois Field of the q
n	The Number of Unknowns in Multivariate Polynomials
p	The Prime Number
q	The k-th power of p: $q = p^k$
$Q(x)$	The Multivariate Quadratic Polynomial
$S(x)$	The System of Multivariate Quadratic Polynomials
x, y, z	Unknowns in Multivariate Polynomials

Abstract

Stream ciphers encrypt messages by xoring with random keystreams. Usually, stream ciphers use linear feedback shift registers or permutation mixing algorithms such as PRNGs (pseudo-random number generators). They can efficiently mix data within a few computations. On the other side, there are some PRNGs, which have the provable security in the theory of public-key cryptosystems. QUAD is such a stream cipher, which is based on evaluation of multivariate quadratic polynomials over finite fields as a cryptographic primitive. The security of QUAD is based on the $\mathcal{MQ}$ (multivariate quadratic) problem: finding the evaluation points given the values of multivariate quadratic systems over finite fields. Because $\mathcal{MQ}$ problem is known to be NP-complete, QUAD is secure on sufficient large quadratic polynomials. However, evaluating large multivariate quadratic polynomials requires to perform many multiplications and additions, in a number which is proportional to the square of the number of variables and to the number of polynomials. Other stream ciphers require at most hundreds computations, QUAD takes ten thousands computations. Therefore, making efficient the evaluation operation for multivariate quadratic polynomials is important.

Parallel computing is a method for accelerating processing. Generally, each polynomials can be evaluated in parallel, and each term in a polynomial can also be computed. Therefore, evaluating multivariate quadratic polynomials is suited to parallel computing method. Using GPUs (graphics processing units) for general processing is a way of parallel computing. Latest GPUs have high computational performance with over thousand cores. However, GPU programming has different limitations from CPU implementations about executing threads, memory loading, etc. Therefore, GPU implementations require optimized models. In this work, we choose CUDA (Compute Unified Device Architecture) as a programming environment with GPUs. This is designed for NVIDIA GPUs.

Specifically, this dissertation is organized as follows:

Chapter 1 presents the background and motivation of this research. Moreover, we show some challenging issues and our contributions.

Chapter 2 explains multivariate polynomials of MPKC (multivariate public-key cryptography) and existing MPKC systems. QUAD is based on MPKC's principles. Moreover, we describe parallel implementation of cryptosystems and existing results using GPUs.

Chapter 3 presents parallelization techniques for evaluating multivariate quadratic polynomials over the binary field. We focused on the fact that evaluating polynomials over the binary field is equivalent to summations of coefficients of non-zero terms. We separated the algorithm for evaluation of multivariate quadratic polynomials into the following three steps, (i)counting non-zero terms in polynomials, (ii)loading coefficients of non-zero terms in polynomials and (iii)summation of such coefficients in each polynomial. Because step (i) is sequential, we implement it on a CPU, and step (ii) and (iii) are implemented on GPU. Moreover, we implement summations in step (iii) by the parallel reduction method. Finally, we reduced the computational cost of evaluating multivairate quadratic polynomials with bitslicing strategy. Because each term in a polynomial is independent, we can parallelize computations of multiplications of monomials. This chapter presents some methods of efficient parallelization and optimization for GPUs. As a result, our implementation shows a throughput of about 12 Mbps by QUAD with a system of 320 polynomials in 160 variables over the binary field. This GPU result is about 18 times faster than CPU.

Chapter 4 discusses parallelization techniques of evaluation of multivariate quadratic polynomials over extension fields. Using extension fields, the parameters (e.g. the number of variables) of QUAD can be chosen smaller than those over the binary field. In this chapter, we seperated the algorithm for evaluation of multivariate quadratic polynomials into the following three steps, (i)multiplying common quadratic terms $x_i x_j$ between every polynomial, (ii)multiplying terms and coefficients in polynomials and (iii)summation of terms in each polynomial. We implemented each step on GPU, because they can be computed in parallel. We also used the same parallel reduction method used in chapter 3 in step (iii). Moreover, we compared the efficiency of the multiplication methods over extension fields for step (i) and (ii). Among the six multiplication methods presented over GF(2^{32}), the fastest method is using the bitslicing strategy, the next is using intermediate field GF(2^8). Finally, we proposed a data construction, suited to memory loading on CUDA GPU programming. As a result, our implementation method showed a throughput of about 25 Mbps by QUAD with a system of 48 polynomials in 96 variables over GF(2^{32}). This GPU timing is about 90 times faster than CPU.

Chapter 5 presents parallelization of Petzoldt's method, is evaluating method of multivariate quadratic polynomials using LRS (linear recurring sequence). Petzoldts's method creates some relations between coefficients in each polynomial, and reduces multiplications in evaluating multivariate quadratic polynomials through such relations. However, parallelizing his method is difficult, because it has many sequential steps. Then, we introduced the multi-stream method, which executes several instances of a cryptosystem. Moreover, we constructed extensive data from chapter 4, suited to Petzoldt's method with the multi-stream method. As a result, our implementation method showed a throughput of about 190 Mbps by 256 streams of QUAD with a system of 32 polynomials in 64 variables over GF(2^{32}).

In chapter 6, we parallelize the XL-Wiedemann algorithm, which is amethod for solving multivariate quadratic systems over finite fields. The Wiedemann algorithm, the core process of the XL-Wiedemann algorithm, is separated into the following three steps: (i)generating the sequence of $\{uA^ib\}_{i=0}^{2N}$ (A: $N \times N$ matrix, $b := Ax$, u: random vector), (ii)computing the minimal polynomial of the sequence $\{uA^ib\}_{i=0}^{2N}$, and (iii)calculating the unknown vector x of $Ax = b$. We implemented step (i) and (iii) on GPU and (ii) on CPU, because step (i) and (iii) are mainly products of sparse matrices and dense vectors and (ii) is very sequential (the Berlekamp-Massey algorithm). Moreover, we expanded the cuSPARSE library, for linear algebra of floating point with sparse matrix, to finite prime fields. Finally, we solved a $\mathcal{MQ}$ problem with a system with 48 polynomials in 24 unknowns.

Chapter 7 shows our conclusion and further research issues.

Abstract(Japanese)

ストリーム暗号は，平文と乱数生成された鍵ストリームとの排他的論理和を取る事で，暗号化を行う手法である．多くのストリーム暗号では，擬似乱数生成器（Pseudo-Random Number Generator, PRNG）に線形フィードバックシフトレジスタや攪拌アルゴリズムを利用しており，計算量の少ないままデータの攪拌が可能である．一方で，使用する PRNG の安全性を数学問題の解決困難性に帰着させ，公開鍵暗号と同等の安全性証明を持たせる種類のストリーム暗号の研究が行われている．多変数二次多項式の付値を暗号プリミティブとする QUAD は，その種のストリーム暗号の一つであり，安全性の根拠として有限体上の連立二次方程式の求解問題 (Multivariate Quadratics, MQ) を利用しているため信頼性が高いが，反面，QUAD は鍵ストリームの生成に数万回以上の計算を必要とし計算コストが高い．この為，QUAD を実用化するためには計算の効率化が研究課題となる．

計算を効率化する手法の一つとして並列処理がある．一般に，多項式は各々独立に付値の計算が可能であり，多項式内の項も各々独立である．従って，多項式の付値は独立に計算可能な部分が多く，並列計算に適している．並列計算の実現手法として，GPU(Graphics Processing Units) を利用した手法が注目されている．現在の GPU は 1,000 個を超えるコアを有しており，並列計算に基づいた高い処理能力を有している．反面，GPU 実装はスレッドの実行タイミング，メモリローディング等，動作上において CPU 実装とは異なる制約があり，GPU に適した実装を行う事が課題となる．

本学位論文は以下のように構成される．

第 1 章では，本研究の背景と目的を述べる．また，本研究の主要な挑戦課題と貢献についても示す．

第 2 章では，QUAD の基礎となっている有限体上の多項式，及び，既存の多変数公開鍵暗号の手法について説明を行う．また，GPU 上で用いる実装環境について説明し，本論文の GPU 実装で用いる CUDA(Compute Unified Device Architecture) API の説明を行い，暗号における並列実装手法及び GPU を用いた既存の実装成果についても説明する．

第 3 章では，QUAD で用いる GF(2) 上の多変数二次多項式の付値に対する並列化を行った．具体的には，GF(2) 上の多項式の付値は，非零な項の係数の加算のみで計算可能なことに着目し，付値の手順を (i) 多項式内の非零な変数の判定，(ii) 非零な変数と対

応する多項式の係数の読み取り，(iii) 係数の総和の 3 ステップに分割した．このとき，(i) は判定が逐次的であるため，逐次処理に強い CPU で，並列処理可能な (ii),(iii) は GPU 上で実装した．また，(iii) では並列リダクションと呼ばれる手法を利用した．更に，ビットスライス戦略により，一度に 32 の多項式を取り扱う事で計算の回数を削減した．最終的に，本章の実装手法を用いる事で，GF(2) 上の 160 変数 320 多項式の QUAD において約 12Mbps の速度を達成した．これは，CPU 実装と比較して約 18 倍高速であった．

第 4 章では，拡大体上における多変数二次多項式の付値に対する並列化を行った．拡大体を利用する事で，多変数多項式のパラメータである変数の数を小さくとる事が可能である．本章では，拡大体上の多項式に対する付値を (i) 多項式間で共通する項の乗算，(ii) 多項式内の項と係数の乗算，(iii) 多項式内の項の総和の計算，という 3 ステップに分割した．各ステップは並列処理可能であり GPU 上で実装した．(iii) は第 3 章と同様に，並列リダクションを利用した．また，(i),(ii) では拡大体上の乗算が必要となるため，効率的な乗算手法について比較を行った．GF(2^{32}) の乗算手法について 6 つの手法を比較した結果，ビットスライス戦略を用いたものが最速であった．更に，GPU のメモリローディングに適したデータ構造を提案した．最終的に，本章の実装手法を用いる事で，GF(2^{32}) 上で 48 変数 96 多項式の QUAD において約 25Mbps の速度を達成した．これは，CPU 実装と比較して約 90 倍高速であった．

第 5 章では，多変数二次多項式の付値に線形回帰数列を用いた Petzoldt の手法に対する並列化を行った．Petzoldt は多項式の係数に関係性を持たせ，複数の乗算を同時に処理する事で効率化を行っている．しかし，彼の手法は逐次的なステップが多く，単純な並列化では効率化ができない．そこで，複数の暗号を同時に実行する事により，１回の処理で生成される鍵ストリームの数を増やすマルチストリーム手法を導入した．更に，第４章のデータ構造を拡張し，マルチストリームを用いた Petzoldt の手法に適したデータ構造を提案した．最終的に，GF(2^{32}) 上の 32 変数 64 多項式の QUAD を 256 個のストリームで同時に実行する事により，約 190Mbps の実装速度を達成した．

第 6 章では，QUAD の安全性の根拠となる MQ 問題を解決する拡張線形化アルゴリズム（eXtended Linearization, XL）の並列化を行った．XL は解決可能な線型方程式を構成とその線型方程式の解決の，2 つのステップで構成される．線型方程式の解決には，方程式の係数行列の逆行列を求める手法が知られているが，XL が構成する行列は疎であるため，疎行列に適したアルゴリズムを用いることによる効率化が期待できる．疎行列に適したアルゴリズムの中でも，Wiedemann アルゴリズムは並列性が高く，8PC クラスタを用いた並列実装による MQ 問題の解決結果が知られている．本研究では，Wiedemann アルゴリズムの主要な 3 ステップ (i) 数列の生成, (ii) 数列に対する最小多項式の導出, (iii) 解ベクトルの導出について，並列性の高い (i),(iii) を GPU，逐次性の強い (ii) を CPU で実装した．最終的に，GF(7) 上 24 変数の 48 連立二次方程式を約 10 時間で解決した．これは，CPU 実装と比較して約 38 倍の高速化に相当する．

第 7 章では，本研究の結論と今後の研究課題について述べる．

Acknowledgment

I would like to thank all the researchers and colleagues who helped me in this thesis. Especially I would like to thank my supervisor, Professor Kouich Sakurai, who has supervised me for years, introduced me to this field of research, and provided me valuable discussions, critics, and advices at any time. I want to express my gratitude to Professor Yoshiaki Hori from Saga University, Associate Professor Takashi Nishide from Tsukuba University, Special Associate Professor Xavier Dahan from Ochanomizu University and Assistant Professor Junpei Kawamoto, for giving me helpful comments daily over these years.

Moreover, I would like to thank Doctor Takanori Yasuda who has given me profitable comments for the multivariate cryptography, and Doctor Hirofumi Muratani, who is my external advisory. Furthermore, I am extremely grateful to people who helped me for studying GPU implementations in Taiwan University, especially, Professor Chen-Mou Cheng, who gave me a chance to study in Taiwan, Docter Bo-Yin Yang, who gave me valuable advice about the computational cost and the time complexity of multivariate cryptography, and Mister Tung Chou, who advised me about GPU implementations and helped in getting settled in Taiwan.

At last, I would like to thank my parents, who have given me daily support during these years of work.

Chapter 1

Introduction

1.1 Background

Stream cipher belongs to symmetric cryptography, which generates random keystreams through a pseudo-random number generator (PRNG). Generally, stream ciphers are known to encrypt faster and to require lower computational resources than block ciphers. Another aspect of the design of efficient and secure stream cipher is to set standards for the parameters involved. Discussions of such a stream cipher is based on the security of the PRNG. A stream cipher can be shown to be provably secure with the theory of public key cryptography. For example, Blum-Blum-Shub provably secure stream cipher uses the theory of the integer factorization.

QUAD is a kind of multivariate cryptography[14]. It is a stream cipher, which uses a multivariate polynomial system as a PRNG. The security of QUAD also depends on the hardness of solving a multivariate quadratic system over a finite field, which is called the multivariate quadratic problem ($\mathcal{MQ}$). Therefore, QUAD holds provable security like public key cryptography though it is a symmetric cipher. QUAD has high security, but it is very slow compared with other symmetric ciphers.

1.2 Related Works

The security discussions of QUAD is based on the PRNG. Yang *et al.* [55] shows some weaknesses in QUAD with a small number of variables. Thus practical QUAD requires the construction of multivariate quadratic systems with many variables, along with achieving a high speed of encryption.

Petzoldt proposed an efficient method for evaluating QUAD, which reduces the computational cost from $\mathcal{O}(mn^2)$ to $\mathcal{O}(mn)$ [42]. His idea is to use linear recurring sequences (LRS). Coefficients of LRS QUAD are powers of generators of finite fields. Then, LRS QUAD computes several multiplications at a time sequentially.

1.3 Challenging Issue

Parallel Evaluating Method of Multivariate Quadratic Polynomial

Our main contributions are two effective computing methods of evaluating multivariate quadratics system. One computes a summation of multivariate polynomials as a rectangle matrix. The other handles a summation of multivariate polynomials as a long vector. Moreover, we evaluate efficiency of contributions with numbers of additions and multiplications. Fast evaluation of multivariate quadratic polynomials is necessary to construct practical QUAD. Our challenge is to make it efficient through two approaches: parallelizations and using extension fields.

The number of monomials in a quadratic polynomial in n variables is given by $\binom{n+2}{2}$. A parallel algorithm for summations named parallel reduction is executed in $\lceil \log T \rceil$ steps, where T is the number of terms to be summed (exactly $T = \binom{n+2}{2}$). However, it executes a surplus step for some n. For example, when $n = 64$, $T = 2,145 > 2,048$ and it takes 12 steps. Therefore, it is desirable to reduce the number of monomials in each quadratic polynomial under 2,048 for $n = 64$. Although the number of reducing terms for each polynomial should be the same for parallelizations, choosing different combinations is difficult. Hence, reducing monomials of quadratic polynomials is an issue.

Moreover, we use Compute Unified Device Architecture(CUDA) API [3], provided by NVIDIA [6], for GPU implementations. In CUDA implementations, we have two sub-issues regarding the tuning of the parallelizations on GPU. One is avoiding the surplus steps of GPU kernels (functions). Indeed, in CUDA, parallelization of kernels is achieved with blocks and threads. However, actually threads are divided by warp, the maximal number of parallel threads in a block executed at a time. Therefore, we should tune the number of threads in order that it is a multiple of the warp size to avoid surplus steps. We should optimize this number for every construction.

Efficient Multiplications over Extension Fields

Using large finite fields can make small polynomial systems (i.e. the number of unknowns, polynomials). However, operations over large finite fields are heavier than those over small fields. Operations over large prime fields spend time for modular operations required by addition and multiplication. On the other hand, although, over large extension fields we can compute additions easily, but multiplications are more complicated than over prime fields.

There is a challenging issue concerning extension fields: generally, multiplication over extension fields is more complicated. Although in small cases (e.g. GF(2^8)), we can make it efficient with lookup tables, in large cases (like GF(2^{32})) we cannot, because of the size of the table takes up to 32EB!. There exists some related works, which discuss fast hardware implementations of binary extension fields [44] and GPU implementations over extension fields [37], however they do not discuss GPU implementations of extensions of the binary field. Hence, fast implementations of the multiplication over binary field's extensions on GPU is an important issue.

Parallelization of MPKC using Linear Recurrence Sequence

In Petzoldt's implementation, he used GF(2^8). The period of generators over GF(2^8) is at most 255. However, his quadratic polynomial has 378 terms. Hence, there are some relations between some terms in it. Therefore, there is a risk that the security might be reduced.

Petzoldt claimed that his method can be parallelized easily as follows. Each quadratic polynomial in QUAD is independent, so it is easy to parallelize at the polynomial level. However, the degree of parallelization is proportional to the number of polynomials in QUAD, which may not be enough for effectively exploiting the full computational power available on modern GPUs. In this paper, we shall consider further parallelization in evaluating LRS quadratic polynomials for GPU implementation.

Security Evaluation of QUAD Stream Cipher

So far, we have not seen any implementation of the XL-Wiedemann algorithm on GPU, which is a candidate for further speed-up because several steps of the XL-Wiedemann algorithm can be parallelized. Therefore, we consider accelerating XL-Wiedemann on GPU. However, GPU implementation poses a set of very different limitations from its CPU counterpart. Hence, in this thesis we shall detail these challenges and how we have dealt with them.

1.4 Contribution

Parallelization of Evaluation of Multivariate Quadratic Polynomial

We provide two techniques to implement QUAD stream cipher. One is a parallel implementation for computing multivariate polynomials. The other is an optimization technique for implementing QUAD on GPUs.

Moreover, we discuss the computational time for generating keystreams of QUAD in more detail than in [47]. Also, we report results of implementation of QUAD stream cipher over $GF(2)$, $GF(2^2)$, $GF(2^4)$, and $GF(2^8)$ on GPU.

Effective Method for the Multiplication over Finite Fields

We reduce the number of terms of multivariate quadratic polynomials from $\binom{n+2}{2}$ to $\binom{n-k+2}{2}$ by removing variables. Our method removes different variables for each polynomial.

We implement multiplications through the polynomial basis, the normal basis, Zech's logarithm, using intermediate fields and bitslicing and discover the most suited method for GPU. For GF(2^{32}), we find out that the best way is bitslicing the polynomial basis over GF(2^{32}). It shows a throughput of 800 Gbps.

We tune our QUAD implementations for CUDA. We choose k, an integer multiple of divisible by 32. Also, we choose the best multiplications in our experimentations, then implement QUAD over GF(2^{32}) on GPU. Moreover, we construct a data structure for QUAD on GF(2^{32}).

We then achieve throughputs of QUAD($2^{32}, 48, 48$) and QUAD($2^{32}, 64, 64$) of 24.827 Mbps and 19.4196 Mbps respectively. These are over 90 times faster than CPU ones. This is the first implementation of QUAD stream cipher over GF(2^{32}).

Parallelization of MPKC using Linear Recurrence Sequence

We choose GF(2^{32}) for the finite field of LRS QUAD stream cipher. The period of generators over GF(2^{32}) is $2^{32}-1$. This is enough for coefficients of quadratic polynomials, as the number of terms in a quadratic polynomial of n variables is only $\binom{n+2}{2}$.

Then, we implement two versions of parallelized Pezoldts's LRS QUAD stream cipher [42] on GPU. The first version is the naïve parallelization with parallelization only at the polynomial level. The second version parallelizes computations in quadratic polynomials, e.g., calculating $\alpha_{i,j}x_ix_j$. The result shows that the latter is 2.5 times faster than the former, making it more suitable for GPU implementation of LRS QUAD.

To further exploit the available computational power on modern GPUs, we adopt the multi-stream strategy used by Chen *et al* [20], in which multiple QUAD instances are executed in parallel. We have implemented multi-stream QUAD over GF(2^{32}) and achieved a throughput of 193.40 Mbps for 256 streams of QUAD with 64 polynomials in 32 variables. To the best of our knowledge, this is the best throughput performance result for software implementation of QUAD. To achieve this performance for Petzoldt's LRS QUAD, we have introduced three data structures specifically for efficient handling of memory loading with CUDA API, the most popular programming environment for NVIDIA GPUs.

Security Evaluation of QUAD Stream Cipher

We present several GPU implementations of the XL-Wiedemann algorithm, in which multiplication of a sparse matrix with a dense vector is parallelized on GPU. Additionally, we computed benchmarks of an implementation based on the cuSPARSE library using floating-point arithmetic. Finally, we show the experimental results of solving $\mathcal{MQ}$ instances over GF(2), GF(3), GF(5), and GF(7). Our implementation can solve $\mathcal{MQ}$ instances of a system 74 equations in 37 unknowns over GF(2) in 36,972 seconds, of a system of 48 equations in 24 unknowns over GF(3) in 933 seconds, as well as of a system of 42 equations in 21 unknowns over GF(5) in 347 seconds. The largest instance of a matrix that we have solved is occurring in a $\mathcal{MQ}$ problem consisting of a system of 80 equations in 40 unknowns over GF(2) in 295,776 seconds. This matrix has 3.78 billion non-zero elements. On the other hand, the most difficult instance in term of solving time is a system of 48 equations in 24 unknowns over GF(7) solved in 34,883 seconds, whose complexity is around $\mathcal{O}(2^{67})$ if we use a brute-force approach.

The cuSPARSE library only supports floating-point arithmetic, not integer arithmetic, let alone finite field arithmetics. Therefore, we need to use cuSPARSE functions to implement finite field arithmetics via additional operations such as modular operations.

Figure 1.1 shows that the relationship between contributions proposed in this dissertation. This dissertation presents two aspects of the speed-up and in the security evaluation of the cryptographic primitives. One is the implementation side. Here, we assume the characteristics of normal and honest users. Under this assumption, they use only several devices (in particular, they have one desktop and/or one laptop PC and/or one mobile phone or smartphone). Moreover, they don't use any distributed algorithms between devices. Therefore, they can use only the full power of one device. In other words, they can only use several CPU-cores and several GPUs inside PC under this limitation. Our parallel method of evaluation for multivariate quadratic polynomials is under this limitations

Another one is on the attacker side. This side shows that the stance of attackers or malicious users. They want to know information of communications between honest users. In this situation, attackers know that honest users use QUADs to encrypt messages in prior of communications. Hence, attackers try to break QUAD. Generally, we assume that they can use as much devices as possible to break it in a reasonnable amount of time (e.g. in a year). In other words, they can use over 1,000,000 PC-cluster to break MQ. These attacks permit to set upper bounds of solving time MQ. Therefore, we should select parameters of QUAD to exceed this security evaluation.

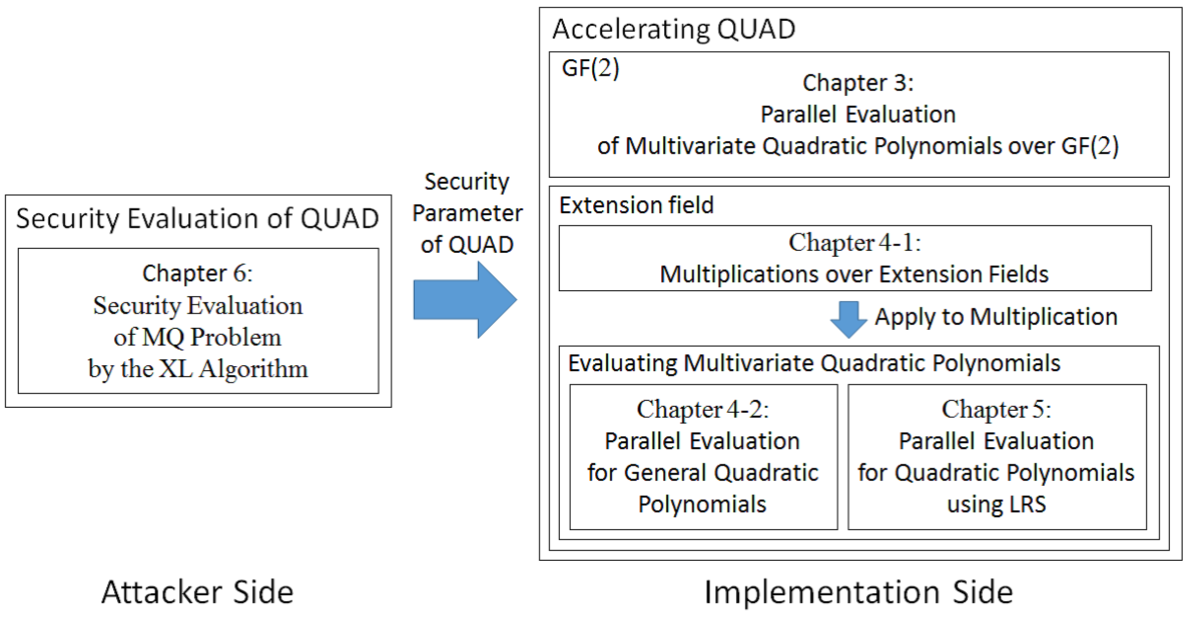


Figure 1.1: Relationship chart of contributions

Chapter 2

Preliminary

2.1 Multivariate Quadratic Polynomial

2.1.1 Polynomial

Let $\boldsymbol{x} = \{x_1, \ldots, x_n\}$. We call primitives, which are denoted multiplications of single constant and multi variables, that term. Polynomials are constructed by summations of multi terms. When terms are constructed by different variables (e.g. 'x' and 'y'), polynomials are called multivariate polynomials. Generally, a k-degree n-unknowns multivariate polynomial $P(\boldsymbol{x})$ denotes Formula (2.1). x_{i_j} is an unknown ($1 \leq i \leq n, 1 \leq j \leq k$), and $\alpha^{(j)}_{i_1,\ldots,i_j}$ is a coefficient.

$$P(\boldsymbol{x}) = \alpha^{(0)} + \sum_{1 \leq i_1 \leq n} \alpha^{(1)}_{i_1} x_{i_1} + \ldots + \sum_{1 \leq i_1 \leq \ldots \leq i_k \leq n} \alpha^{(k)}_{i_1,\ldots,i_k} x_{i_1} \ldots x_{i_k} \tag{2.1}$$

Especially when $k = 2$, polynomials are called multivariate quadratic polynomials. Formula (2.2) presents a multivariate quadratic polynomial $Q(\boldsymbol{x})$ in n variables.

$$Q(\boldsymbol{x}) = \sum_{1 \leq i \leq j \leq n} \alpha_{i,j} x_i x_j + \sum_{1 \leq k \leq n} \beta_k x_k + \gamma \tag{2.2}$$

2.1.2 Evaluating Multivariate Quadratic System

A multivariate quadratic polynomial system $S(\boldsymbol{x})$ is constructed by several multivariate quadratic polynomials over GF(q). It is denoted Formula (5.2) by multivariate polynomials

$$S(x) = \{Q_1(\boldsymbol{x}), Q_2(\boldsymbol{x}), \ldots, Q_m(\boldsymbol{x})\} \tag{2.3}$$

Equation (5.2) can be interpreted as a function of $\text{GF}(p^k)^n \mapsto \text{GF}(p^k)^m$. Evaluation of multivariate quadratic polynomials consists in computing the value $S(\boldsymbol{x}) = \{f_1(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})\}$. The number of monomials in a quadratic polynomial with n variables is $\binom{n+1}{2} + n + 1 = \binom{n+2}{2}$. Therefore, evaluating a quadratic polynomial require $\binom{n+2}{2} - 1 = n(n+3)/2$ additions. Moreover, each quadratic monomial and each linear term require 2 and 1 multiplications over finite fields. Hence, the number of multiplications in evaluating a quadratic polynomial is $2\binom{n+1}{2} + n = n(n+2)$. Finally, evaluating a system of m quadratic polynomials in n variables requires $mn(n+3)/2$ additions and $mn(n+2)$ multiplications over finite fields.

2.1.3 Berbain-Billet-Gilbert's Evaluation for Multivariate Quadratic Polynomial Systems

Berbain, Billet and Gilbert provide several efficient implementation techniques of evaluating multivariate quadratic polynomial system [13]. In this paper, we use the following strategies from their work.

- Variables are treated as vectors. For example, C language defines `int` as a 32-bit integer variable. Therefore, we can use `int` as a 32-vector of boolean variables. This technique is often referred to as "bitslicing" in the literature.
- We precompute each quadratic term. Because in multivariate quadratic systems,

we must compute the same $x_i x_j$ for every polynomial, so precomputing helps to save some computations.

- We compute only non-zero terms in $\mathbb{GF}(2)$. The probability of $x_i = 0$ is $1/2$, and the probability of $x_i x_j = 0$ is $3/4$. Therefore, we can reduce computational cost to about $1/4$.

2.1.4 Petzoldt's Evaluating Method with Linear Recurring Sequences

Petzoldt provides another evaluating method for multivariate quadratic polynomials citepetzoldt2013speeding.. His idea is including several multiplications in a polynomial into one multiplications by making relations between coefficients of polynomials using linear recurring sequences (LRS). Let $\gamma_1, \gamma_2, \ldots, \gamma_L$ be elements of GF(q). Then, a LRS of length L: $\{s_1, s_2, \ldots | s_i \in \mathrm{GF}(q)\}$ is given as follows:

$$s_j = \gamma_1 \cdot s_{j-1} + \gamma_2 \cdot s_{j-1} + \cdots + \gamma_L \cdot s_{j-L} \ \ \forall j > L. \tag{2.4}$$

The values $s_1, \ldots, s_L$ are the initial values of the LRS.

Alternatively, a multivariate quadratic polynomial $Q(X)$ given by Equation (5.1) can also be written as follows:

$$f(\hat{X}) = \hat{X} \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ 0 & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{n,n} & b_n \\ 0 & 0 & \cdots & 0 & c \end{pmatrix} \hat{X}^T$$
$$\hat{X} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n & x_{n+1}(=1) \end{pmatrix}. \tag{2.5}$$

Now, we assume that $\gamma \in \mathrm{GF}(q)$ is a generator of $\mathrm{GF}(q)$. Then, there is an LRS:

$$T_i = \gamma \cdot T_{i-1} + M_{i,i} \cdot x_i (i \geq 2), \tag{2.6}$$

where $M_{i,i} = \gamma^{\sum_{j=1}^{i-1} n-j+2}$, and the initial value $T_1 = x_1$. Then, every term $x_i T_i$ can be denoted as follows:

$$x_i T_i = \sum_{j=1}^{i} \gamma^{i-j} \cdot M_{j,j} \cdot x_i x_j. \tag{2.7}$$

Equation (5.6) shows that $x_i T_i$ includes every $x_i x_j$, where $i \leq j$. Hence, quadratic polynomials can be computed via the following summation:

$$f(\hat{X}) = \sum_{i=1}^{n+1} x_i T_i. \tag{2.8}$$

That is, Equation (5.7) essentially computes the following matrix:

$$f(\hat{X}) = \hat{X} \begin{pmatrix} 1 & \gamma & \cdots & \gamma^{n-1} & \gamma^{n} \\ 0 & \gamma^{n+1} & \cdots & \gamma^{2n-1} & \gamma^{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \gamma^{\binom{n+2}{2}-3} & \gamma^{\binom{n+2}{2}-2} \\ 0 & 0 & \cdots & 0 & \gamma^{\binom{n+2}{2}-1} \end{pmatrix} \hat{X}^T \tag{2.9}$$

2.2 QUAD Stream Cipher

QUAD is a stream cipher proposed by Berbain, Gilbert and Patarin [14]. QUAD uses systems of multivariate quadratic polynomials to obtain the random keystream. Therefore, it is a family of the multivariate public-key cryptography (MPKC). In other words, MPKC is a cryptographic primitive of QUAD. One of advantages of QUAD against other stream ciphers is that it has a provable security. The security of QUAD is based on the MQ assumption just like other MPKC instances, and is proved by Berbain, Gilbert and Patarin [14].

2.2.1 Multivariate Public-Key Cryptography

Cryptography is a technique to prevent data from being leaked by adversaries. Mainly, we use it on network communication. Cryptography is categorized into two types, one is symmetric key cryptography and the other is asymmetric key cryptography.

Symmetric Key Cryptography

Symmetric key cryptography uses the same keys or functions in encryption and decryption. It has two types, block cipher and stream cipher. Block cipher encrypts message block by block size. Stream cipher uses pseudorandom number generators as keystream generators. A message is encrypted with keystream in sequence.

Asymmetric Key Cryptography

Asymmetric key cryptography has two types of keys. One is a public key, which is used for encryption. The other is a private key for decryption.

Multivariate public-key cryptography (MPKC) is a cryptography, which uses a multivariate polynomials over a finite field to encryption. The security of multivariate cryptography depends on complexity of solving a non-linear multivariate polynomial system over a finite field (called the problem of $\mathcal{MQ}$ or $\mathcal{MP}$). In other words, their security is is based on the MQ or MP assumptions, i.e. if $\mathcal{MQ}$ or $\mathcal{MP}$ is hard, MPKCs are also secure.

2.2.2 Constructions of QUAD

Let p be a prime, and $q = p^k$, where $k \geq 1$. We assume that GF(q) is a degree-k extension field over GF(p). Then a multivariate quadratic polynomial in n variables over GF(q) is given by Formula eqrefQuadratics, and a system of m multivariate quadratic polynomials in n variables is defined by Formula eqrefMultiPoly.

