Fast Evaluation of Multivariate Quadratic Polynomials over GF(2^{32}) using Grahpics Processing Units*

Satoshi Tanaka[1,2†], Takanori Yasuda[2], and Kouichi Sakurai[1,2]
[1]Kyushu University, Fukuoka, Japan
tanasato@itslab.inf.kyushu-u.ac.jp, sakurai@csce.kyushu-u.ac.jp
[2]Institute of Systems, Information Technologies and Nanotechnologies, Fukuoka, Japan
yasuda@isit.or.jp

Abstract

QUAD stream cipher is a symmetric cipher based on multivariate public-key cryptography(MPKC), which uses multivariate polynomials as encryption keys. It holds the provable security property based on the computational hardness assumption. More specifically, the security of QUAD depends on the hardness of solving non-linear multivariate quadratic systems over a finite field, which is known as an NP-complete problem. However, QUAD is slower than other stream ciphers, and an efficient implementation, which has a reduced computational cost, is required. In this paper, we propose some implementations of QUAD over GF(2^{32}) on Graphics Processing Units(GPU) and compare them. Moreover, we provide fast multiplications over GF(2^{32}), the core operation of QUAD. Our implementation gives the fastest throughput of QUAD as 24.827 Mbps. We propose an efficient implementation for computing with multivariate polynomials in multivariate cryptography on GPU and evaluate the efficiency of the proposal. GPU is considered to be a commodity parallel arithmetic unit. Our proposal parallelizes an algorithm coming from multivariate cryptography, and makes it efficient by optimizing the algorithm with GPU.

Keywords: efficient implementation, multivariate public-key cryptography, GPU, QUAD, stream cipher

1 Introduction

1.1 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.) [11, 32, 31]. 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-ramdom number generator (PRNG), whose security is provably based on the integer factorization [9]. The QUAD, proposed by Berbain, Gilbert and Patarin, is also such a stream cipher endowed of provable security [8]. It uses the theory of multivariate public-key

Journal of Internet Services and Information Security (JISIS), volume: 4, number: 3, pp. 1-20

*Our GPU implementations of multiplication methods over GF(2^{32}) have appeared in the authors' preliminary paper, "Implementation of Efficient Operations over GF(2^{32}) using Graphics Processing Units," in Proc. of Information & Communication Technology-EurAsia Conference 2014 [29]. This version implements an addiitional method called bitslicing and compare with previous implementations. Moreover, we apply our multiplications to QUAD stream cipher proposed by Berbain, Gilbert and Patarin [8].

†Corresponding author: Room 712, West 2 building, 744 Motooka, Nishi-ku, Fukuoka, Fukuoka 819 -0395, Japan, Tel: +81-928023639

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 MQ. Since MQ is known to be NP-complete [6], 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 [13], 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) [18] and Advanced Encryptions Standard(AES) [23]). 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 progamming libraries and some open libraries (e.g. OpenCL [3]) also allow GPU computations.

1.2 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 [7]. 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 [5]. Hamlet and Brocato provide implementations of QUAD$(2,128,128)$ on a Vertex-4 FPGA and the fastest one gives a 374 Mbps throughput [14]. 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 [8].

1.3 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.

1.3.1 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 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.

1.3.2 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 implementaions of binary extension fields [24] and GPU implementations over extension fields [22], however they do not discuss GPU implementaions of extensions of the binary field. Hence, fast implementations of multiplications over binary field's extensions on GPU is an important issue.

1.3.3 Optimizations of CUDA GPU implementation

In this paper, we use Compute Unified Device Architecture(CUDA) API [2], provided by NVIDIA [4], for GPU implementaions. In CUDA implementations, we have two subissues 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.

1.4 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}).

1.4.1 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 [19]. Li, Zhong, Zhao, et. al. achieve 50 times faster AES on NVIDIA Tesla C2050, GPU citeLi12. 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 [17]. 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 [10]. 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 [26, 27, 28] and studying fast multiplications over finite fields [30, 29]. 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 extenstion fields, we introduce bitslicing techniques. As a result, we have found a more suited multiplication method than in previous results [29].