The QUAD stream cipher uses the Equation (5.2) as a pseudo-random number generator (PRNG) to generate keystreams [14]. Generally, the notation of QUAD(q, n, r) means a construction based on a system of the n-tuple internal state value $\boldsymbol{x} = \{x_1, \ldots, x_n\}^T$ and keystream length r over GF(q) in a cycle of QUAD. In other words, QUAD(q, n, r) has three key constructions. One is the n-tuple key $\boldsymbol{x} = \{x_1, \ldots, x_n\}^T$ over GF(q). Another is the L-bit (in particular, $L = 80$) initialization vector $IV \in \{0, 1\}^L$. The last ones are 4 randomly chosen systems S_{it}, S_{out}, S_0 and S_1. Systems S_{it}, S_0 and S_1 follow from the same construction, are n quadratic equations and in n variables over GF(q). Only S_{out} is different construction, it has r quadratic equations and n variables over GF(q). System S_{out} is used to update the i-th internal state $\boldsymbol{x}_i$ to next $\boldsymbol{x}_{i+1}$, and Q is used to generate the i-th keystream $\boldsymbol{y}_i = \{y_1, \ldots, y_r\}^T$ from $\boldsymbol{x}_i$, where i is an iteration counter. Sometimes, S_{it} and S_{out} are combined to form the system S of $m = n + r$ equations in n variables over GF(q). Both S_0 and S_1 are used in the initialization step. They replace the initial state $\boldsymbol{x}_0$ just like updating $\boldsymbol{x}_{i+1}$with S_{it}.

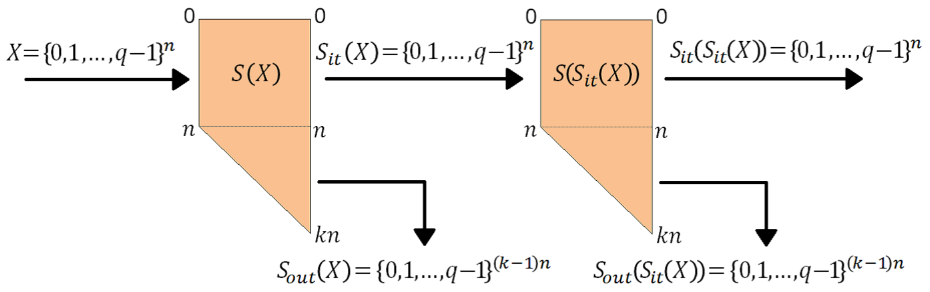
Algorithm of Keystream Generation

The algorithm of QUAD is separated in three parts, key generation, encryption/decryption of the message and initialization step. The algorithm is denoted in algorithm 1

```
Require: $n$ variables $\boldsymbol{x} = \{x_1, \ldots, x_n\}^T$.
Ensure: $m - n$ keystreams $\boldsymbol{y} = \{Q_{n+1}(\hat{x}), \ldots, f_m(\hat{X})\}$.
1: Set $T_1^{(k)} \leftarrow x_1$ for $1 \le k \le m$.
2: Compute $T_i^{(k)} = \gamma^{(k)} \cdot T_{i-1}^{(k)} + M_{i,i}^{(k)} \cdot x_i$ for $2 \le i \le n+1$, $1 \le k \le m$.
3: Compute $f_k(\hat{X}) = \sum_{i=1}^{n+1} x_i T_i^{(k)}$ for $1 \le k \le m$.
4: Output $Y = \{f_{n+1}(\hat{X}), \ldots, f_m(\hat{X})\}$ as keystreams.
5: Set $x_k \leftarrow f_k(\hat{X})$ for $(1 \le k \le n)$.
6: Back to step 1.
```

Algorithm 1: QUAD stream cipher [14]

The generated keystreams are considered to be a pseudorandom bit string and used to encrypt a plaintext with the bitwise XOR operation.



$X=\{0,1,\dots,q-1\}^n$
$S(X)$
$S_{it}(X)=\{0,1,\dots,q-1\}^n$
$S(S_{it}(X))$
$S_{it}(S_{it}(X))=\{0,1,\dots,q-1\}^n$
0
n
kn
$S_{out}(X)=\{0,1,\dots,q-1\}^{(k-1)n}$
$S_{out}(S_{it}(X))=\{0,1,\dots,q-1\}^{(k-1)n}$

Figure 2.1: Generating keystream of QUAD

2.2.3 Key and Initialization of Current State

Berbain *et al.* [14] also provides a technique for initialization of the internal state $X = (x_1, \dots, x_n)$. For QUAD(q, n, r), we use the key $K \in GF(q)^n$, the initialization vector $IV = \{0, 1\}^{|IV|}$ and two carefully randomly chosen multivariate quadratic systems $S_0(X)$ and $S_1(X)$, mapping $GF(q)^n \mapsto GF(q)^n$ to initialize X.

The initialization of the internal state X follows two steps, such that,

Initially Set Step

We set the internal state value X to the key K.

Initially Update Step

We update X for $|IV|$ times. Let i be a number of iterating initially update and $IV_i = \{0, 1\}$ be a value of i-th element of IV, and we change the value of X to

- $S_0(X)$, where $IV_i = 0$, and
- $S_1(X)$, where $IV_i = 1$.

2.2.4 Computational Cost of QUAD

The computational cost of multivariate quadratic polynomials depends on computing quadratic terms. The summation of quadratic terms requires $n(n + 1)/2$ multiplications and additions. Therefore the computational costs of one multivariate quadratic polynomial is $\mathcal{O}(n^2)$. QUAD(q, n, r) requires to compute m multivariate quadratic polynomials. Since $m = kn$, the computational cost of generating key stream is $\mathcal{O}(n^3)$.

Petzoldt's LRS QUAD

Algorithm 2 shows the keystream generation algorithm in Petzoldt's LRS QUAD stream cipher. This algorithm has two iteration steps for evaluating a quadratic polynomial. The first one is computing LRS values T_i by Equation (5.5), and the other, evaluating quadratic polynomials $f_k(\hat{X})$ by Equation (5.7). The first iteration takes n steps, each of which requires 2 multiplications and 1 addition. In addition, the second iteration takes $n+1$ steps and requires $n+1$ multiplications and n additions. Therefore, evaluating a multivariate quadratic polynomial requires $3n + 1$ multiplications and $2n$ additions. Hence, keystream generation in QUAD with m polynomials in n variables takes $3m \cdot n + m$ multiplications and $2m \cdot n$ additions.

```
Require: $(n+1)$ variables $\hat{X} = \{x_1, \dots, x_{n+1}\}$, where $x_{n+1} = 1$.
Ensure: $m - n$ keystreams $Y = \{f_{n+1}(\hat{X}), \dots, f_m(\hat{X})\}$.
1: Set $T_1^{(k)} \leftarrow x_1$ for $1 \le k \le m$.
2: Compute $T_i^{(k)} = \gamma^{(k)} \cdot T_{i-1}^{(k)} + M_{i,i}^{(k)} \cdot x_i$ for $2 \le i \le n+1$, $1 \le k \le m$.
3: Compute $f_k(\hat{X}) = \sum_{i=1}^{n+1} x_i T_i^{(k)}$ for $1 \le k \le m$.
4: Output $Y = \{f_{n+1}(\hat{X}), \dots, f_m(\hat{X})\}$ as keystreams.
5: Set $x_k \leftarrow f_k(\hat{X})$ for $(1 \le k \le n)$.
6: Back to step 1.
```

Algorithm 2: Petzoldt's LRS QUAD [42]

2.2.5 Security parameters of QUAD

The security level of QUAD is based on the MQ assumption, since Berbain, Gilbert and Patarin prove that solving QUAD needs solving MQ problem [14]. The eXtended Linearization(XL) algorithm [22] is a solving method of the $\mathcal{MQ}$. The XL constructs a polynomial system of the degree D by products of quadratic equations and monomials of the degree d, where $1 \leq d \leq D$, and solves the system as linear algebra. Then, the running time of XL depends on D. The minimal D is called the degree of regularity. Yang, Chen, Bernstein et. al. [55] show that the degree of regularity of $\mathcal{MQ}$ in QUAD(q, n, n) is given by the degree of the lowest term with a non-positive coefficient in the following polynomial,

$$G(t) = ((1-t)^{(-n-1)}(1-t^2)^n(1-t^4)^n). \tag{2.10}$$

Moreover, they give the expected running time of the XL-Wiedemann C_{XL} as the following formula.

$$C_{XL} \sim 3\tau T\mathfrak{m}. \tag{2.11}$$

A multivariate polynomial $f(X)$ can be considered as a multivariate function, which computes results with some given variables. A multivariate polynomial system is a group of such functions. The multivariate polynomial system $MP(X)$ which is constructed with m d-dimensional polynomials in n unknowns is given in Formula (3.2).

$$MP(X) = \{f_1^{(d)}(X), \ldots, f_m^{(d)}(X)\} \tag{2.12}$$

A multivariate quadratic system is a special case of the multivariate polynomial system, which uses quadratic functions $Q(X)$. The multivariate quadratic system $\mathcal{MQ}(X)$ which is constructed with n unknowns and m quadratics is also given in Formula (3.3).

$$\mathcal{MQ}(X) = \{Q_1(X), \ldots, Q_m(X)\} \tag{2.13}$$

We assume that $MP(X)$ is constructed with m d-dimensional polynomials. MP problem is to find $X = (x_1, \ldots, x_n)$ where $f_i^{(d)}(X) = 0$ for all $1 \leq j \leq m$. MP problem on a finite field is known as an NP-hard problem [40]. We can also define the $\mathcal{MQ}$ for multivariate quadratic systems $\mathcal{MQ}(X)$. It is also known as an NP-hard problem.

Solving the multivariate quadratic system means the following. Assume that we have

Table 2.1: Existing recommended parameters of QUAD.

Recommended by	Field	Unknowns	Polynomials	Parameter
Berbain *et al.* [14]	GF(2)	160	160	Broken by Yang *et al.* [55]
		256	256	80-bit security for $L = 2^{22}$.
		350	350	80-bit security for $L = 2^{40}$.
	GF(2^4)	40	40	Broken by Yang *et al.* [55]
	GF(2^{16})	20	20	Broken by Yang *et al.* [55]

known the system A of m quadratic polynomials in n variables over a finite field GF(q), given by Equation (5.2). Let $\boldsymbol{y} = \{y_1, \dots, y_m\}^T$ be a m-degree column vector, generated by multiplying the system A and the n-degree unknown column vector $\boldsymbol{x} = \{x_1, \dots, x_n\}^T$. The system (5.2) is equivalent to:

$$A\boldsymbol{x} = \boldsymbol{y}. \tag{2.14}$$

Then, the problem of finding the unknown column vector $\boldsymbol{x}$ with given A and $\boldsymbol{y}$ is called $\mathcal{MQ}$. ($\mathcal{MQ}$ means "multivariate quadratic"). More generally, solving systems of cubic or higher degree polynomials is sometimes called MP. Both $\mathcal{MQ}$ and $\mathcal{MP}$ are known to be NP-complete over GF(q) for any q [11].

The security of QUAD is based on the security of PRNG. Berbain *et al.* give the recommended parameters over GF(2), GF(2^4) and GF(2^{16}) [14]. Table 2.1 shows that existing security parameters and their status.

Standard Security Parameters in ISO/IEC 18031

ISO/IEC discusses the PRNG (called deterministic random bit generator (DRBG)) for cryptography in 18031 [7] It shows the standard security parameters of the MPKC based PRNG (i.e. QUAD). Table 2.2 shows that their recommended parameters. $\Lambda = \lambda r$ is the maximal blocks of QUAD. Therefore, $L = \Lambda k$ bits over GF(2^k). In these parameters, there exists two additional limitations and two additional parameters. First, it limits the number of variables n. Let's S be security strength bits of QUAD(q, n, r). Then, QUAD should have parameters as $nk \geq S$. Moreover, the standard requires $n \geq r$.

Table 2.2: Recommended parameters of ISO/IEC 18031 standard [7].

requested_ strength (bit)	requested_block_length (bit)			
	1-112	113-128	129-192	193-256
1-80	$n = r = 112$ $\Lambda = 2^{23}$ GF(2) $\rho_{min} = 106$ $l_{max} = 4$	$n = r = 32$ $\Lambda = 2^{12}$ GF(2^4) $\rho_{min} = 30$ $l_{max} = 5$	$n = r = 32$ $\Lambda = 2^{12}$ GF(2^6) $\rho_{min} = 30$ $l_{max} = 5$	$n = r = 32$ $\Lambda = 2^{12}$ GF(2^8) $\rho_{min} = 30$ $l_{max} = 5$
81-112	$n = 120, r = 112$ $\Lambda = 2^{26}$ GF(2) $\rho_{min} = 114$ $l_{max} = 4$	$n = r = 128$ $\Lambda = 2^{32}$ GF(2) $\rho_{min} = 122$ $l_{max} = 5$	$n = r = 48$ $\Lambda = 2^{12}$ GF(2^4) $\rho_{min} = 44$ $l_{max} = 5$	$n = r = 64$ $\Lambda = 2^{21}$ GF(2^4) $\rho_{min} = 60$ $l_{max} = 5$
113-128	-	$n = r = 128$ $\Lambda = 2^{28}$ GF(2) $\rho_{min} = 122$ $l_{max} = 4$	$n = r = 64$ $\Lambda = 2^{16}$ GF(2^3) $\rho_{min} = 60$ $l_{max} = 5$	$n = r = 64$ $\Lambda = 2^{17}$ GF(2^4) $\rho_{min} = 60$ $l_{max} = 5$
129-192	-	-	$n = 200, r = 192$ $\Lambda = 2^{32}$ GF(2) $\rho_{min} = 192$ $l_{max} = 4$	$n = r = 128$ $\Lambda = 2^{30}$ GF(2^2) $\rho_{min} = 124$ $l_{max} = 4$
193-256	-	-	-	$n = 272, r = 256$ $\Lambda = 2^{32}$ GF(2) $\rho_{min} = 264$ $l_{max} = 4$

Also, ISO/IEC 18031 standards added two parameters for the rank of matrix. One is the minimum rank of matrix ρ_{min}. This parameter requires that the system matrix of a multivariate quadratic equation system in QUAD should have the rank at least ρ_{min}. The other is the maximum weight l_{max}. It requires that all sums of at most l_{max} of quadratic equations in QUAD should have at least ρ_{min}.

Drelikhov, Marshalko, and Pokrovskiy gives further evaluation of these standards by the meet-in-the-middle technique [23]. They suggest that these standard parameters have less security level (e.g. for $n = 200$ and $r = 192$ (192-bit security) downs to 129-bit security).

2.3 Finite fields of MPKC

MPKC is assumed that it works over finite fields. 2, power of 2, and odd primes are expected to q of GF(q) in MPKC. Each field has different characteristic, hence, we should carefully select it.

Binary field

The binary field (i.e. GF(2)) is the simplest in the finite fields. In the binary field, there exists only 2 values 0 and 1. Therefore, sometimes, the binary field can be handled as the boolean algebra. Then, the exclusive-or (xor, XOR) and the logical conjunction (AND) correspond to addition and multiplication over GF(2). Hence, MPKC using GF(2) is the most efficient for speed.

On the other stand, the security of the binary field is the weakest against algebraic attack. Since field equation $x^2 = x$, the size of a linearized system from $mathcalMQ$ is the smallest than other fields for same number of unknowns and equations. Therefore, it is need to choose large numbers of unknowns and equations.

Binary extension field

Binary extension fields are extended from the binary field with a primitive polynomial. In these fields, additions and multiplications are implemented by vector additions and polynomial modular multiplications over the binary field. Hence, additions over these field are efficient, however, multiplications over them are not. Some small extension fields (e.g. GF(2^4) or GF(2^8)) are implemented multiplications as looking up tables. Such a method can be reduced the computational cost of field operations.

On the other stand, the security of these field is more strong than the binary field. Because, the order of them q can be denoted as $q = 2^k$ (k is the extension degree from GF(2). Hence, the field equation $x^q - x = 0$ appears only high degree cases (like $q = 2^4 = 16$). Usually, we choose GF(2^8) for MPKC because of the tradeoff between the efficiency and the security.

Odd Prime Field

Operations of odd prime fields are implemented by modular operations. For example, multiplications $a \times b$ over GF(3) ($a, b \in$ GF(3)) are defined as $a \times b := a * b \text{mod} 3$. Therefore, the efficiency of field operations depends on the number of characteristic. Chen *et al.* show that multiplications over GF(31) is faster than GF(2^4) and GF(2^{16}) [19].

For each odd prime field, the security level is same with other prime fields. The security is only decreases by the degree of regularity of systems D_{reg}. It depends on the field equations $x^q - x = 0$. Usually, we choose GF(31) for MPKC because of the tradeoff between the efficiency and the security.

2.3.1 Selected Fields for QUAD

We select GF(2) and GF(2^{32}) for implementations of QUAD and GF(2), GF(3), GF(5), and GF(7) for security evaluation of QUAD. GF(2^{32}) has an advantage than other fields. There exists many researches, because many machine word size is 32-bit. Therefore, GF(2^{32}) may accelerates by the efficient operations (like optimal extension fields (OEF)). This is reason of our selecting.

$GF(3)$, $GF(5)$ and $GF(7)$ may become alternative fields of the binary field. Because, it is known that the binary field requires large constructions of systems for the security. Therefore, they are candidates of alternative lightweight fields after breaking systems over GF(2). Hence, we discuss the security of MQ with systems over them.

2.3.2 Operations over Extension Field

Let p be a prime and $q = p^k$. Then, there exists degree k extension fields $\mathrm{GF}(p^k) = \mathrm{GF}(q)$ of $\mathrm{GF}(p)$. Generally, $\mathrm{GF}(q)$ can be defined by a degree k primitive polynomial $f(X)$. Then X is a primitive element of $\mathrm{GF}(q)$, if $f(X) = 0$. Since finite extensions of finite fields are Galois extensions, there is a Galois group $\mathrm{Gal}(\mathrm{GF}(q)/\mathrm{GF}(p))$ given by following formula,

$$\mathrm{Gal}(\mathrm{GF}(q)/\mathrm{GF}(p)) = \{\sigma : \mathrm{GF}(q) \mapsto \mathrm{GF}(q) | \mathit{automorphism} : \sigma(\alpha) = \alpha \ (\forall \alpha \in \mathrm{GF}(p))\}.$$

If τ defines the Frobenius mapping of $\mathrm{GF}(q)/\mathrm{GF}(p)$, the $\mathrm{Gal}(\mathrm{GF}(q)/\mathrm{GF}(p))\}$ is cyclic group, generating by τ.

We can denote an element $a \in \mathrm{GF}(q)$ by a vector over $\mathrm{GF}(p)$ as follows:

$$a = \{c_1, \ldots, c_k\}, \ (c_1, \ldots, c_k \in \mathrm{GF}(p)), \tag{2.15}$$

where we have fixed the basis $\{X_1, \ldots, X_k\}$ of the extension $\mathrm{GF}(q)/\mathrm{GF}(p)$:

$$a = c_1 X_1 + \cdots + c_k X_k = \sum_{i=1}^{k} c_i X_i, \tag{2.16}$$

In this paper, we discuss the following 2 bases,

Polynomial basis: constructed by a primitive element $X \in \mathrm{GF}(q)$ such that $\{1(= X^0), X, \ldots, X^{k-1}\}$.

Normal basis [45]: we assume given an element $\alpha \in \mathrm{GF}(q)$ for a finite Galois extension $\mathrm{GF}(q)/\mathrm{GF}(p)$ such that $\{\sigma(\alpha) | \sigma \in \mathit{Gal}(\mathrm{GF}(q)/\mathrm{GF}(p))\}$. Then, basis is given by $\{\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{k-1}}\}$

$\mathrm{GF}(q)$ can be handled as a residue class ring of the polynomial ring $\mathrm{GF}(p)[X]$ modulo $f(X)$. Given $a, b \in \mathrm{GF}(q)$, we denote by $a(X), b(X)$ their representative polynomials in $\mathrm{GF}(p)[X]/\langle f\rangle$. Therefore, additions and multiplications of $\mathrm{GF}(q)$ can be denoted as following formulas,

$$\begin{aligned} a+b &:= a(X)+b(X)\ \mathit{modf}(X), \\ a*b &:= a(X)*b(X)\ \mathit{modf}(X). \end{aligned}$$

Since a can be handled as a vector of $\mathrm{GF}(p)$ like in Equation (4.1), additions of $\mathrm{GF}(q)$ are computed by:

$$a+b := \{a_1+b_1\ \mathit{mod}\ p, \ldots, a_k+b_k\ \mathit{mod}\ p\}, \tag{2.17}$$

Zech's Logarithm

Originally, Zech's logarithm (also called Jacobi's logarithm [45]) is proposed to figure additions for elements represented as powers of a generator of a cyclic group $\mathrm{GF}(q)^* = \mathrm{GF}(q)\setminus\{0\}$. Zech's logarithm is considered to be a method of efficient exponentiation over cyclic groups for cryptosystems [26, 27]. Let γ be a generator of $\mathrm{GF}(q)^*$. Then, $\mathrm{GF}(q)^* = \langle\gamma\rangle$. Therefore, we can represent any element in $\mathrm{GF}(q)^*$ as γ^ℓ, where ℓ is an integer. In particular, $\gamma^\ell \neq \gamma^{\ell'}$, $0 \leq \ell \neq \ell' \leq p^r-2$. In this way, $\mathrm{GF}(q)^*$ can be represented by $[0, p^r-2]$. Hence, multiplications over $\mathrm{GF}(q)^*$ can be computed by integer additions modulo p^r-1.

Intermediate Field [45]

Let k be a composite integer for $q=p^k$. Then, there exists l, where $l \mid k$ and $1<l<k$. $\mathrm{GF}(q^l)$ is an extension field of $\mathrm{GF}(q)$ and a subfield of $\mathrm{GF}(p^k)$. We call $\mathrm{GF}(p^l)$ an intermediate field. Because, any extension of $\mathrm{GF}(q)/\mathrm{GF}(p)$ are isomorphism, we can compute operations of $\mathrm{GF}(p^k)$ as extension from $\mathrm{GF}(p^l)$.

2.4 The $\mathcal{MQ}$ Problem

The security of MPKC is largely based on the complexity of solving a system of multivariate non-linear equations over finite fields. The $\mathcal{MQ}$ is a quadratic case of this problem. Generic $\mathcal{MQ}$ is known to be NP-complete [11].

Let $q = p^k$, where p is a prime, and $\boldsymbol{x} = \{x_1, \ldots, x_n\}$ $(\forall i, x_i \in \mathrm{GF}(q))$. Generally, multivariate quadratic polynomial equations in n unknowns over GF(q) can be described as follows:

$$f(\boldsymbol{x}) = \sum_{1 \le i \le j \le n} \alpha_{i,j} x_i x_j + \sum_{1 \le k \le n} \beta_k x_k + \gamma = 0, \tag{2.18}$$

where $\forall i, j$, $\alpha_{i,j}, \beta_i, \gamma \in \mathrm{GF}(q)$. The $\mathcal{MQ}$ consists solving quadratic polynomial equations given by $\boldsymbol{y} = \{f_1(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})\}$

The original XL algorithm was proposed by Courtois *et al.* in 2000 [22]. The idea of XL is based on a linearization technique, in which new unknowns representing non-linear terms, e.g., $y_{1,2} = x_1 x_2$, are generated and treated as an independent variable. If the number of equations is greater than the number of variables in the resulted linearized system, then we can solve it by, e.g., Gaussian elimination. If not, we can generate new equations from the original ones by raising to a higher degree. For the sake of completeness, the XL algorithm is described in Algorithm 3. Simply put, the *degree of regularity D* is the minimal degree at which the number of linearly independent equations exceeds the number of unknowns in the linearized system.

The XL algorithm generates sparse equations in Step 1 of Algorithm 3. The number of non-zero terms of an equation is only $\binom{n+2}{2}$ out of all possible $\binom{n+D}{D}$ terms, since the generated equations are just a product of the original equations and some monomials. However, the Gaussian elimination is not suited for solving such sparse linear systems, as it cannot take advantage of the sparsity. The XL-Wiedemann algorithm [36] addresses this problem of the original XL by replacing the Gaussian elimination with the Wiedemann algorithm [52], which is more efficient for solving systems of sparse linear equations.

The Wiedemann algorithm [52] is a solving method for a system of linear sparse equations over finite fields. Let A be an $N \times N$ non-singular matrix over GF(q). The Wiedemann algorithm finds a non-zero vector $\mathbf{x}$, where $\mathbf{y} = \mathbf{Ax}$. The original Wiedemann algorithm is described in Algorithm 4.

2.5 Parallel Computing on Graphics Processing Unit

In the early period, computers has some special chips for graphics processing. Their chips supports rendering for the 2D graphics (e.g. filling rectangles). After middle of 1980s, several workstations are provided to the 3 dimensional computer graphics (3DCG). Such a workstation makes innovations for 3DCG. In the 1990s, some personal computers have special graphic chips on their boards. In 1999, NVIDIA provides a graphics cards named NVIDIA GeForce 256 for supporting 3DCG acceleration. Also, NVIDIA names such graphics accelerators to the Graphics Processing Unit .

2.5.1 General Purpose computing on GPUs

Graphics Processing Unit (GPU) is processor for handling 3 dimensional computer graphics. In a 3DCG world, every elements (e.g. arrangement of objects, an environment of light sources, and appearance in a camera) are constructed by computation. Therefore, 3DCG requires large amount of computation to display graphics. Because 3DCG computation is a heavy process, it is desirable that CPUs do not handle 3DCG computation. In order to avoid to use CPUs for 3DCG computation, GPUs was developed. In this chapter, we explain about graphics processing units (GPUs) and application of them to general purpose work called general purpose computing on graphics processing units (GPGPU).

In early period, GPUs did not have a huge computational power. It was just an alternative processor to CPUs. However, recently, the computational power of GPUs have been making rapid progress. Because drawing the high quality 3DCG in real time (e.g. in online network games) is enormous heavy. In order to realize real time drawing, GPUs have been developed by increasing cores. Nowadays, GPUs have hundreds level cores, it is called many cores constructions. Such cores are specialized floating-point computation.

Floating-point operations per second (FLOPS) is a measure of computational power. Although recent GPUs reaches TFLOPS level computational power, recent CPUs have at most 200 GFLOPS.

General Purpose computing on GPU (GPGPU) is a technique for any general process by using GPUs. In cryptography, it is used for some implementations. For example, Manavski proposed an implementation of AES on GPU, which is 15 times faster than an implementation on CPU, in 2007 [33]. Moreover, Osvik *et al.* presented a result of an over 30 Gbps GPU implementations of AES, in 2010 [38]. On the other hand, the GPGPU technique is also used for cryptanalysis. Bonenberger *et al.* used a GPU to generating polynomials of the General Number Field Sieve [16].

Because GPUs are designed based on SIMD, it is better to handle several simple tasks simultaneously. On the other hand, the performance of a GPU core is not higher than CPU. Therefore, if we use GPU for sequential processing, it is not effective. In the GPGPU techniques, how to parallelize algorithms is an important issue.

GPGPU Programming Environment

In early period, GPGPU technique was achieved with graphics libraries as OpenGL and DirectX However, such tools need to output computer graphics while processing work. Therefore, these tools are not efficient.

Then, GPU developers provides programming environments for GPU computing. NVIDIA gives Compute Unified Device Architecture (CUDA) for its products. Also, Advanced Micro Devices (AMD) provided AMD stream to GPGPU for AMD GPUs.

Another movement is development for heterogeneous environments. Khronos Group formulates an open standard for such environments as Open Computing Language (OpenCL).

2.5.2 CUDA Computing

In this dissertation, we focus on programming NVIDIA GPUs on CUDA API. CUDA API is specialized for NVIDIA GPU architectures, hence, it gives an optimized implementation environment for them.

Each GPU is equipped with several Simultaneous Multiprocessors (SMs). The number of SMs is depends on architectures of graphics chips on GPUs. For example, there are from 9 SMs to 16 SMs on the Fermi architecture with the high-end class of the GeForce 400 series and GeForce 500 series. On the other hand, in the Kepler architecture, which is the successor of the Fermi, there are from 2 SMs to 15 SMs.

In CUDA, hosts correspond to computers, and devices correspond to graphic cards. CUDA works by making the host control the device. Kernel is a function the host uses to control the device. Because only one kernel can work at a time, a program requires parallelizing processes in a kernel. A kernel handles some blocks in parallel. A block also handles some threads in parallel. Therefore a kernel can handle many threads simultaneously.

CUDA is a development environment for NVIDIA's GPUs [3]. In CUDA, hosts correspond to computers, whereas devices correspond to GPUs. In CUDA, a host controls one or more devices attached to it. A kernel is a function that the host uses to control the device(s). A kernel handles several number of blocks in parallel. A block also handles multiple threads in parallel. Therefore, a kernel can handle many threads simultaneously.

CUDA API is a development environment for GPU, based on C language and provided by NVIDIA [3]. Pregnancy tools for using GPU have existed before CUDA is proposed. However, such tools as OpenGL and DirectX need to output computer graphics while processing work. Therefore, these tools are not efficient. CUDA is efficient, because CUDA uses computational core of GPU directly.

2.5.3 GPU architectures and CUDA Capability

Since early days of CUDA, NVIDIA has developed GPU architectures with improvements. Sometimes, these architectures are significant changed from previous one.

Before the Fermi architecture

Until the Fermi architecture (actually up to GeForce GTX 200 series and 300 series), NVIDIA had designed GPU chips for each generation.

Memory loading for CUDA

Also, considerations about memory loading are important. Originally, memory loadings in a warp are executed serially. However, when memory requests of threads in a warp are consecutively, these requests are coalesced to 1 large memory request [1]. In other words, such memory loadings are executed at a time. Therefore, data structures should be consecutively for memory requests in a warp.

2.5.4 GPGPU for Cryptography

In cryptography, GPGPU is used for some implementations and cryptanalysis researches. The oldest cryptographic GPGPU work is Kadem *et al.* [29] in 1999. They tried to brute force attack on a graphics processor they called PixelFlow . It was not been a graphics processor are commercially available.

Chapter 3

Parallelisation of Evaluating Multivariate Quadratic Polynomial

3.1 Introduction

Background

Nowadays cryptography is a necessary technology for network communication. Multivariate cryptography uses multivariate polynomials system as a public key. The security of multivariate cryptography is based on the hardness of solving non-linear multivariate polynomial systems over a finite field [11]. Multivariate cryptography is considered to be a promising tool for fast digital signature, because it requires just computing multivariate polynomial system.

QUAD is a stream cipher, which uses a multivariate quadratic system [14]. Symmetric ciphers are used to authentication schemes [35] and signatures [39]. The security of QUAD depends on the multivariate quadratic ($\mathcal{MQ}$) problem. Therefore QUAD has provable security like public key cryptography though it is a symmetric cipher. QUAD has high security, but it is very slow compared with other symmetric ciphers. When QUAD stream cipher is accelerated, we can realize high security communication with QUAD.

Related Works

Berbain *et al.* [13] provided efficient implementation techniques for multivariate cryptography including QUAD stream cipher on CPUs. They implemented 3 cases of QUAD instances, over $GF(2)$, $GF(2^4)$, and $GF(2^8)$. Arditti *et al.* [8] showed FPGA implementations of QUAD for 128, 160, 256 bits blocks over $GF(2)$. Chen *et al.* [20] presented throughputs of a GPU implementation of QUAD for 320 bits blocks over $GF(2)$. However the results show that GPU implementations are slower than ideal CPU implementations.

Most of these related works just implemented several QUAD instances. They did not evaluate computational costs of QUAD stream ciphers. Only Berbain *et al.* [13] showed the computational costs of QUAD with n unknowns and m multivariate quadratics, which are $\mathcal{O}(mn^2)$. We extended several implementation strategies for multivariate quadratic of Berbain *et al.* to GPU implementations and evaluated the computational cost of QUAD [47].

This is an extension work of our previous result [47]. We present extended GPU implementation results from $GF(2)$ case to $GF(2^p)$ cases, and comparisons with other works. Moreover, we refine the evaluations of computational costs of QUAD for general cases and optimized $GF(2)$ cases.

Motivation

Our goal is to implement efficient QUAD stream cipher. Since QUAD has a rigorous security proof as public key cryptography, we can use a fast and secure cipher when QUAD becomes fast like other stream ciphers.

Our Contribution

We provide two techniques to implement QUAD stream cipher. One is a parallel implementation for computing multivariate polynomials. The other is an optimization technique for implementing QUAD on GPUs.

In this paper, we discuss the computational time for generating keystreams of QUAD in more detail than [47]. Moreover, we report results of implementation of QUAD stream cipher over $GF(2)$, $GF(2^2)$, $GF(2^4)$, and $GF(2^8)$ on GPU.

3.2 QUAD Stream Cipher over GF(2)

3.2.1 Multivariate Quadratic Polynomials

We use a finite field $GF(q)$. Let $X = (x_1, \dots, x_n)$ be a n-tuple variable of $GF(q)$, we describe monomials as $\alpha^{(k)}_{s_1,\dots,s_k} \prod_{i=1}^{k} x_{s_i}$, where $k \geq 0$, $1 \leq s_1 \leq \cdots \leq s_k \leq n$. $\alpha^{(k)}_{s_1,\dots,s_k}$ is a coefficient of a k-dimensional monomial. Therefore, they consist of a coefficient and k variables. If a dimension of a monomial is 0, it is called a constant.

Especially when $k = 2$, polynomials are called quadratics. Let $Q(X)$ be a multivariate quadratics, and Formula (3.1) presents $Q(X)$ with n unknowns,

$$Q(X) = \sum_{1 \leq i \leq j \leq n} \alpha_{i,j} x_i x_j + \sum_{1 \leq k \leq n} \beta_k x_k + \gamma, \tag{3.1}$$

where $\alpha_{i,j} = \alpha^{(2)}_{i,j}$, $\beta_k = \alpha^{(1)}_k$ and $\gamma = \alpha^{(0)}$.

Evaluating Multivariate Quadratic System

A multivariate polynomial $f(X)$ can be considered as a multivariate function, which computes results with some given variables. A multivariate polynomial system is a group of such functions. The multivariate polynomial system $\mathcal{MP}(X)$ which is constructed with n unknowns and m d-dimensional polynomials is given in Formula (3.2).

$$\mathcal{MP}(X) = \{f^{(d)}_1(X), \dots, f^{(d)}_m(X)\} \tag{3.2}$$

A multivariate quadratic system is a special case of the multivariate polynomial system, which uses quadratic functions $Q(X)$. The multivariate quadratic system $\mathcal{MQ}(X)$

which is constructed with n unknowns and m quadratics is also given in Formula (3.3).

$$\mathcal{MQ}(X) = \{Q_1(X), \ldots, Q_m(X)\} \tag{3.3}$$

We assume that $MP(X)$ is constructed with m d-dimensional polynomials. MP problem is to find $X = (x_1, \ldots, x_n)$ where $f_i^{(d)}(X) = 0$ for all $1 \leq j \leq m$. MP problem on a finite field is known as an NP-hard problem [40]. We can also define $\mathcal{MQ}$ problem for multivariate quadratic systems $\mathcal{MQ}(X)$. It is also known as an NP-hard problem. The security of QUAD stream cipher depends on the MQ assumption.

3.2.2 Recommended Parameters of QUAD over GF(2)

QUAD is a stream cipher which is proposed by Berbain *et al.* [14]. However, it is a stream cipher, and the security of it is based on the MQ assumption.

Constructions and notation

Generally, the notation of QUAD(q, n, r) means a construction based on a system of the n-tuple internal state value $\boldsymbol{x} = \{x_1, \ldots, x_n\}^T$ and keystream length r over GF(q) in a cycle of QUAD. On the other hand, it shows that a system of QUAD as $m = n + r$ quadratic equations in n variables over GF(q), and a system in QUAD are given in Equation (5.2). Usually, m is set to kn, where $k \geq 2$, and therefore $r = (k-1)n$.

QUAD(q, n, r) has three key constructions. One is the n-tuple key $\boldsymbol{x} = \{x_1, \ldots, x_n\}^T$ over GF(q). Another is the L-bit (in particular, $L = 80$) initialization vector $IV \in \{0,1\}^L$. The last ones are 4 randomly chosen systems P, Q, S_0 and S_1. Systems P, S_0 and S_1 follow from the same construction, are n quadratic equations and in n variables over GF(q). Only Q is different construction, it has n quadratic equations and n variables over GF(q). System P is used to update the i-th internal state $\boldsymbol{x}_i$ to next $\boldsymbol{x}_{i+1}$, and Q is used to generate the i-th keystream $\boldsymbol{y}_i = \{y_1, \ldots, y_r\}^T$ from $\boldsymbol{x}_i$, where i is an iteration counter. Sometimes, P and Q are combined to form the system S of $m = n+r$ equations in n variables over GF(q). Both S_0 and S_1 are used in the initialization step. They replace the initial state $\boldsymbol{x}_0$ just like updating $\boldsymbol{x}_{i+1}$with P.

According to the cryptanalysis of QUAD by Yang *et al.* [55], QUAD$(2, 160, 160)$ has 2^{140} security against direct attack (i.e. solving the $\mathcal{MQ}$ problem with 320 polynomials in 160 unknowns over GF(2)).

Security against Distinguish Attack

Now, we extend the security discussion against the distinguish attack based on the evaluation by Bard [11]. The security proof by Berbain, Gilbert, Patarin citeberbain2006quad shows that if there exists the algorithm A which distinguishes a L-bit $(L = \lambda r = \lambda(k-1)n)$keystream generated by QUAD PRNG from a L-bit uniformly random keystream in time T_A and with the advantage ϵ, then, there exists the algorithm C which preimages the polynomial functions of QUAD in the time T_C with the probability at least $\frac{\epsilon}{8\lambda}$. The time T_C is given as the following:

$$T_C \leq \frac{2^7 n^2 \lambda^2}{\epsilon 2}(T_A + (\lambda+2)T_S + \log(\frac{2^7 n \lambda^2}{\epsilon 2})) + \frac{2^7 n \lambda^2}{\epsilon^2} T_S, \tag{3.4}$$

where T_S is the running time of QUAD. Therefore, T_C shows that On the other stand, if there exists the algorithm C which preimages polynomials of QUAD in time T_C, then, we can distinguish a L-bit keystream from a L-bit uniformly random stream in time T_A by backward of Equ. (3.4). In other words, if C is the best attack algorithm for QUAD, T_A is the lower bound of the distinguish attack. We assume that the best attack for QUAD over GF(2) is solving $\mathcal{MQ}$ by the Gröbner basis attack. Bardet provides an analysis of the complexity of F_5, which is one of the fastest Gröbner implementation, in her Ph.D. thesis [12]. In her approximately evaluation, solving MQ with a system of $m(= kn)$ polynomials in n unknowns by F_5 takes T_{F_5}, which is given as the following:

$$T_{F_5} = \binom{n+1}{D}^{2.37}. \tag{3.5}$$

Over GF(2), the degree of regularity D is close to:

$$(-k + \frac{1}{2} + \frac{1}{2}\sqrt{2k^2 - 10k - 1 + 2(k+2)\sqrt{k(k+2)}})n. \tag{3.6}$$

Moreover, we assume that each inspection is achieved in 1 CPU-cycle, and we don't consider memory loading. Figure 3.1 shows that the security evaluation of QUAD over GF(2) for $L = 2^k$-bit ($k = 10, 20, 30, 40$) stream. According this figure, we can show the security parameters against distinguish attack. Table 3.1 shows the security parameters over GF(2).

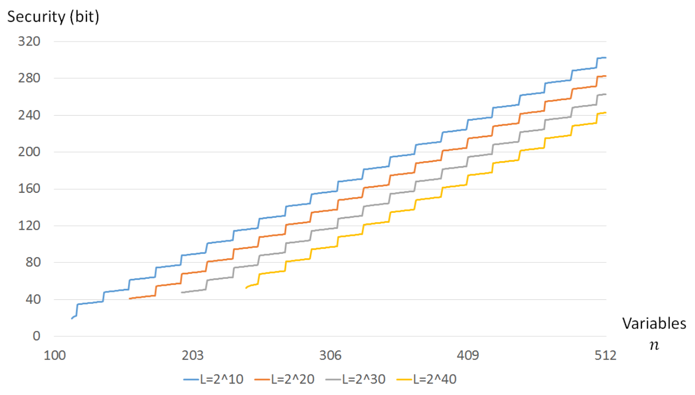


Figure 3.1: Security evaluation against distinguish attack over GF(2).

Table 3.1: Security parameter of QUAD(2,n,r).

L	$\log L$	80-bit	100-bit	120-bit	128-bit
1 Kbit	10	$n = r = 195$	$n = r = 214$	$n = r = 253$	$n = r = 255$
1 Mbit	20	$n = r = 214$	$n = r = 253$	$n = r = 273$	$n = r = 292$
1 Gbit	30	$n = r = 253$	$n = r = 273$	$n = r = 312$	$n = r = 313$
1 Tbit	40	$n = r = 273$	$n = r = 312$	$n = r = 331$	$n = r = 351$

3.3 Parallelization Strategies of Evaluating Quadratic Polynomials

3.3.1 Parallelizing on the GPU

In GPGPU, the most important point is the parallelization of algorithms. Because the performance of a single GPU core is worse than that of CPU, serial implementations with GPU are expected to be slower than CPU implementations.

Since the polynomials of a multivariate quadratic system are independent of each other, parallelization of a system is straightforward. Moreover, we parallelize the evaluation of each polynomial in a multivariate quadratic system. We propose two parallelization techniques, as shown in Figure 3.4.

The Basic Strategy of Parallelization

Let $t_{i,j} = \alpha_{i,j}x_ix_j$. Summation of quadratic terms can be considered as summation of every element of a triangular matrix, as shown on the left side of Figure 3.2. We assume that other elements from the matrix are zero. Therefore, we can compute summation of quadratic terms as summation of a rectangular matrix, as shown on the right side of Figure 3.2. Then, we can compute the summation as $\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_{i,j}x_ix_j = \sum_{i=1}^{n}\sum_{j=1}^{n}t_{i,j}$ as follows.

(1) We compute $S_k(x) = \sum_{i=1}^{n} t_{k,i}$ for all k in parallel.

(2) We compute $\sum_{k=1}^{n} S_k(x)$.

However, such a strategy introduces some overhead caused by the extra unnecessary computations.

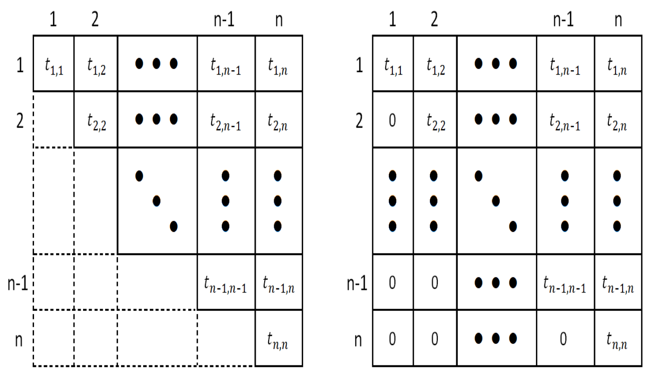


Figure 3.2: Left: Evaluating quadratics on a triangular matrix. Right: Evaluating quadratics on a padded rectangle matrix.

Parallelization Method 1

Next we introduce the first strategy to reduce unnecessary computations. We reshape a triangular matrix to a rectangular matrix as shown in Figure 3.3, in which method of matrix reshaping is depicted. By this reshaping, we can efficiently reduce about 25% of the cost for evaluating a multivariate quadratic polynomial system.

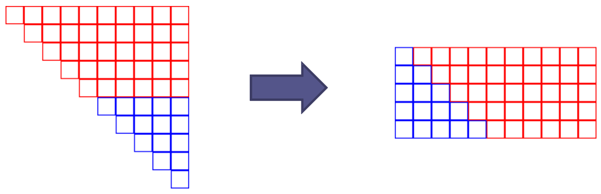


Figure 3.3: Reshaping triangular to rectangular matrix

Parallelization Method 2

In the second strategy, we treat a polynomial as a vector as opposed to a matrix. Assuming that n_c is the number of GPU cores, we separate a vector into n_c-sub-vectors.

Moreover, we use the parallel reduction technique citeparallel reduction to compute all sub-vectors in parallel. The parallel reduction technique works as follows.

(1) We substitute the length of sub-vectors for n_c.

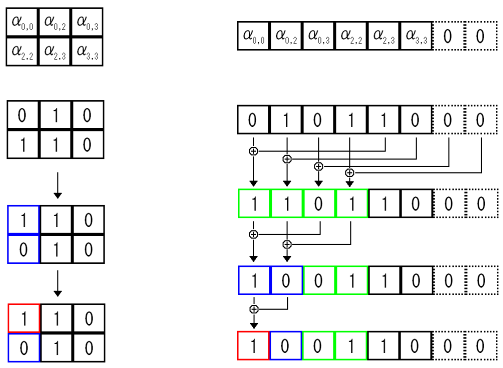

$\alpha_{0,0}$ $\alpha_{0,2}$ $\alpha_{0,3}$
$\alpha_{2,2}$ $\alpha_{2,3}$ $\alpha_{3,3}$
$\alpha_{0,0}$ $\alpha_{0,2}$ $\alpha_{0,3}$ $\alpha_{2,2}$ $\alpha_{2,3}$ $\alpha_{3,3}$ 0 0

Figure 3.4: Parallelization strategies. Left: Strategy 1; right: Strategy 2

(2) We add $n_c/2 + i$-th elements to i-th elements.

(3) We compute $n_c = n_c/2$.

(4) While n_c is larger than 1, we iterate step 2 and 3.

The entire parallel reduction technique consists of $\log n_c$ iterations of the above steps. Therefore, we can evaluate polynomials efficiently.

3.3.2 Optimization on GPU Architectures

On GPU implementations, we must consider its characteristics. Together, the cores on a GPU provide a tremendous amount of computing power, but each single GPU core is much slower than a CPU core. Therefore, we need to minimize the number of inactive GPU cores.

Optimization of Matrix Calculation

An NVIDIA GeForce GTX 580 GPU has 16 SMs, each of which has 32 CUDA cores. Since each SM handles 32 threads at a time, the number of threads should be an integral multiple of 32. In the same way, we should make sure that the algorithm can be handled by 16 SMs in parallel. Together, the total number of threads should be an integral multiple of $32 \times 16 = 512$.

In parallelization method 1, we can compute an summation in a polynomial as multiple co-summations of rows of a matrix. An n-unknown quadratic polynomial has $n(n+1)/2$ monomials. Then the long side of a rectangular matrix that is reshaped from an n-dimensional triangular matrix has n or $n+1$ elements. Although a number of a long side's elements can be counted in a process, counting incurs extra cost in the computation. Therefore, we assume that $n = 31k$, where k is a natural number. By handling a summation in a polynomial as a triangular matrix which elements are k-dimensional square submatrices, we can handle a summation as a 16×31 rectangle matrix, as shown in Figure 3.5. Thus we can parallelize the calculation of a matrix for 16 SMs with 32 CUDA cores per SM.

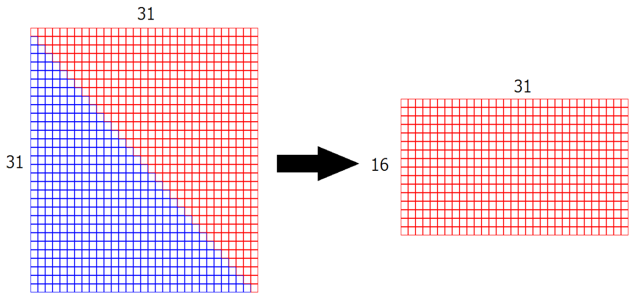


Figure 3.5: Handling as a 16×31 matrix

In parallelization method 2, we can parallelize a summation by the number of cores that can efficiently share data. In CUDA, we can share data in a block. Then we can parallelize a summation by 32 monomials on NVIDIA GeForce GTX 580. Therefore, we assume that $n = 32k$, where k is a natural number. Iterating time of parallelize reduction in a summation is $k(32k+1)/2$.

Further Optimizations

In order to improve the efficiency, we need to break down the computation into small chunks of similar computations for parallel processing. Moreover, GPUs can't handle conditional branches efficiently, so we need to handle conditional branches differently than we do on CPU. In this case, we use a different kernel for each different number of non-zero terms. However, using a kernel for each possible number of non-zero terms would incur an extremely large amount of overhead. Therefore, we make kernels just every number of k. For example, for QUAD$(2, 512, 512)$, the maximum k is 17, so we need only 17 kernels.

3.3.3 Analysis of Potential Speedup

Parallelization Speedup

Originally, each polynomial in QUAD(q, n, n) requires $(n + 1) \times (n + 2)/2$ additions and multiplications. Moreover, QUAD(q, n, n) requires evaluation of $2n$ polynomials. Using the strategies proposed by Berbain *et al.* [14], we can compute each polynomial in QUAD(q, n, n) with $(n + 1) \times (n + 2)/8$ additions and multiplications. Therefore, we can compute QUAD(q, n, n) with $n/16$ times the cost of evaluating a single polynomial using 32-bit vectors.

Such techniques can be used by CPU implementation as well as GPU implementation. By parallelization on GPU, we can compute QUAD(q, n, n) in parallel. We can compute multiplications of a polynomial before additions. We can compute $\alpha_{i,j}x_ix_j$ in n multiplications time by we parallelize multiplications in each i and compute by every j. When we use a multivariate polynomial system over $\mathbb{GF}(2)$, we can compute multiplications by reducing monomials with a strategy of Berbain *et al.*

Parallelization method 1 with optimizations computes a summation in a polynomial by as a rectangle matrix, which elements are k-dimensional square submatrices. Since NVIDIA GeForce GTX 580 has 16×32 cores, each submatrices can be computed on each CUDA cores in parallel. Then, computational time of summations k-dimensional matrices is k^2 additions. Moreover, we should compute submatrices of every polynomials, then it takes mk^2; m is the number of polynomials divided by 32. After that, we compute row co-summations in matrices. So we can compute row co-summations at one time, computational cost of row co-summations is 31 additions. When $m \leq 32$ (the number of polynomials ≤ 1024), we can compute row co-summations of all polynomials in once time. Finally, we compute a summation of row co-summations' result in 15 additions. Then, the computational costs of summations of a multivariate quadratic polynomial system can be denoted by $mk^2 + 46$ additions.

In parallelization method 2, we compute summations by parallel reductions. Parallel reductions can be computed co-summations of 32 elements on NVIDIA GeForce GTX 580 at once. Then, co-summations can be computed in 5 additions. Assuming $n = 32k$, we can compute co-summations of a polynomial by $k(32k + 1)/2$ times. Since we can compute 16 co-summations at once, actually, we can compute co-summations by $\lceil k(32k+1)/32 \rceil$ times. When $n \leq 512 = 32 \times 16$, we can compute co-summations at most $n/2$ times. Finally, we compute summations of co-summations' result of a polynomial as parallelization method 1. Then, the computational costs of summations of a multivariate quadratic polynomial system can be denoted by $(5m + 1)n/2$ additions.

Another Strategy for Multiplications

For evaluating quadratic polynomials, we can achieve an alternative method for multiplications over GF(2). Because, if and only if $x_i = 1$ and $x_j = 1$, then $x_i x_j = 1$, we don't have to compute multiplications over GF(2). Moreover, non-zero variables are common in each polynomial $f_k(x)$. Therefore, we should only check $x_i = 1$. Since this checking is sequential, this step is executed on CPU. In this alternative method, GPU only computes summations of coefficients, which terms $x_i x_j$ are non-zero.

Table 3.2 shows the time complexity of our parallelization methods on GPU with exiting evaluation method [13]. Each addition over GF(2) can computed by xors. Also, multiplications are computed by logical conjunctions. In the table, XOR and AND shows numbers of xors and logical conjunctions. C shows that the number of cores on a GPU.

Table 3.2: Time complexity of evaluation QUAD$(2, n, r)$ $(m = n + r)$.

Evaluating method	Processor	XOR	AND	IF
Naïve method	CPU	$\frac{mn(n+1)}{2}$	$m(n^2+1)$	-
BBG 2006 [13]	CPU	$\lceil \frac{m}{32} \rceil \frac{n(n-1)}{8}$	$\frac{n(n-1)}{8} + \lceil \frac{m}{32} \rceil \frac{n(n-1)}{8}$	n
Our method 1	CPU	-	-	n
	GPU	$\lceil \frac{m}{32} \rceil k^2 + 46$	$\lceil \frac{\lceil \frac{m}{32} \rceil \frac{n(n-1)}{8}}{C} \rceil$	-
Our method 2	CPU	-	-	n
	GPU	$\lceil \frac{m}{32} \rceil \lceil \log \frac{n(n-1)}{8} \rceil$	$\lceil \frac{\lceil \frac{m}{32} \rceil \frac{n(n-1)}{8}}{C} \rceil$	-
Alternative method (with method 2)	CPU	-	-	n
	GPU	$\lceil \frac{m}{32} \rceil \lceil \log \frac{n(n-1)}{8} \rceil$	-	-

3.4 Experiments

In this section, we present and discuss experiment results. We used NVIDIA GeForce GTX 580 GPU, as well as Intel Core i7 875K CPU with 8 GB of memory.

3.4.1 Experiment Setup

We implement the evaluation of systems of $2n$-polynomials in n-unknowns for $n = 32, 64, 96, \ldots, 512$ on CPU and GPU. Finally, we compare the results of GPU and CPU implementations.

CPU Implementation

We implement evaluation of multivariate quadratic polynomial systems on the CPU by C language. We apply strategies of Berbain et al. [13] to CPU implementations.

GPU Implementation

We also apply them to GPU implementations. Moreover, we implement evaluation of multivariate quadratic polynomial systems with the parallelization strategies 1 and 2 as mentioned previously.

3.4.2 Experiment Results

We present the results of evaluation time of multivariate quadratic systems in Table 3.3. Evaluation time with the parallelization strategy 1 increase in the number of unknowns n rapidly. On the other hand, the parallelization strategy 2 increase in n slowly. Therefore, the strategy 2 is more efficient than the strategy 1.

Table 3.3: Evaluation time for multivariate quadratic polynomial systems.

Unknowns	Polynomials	Evaluation time (μsec)		
n	$2n$	CPU	Strategy 1	Strategy 2
32	64	2.7	21.758	15.927
64	128	16.9	23.483	15.849
96	192	52.7	24.110	16.071
128	256	118.8	24.325	16.537
160	320	236.2	25.058	17.166
192	384	417.8	29.845	17.184
224	448	656.5	34.549	18.125
256	512	992.5	41.864	18.651
288	576	1505.4	52.442	19.408
320	640	2322.2	71.663	19.841
352	704	3409.2	90.264	20.236
384	768	4906.2	111.951	20.710
416	832	6666.4	146.331	21.420
448	896	8453.5	193.567	21.892
480	960	10545.1	256.538	22.259
512	1024	12902.0	336.299	22.785

Furthermore, we compare result of QUAD implementations with Berbain *et al.* [13] and Chen *et al.* [20] on QUAD$(2,160,160)$ and QUAD$(2,320,320)$ in Table3.4. Unfortunately, QUAD$(2,160,160)$ with the parallelization strategy 2 is not so fast, compared with the results of Berbain *et al.* [13]. However, QUAD$(2,320,320)$ with the parallelization strategy 2 is 2.3 times faster than Chen *et al.* [20]. Moreover, it is faster than QUAD$(2,160,160)$. Therefore, we think that strategy 2 is suited to QUAD$(2,n,n)$, which n is a large number.

Table 3.4: Encryption throughput of QUAD

		Throughput(Mbps)	
		$\mathrm{QUAD}(2, 160, 160)$	$\mathrm{QUAD}(2, 320, 320)$
CPU		0.646	0.131
GPU	Strategy 1	5.086	3.768
	Strategy 2	11.693	14.567
BBG2006[13]		8.45	—
CCCHNY2010 [20]	CPU	—	6.1
	GPU	—	2.6

Profile of Kernels

Table refKernelGF2 shows that profiles of utilization of kernels. There exists 10 GPU kernels. 1 kernel is used in both method 1 and method 2. 7 kernels are used in method 1. last 2 kernels are used in method 2.

In the parallelized method 1, co-summation of column in sub-matrix shows 38 % in the worst case. However, the occupancy of the kernel depends on the number of polynomials. In fact, the worst case is only for $\mathrm{QUAD}(2, 32, 32)$. Over $\mathrm{QUAD}(2, 64, 64)$, we can achieve at least 75 % occupancy. Also, 4 summation kernels of show that 19 % occupancy. However, there exists only $\lceil m/32 \rceil$ threads. Since, $m \geq 512$ in this experimentations, there no exists more occupancy with parallelizations.

In contrast, both summation kernels of the parallelized method 2 shows 100% occupancy. Hence, the parallelized method 2 is efficient than method 1 for occupancy.

3.5 Conclusion

We presented two parallelization strategies for accelerating the evaluation of multivariate quadratic polynomial systems. A GPU implementation with parallelization strategy 2 is the fastest implementation compared with previous works. Moreover, it might be suited to large finite fields. The security of QUAD depends on the scale of multivariate quadratic polynomial systems. We expect QUADs with the strategy 2 will

Table 3.5: Kernel profile of evaluating multivariate quadratic polynomials over the binary field.

Type	Kernel	Registers per thread	Shared Memory per blocks(bytes)	Occupancy Utilization (%)
Both Methods	Multiplication (k: number of non-zero)	10 ($k \leq 5$), 12 ($k \geq 6$)	128 k	69-100
Method 1	Co-summation of Row	10-26	0	69-100
	Co-summation of Column	10	0	38-100
	4 summation kernels	10	0	19
Method 2	Summation 1	10	2048	100
	Summation 2	12	0	100

become efficient and secure stream ciphers. Our approaches can be applied not only to the QUAD stream cipher but potentially also to other multivariate cryptosystems.

Chapter 4

Effective Method for Multiplications over Extension Fields

4.1 Introduction

Background

Stream ciphers are symmetric cryptosystems, whose encryption is performed by xoring with messages and with keystreams. Basically, the security of stream ciphers is discussed based on parameters of random numbers(i.e. periodicity, unbiassedness, etc.) [18, 54, 53]. In these discussions, security parameters are evaluated by experimentations of known attacks. Several stream ciphers take other approaches for security like provable security, with reductions to known difficult mathematical problems. For example, Blum, Blum and Shub introduced pseudo-random number generator (PRNG), whose security is provably based on the integer factorization [15]. The QUAD, proposed by Berbain, Gilbert and Patarin, is also such a stream cipher endowed of provable security [14]. It uses the theory of multivariate public-key cryptography (MPKC) and generates random numbers by evaluations of multivariate quadratic polynomials over finite fields. Generally, we denote the constructions of QUAD with a system over GF(q) of n unknowns and r bit output stream as QUAD(q, n, r). The security of QUAD depends on the complexity of solving multivariate quadratic equation systems over finite fields, problem called $\mathcal{MQ}$. Since $\mathcal{MQ}$ is known to be NP-complete [11], QUAD is expected to be a practical secure stream cipher.

However, QUAD has problems of computational cost. QUAD requires evaluating multivariate quadratic polynomials over finite fields. Typically, QUAD(q, n, r) takes $mn(n+2)$ additions and $m(n+1)^2$ multiplications over GF(q) for evaluation, where m is the number of polynomials, that is $m = n + r$. Therefore, effective evaluation method of the system is necessary for practical QUAD.

Parallel computing is a possible way to accelerate algorithms. Especially, because of the inherent parallelism between each monomial and each polynomial, evaluation of multivariate quadratic polynomials is suitable for parallelization. Bitslicing is a technique of parallelization. Although it was originally introduced for hardware implementations [24], it is used to apply the Single Input Multiple Data(SIMD) construction virtually. It is already applied to many cryptosystems (e.g. Data Encryption Standard(DES) [31] and Advanced Encryptions Standard(AES) [43]). GPU are hardwares designed for parallel computing. They are appealing for their economic cost (price) against other parallelization methods(Field-Programmable Gate Array(FPGA), PC-clusters, etc.). Nowadays, GPU venders provide GPU programming libraries and some open libraries (e.g. OpenCL [4]) also allow GPU computations.

Related works

One way of making efficient evaluation of multivariate quadratic polynomials over finite fields is reducing the arithmetic operations of polynomials. Berbain, Billet and Gilbert provide such reductions by precomputing monomials, parallelising, bitslicing and dedicated methods for the binary field [13]. They showed throughputs of QUAD$(2, 160, 160)$, QUAD$(2^4, 40, 40)$ and QUAD$(2^8, 20, 20)$ as 8.45 Mbps, 23.59 Mbps and 42.15 Mbps respectively. Petzoldt applied linear recurring sequences (LRS) to QUAD and reduces the computational cost of QUAD(q, n, r) (and $m = n + r$) to $2n$ additions and $3mn + m$ multiplications over GF(q). He showed a throughput of QUAD$(2^8, 26, 26)$ as 872.7kbps and 5.8 faster than QUAD with random constructed polynomials.

In another way, there exists some parallel implementations by FPGA. Arditti, Berbain, Billet *et al.* show throughputs on XCLV25 FPGA of QUAD$(2, 160, 160)$ and QUAD(2, 256, 256) as 3.3 Mbps and 2.0 Mbps respectively [8]. Hamlet and Brocato provide implementations of QUAD$(2, 128, 128)$ on a Vertex-4 FPGA and the fastest one gives a 374 Mbps throughput [25]. However, while their work is efficient, it is still not secure, since their construction is smaller than original recommended parameters by Berbain, Gilbert and Patarin [14].

4.1.1 Challenging issues

Fast evaluation of multivariate quadratic polynomials is necessary to construct practical QUAD. Our challenge is to make it efficient through two approaches: parallelizations and using extension fields. There are three main challenging issues.

Reducing monomials in polynomials

The number of monomials in a quadratic polynomial in n variables is given by $\binom{n+2}{2}$. A parallel algorithm for summations named parallel reduction [5] is executed in $\lceil \log T \rceil$ steps, where T is the number of terms to be summed (exactly $T = \binom{n+2}{2}$). However, it executes a surplus step for some n. For example, when $n = 64$, $T = 2,145 > 2,048$ and it takes 12 steps. Therefore, it is desirable to reduce the number of monomials in each quadratic polynomial under 2,048 for $n = 64$. Although the number of reducing terms for each polynomial should be the same for parallelizations, choosing different combinations is difficult. Hence, reducing monomials of quadratic polynomials is an issue.

Finding fast multiplication methods on GPU

Using large fields is another way of reducing terms of multivariate quadratic polynomials. Since polynomials defined over larger field can yield larger bit streams, smaller polynomials can be used. However, we can reduce the number of variables. There are 2 types of large fields, large prime fields and large field extensions of small prime fields. In the case of MPKC, we often choose extension fields, because additions of extension fields over small prime fields are more efficient than large prime fields. Especially, additions over extension fields can be implemented by vector xoring, we select extensions of the binary field.

There is a challenging issue concerning extension fields: generally, multiplication over extension fields is more complicated. Although in small cases (e.g. GF(2^8)), we can make it efficient with lookup tables, in large cases (like GF(2^{32})) we cannot, because of the size of the table takes up to 32EB!. There are some related works, which discuss fast hardware implementations of binary extension fields [44] and GPU implementations over extension fields [37], however they do not discuss GPU implementations of extensions of the binary field. Hence, fast implementations of multiplications over binary field's extensions on GPU is an important issue.

Optimizations of CUDA GPU implementation

In this paper, we use Compute Unified Device Architecture(CUDA) API [3], provided by NVIDIA [6], for GPU implementations. In CUDA implementations, we have two sub-issues regarding the tuning of the parallelizations on GPU. One is avoiding the surplus steps of GPU kernels (functions). Indeed, in CUDA, kernels parallelization is achieved with blocks and threads in each block. However, actually threads are divided by warp, the maximal number of parallel threads in a block executed at a time. Therefore, we should tune the number of threads in order that it is a multiple of the warp size to avoid surplus steps. We should optimize this number for every construction.

The other is adjusting placement of data. In CUDA, memory loading is suitable for serial data. Hence, we should consider data constructions for suited memory loading on CUDA implementations.

Our contributions

In this paper, we achieve the followings:

Reduction of the computational cost of multivariate quadratic polynomials: we reduce the number of terms of multivariate quadratic polynomials from $\binom{n+2}{2}$ to $\binom{n-k+2}{2}$ by removing variables. Our method removes different variables for each polynomial.

Comparison of several multiplication methods over GF(2^{32}): We implement multiplications through the polynomial basis, the normal basis, Zech's logarithm, using intermediate fields and bitslicing and discover the most suited method for GPU. For GF(2^{32}), we get the best way by bitslicing the polynomial basis over GF(2^{32}). It show a throughput of 800 Gbps.

Optimization of QUAD on GPU: We tune our QUAD implementations for CUDA. We choose k, which is divisible by 32. Also, we choose the best multiplications in our experimentations, then implement QUAD over GF(2^{32}) on GPU. Moreover, we construct a data structure for QUAD on GF(2^{32}).

We then show the throughputs of QUAD($2^{32}, 48, 48$) and QUAD($2^{32}, 64, 64$) as 24.827 Mbps and 19.4196 Mbps respectively. There are over 90 times faster than CPU ones. This is the first implementations of QUAD stream cipher over GF(2^{32}).

Comparison with related works

GPU implementations are a way of parallelizing. Manavski has implemented AES on NVIDIA GeForce GTX 295, GPU resulting in an acceleration by a factor 20 when compared to the CPU implementation, by precomputing T-boxes and using lookup tables [33]. Li, Zhong, Zhao, *et al.* achieve 50 times faster AES on NVIDIA Tesla C2050, GPU citeli2012implementation. They use several techniques, precomputing key-scheduling and T-boxes, using shared memory for T-boxes and CUDA vector datas. Khalid, Bagchi, Paul, *et al.* has implemented HC stream ciphers citekhalid2012optimized. Although the single-data case is slower than CPU implementations, it is 2.8 times faster in the multiple-data case. They conclude GPU is suitable as co-processors of CPU in HC stream ciphers. Jang, Han, Han, *et al.* implement RSA public-key cryptography [28]. They have implemented 1024, 2048 and 4096 bit RSA, and in 1024 bit RSA, they showed 9.2 times faster timings than CPU. Bos and Stefan have implemented the hash functions SHA-3 round-2 candidates [17]. They have evaluated computational time of each algorithm, then they have parallelized them. Our fastest GPU implementation of QUAD over GF(2^{32}) is 90 times faster than CPU. Hence, we can conclude that evaluations of multivariate quadratic polynomials are suitable to be implemented on GPUs.

Besides, we have tried to speed up QUAD stream cipher with 2 approaches, reducing the computational cost of evaluating multivariate quadratic polynomials [46, 47, 48] and studying fast multiplications over finite fields [51, 50]. This paper presents the progresses realized upon those previous works. For evaluating polynomials, we replace and extend our parallelizing method to binary extension fields from the binary prime field. Moreover, in this paper, we reduce monomials of polynomials by a new algorithm. For multiplications over extension fields, we introduce bitslicing techniques. As a result, we have found a more suited multiplication method than in previous results [50].

Parallelization for CUDA

In CUDA, we should consider how to parallelize algorithms on GPUs. Especially, the number of threads in each block is important. This number is defined by GPUs. For example, NVIDIA GTX TITIAN can use 1,024 threads in each block register. On the other hand, this number is also confined by the number of registers in blocks (e.g.