2 Preliminaries

2.1 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 $\mathrm{GF}(p)$. Generaly, $\mathrm{GF}(q)$ 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 extenstions 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:

$$a = \{c_1, \ldots, c_k\}, \ (c_1, \ldots, c_k \in \mathrm{GF}(p)), \tag{1}$$

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

$$a = c_1 X_1 + \cdots + c_k X_k = \sum_{i=1}^{k} c_i X_i, \tag{2}$$

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 [25]: 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 $\{\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{k-1}}\}$

2.1.1 Operations over extension fields

$\mathrm{GF}(q)$ 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,

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

Since a can be handled as a vector of $\mathrm{GF}(p)$ like in Equation (1), additions of $\mathrm{GF}(q)$ are computed by:

$$a+b := \{a_1+b_1 \; mod \; p, \ldots, a_k+b_k \; mod \; p\}, \tag{3}$$

2.1.2 Zech's logarithm

Originally, Zech's logarithm (also called Jacobi's logarithm [25]) is proposed to figure addtions 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 [15, 16]. 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 $p^r - 1$.

2.1.3 Intermediate field [25]

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 $\mathrm{GF}(p^l)$.

2.2 Multivariate 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 $\mathrm{GF}(q)$ can be written in the following form

$$\begin{aligned} f_1(x_1, \ldots, x_n) &= \sum_{1 \leq i \leq j \leq n} \alpha_{i,j}^{(1)} x_i x_j + \sum_{1 \leq i \leq n} \beta_i^{(1)} x_i + \gamma^{(1)} \\ f_2(x_1, \ldots, x_n) &= \sum_{1 \leq i \leq j \leq n} \alpha_{i,j}^{(2)} x_i x_j + \sum_{1 \leq i \leq n} \beta_i^{(2)} x_i + \gamma^{(2)} \\ &\vdots \\ f_m(x_1, \ldots, x_n) &= \sum_{1 \leq i \leq j \leq n} \alpha_{i,j}^{(m)} x_i x_j + \sum_{1 \leq i \leq n} \beta_i^{(m)} x_i + \gamma^{(m)}. \end{aligned} \tag{4}$$

Equation (4) can be interpreted as a function of $\mathrm{GF}(p^k)^n \mapsto \mathrm{GF}(p^k)^m$. Let $\boldsymbol{x} = \{x_1, \ldots, x_n\}$. 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 fileds. 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 quadratic polynomials with n variables and m polynomials requires $mn(n+3)/2$ additions and $mn(n+2)$ multiplications over finite fields.

2.3 The problem MQ

Solving the multivariate quadratic system means the following. Assume that we have known the system A of m quadratic polynomials in n variables over a finite field GF(q), given by Equation (4). Let $\boldsymbol{y}=\{y_1,\ldots,y_m\}^T$ be a m-degree column vector, generated by multiplying the system A and the n-degree unknown column vector $\boldsymbol{x}=\{x_1,\ldots,x_n\}^T$. The system (4) is equivalent to:

$$A\boldsymbol{x}=\boldsymbol{y}. \tag{5}$$

Then, the problem of finding the unknown column vector $\boldsymbol{x}$ with given A and $\boldsymbol{y}$ is called MQ. (MQ means "multivariate quadratic"). More generally, solving systems of cubic or higher degree polynomials is sometimes called MP. Both MQ and MP are known to be NP-complete over GF(q) for any q [6].

2.4 MPKC

Some cryptosystems use a system of multivariate polynomials for encryption/decryption, or for generating/verifying signatures [20]. We call the family of such cryptosystems "multivariate public key cryptography" (MPKC). The security of MPKC is based on the MQ or MP assumptions, i.e. if MQ or MP is hard, MPKCs are also secure.

3 QUAD stream cipher

QUAD is a stream cipher proposed by Berbain, Gilbert and Patarin [8]. 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 [8].

3.1 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 (4). 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. Somtimes, 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.

3.2 Algorithm

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

3.2.1 Keystream generation

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 1. It indicates that the generator outputs keystreams by repeating the above three steps.