65,536 registers per block for GTX TITIAN). Every thread use different registers for variables in kernels. When the total number of registers in every thread is greater than the number of registers in blocks, GPUs shows unexpected behavior (e.g. GPUs are halted). Therefore, we should parallelize algorithms so that the number of registers in blocks is less than these GPU limitations.

Another point of the number of thread is the size of warp. In CUDA, actually, blocks execute a warp, which is a unit of threads at a time. In other words, the number of executing threads of a block is limited by the size of the warp. Therefore, if the number of threads is not divisible by the warp size, it has a surplus iteration. Hence, we should tune the number of threads in order that it is a multiple of the warp size. The size of warp has been 32 since the first version of CUDA.

4.2 Binary Extension Field GF(2^{32})

Let p be a prime and $q = p^k$. Then, there exists degree k extension fields $\mathrm{GF}(p^k) = \mathrm{GF}(q)$ of $\mathrm{GF}(p)$. Generally, $\mathrm{GF}(q)$ can be defined by a degree k primitive polynomial $f(X)$. Then X is a primitive element of $\mathrm{GF}(q)$, if $f(X) = 0$. Since finite extensions of finite fields are Galois extensions, there is a Galois group $\mathrm{Gal}(\mathrm{GF}(q)/\mathrm{GF}(p))$ given by following formula,

$$\mathrm{Gal}(\mathrm{GF}(q)/\mathrm{GF}(p)) = \{\sigma : \mathrm{GF}(q) \mapsto \mathrm{GF}(q) | \mathit{automorphism} : \sigma(\alpha) = \alpha \ (\forall \alpha \in \mathrm{GF}(p))\}.$$

If τ defines the Frobenius mapping of $\mathrm{GF}(q)/\mathrm{GF}(p)$, the $\mathrm{Gal}(\mathrm{GF}(q)/\mathrm{GF}(p))\}$ is cyclic group, generating by τ.

We can denote an element $a \in \mathrm{GF}(q)$ by a vector over $\mathrm{GF}(p)$ as follows:

$$a = \{c_1, \ldots, c_k\}, \ (c_1, \ldots, c_k \in \mathrm{GF}(p)), \tag{4.1}$$

where we have fixed the basis $\{X_1, \ldots, X_k\}$ of the extension $\mathrm{GF}(q)/\mathrm{GF}(p)$:

$$a = c_1X_1 + \cdots + c_kX_k = \sum_{i=1}^{k} c_iX_i, \tag{4.2}$$

In this paper, we discuss the following 2 bases,

Polynomial basis: constructed by a primitive element $X \in \mathrm{GF}(q)$ such that $\{1(= X^0), X, \ldots, X^{k-1}\}$.

Normal basis [45]: we assume given an element $\alpha \in \mathrm{GF}(q)$ for a finite Galois extension $\mathrm{GF}(q)/\mathrm{GF}(p)$ such that $\{\sigma(\alpha)|\sigma \in Gal(\mathrm{GF}(q)/\mathrm{GF}(p))\}$. Then, basis is given by $\{\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{k-1}}\}$

4.2.1 Multiplications over GF(2^{32})

$\mathrm{GF}(q)$ can be handled as a residue class ring of the polynomial ring $\mathrm{GF}(p)[X]$ modulo $f(X)$. Given $a, b \in \mathrm{GF}(q)$, we denote by $a(X), b(X)$ their representative polynomials in $\mathrm{GF}(p)[X]/\langle f\rangle$. Therefore, additions and multiplications of $\mathrm{GF}(q)$ can be denoted as following formulas,

$$\begin{aligned} a + b &:= a(X) + b(X)\ modf(X), \\ a * b &:= a(X) * b(X)\ modf(X). \end{aligned}$$

Since a can be handled as a vector of $\mathrm{GF}(p)$ like in Equation (4.1), additions of $\mathrm{GF}(q)$ are computed by:

$$a + b := \{a_1 + b_1\ mod\, p, \ldots, a_k + b_k\ mod\, p\}, \tag{4.3}$$

Zech's logarithm

Originally, Zech's logarithm (also called Jacobi's logarithm [45]) is proposed to figure additions for elements represented as powers of a generator of a cyclic group $\mathrm{GF}(q)^* = \mathrm{GF}(q) \setminus \{0\}$. Zech's logarithm is considered to be a method of efficient exponentiation over cyclic groups for cryptosystems [26, 27]. Let γ be a generator of $\mathrm{GF}(q)^*$. Then, $\mathrm{GF}(q)^* = \langle\gamma\rangle$. Therefore, we can represent any element in $\mathrm{GF}(q)^*$ as γ^ℓ, where ℓ is an integer. In particular, $\gamma^\ell \neq \gamma^{\ell'}$, $0 \leq \ell \neq \ell' \leq p^r - 2$. In this way, $\mathrm{GF}(q)^*$ can be represented by $[0, p^r - 2]$. Hence, multiplications over $\mathrm{GF}(q)^*$ can be computed by integer additions modulo $p^r - 1$.

Intermediate field [45]

Let k be a composite integer for $q = p^k$. Then, there exists l, where $l \mid k$ and $1 < l < k$. $\mathrm{GF}(q^l)$ is an extension field of $\mathrm{GF}(q)$ and a subfield of $\mathrm{GF}(p^k)$. We call $\mathrm{GF}(p^l)$ an intermediate field. Because, any extension of $\mathrm{GF}(q)/\mathrm{GF}(p)$ are isomorphism, we can compute operations of $\mathrm{GF}(p^k)$ as extension from $\mathrm{GF}(p^l)$.

4.2.2 QUAD Stream Cipher over Extension Field

Multivariate Quadratic Polynomials

Let p be a prime and $q = p^k$. Then, $\mathrm{GF}(q)$ is a degree k extension of the field with p elements. The system A of m quadratic polynomials in n variables over a finite field

GF(q) can be written in the following form

$$\begin{aligned}
Q_1(x_1,\ldots,x_n) &= \sum_{1\le i\le j\le n} \alpha_{i,j}^{(1)} x_i x_j + \sum_{1\le k\le n} \beta_k^{(1)} x_k + \gamma^{(1)} \\
Q_2(x_1,\ldots,x_n) &= \sum_{1\le i\le j\le n} \alpha_{i,j}^{(2)} x_i x_j + \sum_{1\le k\le n} \beta_k^{(2)} x_k + \gamma^{(2)} \\
&\vdots \\
Q_m(x_1,\ldots,x_n) &= \sum_{1\le i\le j\le n} \alpha_{i,j}^{(m)} x_i x_j + \sum_{1\le k\le n} \beta_k^{(m)} x_k + \gamma^{(m)}.
\end{aligned} \tag{4.4}$$

Given the value of every unknowns, we can evaluate multivariate polynomials. Evaluating multivariate polynomials need compute multiplications and summations of terms over a finite field. Therefore, evaluation a multivariate polynomial system is equivalent to evaluate all polynomials.

Construction and Algorithm

QUAD is a stream cipher proposed by Berbain, Gilbert and Patarin [14]. QUAD uses systems of multivariate quadratic polynomials to obtain the random keystream. Therefore, it is a kind of MPKC. One of advantages of QUAD against other stream ciphers is that it has a provable security. The security of QUAD is based on the MQ assumption just like other MPKC instances, and is proved by Berbain, Gilbert and Patarin [14].

Constructions and notation

Generally, the notation of QUAD(q, n, r) means a construction based on a system of the n-tuple internal state value $\boldsymbol{x} = \{x_1,\ldots,x_n\}^T$ and keystream length r over GF(q) in a cycle of QUAD. On the other hand, it shows that a system of QUAD as $m = n + r$ quadratic equations in n variables over GF(q), and a system in QUAD are given in Equation (5.2). Usually, m is set to kn, where $k \geq 2$, and therefore $r = (k-1)n$.

QUAD(q, n, r) has three key constructions. One is the n-tuple key $\boldsymbol{x} = \{x_1, \ldots, x_n\}^T$ over GF(q). Another is the L-bit (in particular, $L = 80$) initialization vector $IV \in \{0, 1\}^L$. The last ones are 4 randomly chosen systems P, Q, S_0 and S_1. Systems P, S_0 and S_1 follow from the same construction, are n quadratic equations and in n variables over GF(q). Only Q is different construction, it has n quadratic equations and n variables over GF(q). System P is used to update the i-th internal state $\boldsymbol{x}_i$ to next $\boldsymbol{x}_{i+1}$, and Q is used to generate the i-th keystream $\boldsymbol{y}_i = \{y_1, \ldots, y_r\}^T$ from $\boldsymbol{x}_i$, where i is an iteration counter. Sometimes, P and Q are combined to form the system S of $m = n+r$ equations in n variables over GF(q). Both S_0 and S_1 are used in the initialization step. They replace the initial state $\boldsymbol{x}_0$ just like updating $\boldsymbol{x}_{i+1}$with P.

Algorithm

The algorithm of QUAD is separated in three parts, key generation, encryption/decryption of the message and initialization step.

Let S be a combined system of P and Q. Then, the keystream generator of QUAD follows three steps:

Computation Step: the generator computes values of system S with the current internal value $\boldsymbol{x}_i = \{x_1^{(i)}, \ldots, x_n^{(i)}\}^T$.

Output Step: the generator outputs r keystreams y_i from the system Q with x_i.

Update Step: the current internal value $\boldsymbol{x}_i = \{x_1^{(i)}, \ldots, x_n^{(i)}\}^T$ is updated to a next internal value with a n-tuple value $\boldsymbol{x}_{i+1} = \{x_1^{(i+1)}, \ldots, x_n^{(i+1)}\}^T$ from system P.

The sketch illustrating the keystream generation algorithm is shown in Figure 4.1. It indicates that the generator outputs keystreams by repeating the above three steps.

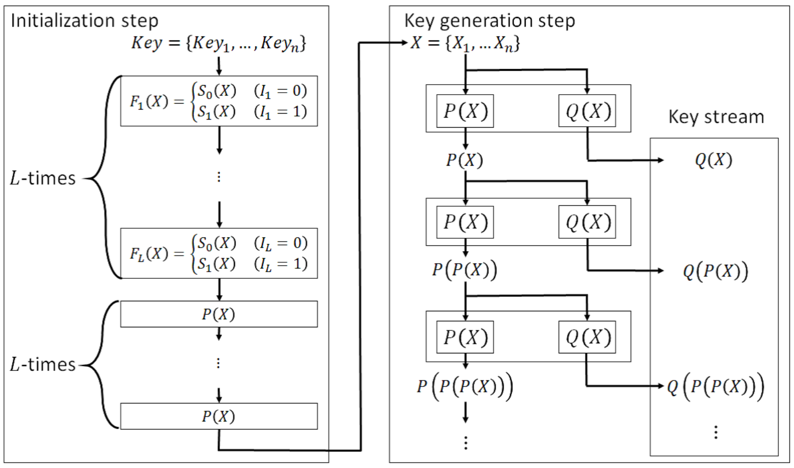

Initialization step
$Key = \{Key_1, \dots, Key_n\}$
$F_1(X) = \begin{cases} S_0(X) & (I_1 = 0) \\ S_1(X) & (I_1 = 1) \end{cases}$
L-times
$F_L(X) = \begin{cases} S_0(X) & (I_L = 0) \\ S_1(X) & (I_L = 1) \end{cases}$
$P(X)$
L-times
$P(X)$
Key generation step
$X = \{X_1, \dots X_n\}$
$P(X)$
$Q(X)$
Key stream
$P(X)$
$Q(X)$
$P(P(X))$
$Q(P(X))$
$P(P(P(X)))$
$Q(P(P(X)))$

Figure 4.1: Image of QUAD key generating algorithm

Encryption/decryption messages

The generated keystreams are considered to be a pseudorandom bit string and used to encrypt a plaintext with the bitwise XOR operation.

Key and initialization of current state

Berbain, Gilbert and Patarin also provides a technique for initialization of the internal state $X = (x_1, \dots, x_n)$ [14]. For QUAD(q, n, r), we use the key $K \in GF(q)^n$, the initialization vector $IV = \{0,1\}^{|IV|}$ and two carefully randomly chosen multivariate quadratic systems $S_0(X)$ and $S_1(X)$, mapping $\mathrm{GF}(q)^n \mapsto \mathrm{GF}(q)^n$ to initialize X. The initialization of the internal state X follows two steps:

Initially set step: we set the internal state value X to the key K.

Initially update step: we update X for $|IV|$ times. Let i be an iteration counter of initially update and $IV_i = \{0,1\}$ be a value of i-th element of IV. We change the value of X to $S_0(X)$, when $IV_i = 0$, or to $S_1(X)$, when $IV_i = 1$.

Computational cost of QUAD

The computational cost of multivariate quadratic polynomials depends on computing quadratic terms. The summation of quadratic terms requires $n(n+1)/2$ multiplications and additions. Therefore the computational costs of one multivariate quadratic polynomial is $\mathcal{O}(n^2)$. QUAD(q, n, r) requires to compute m multivariate quadratic polynomials. Since $m = kn$, the computational cost of generating key stream is $\mathcal{O}(n^3)$.

Security level of QUAD

The security level of QUAD is based on the MQ assumption, since Berbain, Gilbert and Patarin prove that solving QUAD needs solving $\mathcal{MQ}$ problem [14]. The eXtended Linearization(XL) algorithm [22] is a solving method of the $\mathcal{MQ}$. The XL constructs a polynomial system of the degree D by products of quadratic equations and monomials of the degree d, where $1 \leq d \leq D$, and solves the system as linear algebra. Then, the running time of XL depends on D. The minimal D is called the degree of regularity. Yang, Chen, Bernstein *et al.* [55] show that the degree of regularity of the $\mathcal{MQ}$ in QUAD(q, n, n) is given by the degree of the lowest term with a non-positive coefficient in the following polynomial,

$$G(t) = ((1-t)^{(-n-1)}(1-t^2)^n(1-t^4)^n). \tag{4.5}$$

Moreover, they give the expected running time of the XL-Wiedemann C_{XL} as the following formula.

$$C_{XL} \sim 3\tau T \mathfrak{m}. \tag{4.6}$$

T is the number of monomials in equations (for large q, $T = \binom{n+D}{D})$), $\tau = \lambda T$ is the total number of monomials in all equations (λ is the average terms in original quadratic equations), and $\mathfrak{m}$ is the cycle of field multiplications. According to their QUAD analysis, QUAD$(2^8, 20, 20)$ has 45-bit security, QUAD$(2^4, 40, 40)$ has 71-bit security, and

QUAD$(2, 160, 160)$ has less than 140-bit security. Actually, secure QUAD requires larger constructions such as QUAD(2,256,256) or QUAD(2,320,320).

4.3 Evaluating Multivariate Quadratic Polynomials on GPU

4.3.1 Evaluating Polynomials by SIMD

GPU is suitable for implementations of SIMD constructions. SIMD is a parallelization method, which computes multiple data by single function call. In CUDA API, GPU kernels achieve SIMD on GPU by single function call and execute multi threads. In this paper, we evaluate multivariate quadratic polynomials through the following 3 steps. 1) precompute quadratic monomials $x_i x_j$, 2) compute all monomials $\alpha_{i,j}^{(k)} x_i x_j$, $\beta_i^{(k)} x_i$, 3) calculate summations $\sum_{1\leq i\leq j\leq n} \alpha_{i,j}^{(k)} x_i x_j + \sum_{1\leq i\leq n} \beta_i^{(k)} x_i$.

Precomputing $x_i x_j$

This step is based on the precomputing method of Berbain, Billet and Gilbert [13]. There are $\binom{n+1}{2} = n(n+1)/2$ quadratic monomials in a quadratic polynomial with n unknowns. Therefore, we compute each $x_i x_j$ by each thread. Then, thread t can be computed from (i, j) by the following formula.

$$t = \frac{j(j-1)}{2} + i, \tag{4.7}$$

However, computing (i, j) from t is inefficient. Hence, we construct a lookup table of (i, j) from t.

Compute all monomials

Before computing monomials, we store $x_i x_j$ into x_{n+t}, where t is given by Equation (4.7). Also, we assume that $\beta_{n+t}^{(k)} = \alpha_{i,j}^{(k)}$, then we compute $\beta_i^{(k)} x_i$ $(1 \leq i \leq (n+1)(n+2)/2)$ in parallel.

calculate summations $\sum_{1\le i\le j\le n} \alpha_{i,j}^{(k)} x_i x_j + \sum_{1\le i\le n} \beta_i^{(k)} x_i$

In this paper, we use parallel reduction technique like the method of Tanaka, Nishide, Sakurai [48]. It takes $\lceil \log T \rceil$ steps for a summation, where T is the number of terms of the summation. Actually, T is the number of monomials in a polynomial.

4.3.2 Reducing Terms in Polynomials

The parallel reduction takes $\lceil \log T \rceil$ steps for a summation of a polynomial. When $n = 64$, $T = 2,145$. Therefore, it takes $\lceil \log 2,145 \rceil = 12$ steps for a summation. Since if $T \le 2,048$, it takes only 11 steps, reducing monomials of polynomials is desirable. Now, we provide removing method of monomials in quadratic polynomials by variable-base reduction. We remove variables x_i for each polynomial f_j, where i are given by the following formula.

$$\begin{cases} i \equiv n - j - 1 & (mod\ \frac{n}{k}) & (k \mid n) \\ k(i-1) + j > n - k - 1 & (mod\ n) & (k \nmid n) \end{cases} \tag{4.8}$$

After that, we construct new $n-k$-tuple variables $\boldsymbol{x}'_j$ for each polynomial f_j, where $1 \le j \le m$. Then, the number of terms in each polynomial is reduced from $\binom{n+2}{2}$ to $\binom{n-k+2}{2}$. Figure 4.2 shows that the image of this variable-base reduction of $k = 1$. Since each tuple is different, systems of quadratic polynomial equations with $k+1$ polynomials constructs also n unknown system. Hence, we expect this method does not reduce the security parameters for small k.

Security Looseness with Reducing Terms

Our reducing technique removes k variables from polynomials. Looking through the whole of polynomials, polynomials still have a system of m polynomials in n unknowns. However, the number of terms in polynomial is decreased to $\binom{n+2-k}{2}$ from $\binom{n+2}{2}$. According to analysis of QUAD with the XL-Wiedemann by Yang *et al.* [55], the expected

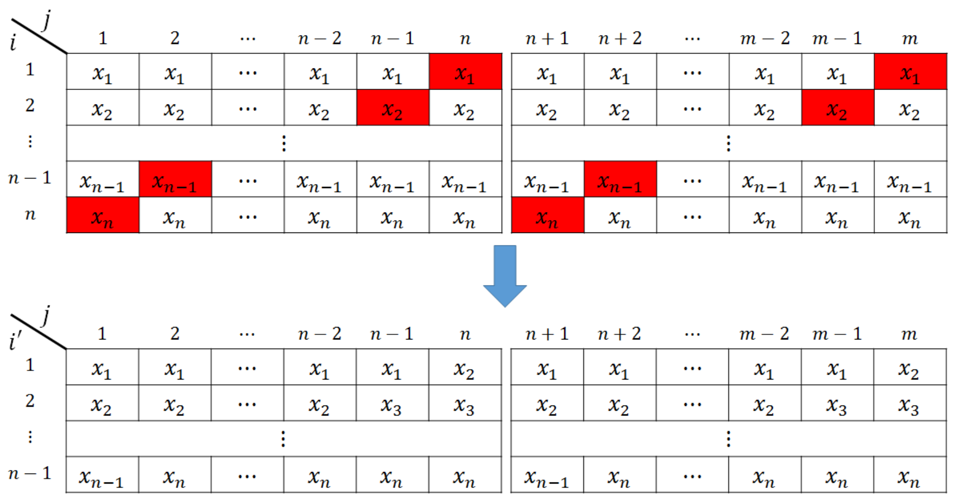


Figure 4.2: Removing variables x_i from quadratic polynomial f_j, where $k = 1$.

running time is given by Formula (4.6) as following:

$$C_{XL} \sim 3\tau T\mathfrak{m} = 3\lambda T^2\mathfrak{m}.$$

Because the number of unknowns in reduced systems equals to in full systems, the degree of regularity D of reduced systems is same to of full systems. Hence T is not decreased. On the other hand, the average number of term in polynomials λ is decreased to $\binom{n+2-k}{2}$ from $\binom{n+2}{2}$. Therefore, the expected time of solving $\mathcal{MQ}$ with a k-reduced system of m polynomials in n unknowns is $(n + 2 - k)(n + 1 - k)/(n + 2)(n + 1)$ times smaller than a full system. For example, $\mathcal{MQ}$ with a 2-reduced system of 64 polynomials in 32 unknowns is solved in 88.4 % of solving time with a full system. Figure 4.3 shows that such security looseness by removing 1, 2, 5, and 10 variables from original quadratic polynomials by our method. Also, Figure 4.4 shows the security ratio between original quadratic polynomials and reduced ones.

In detail, Yang *et al.* give the analysis [55], the XL-Wiedemann algorithm can solve

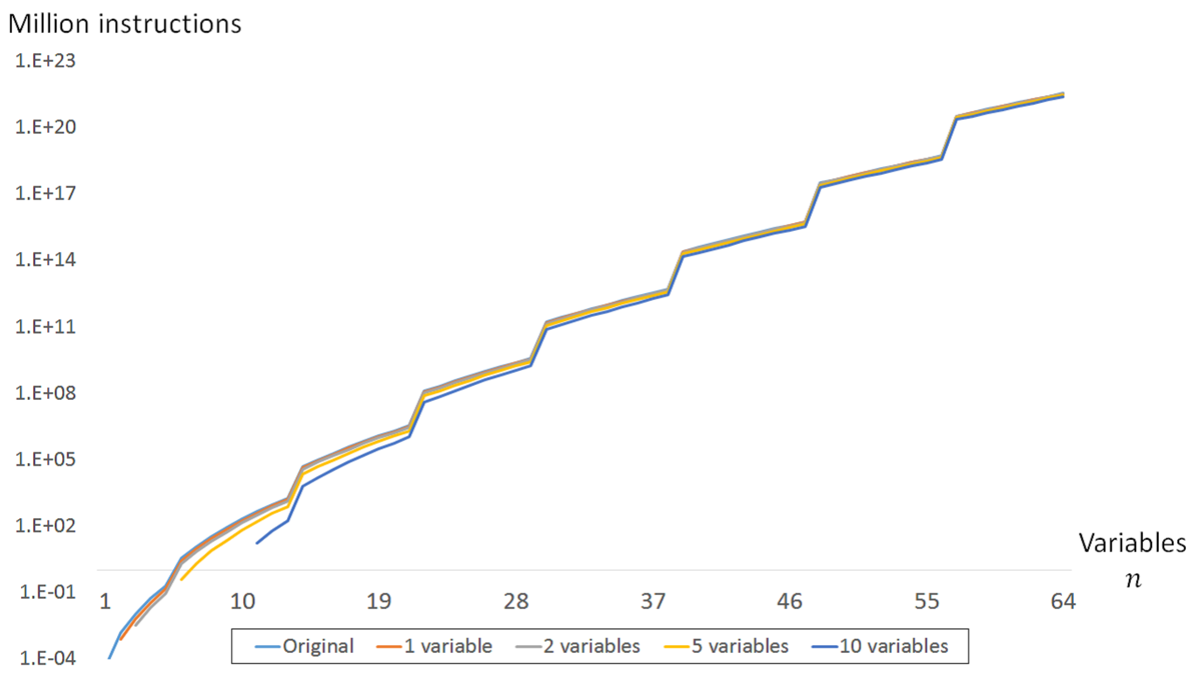


Figure 4.3: Security looseness by reducing variables from polynomials

in time C_{XL} as the following:

$$C_{XL} = 3\tau T \mathfrak{m} = 3w_B T^2 \mathfrak{m},$$

where, T is the number of variables in the linearized system by XL, $\tau = w_B T$ is the total number of monomials in that system, and $\mathfrak{m}$ is a computational cost of multiplications. Let $\mathfrak{m} = 1$, i.t. each multiplication is calculated in a cycle. Then, C_{XL} for a $\mathcal{MQ}$ with a system of 64 equations in 32 variables (the case of QUAD$(2^{32}, 32, 32)$) in roughly 3.98×10^{11} instructions. According to evaluation of security equivalent between symmetric ciphers and asymmetric cryptosystems by Lenstra and Verheul [32], it corresponds to 76-bit security of symmetric cipehrs. When we reduce 2 variables from each equation by our method, C_{XL} is down to 3.52×10^{11} instructions. However, it keeps the 76-bit security.

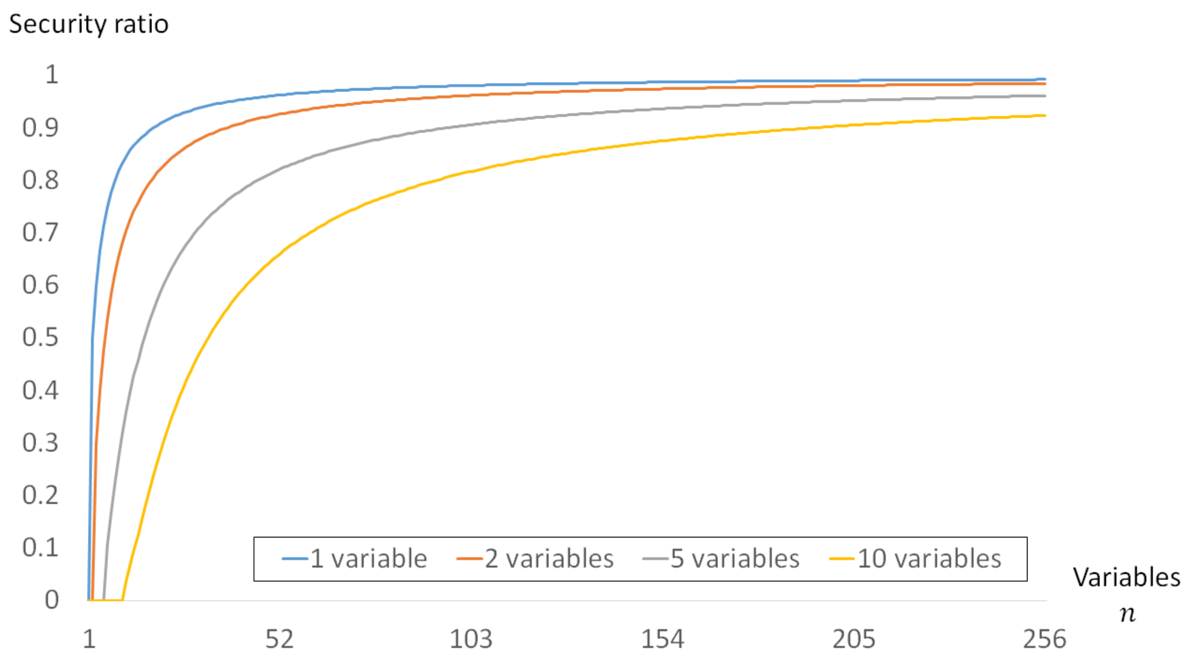


Figure 4.4: Security ratio between original polynomials and reduced polynomials.

4.4 Analysis of Multiplication Algorithms over Extension Fields

The computational costs of multiplications differs regarding the choice of the basis and of the approaches. We discuss 6 multiplication methods. 1) polynomial basis, 2) normal basis, 3)Zech's logarithm, 4) lookup table, 5) using intermediate fields and 6) bitslicing method. Now, we assume that $\mathrm{GF}(q) = \mathrm{GF}(p^k)$ is an extension field and $f(X)$ is a primitive polynomial of $\mathrm{GF}(q)/\mathrm{GF}(p)$. Also, let $c := a * b \in GF(q)$.

4.4.1 Polynomial basis

Let $\mathrm{GF}(q)$ be a set of polynomials over $\mathrm{GF}(p)$. Then, we can compute the multiplication $e_1 * e_2$, where $e_1, e_2 \in \mathrm{GF}(q)$, by:

$$e_1 * e_2 := e_1(X) * e_2(X) \mod f(X), \tag{4.9}$$

Let $e_1, e_2 \in \mathrm{GF}(q)$ be $c_{k-1}x^{k-1}+\cdots+c_1x+c_0$ and $c'_{k-1}x^{k-1}+\cdots+c'_1x+c'_0$, respectively. The product $e_1 * e_2$ can be computed by:

$$e_1 * e_2 = c_{k-1}c'_{k-1}x^{2k-2} + \cdots + c_0c'_0 \mod f(X).$$

In this method, we need to compute the multiplications $c_ic'_j$ for $0 \leq i, j < k$ and the summations $\sum_{i+j=t,i,j\geq 0} c_ic'_j$ for $0 \leq t \leq 2(k-1)$ over $\mathrm{GF}(p)$. The summation $\sum_{i+j=t,i,j\geq 0} c_ic'_j$ requires t additions for $0 \leq t < k$ and $2k-t-2$ additions for $k \leq t \leq 2(k-1)$. Therefore, it requires $(k-1)^2$ additions and k^2 multiplications over $\mathrm{GF}(p)$ if schoolbook multiplication is used. Moreover, $e_1 * e_2$ takes $k\lceil \log_2 p^m \rceil \simeq n\lceil \log_2 p \rceil$ bits of memory.

Normal basis

Given a finite Galois extension $\mathrm{GF}(q)/\mathrm{GF}(p)$, there exists an $\alpha \in \mathrm{GF}(q)$ such that $\{\sigma(\alpha)|\sigma \in \mathrm{Gal}(\mathrm{GF}(q)/\mathrm{GF}(p))\}$ is an $\mathrm{GF}(p)$-basis of $\mathrm{GF}(q)$, which is called a normal basis of $\mathrm{GF}(q)/\mathrm{GF}(p)$. A normal basis of $\mathrm{GF}(q)/\mathrm{GF}(p)$ can thus be denoted by:

$$\{\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{k-1}}\}. \tag{4.10}$$

Then, an element $a \in \mathrm{GF}(q)$ can uniquely be written as:

$$a = c_0\alpha + c_1\alpha^{p^m} + \cdots + c_{k-1}\alpha^{p^{(k-1)m}},\ c_0, \ldots, c_{k-1} \in \mathrm{GF}(p). \tag{4.11}$$

Let $a = [c_0, c_1, \ldots, c_{k-1}]_n \in \mathrm{GF}(q)$ be defined in Equation (4.11). Then Frobenius map σ_0 applied to α gives:

$$\sigma_0(a) = a^q = [c_{k-1}, c_0, c_1, \ldots, c_{k-2}]_n. \tag{4.12}$$

In other words, $\sigma_0(a)$ is simply a right circular shift [34].

Furthermore, let $a = [c_0, c_1, \ldots, c_{k-1}]_n, b = [c'_0, c'_1, \ldots, c'_{k-1}]_n \in \mathrm{GF}(q)$, and the result of the multiplication $a * b$ be $[d_0, d_1, \ldots, d_{k-1}]_n$. Then, every d_i, where $0 \leq i < k$, can be computed by evaluating the quadratic polynomials of $c_0, c_1, \ldots, c_{k-1}$, $c'_0, c'_1, \ldots, c'_{k-1}$ over $\mathrm{GF}(p)$. Let $d_i = p_i(c_0, \ldots, c_{k-1}, c'_0, \ldots, c'_{k-1})$, $\forall 0 \leq i < k$. According to Equation (4.12), we can compute $\sigma_0(a * b)$ by:

$$\begin{aligned} \sigma_0(a * b) &= [d_{k-1}, d_0, d_1, \ldots, d_{k-2}]_n \qquad (4.13) \\ &= \sigma_0(a) * \sigma_0(b) \\ &= [c_{k-1}, c_0, \ldots, c_{k-2}]_n * [c'_{k-1}, c'_0, \ldots, c'_{k-2}]_n \\ &= [p_0(c_{k-1}, c_0, \ldots, c_{k-2}, c'_{k-1}, c'_0, \ldots, c'_{k-2}), \ldots, \\ &\quad\ p_{k-1}(c_{k-1}, c_0, \ldots, c_{k-2}, c'_{k-1}, c'_0, \ldots, c'_{k-2})]_n. \end{aligned}$$

By comparing coefficients, d_{k-2} can be computed by:

$$d_{k-2} = p_{k-1}(c_{k-1}, c_0, c_1, \ldots, c_{k-2}, c'_{k-1}, c'_0, c'_1, \ldots, c'_{k-2}),$$

with Equation (4.13). In the same way, we can compute $\sigma_0^2(a * b), \ldots$, for all i by performing right circular shifts and computing all the d_r's by evaluating p_{k-1}.

Let $a, b \in \mathrm{GF}(q)$ be $[c_0, \ldots, c_{k-1}]_n$ and $[c'_0, \ldots, c'_{k-1}]_n$, respectively. An addition over $\mathrm{GF}(q)$ takes k additions over $\mathrm{GF}(p)$, similar to the polynomial basis method. On the other hand, the multiplication $a * b$ takes $2(k-1)$ right circular shift operations and k evaluations of a fixed (quadratic) polynomial $p_{k-1}(c_0, \ldots, c_{k-1}, c'_0, \ldots, c'_{k-1})$. An evaluation of a quadratic polynomial takes $k^2 - 1$ additions and $2k^2$ multiplications over $\mathrm{GF}(p)$. We can further speed up such an evaluation by precomputing common multiplications $c_i c_j$ over $\mathrm{GF}(p)$, where $0 \leq i, j \leq k-1$. Moreover, we can modify formula for c_i, c_j, c'_i, c'_j to :

$$\begin{aligned} & p_{k-1}(c_0, \ldots, c_{k-1}, c'_0, \ldots, c'_{k-1}) \\ = \ & c_0 c'_0 + \textstyle\sum_{0 \le i < j < k} s_{i,j}(c_i + c_j)(c'_i + c'_j) \quad \forall (i,j), s_{i,j} \in \mathrm{GF}(p), \end{aligned}$$

where $i \neq j$. Therefore, a multiplication over GF(q) requires $k(k-1)(k+2)/2$ additions and $k(k^2+1)/2$ multiplications over GF(p) plus $2(k-1)$ right circular shift operations. Moreover, the normal basis method needs $(k^2 - k + 2)\lceil \log_2 p^m \rceil / 2$ bits of memory.

4.4.2 Zech's logarithm

In this method, a multiplication over GF(q) needs one integer addition modulo $k-1$. On the other hand, addition is not simple. Therefore, we convert it to the polynomial basis for additions and convert it back to the cyclic group representation for multiplications. Therefore, a multiplication needs three such conversions. One is for converting from polynomial to cyclic group representation, while the other is the opposite. Therefore, an addition takes k additions over GF(p), similar to the polynomial basis representation, and a multiplication needs one integer addition modulo $k-1$ plus three conversions between polynomial and cyclic group representations. Moreover, since the tables represent maps from GF(q) to itself, Zech's method needs $2p^n \lceil \log_2 p^n \rceil$ bits of memory.

4.4.3 Multiplication Tables

We create a multiplication table by offline precomputing all combinations of multiplications over GF(q). Then, we can compute multiplications by looking up the multiplication table.

An addition over GF(q) can be computed using k additions over GF(p). On the other hand, a multiplication over GF(q) needs only one table look-up. Since the entire multiplication table needs to store every possible combination of multiplications over

GF(q), this method requires $p^{2n}\lceil\log_2 p^n\rceil$ bits of memory.

4.4.4 Using intermediate Fields

Although the multiplication table method is impractical for GF(2^{32}), for GF(2^8) the table requires only 256 KB. Also, Zech's logarithm over GF(2^{32}) needs just 256 KB. Here, we consider a method using an intermediate field GF(2^l) for GF(2^{32})/$GF(2)$, where $l = 2, 4, 8, 16$. In this method, we can compute multiplications over GF(2^{32}) by considering it as an extension field over GF(2^l) and by using the polynomial basis method or the normal basis method. For example, since the extension degree $k = 4$ for GF(2^{32})/GF(2^8), we can compute multiplication over GF(2^{32}) by 9 additions over GF(2^8)(72 XORs), 16 table look-ups, and one modulo over GF(2^8) with the polynomial basis method, or 288 XORs and 34 table look-ups with the normal basis method.

4.4.5 Bitslicing Method

Bitslicing method is a method of parallelization. Usually, one data is stored in one variable (e.g. 32-bit integer). This method slices datas and stores bit-datas of some variables in one variable. For example, 32 datas of 32-bit integers $x_1, \ldots, x_{32}$. We translate to bitsliced datas $y_1, \ldots, y_{32}$. Then, y_k has the datas of k-bit of every x_i. The ith bit of of y_k is stored the k-th bit of x_i

Bitslicing method achieves simple SIMD constructions and compressing data. In other words, since it can handle several variables at a time, it becomes more efficient. Also, when every data length is shorter than the size of variables, we can reduce the memory size by removing unused bits. On the other side, functions in programming languages are built for normal data. Therefore, we must build special functions for bitsliced data.

Table 4.1: Cost of multiplication over GF(2^{32}).

Intermediate	Method		Computational cost					Memory
field GF(2^l)	GF(2^l)/GF(2)	GF(2^k)/GF(2^l)	XOR	AND	LOOKUP	MOD	ADD	space
Direct	-	Polynomial basis	1,096	1,024	0	0	0	4B
		(Bitslicing)	34.25	32	0	0	0	128B
		Normal basis	16,864	16,400	0	0	0	125B
		(Bitslicing)	527	512.5	0	0	0	128B
		Zech's logarithm	0	0	3	1	1	32GB
		Multiplication table	0	0	1	0	0	64EB
GF(2^2)	Multiplication table	Polynomial basis	450	0	512	1	0	6B
		Normal basis	4,320	0	4,112	0	0	35B
GF(2^4)		Polynomial basis	196	0	256	1	0	132B
		Normal basis	1,120	0	1,040	0	0	143B
GF(2^8)		Polynomial basis	72	0	128	1	0	64KB
		Normal basis	288	0	272	0	0	64KB
GF(2^{16})	Zech's logarithm	Polynomial basis	16	0	12	4	5	256KB
		Normal Basis	64	0	15	5	5	256KB

4.4.6 Costs of Multiplications over GF(2^{32})

Table 4.1 shows the costs of multiplications over GF(2^{32}). The polynomial basis method and the normal basis method shows a much higher computational cost. On the other hand, Zech's logarithm and using multiplication table are impractical, as it needs 32 GB and 64 EB of memory space, respectively. Similarly, we estimate the computational costs of multiplications over GF(2^{32}) using GF(2^2), GF(2^4) or GF(2^{16}). We show the computational costs of multiplications over GF(2^{32}) using these intermediate fields in Table 4.1. Moreover, we assume that the bitslicing method reduces the computational cost of multiplications to 1/32 by 32-bit integers. Then, the estimation of bitslicing method in Table 4.1 shows an average computational cost of 1 multiplications of 32 bitsliced multiplications.

4.4.7 Experimentation of Multiplications

We implement the three basic multiplication methods, namely, polynomial basis, Zech's logarithm, and normal basis, over GF(2^{32}) on CPU and GPU. We evaluate and compare the running time of 67,108,864 multiplications with random elements over GF(2^{32}) for each method. Similarly, we also implement and perform the same experiment using intermediate fields and bitslicing methods as follows:

(1) Multiplication table + polynomial basis method: GF(2^{32})/GF(2^k)/GF(2) (k =

$1, 2, 4, 8)$

(2) Multiplication table + normal basis method: $\mathrm{GF}(2^{32})/\mathrm{GF}(2^k)/\mathrm{GF}(2)$ $(k = 1, 2, 4, 8)$

(3) Zech's logarithm + polynomial basis method: $\mathrm{GF}(2^{32})/\mathrm{GF}(2^{16})/\mathrm{GF}(2)$

(4) Zech's logarithm + normal basis method: $\mathrm{GF}(2^{32})/\mathrm{GF}(2^{16})/\mathrm{GF}(2)$

(5) Bitslicing + polynomial basis method: $\mathrm{GF}(2^{32})/\mathrm{GF}(2)$

(6) Bitslicing + normal basis method: $\mathrm{GF}(2^{32})/\mathrm{GF}(2)$

Hereunder, are the primitive polynomials used for each field extension.

(1) $\mathrm{GF}(2^{32})/\mathrm{GF}(2)$: $Y^{32} + Y^{22} + Y^2 + Y + 1 = 0$

(2) $\mathrm{GF}(2^{32})/\mathrm{GF}(2^2)/\mathrm{GF}(2)$: $Y^{16} + Y^3 + Y + X = 0$

(3) $\mathrm{GF}(2^{32})/\mathrm{GF}(2^4)/\mathrm{GF}(2)$: $Y^8 + Y^3 + Y + X = 0$

(4) $\mathrm{GF}(2^{32})/\mathrm{GF}(2^8)/\mathrm{GF}(2)$: $X^4 + Y^2 + (X + 1)Y + (X^3 + 1) = 0$

(5) $\mathrm{GF}(2^{32})/\mathrm{GF}(2^{16})/\mathrm{GF}(2)$: $Y^2 + Y + X^{13} = 0$

All the experiments are performed on Ubuntu 10.04 LTS 64bit, Intel Core i7 875K and NVIDIA GTX TITIAN with 8 GB of DDR3 memory.

Experimental result

Table 4.2 shows the result of implementations computational time for 67,108,864 multiplications. Table 4.3 shows throughputs of the result with our previous work [50]. In CPU implementations, the normal basis method using $\mathrm{GF}(2^{16})$ is the fastest, possibly because it needs the fewest computations among all methods. On the other hand, in GPU implementations, bitslicing method of the polynomial basis method is the fastest. Compared with our previous result on NVIDIA GTX GeForce 580 [50], the polynomial

Table 4.2: Computing time of 67,108,864 multiplications over GF(2^{32}).

Intermediate field GF(2^l)	Multiplication method GF(2^l)/GF(2)	Computational time (sec)					
		Intel Core i7 875K		NVIDIA GTX TITAN		NVIDIA GeForce GTX 580 [50]	
		Polynomial basis	Normal basis	Polynomial basis	Normal basis	Polynomial basis	Normal basis
Direct (bitslicing)	-	338.077	575.096	0.0764	8.657	1.552	25.064
		18.484	15.425	0.00246	0.94s	N.A.	N.A.
GF(2^2)	Multiplication table	121.997	159.989	1.548	5.099	1.242	3.813
GF(2^4)		31.651	38.281	0.368	0.485	0.583	0.776
GF(2^8)		8.627	9.121	0.0479	0.129	0.0555	0.0621
GF(2^{16})	Zech's logarithm	3.510	3.015	0.153	0.0764	0.195	0.153

Table 4.3: Throughputs of multiplications over GF(2^{32}).

Intermediate field $GF(2^l)$	Multiplication method $GF(2^l)/GF(2)$	Intel Core i7 875K (Mbps)		NVIDIA GTX TITAN (Gbps)		NVIDIA GeForce GTX 580(Gbps) [50]	
		Polynomial basis	Normal basis	Polynomial basis	Normal basis	Polynomial basis	Normal basis
Direct (bitslicing)	-	6.058	3.561	26.180	0.231	1.289	0.0798
		110.802	132.773	812.018	2.108	N.A.	N.A.
$GF(2^2)$	Multiplication table	16.787	12.801	0.392	4.122	1.610	0.525
$GF(2^4)$		64.706	53.499	5.434	4.122	3.431	2.577
$GF(2^8)$		237.394	224.537	41.796	15.472	36.036	32.206
$GF(2^{16})$	Zech's logarithm	583.476	679.270	13.096	26.170	10.256	13.072

basis of GF(2^{32})/GF(2) is 20 times faster. In this experimentation, we optimize the placement of data for memory loading reduced for multiplications, because loading data in a warp is required as a straight chunk in CUDA. We believe that this optimization makes efficient multiplications. We believe that the GPU cannot efficiently access the global memory the tables in Zech's logarithm over GF(2^8), as these tables are too large to fit into the fast memory on GPU.

4.5 Experiments of Parallel QUAD stream Cipher on GPU

4.5.1 Target constructions of QUAD

In this paper, we discuss three instances of QUAD constructions, QUAD($2^{32}, 32, 32$), QUAD($2^{32}, 48, 48$) and QUAD($2^{32}, 64, 64$). They output respectively 1,024, 1,536 and 2,048 bit keystreams at a time. Table 4.4 shows other parameters of these constructions. Security parameters in Table 4.4 are roughly evaluated with formula (4.5) and (4.6) by analysis es of Yang, Chen, Bernstein, *et al.* [55]. For example, $D = 20$ for QUAD(2^{32}, 64, 64). Then the number of monomials $T = \binom{64+20}{20} \simeq 1.0736{\times}10^{19}$. Similarly, the average of

Table 4.4: Parameters of QUAD instances

	Constructions	QUAD($2^{32}, 32, 32$)	QUAD($2^{32}, 48, 48$)	QUAD($2^{32}, 64, 64$)
	Variables	32	48	64
	Polynomials	64	96	128
	Monomials	561	1,225	2,145
	Output (bit)	1,024	1,536	2,048
Memory size	System (KB)	140.25	459.375	1,072.5
	Key(Byte)	128	192	256
	Security (bit)	≤ 78	≤ 104	≤ 134
	Security looseness (%)	88.4	80.8	94.0

monomials in quadratic terms $\lambda \simeq \binom{64+2}{2} = 2145$, and $\tau = \lambda T \simeq 2.3029 \times 10^{22}$. Therefore, the running time of XL-Wiedemann $C_{XL} = 3\tau T\mathfrak{m} \simeq 3 \times 2.3029 \times 10^{22} \times 1.0736 \times 10^{19}\mathfrak{m} \simeq \lambda T = 7.4171 \times 10^{41}\mathfrak{m}$ multiplications over GF(2^{32}). From our GPU implementations of multiplications over GF(2^{32}), we assume that $\mathfrak{m} = 0.03$ from our multiplication result on GPU. Hence, $C_{XL} \simeq 27.4171 \times 10^{41} \times 0.03 \simeq 2.2251 \times 10^{40} \leq 2^{134}$.

4.5.2 Optimizing Evaluation of Polynomials on CUDA API

We should consider the size of warp, which is the maximal number of parallel threads of each block at a time. Let W be a number of warps, T be the number of threads in a kernel. The kernel is executed with $\lceil T/W \rceil$ iterations. Therefore, when $W \nmid T$, the kernel is running redundant steps. Hence, we should tune the number of threads in order that it is a multiple of W. In CUDA, $W = 32$. Then, we consider the case of $n = 64$. The number of terms with $n = 64$ is $\binom{64+2}{2} = 33 \times 65 \mid 32$. We reduce the terms by remove in variables. Now, we remove 2 variables for each polynomial ($k = 2$). Then, the number of terms is reduced to $\binom{64-2+2}{2} = 32 \times 63$. Similarly, we chose $k = 2$ and 5 for $n = 32$ and 48, respectively.

From our experimentation result in Table 4.2, we choose bitslicing method of the polynomial basis as the multiplication over GF(2^{32}) for QUAD on GPU. Then, we can handle 32 polynomials at a time with a 32-bit integer variable. Figure 4.5 shows placements of polynomials for each QUAD construction. Each bit of variables have bit data of terms in different polynomials and each term is constructed by 32 bits over GF(2^{32}). Therefore, they require 32 memory loading in a kernel. Since loading data in a kernel should be as a straight chunk in CUDA, our data constructions are separated into bit-data chunks.

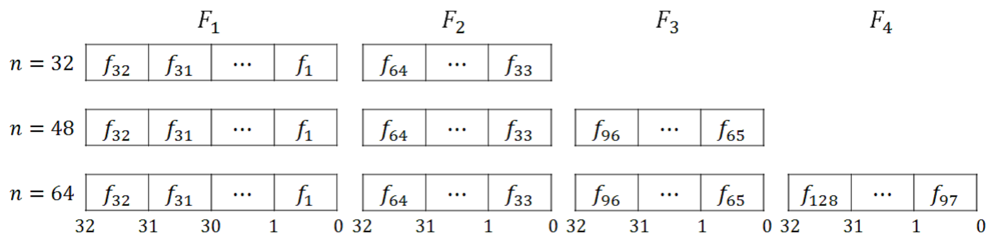


Figure 4.5: Placements of polynomials for each QUAD construction.

4.5.3 Experimental Result

We implement QUAD stream ciphers over GF(2^{32}) on CPU and GPU. In this work, we implement three constructions about QUAD(2^{32}, 32, 32), QUAD(2^{32}, 48, 48) and QUAD(2^{32}, 64, 64). Moreover, we measured encryption time of each construction with 10MB data. We show the result in Table 4.5.

Table 4.5: Encrypting time of QUAD over $GF(2^{32})$

Variables	32	48	64
Polynomials	64	96	128
CPU (Intel Core i7 875K)			
Encrypting time (sec)	205.105	298.842	392.277
Throughputs (Kbps)	399.408	274.126	208.832
GPU (NVIDIA GTX TITAN)			
Encrypting time (sec)	4.032	3.222	4.120
Throughputs (Mbps)	19.841	24.827	19.419
Speed up factor	50.869	92.743	95.220

Also, we show the comparison with related works in Table 4.6 and 4.7. Table 4.6 shows comparisons with other QUAD implementations. Our result is not the fastest. However, the faster constructions, QUAD(2, 128, 128), QUAD(2^4, 40, 40) and QUAD(2^8, 20, 20) are

Table 4.6: Comparison with other QUAD implementations

	Implementation environment	Constructions			Output (bit)	Key (KB)	Throughputs (Mbps)	Security (bit)
		q	n	m				
BGP06 [14]	Pentium 4	2	160	320	160	503.164	5.7	≤ 140
BBG067 [13]	Opetron 64 bit	2	160	320	160	503.164	8.45	≤ 140
		2^4	40	80	160	33.633	23.59	≤ 71
		2^8	20	40	160	9.023	42.15	≤ 45
ABBG07 [8]	FPGA	2	256	512	256	2056.063	2.0	≤ 140
HB13 [25]	Virtex-4, FPGA	2	128	256	128	262.031	374.7	≤ 118
TNS [48]	NVIDIA GeForce GTX 480, GPU	2	160	160	160	503.164	4.872	≤ 140
		2	256	512	256	2056.063	4.115	≤ 160
		2	320	640	320	4012.578	3.656	≤ 320
Our work	NVIDIA GTX TITAN, GPU	2^{32}	32	64	1,024	140.250	19.841	≤ 76
		2^{32}	48	96	1,536	459.375	24.827	≤ 103
		2^{32}	64	128	2,048	1,008.000	19.419	≤ 132

less secure than QUAD($2^{32}, 64, 64$). Hence, our implementations seems to be a trade-off point between speed and security. Table 4.7 shows comparisons with other GPU implementations. Our GPU implementations are 50-95 times faster than CPU. Hence, our implementations make more efficient than our previous work [48]. Moreover, these factors show that QUAD stream cipher is suitable for parallel implementations.

Table 4.7: Comparison with previous GPU implementations of QUAD.

	GPU	Algorithm	Throughputs	Speed up factor
TNS13 [48]	NVIDIA GeForce GTX 480	$QUAD(2, 160, 160)$	4.872Mbps	10.00
		$QUAD(2, 256, 256)$	4.115Mbps	21.32
		$QUAD(2, 320, 320)$	3.656Mbps	29.72
Our work	NVIDIA GTX TITIAN	$QUAD(2^{32}, 32, 32)$	19.841 Mbps	50.869
		$QUAD(2^{32}, 48, 48)$	24.827 Mbps	92.743
		$QUAD(2^{32}, 64, 64)$	19.419 Mbps	95.220

Profile of Kernels

Our multiplications over GF(2^{32}) has 132 registers. Each block has only 65,536 registers. Then, we can only run 496 threads (15.5 warps) in a block. Also, each SM can have 2,048 threads over blocks, if the sum of registers is at most 65,536. The occupancy of utilization shows 100 % when each SM has 64 warps(2,048 threads). If a kernel has 135 registers per threads, each SM can have only 12 warps. Therefore, we can only achieved 19 % occupancy for multiplications.

On the other side, summations of terms in polynomials have only 11 registers. If the number of registers in a kernel is less than 32, each SM can have 2,048 threads. Moreover, each QUAD instance have 256, 512, 1,024 threads in a block. Then, we achieve 100 % occupancy.

4.6 Conclusion

In this work, we discuss fast implementations of QUAD over GF(2^{32}). We discuss 3 approaches of accelerating QUAD, parallelization of evaluating multivariate quadratic polynomials, finding the most suited multiplication method on GPU and optimizing on CUDA. In the parallelization approach, we also provide the variable-base reduction method of terms in polynomials.

By the experimentation of multiplication over GF(2^{32}). We find a more suited method than in our previous work [50]. The multiplication using bitslicing show a throughput of over 800 Gbps.

Finally, we show implementations of QUAD steam cipher over GF(2^{32}) on GPU with several optimizations. QUAD($2^{32}, 48, 48$) and QUAD($2^{32}, 64, 64$) show speed up factors of over 90 times compared to CPU. We consider that our implementation result is a tradeoff point between speed and security.

At a future work, we would like to discuss the security of our QUAD implementations. For example, evaluating how decrease the security in our variable-base reduction method. Also, we are interested to generalizations to other extensions of the binary field (e.g. GF(2^{16}) or GF(2^{64})).

Chapter 5

Parallelizations of MPKC using Linear Recurrence Sequence

5.1 Introduction

Background

Stream ciphers are a type of efficient symmetric cryptosystems, whose encryption is performed by xoring messages with keystreams. The security of a stream cipher is largely determined based on its pseudo-randomness, e.g., periodicity, unbiasedness, etc [18, 54]. In these discussions, security parameters are evaluated by experimentation with known attacks. In contrast, there are stream ciphers that provide *provable security*, with reductions to well-known mathematically difficult problems. For example, Blum, Blum, and Shub introduced a construction based on PRNG, whose security can be reduced to the difficulty of the integer factorization problem [15]. Proposed by Berbain, Gilbert, and Patarin, QUAD is also such a stream cipher endowed with provable security [14]. It uses the theory of multivariate public-key cryptography (MPKC) and generates keystreams by evaluating multivariate quadratic polynomials over finite fields. Generally, we denote the construction of QUAD with a system over GF(q) of n unknowns and r-bit output streams as QUAD(q, n, r). The security of QUAD depends on the complexity of solving multivariate quadratic equation systems over finite fields, i.e., the $\mathcal{MQ}$ problem. Since the $\mathcal{MQ}$ problem is known to be NP-complete [11], QUAD is a provably secure stream cipher.

However, QUAD has a problem when it comes to computational cost. QUAD requires evaluating multivariate quadratic polynomials over finite fields. Typically, QUAD(q, n, r) needs $mn(n+2)$ additions and $m(n+1)^2$ multiplications over GF(q) for evaluation, where m is the number of polynomials, that is, $m = n + r$. Therefore, efficient evaluation is necessary for making QUAD faster.

Related work

Petzoldt proposed an efficient method of evaluating QUAD, which reduces the computational cost from $\mathcal{O}(mn^2)$ to $\mathcal{O}(mn)$ [42]. His idea is to use LRS. Coefficients of LRS QUAD are powers of generators of finite fields. Then, LRS QUAD computes several multiplications at a time sequentially.

Parallelization technique is an effective method for accelerating QUAD. Hamlet and Brocato provided implementations of parallelized QUAD on field-programmable gate array (FPGA) [25]. They showed a throughput of 374 Mbps for QUAD with 256 quadratic polynomials in 128 variables over GF(2). Tanaka, Yasuda, and Sakurai proposed parallelized QUAD on graphics processing units (GPU) [49]. They showed a throughput of 24.8 Mbps for QUAD with 96 quadratic polynomials in 48 variables over GF(2^{32}).

The challenges

In his implementation, Petzoldt used GF(2^8). The period of generators over GF(2^8) is at most 255. However, his quadratic polynomial has 378 terms. Hence, there are some relations between some terms in his quadratic polynomials. Therefore, there is a risk that the security might be reduced.

Petzoldt claimed that his method can be parallelized easily as follows. Each quadratic polynomial in QUAD is independent, so it is easy to parallelize at the polynomial level. However, the degree of parallelization is proportional to the number of polynomials in QUAD, which may not be enough for effectively exploiting the full computational power available on modern GPUs. In this paper, we shall consider further parallelization in evaluating LRS quadratic polynomials for GPU implementation.

Our contributions

We choose GF(2^{32}) for the finite field of LRS QUAD stream cipher. The period of generators over GF(2^{32}) is $2^{32}-1$. This is enough for coefficients of quadratic polynomials, as the number of terms in a quadratic polynomial of n variables is only $\binom{n+2}{2}$.

In this work, we implement two versions of parallelized Pezoldts's LRS QUAD stream cipher [42] on GPU. The first version is the naïve parallelization with parallelization only at the polynomial level. The second version parallelizes computations in quadratic polynomials, e.g., calculating $\alpha_{i,j}x_ix_j$. The result shows that the latter is 2.5 times faster than the former, making it more suitable for GPU implementation of LRS QUAD.

To further exploit the available computational power on modern GPUs, we adopt the multi-stream strategy used by Chen *et al* [20], in which multiple QUAD instances are executed in parallel. We have implemented multi-stream QUAD over GF(2^{32}) and achieved a throughput of 193.40 Mbps for 256 streams of QUAD with 64 polynomials in 32 variables. To the best of our knowledge, this is the best throughput performance result for software implementation of QUAD. To achieve this performance for Petzoldt's LRS QUAD, we have introduced three data structures specifically for efficient handling of memory loading with CUDA API, the most popular programming environment for NVIDIA GPUs.

5.2 The QUAD Stream Cipher using the Linear Recurring Sequence

Let p be a prime, and $q = p^k$, where $k \geq 1$. We assume that GF(q) is a degree-k extension field over GF(p). Then a multivariate quadratic polynomial in n variables over GF(q) is given as follows:

$$f(X) = \sum_{1\leq i\leq j\leq n} a_{i,j}x_ix_j + \sum_{1\leq k\leq n} b_kx_k + c$$
$$(a_{i,j}, b_k, c \in \mathrm{GF}(q), (1 \leq i \leq j \leq n, 1 \leq k \leq n)). \tag{5.1}$$

Moreover, a system of m multivariate quadratic polynomials in n variables is denoted as follows:

$$S(X) = \{f_1(X), \ldots, f_m(X)\}. \tag{5.2}$$

The QUAD stream cipher uses the Equation (5.2) as a PRNG to generate keystreams [14]. When $m = tn$ (where $t \geq 2$), the PRNG of QUAD divides the system of m multivariate quadratic polynomials $S(X)$ into two subsystems $S_{it}(X)$ of n polynomials and $S_{out}(X)$ of $(t-1)n$ polynomials. Then, QUAD uses $S_{it}(X)$ to update internal states X and $S_{out}(X)$ to output keystreams. Finally, it encrypts messages by xoring with keystreams.

5.2.1 Quadratic Polynomials Generated with LRS

Let $\gamma_1, \gamma_2, \ldots, \gamma_L$ be elements of GF(q). Then, a linear recurring sequence (LRS) of length L: $\{s_1, s_2, \ldots | s_i \in \mathrm{GF}(q)\}$ is given as follows:

$$s_j = \alpha_1 \cdot s_{j-1} + \alpha_2 \cdot s_{j-1} + \cdots + \alpha_L \cdot s_{j-L} \ \ \forall j > L. \tag{5.3}$$

The values $s_1, \ldots, s_L$ are the initial values of the LRS.

Alternatively, a quadratic polynomial $f(X)$ given by Equation (5.1) can also be written as follows:

$$f(\hat{X}) = \hat{X} \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ 0 & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{n,n} & b_n \\ 0 & 0 & \cdots & 0 & c \end{pmatrix} \hat{X}^T$$

$$\hat{X} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n & x_{n+1}(=1) \end{pmatrix}. \tag{5.4}$$

Now, we assume that $\gamma \in \mathrm{GF}(q)$ is a generator of $\mathrm{GF}(q)$. Then, there is an LRS:

$$T_i = \gamma \cdot T_{i-1} + M_{i,i} \cdot x_i (i \geq 2), \tag{5.5}$$

where $M_{i,i} = \gamma^{\sum_{j=1}^{i-1} n-j+2}$, and the initial value $T_1 = x_1$. Then, every term $x_i T_i$ can be denoted as follows:

$$x_i T_i = \sum_{j=1}^{i} \gamma^{i-j} \cdot M_{j,j} \cdot x_i x_j. \tag{5.6}$$

Equation (5.6) shows that $x_i T_i$ includes every $x_i x_j$, where $i \leq j$. Hence, quadratic polynomials can be computed via the following summation:

$$f(\hat{X}) = \sum_{i=1}^{n+1} x_i T_i. \tag{5.7}$$

That is, Equation (5.7) essentially computes the following matrix:

$$f(\hat{X}) = \hat{X} \begin{pmatrix} 1 & \gamma & \cdots & \gamma^{n-1} & \gamma^{n} \\ 0 & \gamma^{n+1} & \cdots & \gamma^{2n-1} & \gamma^{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \gamma^{\binom{n+2}{2}-3} & \gamma^{\binom{n+2}{2}-2} \\ 0 & 0 & \cdots & 0 & \gamma^{\binom{n+2}{2}-1} \end{pmatrix} \hat{X}^T \tag{5.8}$$

Petzoldt's LRS QUAD

Algorithm 2 shows the keystream generation algorithm in Petzoldt's LRS QUAD stream cipher. This algorithm has two iteration steps for evaluating a quadratic polynomial. The first one is computing LRS values T_i by Equation (5.5), and the other, evaluating quadratic polynomials $f_k(\hat{X})$ by Equation (5.7). The first iteration takes n steps, each of which requires 2 multiplications and 1 addition. In addition, the second iteration takes $n+1$ steps and requires $n+1$ multiplications and n additions. Therefore, evaluating a multivariate quadratic polynomial requires $3n+1$ multiplications and $2n$ additions. Hence, keystream generation in QUAD with m polynomials in n variables takes $3m \cdot n + m$ multiplications and $2m \cdot n$ additions.

5.3 Parallelization of the LRS QUAD

5.3.1 Naïve Method

A naïve parallelization method for Petzoldt's LRS QUAD is suggested in his own work [42]. This method parallelizes between each evaluation of quadratic polynomials $f_k(\hat{X})$ in QUAD of m polynomials in n variables.

In addition, we can use the bitslicing method of Berbain, Billet, and Gilbert [13] for parallel evaluation of quadratic polynomials. Usually, one data is stored in one variable, e.g., 32-bit integer register. The bitslicing method slices datas and stores bit-datas of multiple variables in one register. Take 32 datas of 32-bit integers $x_1, \ldots, x_{32}$ as an example. We translate to bitsliced datas $y_1, \ldots, y_{32}$. Then, y_k has the datas of k-bit of every x_i. The i-th bit of y_k is stored the k-th bit of x_i. In QUAD stream cipher, we can handle 32 quadratic polynomials at a time by bitslicing.

5.3.2 Parallel Evaluation of Quadratic Polynomials

In Algorithm 2, there are three places that can be effectively parallelized. The first is in computing $M_{i,i} \cdot x_i$ (for $i \geq 2$) in step 2. Although the computation of $\gamma \cdot T_{i-1}$ is sequential for i, the computation of $M_{i,i} \cdot x_i$ is independent of each other. Then, calculating x_i can be parallelized as precomputing of $T_i = \gamma \cdot T_{i-1} + M_{i,i} \cdot x_i$. Similarly, computing $x_j T_i$ in step 3 can be handled in parallel before evaluating $f(\hat{X}) = \sum_{i=1}^{n+1} x_i T_i$. This is the second place for parallelization.

The third one is the summation $\sum_{i=1}^{n+1} x_i T_i$ in step 3. In this place, we can use the parallel reduction technique [5] as used by Tanaka, Yasuda, and Sakurai in their parallel QUAD implementation [49]. This technique computes the summations of N terms in $\lceil \log N \rceil$ steps.

5.3.3 Multi-Stream Strategy

Nowadays, the latest GPU has more than 1,000 cores in a chip. However, the inherent parallelism in Petzoldt's LRS QUAD stream cipher is not enough for filling modern GPUs. Especially, computing $T_i = \gamma \cdot T_{i-1} + M_{i,i} \cdot x_i$ ($M_{i,i} \cdot x_i$ is precomputed) takes n sequential steps for QUAD of m quadratic polynomials in n variables. Therefore, in this step, we can parallelize only $\lceil m/32 \rceil$ for the QUAD with the bitslicing method. Hence, these steps are bottleneck of this version of QUAD. For example, NVIDIA GeForce GTX TITAN is 2,688-core GPU. However, even QUAD with 1,024 quadratic polynomials is parallelized to only 32 threads. Therefore, the QUAD uses only 1.2 % of cores on NVIDIA GeForce GTX TITAN.

Then, we use the multi-stream strategy by Chen *et al* [20]. This strategy runs several QUAD instances in parallel. When running k streams of QUAD of m quadratic polynomials, we can compute $k\lceil m/32 \rceil$ in a step of $T_i = \gamma \cdot T_{i-1} + M_{i,i} \cdot x_i$. This way, the GPU utilization rate can be increased significantly.

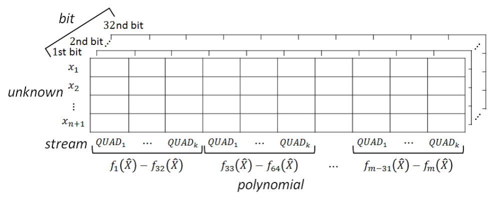


Figure 5.1: Data structures used to realize LRS QUAD on GPU.

5.3.4 Data Structures

We use three data structures for realizing Petzoldt's LRS QUAD on GPU. The first data structure stores the unknowns $\hat{X} = \{x_1, \ldots, x_{n+1}\}$, the second, temporary values $T_i^{(k)}$, while the third, the results $f_k(\hat{X})$. Each data structure is constructed by 3- or 4-dimensional arrays, *bit*, *unknowns*, *polynomial*, and *stream*. The dimension of *bit* shows the index of the bit data of a variable, as each bit in a variable is separated into different variables for the bitslicing method. The unknown data structure doesn't has the *polynomial* dimension, because the unknowns in a stream of QUAD are common for each polynomial in the stream. The *stream* is used only in the version using the multi-stream strategy.

We need to decide an order of these dimensions in the data structures. This order is important in optimizing memory loading in CUDA. Generally, CUDA kernels (GPU functions) execute several blocks of threads in parallel. Each block divides its own threads into some small units called *warps*, and the number of threads in a warp is at most 32. This means that each block executes maximally 32 threads at a time. Originally, memory loadings in a warp are executed serially. However, when there are multiple consecutive memory requests from threads in a same warp, these requests are coalesced to one single large memory request [1]. In other words, such memory loadings are executed at a time and hence more efficient. See Figure 5.1 for a pictorial depiction of the data structures we have used.

Table 5.1: Constructions of QUAD instances.

Unknowns	Polynomials	Key size (KB)
32	64	8.25
64	128	32.5
128	256	129
256	512	514
512	1,024	2,052

5.4 Experimental Results

We measure the running time of encrypting 10 MB of data. We have implemented both versions of parallelized Petzoldt's LRS QUAD stream cipher on GPU. Table 6.1 shows the constructions of the implemented instances. We select GF(2^{32}) as the base finite field of our constructions. The reason of this choice is due to the period of generators in finite fields. The Petzoldt's LRS QUAD uses generators for coefficients of quadratic polynomials. The period of generators over GF(q) is $q-1$. If the number of coefficients in a quadratic polynomial exceeds the period, some pairs of unknowns would have some relationship, and there is a risk of reducing the security. QUAD with 512 variables has 131,841 terms in a quadratic polynomial. Therefore, we choose GF(2^{32}).

Moreover, we use the multi-stream strategy for the second version. We run 1, 8, 16, 32, 64, 128, and 256 streams for each construction. We use Intel core i7 875K for CPU, NVIDIA GeForce TITAN GTX for GPU in all our experiments.