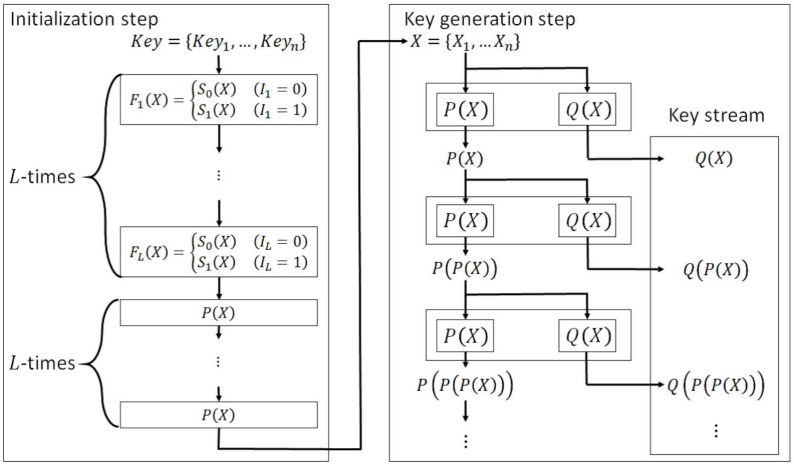

Figure 1: Image of QUAD key generating algorithm

3.2.2 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.

3.2.3 Key and initialization of current state

Berbain, Gilbert and Patarin also provides a technique for initialization of the internal state $X = (x_1, \ldots, x_n)$ [8]. 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 $IV_i = 1$.

3.2.4 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 $O(n^2)$. QUAD(q,n,r) requires to compute m multivariate quadratic polynomials. Since $m = kn$, the computational cost of generating key stream is $O(n^3)$.

3.2.5 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 MQ problem [8]. The eXtended Linearization(XL) algorithm [12] is a solving method of 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. [33] show that the degree of regularity of MQ in QUAD(q,n,n) is given by the degree of the lowest term with a non-positive coefficient in the following polynomial,

$$G(t) = ((1-t)^{(-n-1)}(1-t^2)^n(1-t^4)^n). \tag{6}$$

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

$$C_{XL} \sim 3\tau T\mathfrak{m}. \tag{7}$$

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 GPGPU via CUDA

GPU is a special-purpose processor for accelerating computer graphics computations. Due to the nature of its computational tasks, GPUs can handle many operations in parallel at a high speed.

General-purpose GPU (GPGPU) computing is a technique that uses GPUs for general-purpose computation. Since GPUs are designed for SIMD operations, they are quite efficient for parallel processing. On the other hand, they are not so efficient when there is only a limited amount of parallelism. Therefore, the most important task in GPGPU is to identify or manufacture parallelism in the algorithms to be implemented.

4.1 CUDA API

CUDA is a development environment for NVIDIA's GPUs [2]. 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.

4.2 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.

4.3 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.

5 Evaluating multivariate quadratic polynomials on GPU

5.1 Evaluating polynomials by SIMD

GPU is suitable for implementations of SIMD constructions. SIMD is a pallalelization 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 $\sum_{1\le i\le j\le n} \alpha_{i,j}^{(k)} x_i x_j + \sum_{1\le i\le n} \beta_i^{(k)} x_i$.

5.1.1 Precomputing $x_i x_j$

This step is based on the precomputing method of Berbain, Billet and Gilbert [7]. 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.

$$t = \frac{j(j-1)}{2} + i, \tag{8}$$

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

5.1.2 Compute all monomials

Before computing monomials, we store $x_i x_j$ into x_{n+t}, where t is given by Equation (8). Also, we assume that $\beta_{n+t}^{(k)} = \alpha_{i,j}^{(k)}$, then we compute $\beta_i^{(k)} x_i$ $(1 \le i \le (n+1)(n+2)/2)$ in parallel.

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

In this paper, we use parallel reduction technique like the method of Tanaka, Nishide, Sakurai [28]. 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.

5.2 Reducing terms of 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 $T \leq 2,048$, 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.

$$\begin{cases} i \equiv n-j-1 & (mod\ \frac{n}{k}) \quad (k \mid n) \\ k(i-1)+j > n-k-1 & (mod\ n) \quad (k \nmid n) \end{cases} \tag{9}$$

After that, we construct new $n-k$-tuple variables $\boldsymbol{x}'_j$ for each polynomial f_j, where $1 \leq j \leq m$. Then, the number of terms in each polynomial is reduced from $\binom{n+2}{2}$ to $\binom{n-k+2}{2}$. Figure 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.