Table 5.2 shows the evaluation time of both versions. Also, it shows the result of CPU implementations for comparison. CPU implementations use only the bitslicing method for parallelization. Overall, the second version is about 2.5 times faster than the first naïve version for each construction. However, both implementations are slower than CPU implementation for 32 and 64 unknowns. We suspect that this is because there is only a limited amount of parallelism for LRS QUAD.

Generally, a core of GPU is slower than that of CPU. For example, a core of NVIDIA GeForce GTX titan is about 3.5 times slower than of Intel core i7 875K. Also, GPU's

Table 5.2: Evaluating time of 10 MB by Petzoldt's LRS QUAD.

Unknowns	Encrypting time (sec)		
	CPU	GPU	
		version 1	version 2
32	29.811	190.531	74.922
64	59.225	190.053	74.426
128	120.082	189.540	73.745
256	249.737	190.677	72.182
512	602.434	200.517	84.757

memory performance is not as good as that of CPU. Therefore, GPU implementations require larger parallelism for higher efficiency. However, QUAD with 64 unknowns only has 128 quadratic polynomials. They are stored in only 4 variables. In other words, GPU can parallelize at most 4 threads in sequential steps. Hence, this limited parallelism is a bottleneck of GPU implementations, making them slower than their CPU counterparts.

Table 5.3 shows the evaluation time of the second version using the multi-stream strategy. Roughly, evaluation time seems inversely proportional to the number of multi-streams. The fastest result, running 256 streams of QUAD with 64 quadratic polynomials in 32 unknowns, achieves a throughput of 193.396 Mbps.

5.4.1 Comparing with Related Works

Table 5.4 shows the comparison with other QUAD implementations [8, 13, 14, 25, 48, 49]. The constructions q, n, and m mean using finite fields GF(q), the number of unknowns, and the number of quadratic polynomials. Our parallelized Petzoldt's LRS QUAD is the fastest software implementation. Although our implementation seems slower than the FPGA implementation by Hamlet and Brocato [25], our 256 streams of QUAD with 128 quadratic polynomials in 64 unknowns has a higher level of security than their QUAD. Hence, our implementation can be viewed as providing a trade-off between speed and security.

Table 5.3: Evaluation time of 10 MB using the multi-streams strategy.

Streams	Encrypting time (sec)				
	Unknowns				
	32	64	128	256	512
1	74.922	74.426	73.745	72.182	84.757
2	37.501	37.165	37.141	38.654	43.219
3	25.116	24.926	24.885	26.337	29.342
4	18.736	18.622	18.679	20.136	22.151
5	15.134	15.058	15.262	16.396	18.053
6	12.619	12.564	12.840	13.752	15.296
7	10.880	10.841	11.369	11.920	13.185
8	9.396	9.353	10.378	10.503	11.529
16	4.735	4.755	5.297	5.371	6.017
32	2.415	2.562	2.748	2.831	3.367
64	1.188	1.355	1.448	1.641	2.162
128	0.696	0.740	0.844	1.076	1.629
256	0.414	0.469	0.600	1.154	2.083

Table 5.4: Comparison of various QUAD implementations

	Implementation environment	Constructions			Key (KB)	Throughputs (Mbps)	Security (bit)
		q	n	m			
BGP06 [14]	Pentium 4	2	160	320	503.164	5.7	≤ 140
BBG06 [13]	Opetron 64 bit	2	160	320	503.164	8.45	≤ 140
		2^4	40	80	33.633	23.59	≤ 71
		2^8	20	40	9.023	42.15	≤ 45
ABBG07 [8]	FPGA	2	256	512	2056.063	2.0	≤ 140
HB13 [25]	Virtex-4, FPGA	2	128	256	262.031	374.7	≤ 118
TNS13 [48]	NVIDIA GeForce GTX 480, GPU	2	160	160	503.164	4.872	≤ 140
		2	256	512	2056.063	4.115	≤ 160
		2	320	640	4012.578	3.656	≤ 320
TYS14 [49]	NVIDIA GTX TITAN, GPU	2^{32}	32	64	140.250	19.841	≤ 76
		2^{32}	48	96	459.375	24.827	≤ 103
		2^{32}	64	128	1,008.000	19.419	≤ 132
Our work (256 streams)	NVIDIA GTX TITAN, GPU	2^{32}	32	64	2,112	193.396	≤ 76
		2^{32}	64	128	8,320	170.476	≤ 132

5.5 Conclusion

In this paper, we discuss how to parallelize on GPU Petzoldt's LRS QUAD [42]. We show two designs and their corresponding implementations. The first version is a naïve parallelization, while the second exploits parallelism in evaluating quadratic polynomials. The latter is about 2.5 times faster than the former on modern GPU.

Moreover, we use the multi-stream strategy of Chen *et al* [20]. Running 256 streams of QUAD with 64 polynomials in 32 unknown over GF(2^{32}) achieves a throughput of 193.396 Mbps, while 128 polynomials in 64 unknowns, 170.476 Mbps. We can see that running 256 streams of QUAD with 128 polynomials in 64 unknowns is a new trade-off between throughput and security.

One next step is a more careful security analysis of Petzoldt's LRS QUAD. Especially, we would like to analyze the potential security loss when there are some special relationships between some pairs of unknowns in the system.

Chapter 6

Accelerating Extended Linearization Algorithm

6.1 Introduction

The problem of finding roots of non-linear multivariate polynomial equations over finite fields lies at the core of the security for multivariate public-key cryptography (MPKC). Many MPKCs, e.g., Unbalanced Oil and Vinegar (UOV) [30], Hidden Field Equations (HFE) [41], and the QUAD stream cipher [14], base their security on the quadratic case of such problems, which we will refer to as the $\mathcal{MQ}$ problem. Therefore, estimating the complexity of the $\mathcal{MQ}$ problem is of crucial importance for determining the security of these MPKCs.

To this date, there are two kinds of efficient algorithms for solving the $\mathcal{MQ}$ problem. One is the Gröbner basis method, and the other is the eXtended Linearization (XL) algorithm. Both algorithms generate new equations from the original systems. Although XL is shown to be a redundant variant of a Gröbner basis algorithm F_4 [9], it does have the advantage of having a smaller memory footprint in practice [55].

The bottleneck computation in XL is the solving step of linearized systems. For sparse systems generated by XL, the Wiedemann algorithm can be used to efficient solve an $N \times N$ non-singular matrix system with row sparsity k in $\mathcal{O}(kN^2)$ complexity in terms of multiplications and additions. Here N is determined by the *degree of regularity* for the $\mathcal{MQ}$ problem, which we will give more detail later in this paper.

There are several implementations of the XL-Wiedemann algorithm. Yang *et al.* estimate the solving time for $\mathcal{MQ}$ instances in 6–15 unknowns by their C++ implementation [55]. Moreover, they show that the expected time for solving an $\mathcal{MQ}$ instance of 40 equations in 20 unknowns over GF(2^8) is around 2^{45} CPU cycles. Cheng *et al.* implement the XL-Wiedemann algorithm on a cluster of 8 PCs in NUMA architecture [21]. As a result, they can solve $\mathcal{MQ}$ instances of 36 equations in 36 unknowns over GF(2) in 46,944 seconds, 64 equations in 32 unknowns over GF(16) in 244,338 seconds, as well as 58 equations in 29 unknowns over GF(31) in 12,713 seconds.

So far, we have not seen any implementation of the XL-Wiedemann algorithm on GPU, which is a candidate for further speed-up because several steps of the XL-Wiedemann algorithm can be parallelized. Therefore, we consider accelerating XL-Wiedemann on GPU. However, GPU implementation poses a set of very different limitations from its CPU counterpart. Hence, in this paper we shall detail these challenges and how we have dealt with them.

Our contributions include the following. We present several GPU implementations of the XL-Wiedemann algorithm, in which multiplication of a sparse matrix with a dense vector is parallelized on GPU. Moreover, we benchmark an implementation based on the cuSPARSE library using floating-point arithmetic. Finally, we show the experimental results of solving $\mathcal{MQ}$ instances over GF(2), GF(3), GF(5), and GF(7). Our implementation can solve $\mathcal{MQ}$ instances of 74 equations in 37 unknowns over GF(2) in 36,972 seconds, 48 equations in 24 unknowns over GF(3) in 933 seconds, as well as 42 equations in 21 unknowns over GF(5) in 347 seconds. The largest instance for matrix we have solved is 80 equations in 40 unknowns over GF(2) in 295,776 seconds, it has 3.78 billions terms in a matrix. On the other hand, the largest instance for solving time, we have solved is 48 equations in 24 unknowns over GF(7) in 34,883 seconds, whose complexity is around $\mathcal{O}(2^{67})$ if we use a brute-force kind of approach.

The cuSPARSE library only supports floating-point arithmetic, not integer arithmetic, let alone finite field arithmetics. Therefore, we need to use cuSPARSE functions to implement finite field arithmetics via additional operations such as the modular operations.

6.2 The XL-Wiedemann Algorithm for $\mathcal{MQ}$ Problem

The security of MPKC is largely based on the complexity of solving a system of multivariate non-linear equations over finite fields. The $\mathcal{MQ}$ problem is a quadratic case of this problem. Generic $\mathcal{MQ}$ is known to be NP-complete [11].

Let $q = p^k$, where p is a prime, and $\boldsymbol{x} = \{x_1, \ldots, x_n\}$ $(\forall i, x_i \in \mathrm{GF}(q))$. Generally, multivariate quadratic polynomial equations in n unknowns over GF(q) can be described as follows:

$$f(\boldsymbol{x}) = \sum_{1 \le i \le j \le n} \alpha_{i,j} x_i x_j + \sum_{1 \le k \le n} \beta_k x_k + \gamma = 0, \tag{6.1}$$

where $\forall i, j, k$, $\alpha_{i,j}, \beta_k, \gamma \in \mathrm{GF}(q)$. The $\mathcal{MQ}$ problem consists solving quadratic polynomial equations given by $\boldsymbol{y} = \{f_1(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})\}$

The original XL algorithm was proposed by Courtois *et al.* in 2000 [22]. The idea of XL is based on a linearization technique, in which new unknowns representing non-linear terms, e.g., $y_{1,2} = x_1 x_2$, are generated and treated as an independent variable. If the number of linearly independent equations is greater than the number of variables in the resulted linearized system, then we can solve it by, e.g., Gaussian elimination. If not, we can generate new equations from the original ones by raising to a higher degree. For the sake of completeness, the XL algorithm is described in Algorithm 3. Simply put, the *degree of regularity* D is the minimal degree at which the number of linearly independent equations exceeds the number of unknowns in the linearized system.

```
Require: $m$ quadratic polynomial equations $F = \{f_1, \dots, f_m\}$, $m$-th vector $\boldsymbol{y} = F(\boldsymbol{x})$,
    and the degree of regularity $D$.
Ensure: The $n$-th unknown vector $\boldsymbol{x} = \{x_1, \dots, x_n\}$.
1: Multiply: Generate products between all polynomial equations and all unknowns of
    the form $\prod_{j=1}^{D-2} x_{i_j}$.
2: Linearize: Treat each monomial in $x_i$ of degree $\leq D$ as a new, independent
    unknown and perform an elimination algorithm on the linearized equations
    obtained in Step 1 to derive a univariate equation.
3: Solve: Solve the univariate equations obtained in Step 2 over GF($q$).
4: Back-substitute: Find the values of the other unknowns by back-substitution into
    the linearized system.
```

Algorithm 3: The XL algorithm [22]

The XL algorithm generates sparse equations in Step 1 of Algorithm 3. The number of non-zero terms of an equation is only $\binom{n+2}{2}$ out of all possible $\binom{n+D}{D}$ terms, since the generated equations are just a product of the original equations and some monomials. However, the Gaussian elimination is not suited for solving such sparse linear systems, as it cannot take advantage of the sparsity. The XL-Wiedemann algorithm [36] addresses this problem of the original XL by replacing the Gaussian elimination with the Wiedemann algorithm [52], which is more efficient for solving systems of sparse linear equations.

6.2.1 The Wiedemann Algorithm in the XL

The algorithm of XL can be separated to 2 major steps, (i) generating a large linearized system, and (ii) solving the system constructed in (i). A system of linear equation can be denoted as finding unknown vector $\boldsymbol{x}$ with a given vector $\boldsymbol{x}$ from $A\boldsymbol{x} = \boldsymbol{b}$, where A is a matrix of the system. In other words, solving a system is finding the inverse matrix A^{-1} of A, because $A^{-1}\boldsymbol{b} = A^{-1}A\boldsymbol{x} = I\boldsymbol{x} = \boldsymbol{x}$, where I is the unit vector. There exits several methods for solving the linear algebra.

Gaussian Elimination

The Gaussian elimination is a most basic method for the linear algebra. This method is separated to 2 steps, (i)forward elimination, and (ii)back substitution. The forward elimination step generates a upper triangular matrix from A by substitution i-th column element of j-th row equation from i-th row equation, where $i < j$. After this step, we will get a univariate equation. The back substitution step computes the root of the univariate equation, and then, substitutes the value of the root into other equations. In the step, we iterate up to find all values of unknowns.

LU decomposition

The LU decomposition is a generalized version of the Gaussian elimination. Generally, a square matrix A can be decomposed to a product a lower triangular matrix L and an upper triangular matrix U as $A = LU$. Let $\boldsymbol{y} = U\boldsymbol{x}$. Therefore, $A\boldsymbol{x} = LU\boldsymbol{x} = L\boldsymbol{y} = \boldsymbol{b}$. Then, we find a vector $\boldsymbol{y}$ from $L\boldsymbol{y} = \boldsymbol{b}$. After that, we finally get a vector $\boldsymbol{x}$ from $U\boldsymbol{x} = \boldsymbol{y}$.

The Wiedemann Algorithm

The Wiedemann algorithm [52] is a solving method for a system of linear sparse equations over finite fields. Let A be an $N \times N$ non-singular matrix over GF(q). The Wiedemann algorithm finds a non-zero vector $\mathbf{x}$, where $\mathbf{y} = \mathbf{A}\mathbf{x}$. The original Wiedemann algorithm is described in Algorithm 4.

The Wiedemann algorithm computes mainly products $A^i\boldsymbol{b}$ and the Berlekamp-Massey algorithm. Since $A^i\boldsymbol{b}$ can be computed by $A(A^{i-1}\boldsymbol{b})$, we can keep the sparsity of A. Therefore, sparse matrices are suited to the Wiedemann. Also, the XL algorithm generates a sparse matrix from a system of quadratic equations. Hence, sometimes, XL uses the Wiedemann algorithm for the linear algebra. Yang *et al.* uses XL with the Wiedemann and analyses $\mathcal{MQ}$ of the QUAD citeyang2007analysis. Cheng *et al.* parallelizes XL with the block version of the Wiedemann (called the block Wiedemann) on the 8 PC

```
Require: $N \times N$ non-singular matrix $A$ and vector $\boldsymbol{b}$, where $A\boldsymbol{x} = \boldsymbol{b}$.
Ensure: The unknown solution vector $\boldsymbol{x}$.
1: Set $\boldsymbol{b}_0 = \boldsymbol{b}$, $k = 0$, $\boldsymbol{y}_0 = 0$, and $d_0 = 0$.
2: Compute the matrix sequence $s_i = \boldsymbol{u}_{k+1} A^i \boldsymbol{b}_k$ for $0 \le i \le 2(N - d)$, with a random
   vector $u_{k+1}$.
3: Set $f(\lambda)$ to the minimum polynomial of the sequence of $s_i$ using the
   Berlekamp-Massey algorithm.
4: Set $\boldsymbol{y}_{k+1} = \boldsymbol{y}_k + f^-(A)\boldsymbol{b}_k$, where $f^-(\lambda) := \frac{f(\lambda)-f(0)}{\lambda}$, $\boldsymbol{b}_{k+1} = \boldsymbol{b}_0 + A\boldsymbol{y}_{k+1}$, and
   $d_{k+1} = d_k + \deg f(\lambda)$.
5: If $\boldsymbol{b}_{k+1} = 0$, then the solution is $\boldsymbol{x} = \boldsymbol{y}_k$
6: Set $k = k + 1$ and go to Step 2.
```

Algorithm 4: The Wiedemann algorithm [52]

cluster [21].

6.3 Linear Algebra on GPU with CUDA

Provided by NVIDIA, CUDA is a development environment for GPU based on C language. Proprietary tools for using GPU have existed before CUDA; such tools often need to tweak OpenGL and/or DirectX and disguise computation as graphics rendering commands. Therefore, these tools are not easy to use, whereas CUDA is efficient because it can use GPU's computational cores directly.

In CUDA, hosts correspond to PC, and devices correspond to GPU. CUDA works by making the host control the device via kernels. Because only one kernel can be executed at a time, we need to parallelize processing inside a kernel. A kernel handles some blocks in parallel. A block also handles some threads in parallel. Therefore, a kernel can handle many threads simultaneously.

NVIDIA provides several libraries for linear algebra. For example, the cuBLAS library provides functions of the Basic Linear Algebra Subprograms (BLAS) library. BLAS is classified into three levels of functionalities. Level 1 functions provide operations on vectors, level 2 operations on vectors and matrices, while level 3 allows matrix-matrix operations. The cuSPARSE library is actually the sparse version of the cuBLAS library.

Therefore, cuSPARSE also provides these three levels of functions.

We assume that D is the degree of regularity for the XL algorithm. Then, XL constructs an $\binom{n+D}{D} \times \binom{n+D}{D}$ linearized matrix from the $\mathcal{MQ}$ instances of m equations in n unknowns over GF(q). However, quadratic polynomial equations in n unknowns have only $\binom{n+2}{2}$ terms. Therefore, we can reduce computations of matrix-vector product as well as the memory footprint if we store the matrix in sparse form.

Let N be the degree of row and column in a matrix, and num_{NZ} be the number of non-zero elements in the matrix. Sparse matrix forms have value, row-index and column-index data of non-zero elements in a matrix. There are some sparse matrix formats such as the following [2]

- The COO (coordinate) format is the most basic one. It simply holds value, row-index and column-index data of non-zero elements in the matrix. Therefore, it requires $3num_{NZ}$ for the memory space.
- The CSR (compressed storage row) assumes that the data vector is ordered by the row-index. It differs only row-index from the COO formats, in which it holds the head number of non-zero terms in each row-vector of the matrix instead of row-index data. Then, it requires $2num_{NZ} + N$ memory.
- The ELL (Ellpack-Itpack) format uses two dense $N \times max_{NZ}$ matrices, where max_{NZ} is the maximal number of non-zero terms in a row-vector. One matrix shows the value of non-zero matrix, and the other shows the column-index.

Figure 6.1 shows examples of each of the three formats.

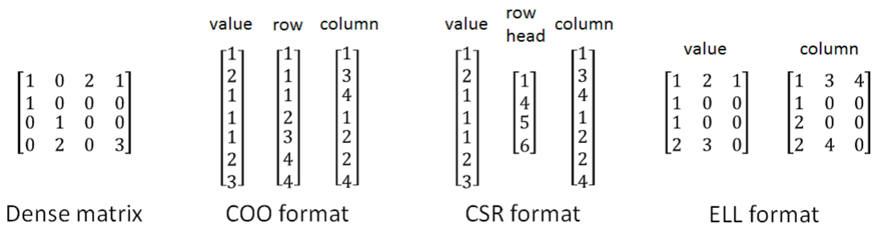


Figure 6.1: Sparse matrix formats.

6.4 Implementing XL-Wiedemann on GPU

6.4.1 Degrees of Regularity over Small Fields

The bottleneck of the XL-Wiedemann algorithm is the linear algebra part that solves an $N \times N$ matrix system. Here N is determined by the degree of regularity D as $N = \binom{N+D}{D}$. The degree of regularity is the minimal degree where the number of linearly independent equations exceeds the number of linearized unknowns. We can figure out the number of linearized unknowns N for the degree d as $N = \binom{N+d}{d}$ easily. Rønjon and Raddum citeronjom2008number gave that an upper bound for the number of linearly independent equations I is decided using the following formula:

$$I = \sum_{i=0}^{\frac{D_m}{D_e}} (-1)^i \binom{m+i}{i+1} \sum_{j=0}^{D_m - i \cdot D_e} \binom{n}{j}. \tag{6.2}$$

Here, D_m is the maximal degree of the monomials, and D_e is the degree of the original equations. For the $\mathcal{MQ}$ problem, $D_m = D - 2$ and $D_e = 2$. Therefore, we can find the minimal degree D, where $I \geq N(= \binom{N+D}{D})$ by Formula (6.2). Figure 6.2 shows degrees of regularity for $\mathcal{MQ}$ instances of $2n$ equations in n unknowns over GF(2), GF(3), GF(5), and other prime fields for $n \leq 64$. The cases of GF(5) and other larger prime fields are actually quite similar. Only GF(2) and GF(3) differ from the other cases because we need to take into consideration field equations $\alpha^q = \alpha$.

From the definition of the degree of regularity, it is obvious that $I \geq N$. However, for

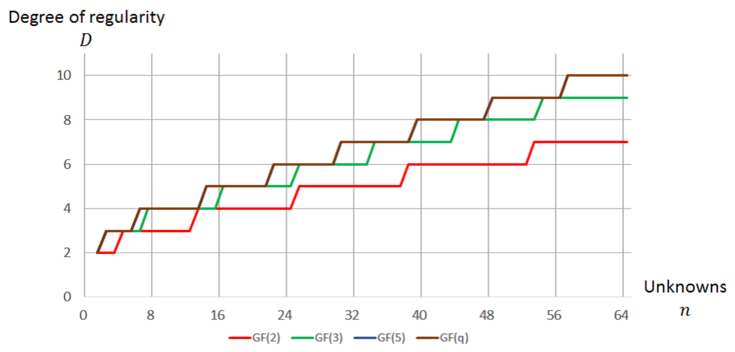


Figure 6.2: The degrees of regularity for $m = 2n$ cases for $n \leq 64$.

the Wiedemann algorithm to work, we need to reduce to N from I. The simplest way is to randomly remove certain equations, which is our strategy in our implementation.

6.4.2 The Wiedemann Algorithm

The Wiedemann algorithm has three separate steps. The first step is to generate the sequence $\{(\boldsymbol{u}, A^i\boldsymbol{b})\}_{i=0}^{2N}$ for an $N \times N$ matrix A and a vector $\boldsymbol{b}$, where $A\boldsymbol{x} = \boldsymbol{b}$, as well as a random vector $\boldsymbol{u}$. The second step is to find the minimal polynomial of the generated sequence $f(\lambda)$ using the Berlekamp-Massey algorithm. The final step is to compute $f^-(A)\boldsymbol{b}$, where $f^-(\lambda) = \frac{f(\lambda)-f(0)}{\lambda}$. In this work, we only implement the first step and the final step on GPU. This is because the Berlekamp-Massey algorithm is sequential in nature and hence might not benefit from parallelization. For example, it has many conditional branches, which are not suitable for GPU implementation. Therefore, we implement the second step on CPU.

6.4.3 Generating Sequence $\{(\boldsymbol{u}, A^i\boldsymbol{b})\}_{i=0}^{2N}$

This step requires multiplying the sparse matrix A and the dense vector $A^{i-1}\boldsymbol{b}$, as well as taking dot product $(\boldsymbol{u}, A^i\boldsymbol{b})$. However, we can choose the random vector $\boldsymbol{u}$ as $\boldsymbol{u} = \{1, 0, \ldots, 0\}$. Therefore, taking dot product amounts to looking up the first coordinate in the vector $A^i\boldsymbol{b}$. Hence, we should consider only multiplication of the sparse matrix A and the dense vector $A^{i-1}\boldsymbol{b}$.

Multiplying the sparse matrix A and the dense vector $A^{i-1}\boldsymbol{b}$ takes two steps. The first one is multiplying non-zero elements in the matrix with the elements in the vector. The other is summing the results of the partial multiplications for each row.

We choose the ELL format for representing sparse matrices. One advantage is that every column width is the same in a matrix, and the multiplication result also has such width. In CUDA kernels, the column width corresponds to the number of threads, while the row height corresponds to the number of blocks. To achieve maximal efficiency, each block should have the same number of threads. Therefore, the ELL format is best suited for GPU implementation.

In summing the partial multiplication results, we use the parallel reduction technique [5]. Such a technique allows computing summation of n items in $\mathcal{O}(\log n)$ steps.

6.4.4 Computing $f^{-}(A)\boldsymbol{b}$

Since $f^{-}(A)\boldsymbol{b} = \sum_{i=1}^{d} c_i A^{i-1}\boldsymbol{b}$, where d is the degree of $f(\lambda)$, this step amounts to summing $c_i A^{i-1}\boldsymbol{b}$, using the same partial sums from the previous step. Hence, there are two strategies for computing $A^i\boldsymbol{b}$. The first one is to store the result of $A^i\boldsymbol{b}$ on GPU. This strategy can avoid recomputing $A^i\boldsymbol{b}$. However, it needs about $\mathcal{O}(N^2)$ memory for storing $A^i\boldsymbol{b}$, where $0 \leq i \leq N$ (since $d \leq N$). Therefore, this strategy can only work for smaller matrices.

The other strategy is to recompute $A^i\boldsymbol{b}$ on the fly. Although it repeats the computation of d products of $A^i\boldsymbol{b}$, it only requires $\mathcal{O}(A^{i-1}\boldsymbol{b})$ memory to hold the last vector of $A^i\boldsymbol{b}$. Therefore, this strategy is more suitable for large matrices.

6.4.5 cuSPARSE

The cuSPARSE library [2] provides functions that multiply a sparse matrix with a dense vector. Therefore, we consider using cuSPARSE as an alternative implementation

Table 6.1: $\mathcal{MQ}$ instances in our experiments..

Field GF(q)	GF(2)		GF(3)		GF(5)		GF(7)	
Degree of regularity D	4	5	4	5	4	5	4	5
Unknowns n	24	37	15	24	13	21	13	21
Equations m	48	74	30	48	26	42	26	42
Matrix								
Linearized terms	12,950	510,415	3,635	110,954	2,379	65,758	2,379	65,779
Nonzero terms	301	704	136	325	105	253	105	253

for computing A and $A^{i-1}\boldsymbol{b}$. There are two important issues with implementations. First, the interface is fixed and opaque. The cuSPARSE library only provides this function for CSR format: $\boldsymbol{y} \leftarrow \alpha A\boldsymbol{x} + \beta\boldsymbol{y}$, where A is a matrix, $\boldsymbol{x}$, $\boldsymbol{y}$ are vectors, and α, β are scalars. Therefore, we set $\beta = 0$ for the first step. Moreover, we are stuck with CSR format for representing sparse matrices when we use cuSPARSE library.

The second issue is the restriction of the unknown type. The cuSPARSE library only supports floating-point arithmetic, not integer arithmetic, let alone finite field arithmetics. Therefore, we need to use cuSPARSE functions to implement finite field arithmetics via additional operations such as the modular operations.

6.5 Experimental Results

We implement the XL-Wiedemann algorithm on GPU using two strategies, integer version and cuSPARSE (floating-point) version. We experiment with solving the largest cases for $D = 4, 5$ over GF(2), GF(3), GF(5), and GF(7) by both implementation strategies and summarize these instances in Table 6.1.

Table 6.2 shows the overall experimental results, and Table 6.3 shows the profiling results of the Wiedemann algorithm. Despite the overhead brought by the two issues mentioned previously, the cuSPARSE version seems to outperform integer version for larger cases. In our experiments, the Berlekamp-Massey algorithm can occupy a significant portion of the total running time and hence may be worth further optimization.

Table 6.2: Running time of XL-Wiedemann on GPU.

	Field GF(q)	GF(2)		GF(3)		GF(5)		GF(7)	
	Degree of regularity D	4	5	4	5	4	5	4	5
	Unknowns n	24	37	15	24	13	21	13	21
	Equations m	48	74	30	48	26	42	26	42
Integer	Solving time (sec)	14.7358	83,782.11	0.5847	2,089.30	0.4415	601.124	0.4856	670.963
	Extension (sec)	0.1248	130.98	0.0116	7.29	0.0059	3.347	0.0053	2.913
	Wiedemann (sec)	14.6101	83,651.08	0.5729	2,082.01	0.4355	597.777	0.4802	668.049
cuSPARSE	Solving time (sec)	8.8982	36,971.85	0.8684	932.95	0.4852	346.571	0.5063	387.121
	Extension (sec)	0.0885	128.28	0.0098	8.00	0.0050	3.366	0.0050	3.354
	Wiedemann (sec)	8.8077	36,843.49	0.8583	924.95	0.4800	343.204	0.5012	383.764

Table 6.3: Profiling results for the Wiedemann algorithm.

	Field GF(q)	GF(2)		GF(3)		GF(5)		GF(7)	
	Degree of regularity D	4	5	4	5	4	5	4	5
	Unknowns n	24	37	15	24	13	21	13	21
	Equations m	48	74	30	48	26	42	26	42
Integer	Running time (sec)								
	Wiedemann	14.6101	83,651.08	0.5729	2,082.01	0.4355	597.777	0.4802	668.049
	Generating Sequence	9.5806	49,719.75	0.3030	1,104.82	0.2131	302.236	0.2304	336.292
	Berlekamp-Massey	4.9253	9,035.16	0.2379	439.1057	0.19	148.328	0.2195	167.483
	Computing $f^{-}(A)\boldsymbol{b}$	0.0937	24,895.43	0.0305	537.99	0.0273	147.188	0.0295	164.249
	Memory Usage (MB)								
	Matrix	29.74	2741.49	5.66	412.67	2.86	190.39	2.86	190.46
	Stream	1279.47	0	100.81	0	43.22	0	43.22	0
cuSPARSE	Running time (sec)								
	Wiedemann	8.8077	36,843.49	0.8583	924.94	0.4800	343.204	0.5012	387.764
	Generating Sequence	3.8079	22,215.69	0.4284	325.75	0.2418	108.0073	0.2393	114.814
	Berlekamp-Massey	4.8855	9,059.83	0.4284	325.75	0.1999	183.685	0.2223	214.049
	Computing $f^{-}(A)\boldsymbol{b}$	0.1045	5,567.20	0.0403	160.77	0.0372	51.473	0.0386	54.863
	Memory Usage (MB)								
	Matrix	44.66	4114.18	5.67	413.10	2.87	190.64	2.87	190.71
	Stream	1279.47	0	100.81	0	43.22	0	43.22	0

We can also use high-quality, state-of-the-art implementations from commercial computer algebra systems like MAGMA.

Finally, we solve the largest case of $D = 6$ over GF(7), which has a system of 24 unknowns and 48 polynomials.. We choose the version of using the cuSPARSE library as a solver of the $\mathcal{MQ}$ instance, because of the result of $D = 5$ cases. Table refD6GF7 shows the construction and experimental result of solving the $\mathcal{MQ}$ instance.

Profile of Kernels

Table refKernelXL shows that profiles of utilization of kernels. Both of integer and cuSPARSE versions use kernels of inner products and additions vector and vector. Also, the integer version implements multiplications matrix and vector. On the other stands, the cuSPARSE version uses the library function of multiplications. Moreover, it uses modular kernels.

The inner product kernel shows 19 % occupancy. However, this kernel has only 1

Table 6.4: Solving the $\mathcal{MQ}$ of 24 unknowns and 48 equations over GF(7)

Constructions	Unknowns n	24
	Equations m	48
Matrix	Linearized terms	593,774
	Non-zero terms	325
Memory (MB)	Matrix	2,208.44
Running time (sec)	XL-Wiedemann	34,881.637
	Linearization	580.406
	Wiedemann	34,301.231
Wiedemann algorithm (sec)	Generating Sequence	11,046.464
	Berlekamp-Massy	17,698.748
	Compute $f^{-}(A)\boldsymbol{b}$	5,555.593

kernel. Hence, it is sufficient. The occupancy of additions between 2 vectors depends on the size of system matrix. This occupancy is from 19 to 100 %. There may exists some improvements by modifying threads.

The number of threads in multiplications kernel of the integer version depends on the maximal number of non-zero term n_{max} in the system. The occupancy of this kernel is less than 56 % is only $n_{max} \leq 96$. The expected number of n_{max} is $\binom{n+2}{2}\frac{q-1}{q}$ for GF(q), where $k > 2$ $((\binom{n+1}{2} + 1)/2$ for GF(2)). Then, probability , we don't have to consider the occupancy in the kernel for $n > 15$ for GF(q) (and $n > 18$ for GF(2)).

In the cuSPARSE version, threads of multiplications between matrix and vector are aromatically modified by CUDA. Hence, we should consider only the kernel of modular operations. The number of threads in this kernel depends on the size of system matrix. Then, tuning of the kernel is similar to additions between 2 vectors.

Table 6.5: Kernel profile of the XL-Wiedemann algorithm.

Type	Kernel	Registers per thread	Shared Memory per blocks(bytes)	Occupancy Utilization (%)
Both	Inner product	10	0	19
	Addition	6-8	0	19-100
Integer	Multiplication	12	4 n_{max}	19-100
cuSPARSE	Modular	9	0	19-100

Table 6.6: Comparing with CPU and GPU implementations

		CPU	GPU
Wiedemann algorithm		34,301.231	1,412,393.162
Wiedemann algorithm (sec)	Step1: Generating Sequence	11,046.464	927,983.902
	Step2: Berlekamp-Massey	17,698.748	17,698.748
	Step3: Compute $f^{-}(A)\boldsymbol{b}$	5,555.593	466,710.512
Product Matrix and Vector (msec)		9.828	825.624

6.5.1 Comparison with CPU Implementations

Table 6.6 shows that the comparison with the CPU implementations of solving a system with 48 equations in 24 unknowns over GF(7). Our CPU evaluations is based on the running time of products sparse matrices and dense vectors. It can compute products of $500,000 \times 500,000$ sparse (row sparsity is 128) and 500,000-degree vector is in 331.244 msec. Then, we assume that running time of matrices and vectors is proportion to the number of rows in matrix and the row sparsity. Therefore, times of step 1 and 3 in Table 6.6 are calculated with a ratio between CPU and GPU. Also, since the Berlekamp-Massey algorithm in step 2 is executed on CPU. Hence, step 2 is assumed as same time of GPU implementations. Finally, our GPU implementations is about 41.176 times faster than CPU implementations.

6.5.2 Comparison with Related Works

Table Comp shows that the comparison with the related work. Yang et. al. solve the system of n quadratic and n quotic equations in n unknowns [55]. The row of multiplications in Table Comp means the expected number of multiplications in the XL-Wiedemann algorithm (given by $3num_{NZ}N$ [55]). It shows that our $\mathcal{MQ}$ is about 1.2 times larger than Yang. et. al. Then, our result is roughly 4 times faster than their result.

Table 6.7: Comparison with the related work.

		Yang et. al. [55]	our works
Implementation Environment		CPU	GPU
Field	$GF(2^8)$	$GF(2^4)$	GF(7)
Constructions	Unknowns	15	24
	Equations	15 quadratic, 15 quotic	48
	Degree of regularity D	8	6
Matrix	Linearized terms	490,314	593,774
	Non-zero terms	385 (average of row)	325
Memory (MB)	Matrix	N.A.	2,208.44
Operations	Multiplications	$2.78 \cdot 10^{14}$	$3.44 \cdot 10^{14}$
Running Time(sec)		$1.17 \cdot 10^6$	$3.49 \cdot 10^5$

6.6 Conclusion

We provide GPU implementations of the XL-Wiedemann algorithm using both integer arithmetic and floating-point arithmetic via cuSPARSE library. Our implementation can solve $\mathcal{MQ}$ instances of 74 equations in 37 unknowns over GF(2) in 36,972 seconds, of 48 equations in 24 unknowns over GF(3) in 933 seconds, as well as 42 equations in 21 unknowns over GF(5) in 347 seconds by using cuSPARSE library. Finally, we solve the largest case of $D = 7$ over GF(7), $\mathcal{MQ}$ of 24 unknowns and 48 equations. The version of using the cuSPARSE library solves the $\mathcal{MQ}$ in 34,882 seconds. Our next goal is to estimate the expected solving time for larger-degree cases.

Chapter 7

Concluding Remarks and Future Work

7.1 Concluding Remarks

We proposed that GPU parallelization techniques for evaluation of a multivariate quadratic system. Moreover, we implemented our technique, and measured evaluating times of multivariate quadratic systems.

First, we presented and evaluated the GPU implementation techniques for QUAD stream cipher. Also we provided optimization techniques of QUAD to suit NVIDIA GeForce GTX 480. Moreover, we carried out the experiments on the implementations of QUAD over $GF(2)$, $GF(2^2)$, $GF(2^4)$ and $GF(2^8)$. As a result, the larger the number of unknowns n is, the slower the throughput of QUAD is. However, when $tn(n+2) \leq 32C$, it is stable. The condition for stable throughputs depends on the number of cores C. Although the GTX 480 has only 480 cores, the GTX 680, which is the latest high-performance GPU, has 1536 cores. Therefore, the throughput of QUAD$(2, n, n)$ is stable if $n \leq 439$. We expect that future GPUs allow efficient implementation of QUAD$(2, 512, 512)$ and more heavy constructions of QUAD.

Second, we discuss fast implementations of QUAD over GF(2^{32}). We discuss 3 approaches of accelerating QUAD, parallelization of evaluating multivariate quadratic polynomials, finding the most suited multiplication method on GPU and optimizing on CUDA. In the parallelization approach, we also provide the variable-base reduction method of terms in polynomials. By the experimentation of multiplication over GF(2^{32}). We find a more suited method than in our previous work [50]. The multiplication using bitslicing show a throughput of over 800 Gbps. Moreover, we show implementations of QUAD steam cipher over GF(2^{32}) on GPU with several optimizations. QUAD($2^{32}, 48, 48$) and QUAD($2^{32}, 64, 64$) show speed up factors of over 90 times compared to CPU. We consider that our implementation result is a tradeoff point between speed and security.

Third, we discuss how to parallelize on GPU Petzoldt's LRS QUAD [42]. We show two designs and their corresponding implementations. The first version is a naïve parallelization, while the second exploits parallelism in evaluating quadratic polynomials. The latter is about 2.5 times faster than the former on modern GPU. Moreover, we use the multi-stream strategy of Chen *et al* [20]. Running 256 streams of QUAD with 64 polynomials in 32 unknown over GF(2^{32}) achieves a throughput of 193.396 Mbps, while 128 polynomials in 64 unknowns, 170.476 Mbps. We can see that running 256 streams of QUAD with 128 polynomials in 64 unknowns is a new trade-off between throughput and security.

Finally, we provide GPU implementations of the XL-Wiedemann algorithm using both integer arithmetic and floating-point arithmetic via cuSPARSE library. Our implementation can solve $\mathcal{MQ}$ instances of 74 equations in 37 unknowns over GF(2) in 36,972 seconds, of 48 equations in 24 unknowns over GF(3) in 933 seconds, as well as 42 equations in 21 unknowns over GF(5) in 347 seconds by using cuSPARSE library. Also, we solve the largest case of $D = 7$ over GF(7), $\mathcal{MQ}$ of 24 unknowns and 48 equations. The version of using the cuSPARSE library solves the $\mathcal{MQ}$ in 34,882 seconds.

7.2 Further Issues

We have some issues for further improvements.

Extending QUAD Implementations to General Finite Fields

Generalizations of fields for evaluating quadratic polynomials : we would like to discuss the security of our QUAD implementations. For example, evaluating how decrease the security in our variable-base reduction method. Also, we are interested to generalizations to other extensions of the binary field (e.g. GF(2^{16}) or GF(2^{64})).

Improvements of Multiplications over GF(2^{32})

Our multiplication techniques are specialized to the parallelization on GPUs. However, each multiplication method is plain. Therefore, we can achieve the known multiplication methods. For example, we can use methods of the optimal extension fields (OEFs) [10]. OEFs are efficient fields specialized to the word size. These fields are GF(p^m), where $p = 2^\omega - c$, $\log c \leq \lfloor \frac{n}{c} \rfloor$ and m is the word-size bit. Although, OEF should compute over GF(p) for additions and multiplications, computational costs of them are low. For additions over GF(p), they require 2 integer additions or subtractions. On the other side, for multiplications over GF(p), they require 1 integer multiplication, 6 additions and subtractions, and 6 shifts. We can choose $p = 2^{32} - 5$ and $m = 1$ as a parameter for GF(p^m). About our textbook multiplications, they require 34.25 XORs and 32 ANDs for each multiplication (actually, 1,096 XORs and 1,024 ANDs for 32 multiplications because of using the bitslicing methods). Also, our additions over GF(2^{32}) takes 1 XOR. Therefore, if we use OEF $GF(2^{32} - 5)$ to QUAD($2^{32} - 5, 32, 32$), we can accelerate roughly to 9.3 times faster than QUAD($2^{32}, 32, 32$) without memory loading. Also, QUAD($2^{32} - 5, 64, 64$) is 3.1 times faster than QUAD($2^{32}, 64, 64$).

Reducing Registers in Kernels

On the other standing, we should reduce the registers in kernels. In CUDA version 3.5, each SM can have 64 warps (i.e. 2,048 threads). However, this number is also limited by the number of registers. Each SM has only 65,536 registers. If each block in a kernel has over 65,536 registers, the kernel does not work. In contrast, if a sum of registers in several blocks is less than 65,536 registers, SMs can handle such blocks simultaneously. SMs switch different warps per cycle, and handle at most 2 warps in a cycle. This switching makes hiding latency of memory loading. To achieve maximal efficiency, we should use at most 32 registers in a thread.

Especially, we should improve our multiplications over GF(2^{32}). Because, our multiplication kernels have 135 threads in a kernel. Then, each SM has only 19 If we improve the occupancy to 100 %, we can accelerate 2.3 times faster for QUAD($2^{32}, 32, 32$) and 5.7 times faster for QUAD($2^{32}, 64, 64$).

Appendix A

Source Code

A.1 Evaluating Polynomial over GF(2)

A.1.1 Main Function of Evaluation

This code shows the entire of evaluating a multivariate quadratic system.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cutil.h>

/*
    Definitions
    numUnk        number of unknowns
    numUnkP1    (numUnk + 1)
    numUnkP2    (numUnk + 2)
    numPol        number of polynomials
    numPol_32    bitsliced polynomials
*/
#define numUnk 320
#define numUnkP1 321
#define numUnkP2 322
#define numPol 640
#define numPol_32 20 /* POLYNOMIALS >> 5 */

#define SquredUNK 103684
```

```
#define Sum1ToUNK 51360
#define LOOP 131072

/* Main Function */
int main(int argc, char** argv) {
    /* Initialization */
    srand((unsigned)time(NULL));

    int i, j, k, l;

    /* CPU Variables */
    int ***Coeffs;                /* Coefficients */
    int *x, *xx;                /* Unknowns */
    int *S, *Sout;                /* Values */

    /* GPU Variables */
    int *GA;                /* Coefficients */
    int *GSout, *GS;
    int *GT1, *GT2, *GT3, *GT4, *GT5;    /* Temporary Datas */
    int *GAxx, *Gxx;

    int k30;

    /* GPU start up */
    CUT_DEVICE_INIT(argc, argv);
    if ((fp = fopen("plaintext.dat", "r")) == NULL) {
        CUT_EXIT(argc, argv);
        exit(1);
    }

    /* Ensure CPU variables */
    Coeffs = (int***)malloc(numPol_32*sizeof(int**));
    for (i = 0; i < numPol_32; i++) {
        Coeffs[i] = (int**)malloc(numUnkP2*sizeof(int*));
        for (j = 0; j < numUnkP2; j++) {
            Coeffs[i][j] = (int*)malloc(numUnkP2*sizeof(int));
        }
    }
    x = (int*)malloc(numUnkP2 * sizeof(int));
    xx = (int*)malloc(352 * sizeof(int));
    S = (int*)malloc(30*numPol_32*sizeof(int));
    Sout = (int*)malloc(VARPerInt*sizeof(int));

    /* Set coeffcients */
    for (k = 0; k < numPol_32; k++) {
        for (j = 0; j < numUnkP2; j++) {
            for (i = 0; i < numUnkP2; i++) {
                A[k][j][i] = 0;
                if (j > i) { continue; }
                if (i == numUnkP1) { continue; }

                for (l = 0; l < sizeof(int) * 8; l++) {
                    A[k][j][i] = A[k][j][i] << 1 | (rand() % 2);
                }
            }
        }
    }

    /* Set IV */
    for (i = 0; i < numUnk; i++) {
        x[i] = rand() % 2;
        printf("%d ", x[i]);
    }
    x[numUnk] = 0x01;
    x[numUnkP1] = 0x00;

    /* Ensure GPU variables */
    cudaMalloc((void**)&GA, numPol_32 * numUnkP2 * numUnkP2 * sizeof(int
));
    cudaMalloc((void**)&Gxx, 352 * sizeof(int));
    cudaMalloc((void**)&GAxx, 180 * 352 * sizeof(int));
    cudaMalloc((void**)&GT1, 352 * 32 * numPol_32 * sizeof(int));
    cudaMalloc((void**)&GT2, 480 * numPol_32 * sizeof(int));
    cudaMalloc((void**)&GT3, 120 * numPol_32 * sizeof(int));
    cudaMalloc((void**)&GT4, 30 * numPol_32 * sizeof(int));
    cudaMalloc((void**)&GT5, 5 * numPol_32 * sizeof(int));
    cudaMalloc((void**)&GS, numPol_32*sizeof(int));
    cudaMalloc((void**)&GSout, VARPerInt*sizeof(int));

    /* Transfer coefficients */
    for (j = 0; j < numPol_32; j++) {
        for (i = 0; i < numUnk; i++) {
            int* tmp = Coeffs[j][i];
            cudaMemcpy((int*)GA + numUnkP2 * numUnkP2 * j + numUnkP2 *
i, tmp, numUnkP2 * sizeof(int), cudaMemcpyHostToDevice);
        }
    }

    /* Dimensions */
    dim3 Grid15x1, Grid15x2, Grid15x4;
    dim3 Block32xPIntx1, Block32x6x3, Block32x5x6, Block4x8x5;

    Grid15x1=dim3(15,1,1); Grid15x2=dim3(15,2,1);
    Grid15x4=dim3(15,4,1);
    Block32xPIntx1=dim3(32,numPol_32,1); Block4x8x5=dim3(4,8,5);
    Block32x6x3=dim3(32,6,3); Block32x5x6=dim3(32,5,6);

    for (k = 0; k < LOOP; k++) {
        /* Cout non-zero unknowns */
        k30 = checkXX(x, xx);

        /* Transfer unknonws */
        cudaMemcpy(Gxx, xx, 352 * sizeof(int), cudaMemcpyHostToDevice);

        switch (k30) {
        case 6:
            /* Set a trinanguller matrix */
            SetArray06<<<Grid15x2, Block32x6x3>>>(Gxx,GAxx);

            /* Compute summations of submatrix rows */
            ComputeSRow06<<<Grid15x4, Block32x5x6>>>(GAxx, GT1, GA);
            /* Compute summations of submatrix culmns */
            ComputeSCul06<<<Grid15x1, Block32xPintx1>>>(GT1, GT2);

            break;
        case 7: case 5: case 8: case 4: case 9:
        case 3: case 10: case 2: case 11: case 1:
        }

        /* Compute 15x32 matrix */
        ComputeSRow_1<<<Grid15x1, Block4x8x5>>>(GT2, GT3);
        ComputeSRow_2<<<30, 20>>>(GT3, GT4);
        ComputeSCul_1<<<5, 20>>>(GT4, GT5);
        ComputeSCul_2<<<1, 20>>>(GT5, GS);

        /* Transfer result */
        cudaMemcpy(S, GS, numPol_32 * sizeof(int), cudaMemcpyDeviceToHos
t);

        /* Output keystream */
        outS(S, Sout);
        /* Update next unknowns */
        upX(S, x);

        for (j = 0; j < VARPerInt; j++) {
            s[k+j] ^= Sout[j];
            //printf("%d %d ", S[2*j], S[2*j + 1]);
        }
    }

    /* Release GPU variables*/
    cudaFree(GA);
    cudaFree(Gxx);    cudaFree(GAxx);
    cudaFree(GT1);    cudaFree(GT2);    cudaFree(GT3);
    cudaFree(GT4);    cudaFree(GT5);
    cudaFree(GS);    cudaFree(GSout);

    /* Release CPU variables */
    for (i = 0; i < numPol_32; i++) {
        for (j = 0; j < numUnkP2; j++)
            free(Coeffs[i][j]);

        free(Coeffs[i]);
    }
    free(Coeffs);
    free(x);    free(xx);
    free(S);    free(Sout);

    /* Halt */
    CUT_EXIT(argc, argv);

    return 0;
```

```
}
```

A.1.2 Parallelization for a Summation of Polynomial

Counting Non-zero Unknown

This kernel counts non-zero variables in unknowns.

```
__host__ int checkXX(int *x, int *xx) {
    int i, j, k, l;

    j = 0;    k = 0;    l = 0;
    for (i = 0; i < numUnkP2; i++) {
        if (x[i]) {
            xx[k] = i;
            k++;
            l++;

            if (l == 0x1f) {
                l = 1;
                j++;
            }
        }
    }

    for (l = k; l < numUnkP2; l++) {
        xx[l] = numUnkP1;
    }

    return j+1;
}
```

Setting Summations of Polynomial

This kernel sets a triangular matrix from non-zero monomials. Here is only described a case 151 – 180 non-zero variables.

```
__global__ void SetArray6(int *xx, int *Axx) {
    __shared__ int Pxx1[192];
    int i = blockIdx.x, j = (threadIdx.z << 1) + blockIdx.y,
        k = threadIdx.x, l = threadIdx.y, m, o, p;

    m = ((j << 4) - j) + i;        //block 15j + i
    o = (l << 5) + k;             //thread 32l + k
    p = umul24(m, 192) + o;    //(32k)m + o, xx_mo

    Pxx1[o] = xx[o];
    __syncthreads();

    if (m < o) {
        Axx[p] = umul24(Pxx1[m], numUnkP2) + Pxx1[o-1];
    } else {
        Axx[p] = umul24(Pxx1[179-m], numUnkP2) + Pxx1[191-o];
    }
}
```

Computing summation of matrix row

This kernel computes a summation of row elements in a matrix. Here is only described a case 151 – 180 non-zero variables.

```
__global__ void ComputeSRow6(int *Axx, int *T1, int *A) {
    int j, l, m, p, q, k1, k2, k3, k4, k5, k6;
    int *ptr1, *ptr2;

    ptr1 = Axx;
    ptr2 = A;

    q = (threadIdx.y << 2) + blockIdx.y;
    ptr2 += umul24(SquredUNK, q);
    j = umul24(2880, q);
    m = (threadIdx.z << 4) - threadIdx.z + blockIdx.x;

    l = (m << 5) + threadIdx.x;

    ptr1 += umul24(l, 6);
    p = j + l;
    k1 = *ptr1++;     k2 = *ptr1++;     k3 = *ptr1++;
    k4 = *ptr1++;     k5 = *ptr1++;     k6 = *ptr1;

    T1[p] = *(ptr2+k1) ^ *(ptr2+k2) ^ *(ptr2+k3) ^
*(ptr2+k4) ^ *(ptr2+k5) ^ *(ptr2+k6);
}
```

Computing summations of matrix column

This kernel computes a summation of column elements in a matrix. Here is only described a case 151 – 180 non-zero variables.

```
__global__ void ComputeSCul06(int *T1, int *T2) {
    int j, m, p, k1, k2, k3, k4, k5, k6;
    int *ptr;

    ptr = T1;

    j = umul24(threadIdx.y, 480);

    m = (blockIdx.x << 5) + threadIdx.x;

    p = j + m;
    ptr += umul24(p, 6);

    k1 = *ptr++;    k2 = *ptr++;    k3 = *ptr++;
    k4 = *ptr++;    k5 = *ptr++;    k6 = *ptr;

    T2[p] = k1 ^ k2 ^ k3 ^ k4 ^ k5 ^ k6;
}
```

A.2 Evaluating Polynomials for GF(2^{32})

A.2.1 Header File of Program Files

This configulation file "condQUAD.h" shows constructions of QUAD and threads, blocks and grid of its kernels.

```
#define numUnk          64
#define numUnk_2        62
#define numPol_Half     64
#define numPol          128

/* nQT = comb(n+2, 2) = (n+2)(n+1)/2 */
#define numQuadTerm     1953
#define numLinearBase   1984
#define numPolyTerm     2016
#define widPolyTerm     2048
#define widPolyTermx32 65536
#define numSystemTerm   131072
#define lengInitVector 120

/* MB/sizeof(int) */
#define lengData       2621440
#define BytePerCycle   256
#define IntPerCycle    64

#define iterateTime    40960
#define lengStream     2621440

typedef    unsigned int    element;
```

A.2.2 Main Frame of QUAD

This program shows main function of QUAD encryption test and own timer function.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#include "condQUAD.h"

extern "C" int encryptQUAD(element *QuadSystem[3], element *, element *,
element *, element *, double *);

/* Print function */
void printStream(element *, int);
```

```
/* Time get function */
extern "C" double gettimeofday_sec();

int main(int argc, char *argv[]) {
    /* CPU variables */
    element *QuadSys[3];    /* P&Q, S0, S1 */
    element OriKey[numUnk * 32];
    element *Key, *InitVec;
    element *plaintext, *ciphertext;

    /* Timer variable */
    double timeEncrypt[7];
    double timeBase;
    double timeStart, timeEnd;
    double timeMalloc, timeFree;
    double timeGenerate;
    double timeEnc;

    /* counter */
    int i, j, k;
    int idx;

    timeStart = gettimeofday_sec();

    srand((unsigned int)timeStart);

    /* Memory allocation */
    timeBase = gettimeofday_sec();
    QuadSys[0] = (element *)malloc(2*numSystemTerm * sizeof(element));
    QuadSys[1] = (element *)malloc(numSystemTerm * sizeof(element));
    QuadSys[2] = (element *)malloc(numSystemTerm * sizeof(element));

    Key = (element *)malloc(widPolyTermx32 * sizeof(element));
    InitVec = (element *)malloc(lengInitVector * sizeof(element));

    plaintext = (element *)malloc(lengData * sizeof(element));
    ciphertext = (element *)malloc(lengData * sizeof(element));
    timeMalloc = gettimeofday_sec() - timeBase;

    fprintf(stderr, "CPU memory allocated.\n");
    timeBase = gettimeofday_sec();

    /* Create/load QUAD data */
    idx = 0;
    for (i = 0; i < 2; ++i) {
        for (j = 0; j < 32; ++j) {
            for (k = 0; k < numPolyTerm; ++k) {
                idx = (i << 16) | (j << 11) | k;
                QuadSys[0][idx] = rand();
                QuadSys[0][idx+numSystemTerm] = rand();
                QuadSys[1][idx] = rand();
                QuadSys[2][idx] = rand();
            }
            for (k = numPolyTerm; k < widPolyTerm; ++k) {
                idx = (i << 16) | (j << 11) | k;
                QuadSys[0][idx] = 0;
                QuadSys[0][idx+numSystemTerm] = 0;
                QuadSys[1][idx] = 0;
                QuadSys[2][idx] = 0;
            }
        }
    }
    fprintf(stderr, "Polynomial set.\n");

    /* Create/load key and Initialization vector */
    for (i = 0; i < numUnk * 32; ++i) {
        OriKey[i] = ((rand() >> 3) & 0x01) * 0xFFFFFFFF;
    }
    for (i = 0; i < widPolyTermx32; ++i) { Key[i] = 0; }
    for (i = 0; i < 32; ++i) {
        idx = i << 11;

        for (j = 0; j < 2; ++j) {
            for (k = 0; k < 31; ++k) {
                Key[idx + (j * 31) + k + numLinearBase] = (OriKey[(i <<
6) | (j << 5) | k] << (k+1)) | (OriKey[(i << 6) | (j << 5) | (k+1)] >>
(31 - k));
            }
        }
    }
    Key[2047] = 0xFFFFFFFF;

    fprintf(stderr, "Key set.\n");

    for (i = 0; i < lengInitVector; ++i) {
        InitVec[i] = (rand() >> 7) & 0x01;
    }

    /* Generate/load stream data */
    for (i = 0; i < lengData; ++i) {
        plaintext[i] = (element)rand();
    }
    timeGenerate = gettimeofday_sec() - timeBase;

    //printStream(plaintext, 2000);

    /* Encryption */
    timeBase = gettimeofday_sec();
    encryptQUAD(QuadSys, Key, InitVec, plaintext, ciphertext, timeEncryp
t);
    timeEnc = gettimeofday_sec() - timeBase;

    //printStream(ciphertext, 2000);

    /* Memory release */
    timeBase = gettimeofday_sec();
    free(QuadSys[0]);
    free(QuadSys[1]);
    free(QuadSys[2]);

    free(Key);
    free(InitVec);

    free(plaintext);
    free(ciphertext);
    timeFree = gettimeofday_sec() - timeBase;
    timeEnd = gettimeofday_sec() - timeStart;

    printf("Running time\n  All: %f[sec]\n", timeEnd);
    printf("1.Memory allocation: %f[sec]\n", timeMalloc);
    printf("2.Generate Data: %f[sec]\n", timeGenerate);
    printf("3.Encrypting time: %f[sec]\n", timeEnc);
    printf("3.1.GPU memory allocation: %f[sec]\n", timeEncrypt[0]);
    printf("3.2.Data CPU => GPU: %f[sec]\n", timeEncrypt[1]);
    printf("3.3.Vector Initialize: %f[sec]\n", timeEncrypt[2]);
    printf("3.4.Generate PRN: %f[sec]\n", timeEncrypt[3]);
    printf("3.5.Encrypt message: %f[sec]\n", timeEncrypt[4]);
    printf("3.6.Data GPU => CPU: %f[sec]\n", timeEncrypt[5]);
    printf("3.7.GPU memory release: %f[sec]\n", timeEncrypt[6]);
    printf("4.Memory release: %f[sec]\n", timeFree);

    return 0;
}

void printStream(element *stream, int width) {
    int i;

    for (i = 0; i < width; ++i) {
        printf("%x ", stream[i]);
    }
    putchar('\n');
}

extern "C" double gettimeofday_sec() {
    struct timeval t;

    gettimeofday(&t, NULL);

    return (double)t.tv_sec + (double)t.tv_usec * 1e-6;
}
```

A.2.3 Encrypting Functions of QUAD

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <cuda.h>

#include "condQUAD.h"

extern __global__ void    setX(element *, element *);
extern __global__ void    multiply(element *, element *);

extern __global__ void prodMatVec(element *, element *, element *);
extern __global__ void prodMatVecP(element *, element *);
extern __global__ void prodMatVecPQ(element *, element *, element *, ele
ment *);
extern __global__ void outputStream(element *, element *, element *);
extern __global__ void setNextVec(element *, element *);

extern __global__ void initStreamCounter(element *);

/* Encrypt message */
extern __global__ void computeXOR(element *, element *, element *);

extern "C" double gettimeofday_sec();

extern "C" int encryptQUAD(
    element *QuadSystem[3], /* 0: P&Q, 1: S0, 2: S1 */
    element *key,
    element *InitVec,
    element *plaintext,
    element *ciphertext,
    double  *timeFunc
        ) {

    /* GPU variables */
    element *devSysP = 0, *devSysS0 = 0, *devSysS1 = 0;
    element *devX = 0, *devY = 0, *devAXX = 0, *devVecX = 0;
    element *devPlain = 0, *devCipher = 0;
    element *devStream = 0;
    element *devCou = 0;

    /* Timer variables */
    double  timeBase;

    /* Dimension variable */
    dim3    gridSetX, blockSetX;
    dim3    gridAXX;
    dim3    blockAXX1, blockAXX2;
    dim3    gridAdd1, blockAdd;
    dim3    gridAdd2;
```

```
/* counter */
int i;

gridSetX = dim3(numUnk_2, 1, 1);
blockSetX = dim3(numUnk_2, 32, 1);
gridAXX = dim3(32, 1, 1);
blockAXX1 = dim3(64, 2, 1);
blockAXX2 = dim3(64, 4, 1);
gridAdd1 = dim3(2, 32, 1);
blockAdd = dim3(1024, 1, 1);
gridAdd2 = dim3(4, 32, 1);

/* -1: GPU initialize */
/* GPU initialize */
if (cuInit(0) != CUDA_SUCCESS) {
    fprintf(stderr, "GPU cannot be opened.\n");

    return -1;
}

/* Shared/L1 configulation change */
cudaDeviceSetCacheConfig(cudaFuncCachePreferL1);

/* 0: GPU memory allocation */
timeBase = gettimeofday_sec();
cudaMalloc((void **)&devSysP, 2 * numSystemTerm * sizeof(element));
cudaMalloc((void **)&devSysS0, numSystemTerm * sizeof(element));
cudaMalloc((void **)&devSysS1, numSystemTerm * sizeof(element));

cudaMalloc((void **)&devX, widPolyTermx32 * sizeof(element));
cudaMalloc((void **)&devY, widPolyTermx32 * sizeof(element));
cudaMalloc((void **)&devAXX, 2 * numSystemTerm * sizeof(element));
cudaMalloc((void **)&devVecX, 640 * sizeof(element));

cudaMalloc((void **)&devPlain, lengData * sizeof(element));
cudaMalloc((void **)&devCipher, lengData * sizeof(element));
cudaMalloc((void **)&devStream, lengStream * sizeof(element));
cudaMalloc((void **)&devCou, 64 * sizeof(element));
timeFunc[0] = gettimeofday_sec() - timeBase;
```

```
    /* 1: Data transfer HtoD */
    timeBase = gettimeofday_sec();
    cudaMemcpy(devSysP, QuadSystem[0], 2 * numSystemTerm * sizeof(elemen
t), cudaMemcpyHostToDevice);
    cudaMemcpy(devSysS0, QuadSystem[1], numSystemTerm * sizeof(element),
 cudaMemcpyHostToDevice);
    cudaMemcpy(devSysS1, QuadSystem[2], numSystemTerm * sizeof(element),
 cudaMemcpyHostToDevice);

    cudaMemcpy(devX, key, widPolyTermx32 * sizeof(element), cudaMemcpyHo
stToDevice);
    cudaMemcpy(devPlain, plaintext, lengData * sizeof(element), cudaMemc
pyHostToDevice);
    initStreamCounter<<<1, 64>>>(devCou);

    timeFunc[1] = gettimeofday_sec() - timeBase;

    /* 2: Initialize */
    timeBase = gettimeofday_sec();
    for (i = 0; i < lengInitVector; ++i) {
        /* Precomputation x => x'(={x_(1,1), x_(1_2), x_(2_2), ...} */
        /* 1984 = 62 * 32 */
        setX<<<gridSetX, blockSetX>>>(devX, devY);
        multiply<<<31, 64>>>(devX, devY);

        /* nextX <= P(x) */
        if (InitVec[i] == 0) {
            prodMatVec<<<gridAXX, blockAXX1>>>(devX, devSysS0, devAXX);
        } else {
            prodMatVec<<<gridAXX, blockAXX1>>>(devX, devSysS1, devAXX);
        }
        prodMatVecP<<<gridAdd1, blockAdd>>>(devAXX, devVecX);
        setNextVec<<<64, 31>>>(devVecX, devX);
    }
    cudaThreadSynchronize();
    for (i = 0; i < lengInitVector; ++i) {
        /* Precomputation x => x'(={x_(1,1), x_(1_2), x_(2_2), ...} */
        setX<<<gridSetX, blockSetX>>>(devX, devY);
        multiply<<<31, 64>>>(devX, devY);
        prodMatVec<<<gridAXX, blockAXX1>>>(devX, devSysP, devAXX);
        prodMatVecP<<<gridAdd1, blockAdd>>>(devAXX, devVecX);
        setNextVec<<<64, 31>>>(devVecX, devX);
    }
    cudaThreadSynchronize();
    timeFunc[2] = gettimeofday_sec() - timeBase;

    fprintf(stderr, "Initialized\n");

    /* 3: Generate sequence */
    timeBase = gettimeofday_sec();
    for (i = 0; i < iterateTime; ++i) {
        /* Precomputation x => x'(={x_(1,1), x_(1_2), x_(2_2), ...} */
        setX<<<gridSetX, blockSetX>>>(devX, devY);
        multiply<<<31, 64>>>(devX, devY);

        /* y = Ax' */
        /* stream <= Q(x) */
        prodMatVec<<<gridAXX, blockAXX2>>>(devX, devSysP, devAXX);

        /* nextX <= P(x) */
        prodMatVecP<<<gridAdd2, blockAdd>>>(devAXX, devVecX);
        outputStream<<<2, 32>>>(devVecX, devStream, devCou);
        setNextVec<<<64, 31>>>(devVecX, devX);
    }
    cudaThreadSynchronize();
    timeFunc[3] = gettimeofday_sec() - timeBase;

    fprintf(stderr, "generated.\n");

    /* 4: Encrypt with stream */
    timeBase = gettimeofday_sec();
    computeXOR<<<2560, 1024>>>(devPlain, devCipher, devStream);
    cudaThreadSynchronize();
    timeFunc[4] = gettimeofday_sec() - timeBase;

    fprintf(stderr, "encrypted.\n");

    /* 5: Data transfer DtoH */
    timeBase = gettimeofday_sec();
    cudaMemcpy(ciphertext, devCipher, lengData * sizeof(int), cudaMe
mcpyDeviceToHost);
    timeFunc[5] = gettimeofday_sec() - timeBase;

    fprintf(stderr, "translated.\n");

    /* 6: GPU memory release */
    timeBase = gettimeofday_sec();
    if (devSysP != 0) { cudaFree(devSysP); }
    if (devSysS0 != 0) { cudaFree(devSysS0); }
    if (devSysS1 != 0) { cudaFree(devSysS1); }

    if (devX != 0) { cudaFree(devX); }
    if (devY != 0) { cudaFree(devY); }
    if (devAXX != 0) { cudaFree(devAXX); }
    if (devVecX != 0) { cudaFree(devVecX); }

    if (devPlain != 0) { cudaFree(devPlain); }
    if (devCipher != 0) { cudaFree(devCipher); }

    if (devStream != 0) { cudaFree(devStream); }
    if (devCou != 0) { cudaFree(devCou); }
    timeFunc[6] = gettimeofday_sec() - timeBase;

    return 0;
}
```

A.2.4 GPU Kernels of QUAD over GF(2^{32})

```
#include <cuda.h>

#include "condQUAD.h"

__global__ void setX(element *X, element *Y) {
    //int laneId = threadIdx.x & 0x1f;
    int idx_bl, idx_th, idx_base, idx;
    int value;

    idx_bl = blockIdx.x;
    idx_th = threadIdx.x;
    idx_base = threadIdx.y << 11;
    if (idx_th == 0) {
        value = X[(idx_base) | (idx_bl +  numLinearBase)];
    }
    value = __shfl(value, 0);

    if (idx_bl <= idx_th) { X[idx_base + ((idx_th * (idx_th + 1)) >> 1)
+ idx_bl] = value; }
    if (idx_bl >= idx_th) { Y[idx_base + ((idx_bl * (idx_bl + 1)) >> 1)
+ idx_th] = value; }
}

/* multiple 32 elementxelement */
__global__ void multiply(element *X, element *Y) {
    /* pseudo-code */
    element A00 to A1f;
    element B00 to B1f;

    /* AB00 to AB1f: data of 1st to 32nd bit of A * B */
    element AB00 to AB1f;
    /* Carry00 to Carry1e: data of 33rd to 63rd bit of A * B */
    element Carry00 to Carry 1e;

    /* CXXYY := CarryXX ^ CarryYY */
    element C0B15, C0C16, C0D17, C0E18, C0F19, C101A, C111B, C121C,
            C131D, C141E;

    /* CXXYYZZ := CXXYY ^ CarryZZ */
    element C000A14, C010B15, C020C16, C030D17, C040E18, C050F19,
            C06101A, C07111B, C08121C, C09131D, C0A141E;

    int idx_base, idx;

    idx_base = (blockIdx.x << 6) + threadIdx.x;

    idx = idx_base;

    /* for XX as 00 to 1f */
    A00 = X[idx_base];    B00 = X[idx_base];    idx += 2048;
    A01 = X[idx_base];    B01 = X[idx_base];    idx += 2048;
    ...
```

```
AXX = X[idx_base];    BXX = X[idx_base];    idx += 2048;
...
A1f = X[idx_base];    B1f = X[idx_base];

/* for XX as 00 to 1f */
AB00 = (A00 & B00);
AB01 = (A01 & B00) ^ (A00 & B01);
...
ABXX = (A00 & BXX) ^ (A01 & B(XX-01)) ^ ... ^ (AXX & B00);
...
AB1f = (A00 & B1f) ^ (A01 & B1e) ^ ... ^ (A1f & B00);

/* for ZZ as 00 to 1e */
Carry00 = (A1f & B01) ^ (A1e & B02) ^ ... ^ (A01 & B1f);
Carry01 = (A1f & B02) ^ (A1e & B03) ^ ... ^ (A02 & B1f);
...
CarryZZ = (A1f & B(ZZ-1f)) ^ ... ^ (A(ZZ-1f) & B1f);
...
Carry1e = (A1f & B1f);

C0B15 = Carry0b ^ Carry15;
C0C16 = Carry0c ^ Carry16;
C0D17 = Carry0d ^ Carry17;
C0E18 = Carry0e ^ Carry18;
C0F19 = Carry0f ^ Carry19;
C101A = Carry10 ^ Carry1a;
C111B = Carry11 ^ Carry1b;
C121C = Carry12 ^ Carry1c;
C131D = Carry1e ^ Carry1d;
C141E = Carry14 ^ Carry1e;
C000A14 = Carry00 ^ Carry0a ^ Carry14;
C010B15 = Carry01 ^ C0B15;
C020C16 = Carry02 ^ C0C16;
C030D17 = Carry03 ^ C0D17;
C040E18 = Carry04 ^ C0E18;
C050F19 = Carry05 ^ C0F19;
C06101A = Carry06 ^ C101A;
C07111B = Carry07 ^ C111B;
C08121C = Carry08 ^ C121C;
C09131D = Carry09 ^ C131D;
C0A141E = Carry0a ^ C141E;
```

```
    X[idx_base] = AB00 ^ C000A14;
    idx_base += 2048;
    X[idx_base] = AB01 ^ C000A14 ^ C010B15;
    idx_base += 2048;
    X[idx_base] = AB02 ^ C000A14 ^ C010B15 ^ C020C16;
    idx_base += 2048;
    X[idx_base] = AB03 ^ C010B15 ^ C020C16 ^ C030D17;
    idx_base += 2048;
    X[idx_base] = AB04 ^ C020C16 ^ C030D17 ^ C040E18;
    idx_base += 2048;
    X[idx_base] = AB05 ^ C030D17 ^ C040E18 ^ C050F19;
    idx_base += 2048;
    X[idx_base] = AB06 ^ C040E18 ^ C050F19 ^ C06101A;
    idx_base += 2048;
    X[idx_base] = AB07 ^ C050F19 ^ C06101A ^ C07111B;
    idx_base += 2048;

    X[idx_base] = AB08 ^ C06101A ^ C07111B ^ C08121C;
    idx_base += 2048;
    X[idx_base] = AB09 ^ C07111B ^ C08121C ^ C09131D;
    idx_base += 2048;
    X[idx_base] = AB0a ^ C08121C ^ C09131D ^ C0A141E;
    idx_base += 2048;
    X[idx_base] = AB0b ^ C09131D ^ C0A141E ^ C0B15;
    idx_base += 2048;
    X[idx_base] = AB0c ^ C0A141E ^ C0B15 ^ C0C16;
    idx_base += 2048;
    X[idx_base] = AB0d ^ C0B15 ^ C0C16 ^ C0D17;
    idx_base += 2048;
    X[idx_base] = AB0e ^ C0C16 ^ C0D17 ^ C0E18;
    idx_base += 2048;
    X[idx_base] = AB0f ^ C0D17 ^ C0E18 ^ C0F19;
    idx_base += 2048;

    X[idx_base] = AB10 ^ C0E18 ^ C0F19 ^ C101A;
    idx_base += 2048;
    X[idx_base] = AB11 ^ C0F19 ^ C101A ^ C111B;
    idx_base += 2048;
    X[idx_base] = AB12 ^ C101A ^ C111B ^ C121C;
    idx_base += 2048;
    X[idx_base] = AB13 ^ C111B ^ C121C ^ C131D;
    idx_base += 2048;
    X[idx_base] = AB14 ^ C121C ^ C131D ^ C141E;
    idx_base += 2048;
    X[idx_base] = AB15 ^ C131D ^ C141E ^ Carry15;
    idx_base += 2048;
    X[idx_base] = AB16 ^ Carry00 ^ Carry0a ^ Carry15 ^ Carry16 ^ Carry1e
;
    idx_base += 2048;
    X[idx_base] = AB17 ^ Carry01 ^ Carry0b ^ Carry16 ^ Carry17;
    idx_base += 2048;

    X[idx_base] = AB18 ^ Carry02 ^ Carry0c ^ Carry17 ^ Carry18;
    idx_base += 2048;
    X[idx_base] = AB19 ^ Carry03 ^ Carry0d ^ Carry18 ^ Carry19;
    idx_base += 2048;
    X[idx_base] = AB1a ^ Carry04 ^ Carry0e ^ Carry19 ^ Carry1a;
    idx_base += 2048;
    X[idx_base] = AB1b ^ Carry05 ^ Carry0f ^ Carry1a ^ Carry1b;
    idx_base += 2048;
    X[idx_base] = AB1c ^ Carry06 ^ Carry10 ^ Carry1b ^ Carry1c;
    idx_base += 2048;
    X[idx_base] = AB1d ^ Carry07 ^ Carry11 ^ Carry1c ^ Carry1d;
    idx_base += 2048;
    X[idx_base] = AB1e ^ Carry08 ^ Carry12 ^ Carry1d ^ Carry1e;
    idx_base += 2048;
    X[idx_base] = AB1f ^ Carry09 ^ Carry13 ^ Carry1e;
}


/*
    20 * 641, 321
*/
__global__ void prodMatVec(element *extX, element *A, element *AXX) {
    /* pseudo-code */
    element A00 to A1f;
    element B00 to B1f;

    /* AB00 to AB1f: data of 1st to 32nd bit of A * B */
    element AB00 to AB1f;
    /* Carry00 to Carry1e: data of 33rd to 63rd bit of A * B */
```

```
element Carry00 to Carry 1e;

/* CXXYY := CarryXX ^ CarryYY */
element C0B15, C0C16, C0D17, C0E18, C0F19, C101A, C111B, C121C,
        C131D, C141E;

/* CXXYYZZ := CXXYY ^ CarryZZ */
element C000A14, C010B15, C020C16, C030D17, C040E18, C050F19,
        C06101A, C07111B, C08121C, C09131D, C0A141E;

int idx_base, idx;

idx_base = (blockIdx.x << 6) | threadIdx.x;

idx = idx_base;

/* for XX as 00 to 1f */
A00 = extX[idx];    idx += 2048;
A01 = extX[idx];    idx += 2048;
...
AXX = extX[idx];    idx += 2048;
...
A1f = extX[idx];

idx_base |= threadIdx.y << 16;
idx = idx_base;

/* for XX as 00 to 1f */
B00 = A[idx];    idx += 2048;
B01 = A[idx];    idx += 2048;
...
BXX = A[idx];    idx += 2048;
...
A1f = A[idx];


/* for XX as 00 to 1f */
AB00 = (A00 & B00);
AB01 = (A01 & B00) ^ (A00 & B01);
...
ABXX = (A00 & BXX) ^ (A01 & B(XX-01)) ^ ... ^ (AXX & B00);
```

```
...
AB1f = (A00 & B1f) ^ (A01 & B1e) ^ ... ^ (A1f & B00);

/* for XX as 00 to 1e */
Carry00 = (A1f & B01) ^ (A1e & B02) ^ ... ^ (A01 & B1f);
Carry01 = (A1f & B02) ^ (A1e & B03) ^ ... ^ (A02 & B1f);
...
CarryXX = (A1f & B(XX-1f)) ^ ... ^ (A(XX-1f) & B1f);
...
Carry1e = (A1f & B1f);

C0B15 = Carry0b ^ Carry15;
C0C16 = Carry0c ^ Carry16;
C0D17 = Carry0d ^ Carry17;
C0E18 = Carry0e ^ Carry18;
C0F19 = Carry0f ^ Carry19;
C101A = Carry10 ^ Carry1a;
C111B = Carry11 ^ Carry1b;
C121C = Carry12 ^ Carry1c;
C131D = Carry1e ^ Carry1d;
C141E = Carry14 ^ Carry1e;
C000A14 = Carry00 ^ Carry0a ^ Carry14;
C010B15 = Carry01 ^ C0B15;
C020C16 = Carry02 ^ C0C16;
C030D17 = Carry03 ^ C0D17;
C040E18 = Carry04 ^ C0E18;
C050F19 = Carry05 ^ C0F19;
C06101A = Carry06 ^ C101A;
C07111B = Carry07 ^ C111B;
C08121C = Carry08 ^ C121C;
C09131D = Carry09 ^ C131D;
C0A141E = Carry0a ^ C141E;

idx = idx_base;
AXX[idx] = AB00 ^ C000A14;
idx += 2048;
AXX[idx] = AB01 ^ C000A14 ^ C010B15;
idx += 2048;
AXX[idx] = AB02 ^ C000A14 ^ C010B15 ^ C020C16;
idx += 2048;
AXX[idx] = AB03 ^ C010B15 ^ C020C16 ^ C030D17;
```

```
    idx += 2048;
    AXX[idx] = AB04 ^ C020C16 ^ C030D17 ^ C040E18;
    idx += 2048;
    AXX[idx] = AB05 ^ C030D17 ^ C040E18 ^ C050F19;
    idx += 2048;
    AXX[idx] = AB06 ^ C040E18 ^ C050F19 ^ C06101A;
    idx += 2048;
    AXX[idx] = AB07 ^ C050F19 ^ C06101A ^ C07111B;
    idx += 2048;

    AXX[idx] = AB08 ^ C06101A ^ C07111B ^ C08121C;
    idx += 2048;
    AXX[idx] = AB09 ^ C07111B ^ C08121C ^ C09131D;
    idx += 2048;
    AXX[idx] = AB0a ^ C08121C ^ C09131D ^ C0A141E;
    idx += 2048;
    AXX[idx] = AB0b ^ C09131D ^ C0A141E ^ C0B15;
    idx += 2048;
    AXX[idx] = AB0c ^ C0A141E ^ C0B15 ^ C0C16;
    idx += 2048;
    AXX[idx] = AB0d ^ C0B15 ^ C0C16 ^ C0D17;
    idx += 2048;
    AXX[idx] = AB0e ^ C0C16 ^ C0D17 ^ C0E18;
    idx += 2048;
    AXX[idx] = AB0f ^ C0D17 ^ C0E18 ^ C0F19;
    idx += 2048;

    AXX[idx] = AB10 ^ C0E18 ^ C0F19 ^ C101A;
    idx += 2048;
    AXX[idx] = AB11 ^ C0F19 ^ C101A ^ C111B;
    idx += 2048;
    AXX[idx] = AB12 ^ C101A ^ C111B ^ C121C;
    idx += 2048;
    AXX[idx] = AB13 ^ C111B ^ C121C ^ C131D;
    idx += 2048;
    AXX[idx] = AB14 ^ C121C ^ C131D ^ C141E;
    idx += 2048;
    AXX[idx] = AB15 ^ C131D ^ C141E ^ Carry15;
    idx += 2048;
    AXX[idx] = AB16 ^ Carry00 ^ Carry0a ^ Carry15 ^ Carry16 ^ Carry1e;
    idx += 2048;
    AXX[idx] = AB17 ^ Carry01 ^ Carry0b ^ Carry16 ^ Carry17;
    idx += 2048;

    AXX[idx] = AB18 ^ Carry02 ^ Carry0c ^ Carry17 ^ Carry18;
    idx += 2048;
    AXX[idx] = AB19 ^ Carry03 ^ Carry0d ^ Carry18 ^ Carry19;
    idx += 2048;
    AXX[idx] = AB1a ^ Carry04 ^ Carry0e ^ Carry19 ^ Carry1a;
    idx += 2048;
    AXX[idx] = AB1b ^ Carry05 ^ Carry0f ^ Carry1a ^ Carry1b;
    idx += 2048;
    AXX[idx] = AB1c ^ Carry06 ^ Carry10 ^ Carry1b ^ Carry1c;
    idx += 2048;
    AXX[idx] = AB1d ^ Carry07 ^ Carry11 ^ Carry1c ^ Carry1d;
    idx += 2048;
    AXX[idx] = AB1e ^ Carry08 ^ Carry12 ^ Carry1d ^ Carry1e;
    idx += 2048;
    AXX[idx] = AB1f ^ Carry09 ^ Carry13 ^ Carry1e;
}

__device__ void warpReduce(volatile element *sdata, int tid) {
    sdata[tid] ^= sdata[tid+32];
    sdata[tid] ^= sdata[tid+16];
    sdata[tid] ^= sdata[tid+ 8];
    sdata[tid] ^= sdata[tid+ 4];
    sdata[tid] ^= sdata[tid+ 2];
    sdata[tid] ^= sdata[tid+ 1];
}

/* m(=2) * 32 * 1024 */
__global__ void prodMatVecP(element *AXX, element *NextAXX) {
    __shared__ element tmpData[1024];
    int idxBase;
    int tid;

    tid = threadIdx.x;
    idxBase = (blockIdx.x << 16) | (blockIdx.y << 11) | threadIdx.x;
    tmpData[tid] = AXX[idxBase] ^ AXX[idxBase | 1024];
    __syncthreads();

    /* 2145 = 2 * 1024 + 97 */
    if (tid < 512) { tmpData[tid] ^= tmpData[512 | tid]; }
    __syncthreads();
    if (tid < 256) { tmpData[tid] ^= tmpData[256 | tid]; }
    __syncthreads();
    if (tid < 128) { tmpData[tid] ^= tmpData[128 | tid]; }
    __syncthreads();
    if (tid < 64) { tmpData[tid] ^= tmpData[64 | tid]; }
    __syncthreads();

    if (tid < 32) { warpReduce(tmpData, tid); }

    if (tid == 0) {
        idxBase = (blockIdx.x << 5) | blockIdx.y;
        NextAXX[idxBase] = tmpData[0];
    }
}


/* P(x) => X */
__global__ void setNextVec(
    element *X,
    element *nextX)
{
    int value;
    element v1, v2;
    int idx, idx1, idx2;

    idx = threadIdx.x;
    idx1 = ((blockIdx.x & 0x01) << 5) | idx;
    idx2 = (blockIdx.x & 0x3e) << 10;

    if (idx == 0) {    value = X[blockIdx.x]; }

    value = __shfl(value, 0);

    v1 = ((value >> idx) & 0x01) * 0xFFFFFFFE;
    v2 = ((value >> (idx + 1)) & 0x01) * 0x7FFFFFFF;

    nextX[idx2 | (numQuadTerm + idx2)] =
        (v1 << idx) | (v2 >> (30 - idx));
}
```

```
/*
    Output Stream
    resFx: result of f(x)
    stream: keystream data
    cou: index counter of keystream
*/
__global__ void outputStream(
    element *resFx,
    element *stream,
    element *cou)
{
    int idx = (blockIdx.x << 5) | threadIdx.x;

    stream[cou[idx]] = resFx[64 | idx];
    cou[idx] += 64;
}

/*
    Initialize Counter of Keystream
    cou: index counter of keystream
*/
__global__ void initStreamCounter(
    element *cou)
{
    int idx;

    idx = threadIdx.x;
    cou[idx] = idx;
}

/*
    Encrypting Message
    plain: original plaintext message
    cipher: encrypted ciphertext
    stream: keystream data
*/
__global__ void computeXOR(
    element *plain,
    element *cipher,
    element *stream)
```

```
{
    int    idx;

    idx = (blockIdx.x << 10) | threadIdx.x;

    cipher[idx] = plain[idx] ^ stream[idx];
}
```

References

[1] Cuda c programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, Accessed July 2014.

[2] cusparse::cuda toolkit documentation. http://docs.nvidia.com/cuda/cusparse, Accessed August 2014.

[3] Nvidia developer zone. https://developer.nvidia.com/category/zone/cuda-zone, Accessed July 2014.

[4] Opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/: Accessed August 2014.

[5] Optimizing parallel reduction in cuda. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf: Accessed August 2014.

[6] Visual computing leadership from nvidia. http://www.nvidia.com/page/home.html: Accessed August 2014.

[7] ISO/IEC 18031. Information technology—Security techniques—Random bit generation: second edition. ISO/IEC 18031: 2011(E), International Organization for Standardization, Geneva, Switzerland, 2011.

[8] David Arditti, Côme Berbain, Oliver Billet, and Henri Gilbert. Compact fpga implementations of quad. In *Proc. of the 2nd ACM symposium on information,*

computer and communications security (ASIACCS'07), Singapore, pages 135–147. ACM, March 2007.

[9] Gwénolé Ars, Jean-Charles Faugere, Hideki Imai, Mitsuru Kawazoe, and Makoto Sugita. Comparison between xl and gröbner basis algorithms. In *Advances in Cryptology-ASIACRYPT 2004*, pages 338–353. Springer, 2004.

[10] Daniel V Bailey and Christof Paar. Optimal extension fields for fast arithmetic in public-key algorithms. In *Advances in Cryptology - CRYPTO'98*, pages 472–485. Springer, 1998.

[11] Gregory V. Bard. *Algebraic Cryptanalysis.* Springer, 2009.

[12] Magali Bardet. *Étude des systèmes algébriques surdéterminés. Applications aux codes correcteurs et à la cryptographie.* PhD thesis, Université Pierre et Marie Curie-Paris VI, 2004.

[13] Côme Berbain, Oliver Billet, and Henri Gilbert. Efficient implementations of multivariate quadratic systems. In *Proc. of the 13th International Workshop on Selected Areas in Cryptography (SAC'06), Revised Selected Papers, LNCS*, volume 4356, pages 174–187. Springer-Verlag, August 2006.

[14] Côme Berbain, Henri Gilbert, and Jacques Patarin. Quad: A pratical stream cipher with provable security. In *Proc. of the 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques Advances (EUROCRYPT 2006), St. Petersburg, Russia, LNCS*, volume 4004, pages 109–128. Springer-Verlag, May-June 2006.

[15] Lenore Blum, Manuel Blum, and Michael Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 4(2):364–384, May 1986.

[16] D. Bonenberger and M. Krone. Factorization of rsa-170. Technical report, Tech. rep., Ostfalia University of Applied Sciences, 2010.

[17] Joppe W. Bos and Deian Stefan. Performance analysis of the sha-3 candidates on exotic multi-core architectures. In *Proc. of the 12th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2010), Santa Barbara, USA, LNCS*, volume 6225, pages 279–293. Springer-Verlag, August 2010.

[18] Christophe De Cannière and Bart Preneel. Trivium specifications. Technical report, eSTREAM, the ECRYPT Stream Cipher Project, `http://www.ecrypt.eu.org/stream/p3ciphers/trivium/trivium_p3.pdf`, 2006. Accessed August 2014.

[19] Anna Inn-Tung Chen, Ming-shing Chen, Tien-Ren Chen, Jintai Ding, Eric Li-hsiang Kuo, and Frost Yu-shuang Li. Small odd prime field multivariate pkcs. 2008.

[20] M.-S. Chen, T.-R. Chen, C.-M. Cheng, C.-H. Hsiao, Niederhagen R., and B.-Y. Yan. What price a provably secure stream cipher?, 2010.

[21] Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. Solving quadratic equations with xl on parallel architectures. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 356–373. Springer, 2012.

[22] Nicolas Courtois, Alexander Kilmov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Proc. of the 19th Annual International Conference on the Theory and Applications of Cryptographic Techniques Advances (EUROCRYPT 2000), Bruges, Belgium, LNCS*, volume 1807, pages 392–407. Springer-Verlag, May 2000.

[23] VO Drelikhov, GB Marshalko, and AV Pokrovskiy. On the security of mq_drbg. *IACR Cryptology ePrint Archive*, 2011:548, 2011.

[24] James Goodman and Anantha P. Chandrakasan. An energer-efficient reconfigurable public-key cryptography processor. In *Proc. of the 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000), Worcester, Massachusetts, USA, LNCS*, volume 1965, pages 175–190. Springer-Verlag, August 2000.

[25] Jason R. Hamlet and Robert W. Brocato. Throughput optimized implementations of quad. Technical Report 118, Cryptology ePrint Archive, `http://eprint.iacr.org/2013/118`, February 2013. Accessed August 2014.

[26] Klaus Huber. Some comments on zech's logarithms. *Journal of IEEE Transactions on Information Theory*, 36(4):946–950, July 1990.

[27] Kyoki Imamura. A method for computing addition tables in gf(p^n). *Journal of IEEE Transactions on Information Theory*, 26(4):367–369, May 1980.

[28] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. Accelerating ssl with gpus. In *Proc. of the 16th Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'10), New Delhi, India*, pages 437–438. ACM, August-September 2010.

[29] G. Kedem and Y. Ishihara. Brute force attack on unix passwords with simd computer. In *Proceedings of the 8th conference on USENIX Security Symposium-Volume 8*, pages 8–8. USENIX Association, 1999.

[30] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In *Advances in Cryptology - EUROCRYPT' 99*, pages 206–222. Springer, 1999.

[31] Matthew Kwan. Reducing the gate count of bitslice des. Technical Report 051, Cryptology ePrint Archive, `http://eprint.iacr.org/2000/051`, October 2000. Accessed August 2014.

[32] Arjen K Lenstra and Eric R Verheul. Selecting cryptographic key sizes. *Journal of cryptology*, 14(4):255–293, 2001.

[33] Svetlin A. Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *Proc. of the 2007 IEEE International Conference on Sig-*

nal Processing and Communications (ICSPC 2007), Dubai, United Arab Emirates, pages 65–68. IEEE, November 2007.

[34] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[35] Atsuko Miyaji, Mohammad Shahriar Rahman, and Masakazu Soshi. Efficient and low-cost rfid authentication schemes. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 2(3):4–25, 2011.

[36] Wael Said Abdelmageed Mohamed, Jintai Ding, Thorsten Kleinjung, Stanislav Bulygin, and Johannes Buchmann. Pwxl: A parallel wiedemann-xl algorithm for solving polynomial equations over GF(2). *SCC*, 2010:89–100, 2010.

[37] Andrew Moss, Daniel Page, and Nigel P. Smart. Toward acceleration of rsa using 3d graphics hardware. In *Proc. of the 11th IMA international conference on Cryptography and Coding, Cirencester, United Kingdom, LNCS*, volume 4887, pages 364–383. Springer-Verlag, December 2007.

[38] D. Osvik, J. Bos, D. Stefan, and D. Canright. Fast software aes encryption. In *Fast Software Encryption*, pages 75–93. Springer, 2010.

[39] Nasrollah Pakniat and Ziba Eslami. A proxy e-raffle protocol based on proxy signatures. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 2(3):74–84, 2011.

[40] J. Patarin and L. Goubin. Asymmetric cryptography with s-boxes is it easier than expected to design efficient asymmetric cryptosystems? *Information and Communications Security*, pages 369–380, 1997.

[41] Jacques Patarin. Hidden fields equations (hfe) and isomorphisms of polynomials (ip): Two new families of asymmetric algorithms. In *Advances in Cryptology - Eurocrypt'96*, pages 33–48. Springer, 1996.

[42] Albrecht Petzoldt. Speeding up quad. Technical Report 263, Cryptology ePrint Archive, `http://eprint.iacr.org/2013/263`, July 2013. Accessed August 2014.

[43] Chester Rebeiro, David Selvakumar, and ASL Devi. Bitslice implementation of aes. In *Proc. of the 5th International Conference on Cryptology and Network Security, (CANS 2006), Suzhou, China, LNCS*, volume 4301, pages 203–212. Springer-Verlag, December 2006.

[44] Aresh Reyhani-Masoleh and M. Anwar Hasan. Efficient digit-serial normal basis multipliers over binary extension fields. *Journal of ACM Transactions on Embedded Computing Systems*, 3(3):575–592, August 2004.

[45] Harald Niederreiter Rudolf Lidl. *Introduction to finite fields and their applications, Revised edition.* Cambridge University Press, 1994.

[46] Satoshi Tanaka, Tung Chou, Bo-Yin Yang, Chen-Mou Cheng, and Kouichi Sakurai. Efficient parallel evaluation of multivariate quadratic polynomials on gpus. In *Proc. of the 13th International Workshop on Information Security Applications (WISA 2012), Jeju Island, Korea, LNCS*, volume 7690, pages 28–42. Springer-Verlag, August 2012.

[47] Satoshi Tanaka, Takashi Nishide, and Kouichi Sakurai. Efficient implementation of evaluating multivariate quadratic system with gpus. In *Proc. of the Sixth International Conference on Innovate Mobile and Internet Services in Ubiquitous Computing (IMIS 2012), Palermo, Italy*, pages 660–664. IEEE, July 2012.

[48] Satoshi Tanaka, Takashi Nishide, and Kouichi Sakurai. Efficient implementation for quad stream cipher with gpus. *Journal of Computer Science and Information Systems*, 10(2, special issue):897–911, April 2013.

[49] Satoshi Tanaka, Takanori Yasuda, and Kouichi Sakurai. Fast evaluation of multi-

variate quadratic polynomials over $gf(2^{32})$ using grahpics processing units. *Journal of Internet Services and Information Security (JISIS)*, 4(3):1–20, August 2014.

[50] Satoshi Tanaka, Takanori Yasuda, and Kouichi Sakurai. Implementation of efficient operations over $gf(2^{32})$ using graphics processing units. In *Proc. of the Second International Conference on Information & Communication Technology (ICT-EurAsia 2014), Bali, Indonesia, LNCS*, volume 8407, pages 602–611. Springer-Verlag, April 2014.

[51] Satoshi Tanaka, Takanori Yasuda, Bo-Yin Yang, Chen-Mou Cheng, and Kouichi Sakurai. Efficient computing over $gf(2^{16})$ using graphics processing unit. In *Proc. of the Seventh International Conference on Innovative Mobile and Internet Servicies in Ubiquitous Computing (IMIS 2013), Taichung, Taiwan*, pages 843–846. IEEE, July 2013.

[52] Douglas Wiedemann. Solving sparse linear equations over finite fields. *Information Theory, IEEE Transactions on*, 32(1):54–62, 1986.

[53] Hongjun Wu. A new stream cipher hc-256. In *Proc. of the 11th International Workshop on Fast Software Encryption (FSE 2004), Delhi, India, Revised Papers, LNCS*, volume 3017, pages 226–244. Springer-Verlag, February 2004.

[54] Hongjun Wu. The stream cipher hc-128. Technical report, eSTREAM, the ECRYPT Stream Cipher Project, `http://www.ecrypt.eu.org/stream/p3ciphers/hc/hc128_p3.pdf`, 2008. Accessed August 2014.

[55] Bo-Yin Yang, Daniel J. Bernstein Owen Chia-Hsin Cheng, and Jiun-Ming Chen. Analysis of quad. In *Proc. of the 14th International Workshop on Fast Software Encryption (FSE2007), Luxembourg, Luxembourg, Revised Selected Papers, LNCS*, volume 4593, pages 290–308. Springer-Verlag, March 2007.

Publications

Refereed Journal Papers

(1) Satoshi Tanaka, Takashi Nishide, and Kouichi Sakurai, "Efficient Implementation for QUAD Stream Cipher with GPUs." Journal of Computer Science and Information Systems, vol. 10, issue 2, special issue, pp.897-911, ComSIS Consortium, April 2013.

(2) Satoshi Tanaka, Takanori Yasuda, and Kouichi Sakurai. "Fast Evaluation of Multivariate Quadratic Polynomials over GF(2^{32}) using Grahpics Processing Units." Journal of Internet Services and Information Security (JISIS), vol. 4, issue 3, pp.1-20, ISYOU, ISEP/IPP, August 2014.

Refereed International Conference Papers

(1) Satoshi Tanaka, Tung Chou, Bo-Yin Yang, Chen-Mou Cheng, and Kouichi Sakurai, "Efficient Parallel Evaluation of Multivariate Quadratic Polynomials on GPUs." In Proc. of the 13th International Workshop on Information Security Applications (WISA 2012), Jeju Island, Korea, LNCS, vol. 7690, pp.28-42, Springer-Verlag, August 2012.

(2) Satoshi Tanaka, Takashi Nishide, and Kouichi Sakurai, "Efficient Implementation of Evaluating Multivariate Quadratic System with GPUs." In Proc. of the Sixth

International Conference on Innovate Mobile and Internet Services in Ubiquitous Computing (IMIS 2012), Palermo, Italy, pp. 660-664, IEEE, July 2012.

(3) Satoshi Tanaka, Takanori Yasuda, Bo-Yin Yang, Chen-Mou Cheng, and Kouichi Sakurai, "Efficient Computing over GF(2^{16}) using Graphics Processing Unit." In Proc. of the Seventh International Conference on Innovative Mobile and Internet Servicies in Ubiquitous Computing (IMIS 2013), Taichung, Taiwan, pp. 843-846, IEEE, July 2013.

(4) Satoshi Tanaka, Takanori Yasuda, and Kouichi Sakurai, "Implementation of Efficient Operations over GF(2^{32}) using graphics processing units." In Proc. of the Second International Conference on Information & Communication Technology (ICT-EurAsia2014), Bali, Indonesia, LNCS, vol. 8407, pp. 602-611, Springer-Verlag, April 2014.

(5) Satoshi Tanaka, Chen-Mou Cheng, Takanori Yasuda, Kouichi Sakurai, "Parallelization of QUAD Stream Cipher using Linear Recurring Sequences on Graphics Processing Unit." In Proc. of the 1st International Workshop on Information and Communication Security (WICS'14), pp.543-548, IEEE, December, 2014.

(6) Satoshi Tanaka, Chen-Mou Cheng, Kouichi Sakurai, "Evaluation of Solving Time for Multivariate Quadratic Equation System using XL Algorithm over Small Finite Fields on GPU." In Proc. of the 2nd International Conference on Mathematics and Computing (ICMC 2015), Springer Proceedings in Mathematics & Statistics, Springer, January, 2015.

Unrefereed International Conference Papers

(1) Satoshi Tanaka, Takashi Nishide, Kouichi Sakurai, "Efficient Computation Method of Multivariate Quadratic Polynomial Systems for Cryptography on Graphics Pro-

cessing Unit." In 2013 International Symposium on Information Science and Electrical Engineering, poster session, January, 2013.

(2) Satoshi Tanaka, Takanori Yasuda, Kouichi Sakurai, "Implementation of Efficient Operations over GF(2^{32}) using Graphics Processing Units." In the 8th International Workshop on Security (IWSEC 2013), poster session, September, 2013.

(3) Satoshi Tanaka, Takanori Yasuda, Kouichi Sakurai, "Efficient Implementation of Multiplication on Extension Field Using Grahics Processing Unit." In Forum "Math-for-Industry" 2013, poster session, November, 2013.

(4) Satoshi Tanaka, Takanori Yasuda, Kouichi Sakurai, "Parallel Implementations of QUAD Stream Cipher over Binary Extension Fields on Graphics Processing Units." In the 9th International Workshop on Security (IWSEC 2014), poster session, August, 2014.

Domestic Conference Papers

(1) Satoshi Tanaka, Takanori Yasuda, Kouichi Sakurai, "Notes on Efficient Computing over GF(2^{16}) using Graphics Processing Unit." In IEICE Technical Report, vol. 112, no. 460, pp.143-147, Marchi, 2013.

(2) Satoshi Tanaka, Bo-Yin Yang, Chen-Mou Cheng, Kouichi Sakurai, "Accelerating of Solving Method for Non-linear Multivariate System with Graphics Processing Unit." In Proc. of the 67th Joint Conference of Electrical, Electronics and Engineers in Kyushu (JCEEE 2013), page 184, September, 2013.

(3) Satoshi Tanaka, Takanori Yasuda, Kouichi Sakurai, "Implementation on Efficient Operations over GF(2^{32}) using Graphics Processing Unit." In Proc. of Computer Security Symposium 2013 (CSS 2013), pp.565-572, October, 2013.

(4) Satoshi Tanaka, Takanori Yasuda, Kouichi Sakurai, “Accelerating QUAD Stream Cipher over Extend Field using Graphics Processing Unit.” In Proc. of Forum on Information Technology 2014 (FIT2014), pp.149-150, September, 2014.

(5) Satoshi Tanaka, Takanori Yasuda, Kouichi Sakurai, “Parallel Implementation of QUAD Stream Cipher over GF(2^{32}) using Linear Recurring Sequence.” In Proc. of the 67th Joint Conference of Electrical, Electronics and Information Engineers in Kyushu (JCEEE 2014), September, 2014.

(6) Satoshi Tanaka, Chen-Mou Cheng, Kouichi Sakurai, “Evaluating Solving Time of Multivariate Quadratic Equation System using XL Algorithm over Small Finite Fields.” In Proc. of Computer Security Symposium 2014 (CSS 2014), pp.124-131, October, 2014.

(7) Satoshi Tanaka, Chen-Mou Cheng, Takanori Yasuda, Kouichi Sakurai, “Accelerating QUAD Stream Cipher using Multi-Stream Method on Graphics Processing Unit.” In IEICE Technical Report, vol. 114, no. 319, ISEC2014-56, pp. 1-6, November, 2014.

Index