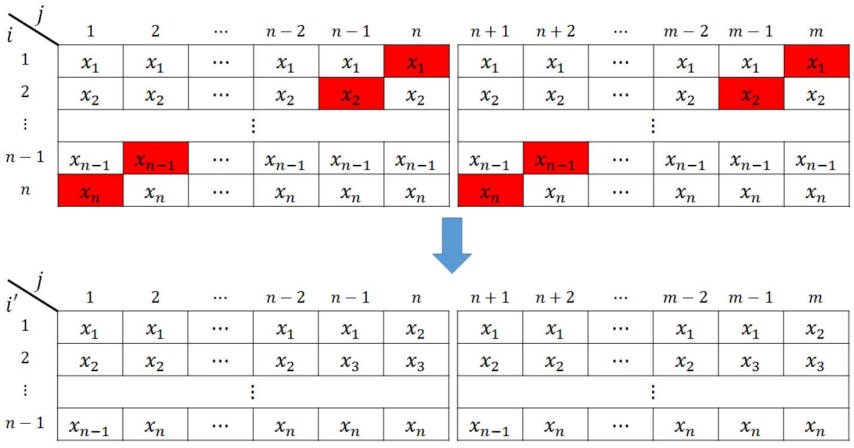

Figure 2: Removing variables x_i from quadratic polynomial f_j, where $k = 1$.

6 Analysis of multiplication algorithms over extension fields

6.1 Multiplication method

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 $c := a * b \in GF(q)$.

6.2 Polynomial basis

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

$$e_1 * e_2 := e_1(X) * e_2(X) \mod f(X), \tag{10}$$

Let $e_1, e_2 \in \mathrm{GF}(q)$ be $c_{k-1}x^{k-1} + \cdots + c_1 x + c_0$ and $c'_{k-1}x^{k-1} + \cdots + c'_1 x + c'_0$, respectively. The product $e_1 * e_2$ can be computed by:

$$e_1 * e_2 = c_{k-1}c'_{k-1}x^{2k-2} + \cdots + c_0 c'_0 \mod f(X).$$

In this method, we need to compute the multiplications $c_i c'_j$ for $0 \le i, j < k$ and the summations $\sum_{i+j=t, i,j\ge 0} c_i c'_j$ for $0 \le t \le 2(k-1)$ over GF(p). The summation $\sum_{i+j=t, i,j\ge 0} c_i c'_j$ requires t additions for $0 \le t < k$ and $2k-t-2$ additions for $k \le t \le 2(k-1)$. Therefore, it requires $(k-1)^2$ additions and k^2 multiplications over 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.

6.2.1 Normal basis

Given a finite Galois extension GF(q)/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 GF(p)-basis of GF(q), which is called a normal basis of GF(q)/GF(p). A normal basis of GF(q)/GF(p) can thus be denoted by:

$$\{\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{k-1}}\}. \tag{11}$$

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

$$a = c_0\alpha + c_1\alpha^{p^m} + \cdots + c_{k-1}\alpha^{p^{(k-1)m}},\ c_0, \ldots, c_{k-1} \in \mathrm{GF}(p). \tag{12}$$

Let $a = [c_0, c_1, \ldots, c_{k-1}]_n \in \mathrm{GF}(q)$ be defined in Equation (12). Then Frobenius map σ_0 applied to α gives:

$$\sigma_0(a) = a^q = [c_{k-1}, c_0, c_1, \ldots, c_{k-2}]_n. \tag{13}$$

In other words, $\sigma_0(a)$ is simply a right circular shift [21].

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 $0 \le i < k$, 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 GF(p). Let $d_i = p_i(c_0, \ldots, c_{k-1}, c'_0, \ldots, c'_{k-1})$, $\forall 0 \le i < k$. According to Equation (13), we can compute $\sigma_0(a * b)$ by:

$$\begin{aligned} \sigma_0(a*b) &= [d_{k-1}, d_0, d_1, \ldots, d_{k-2}]_n \\ &= \sigma_0(a) * \sigma_0(b) \\ &= [c_{k-1}, c_0, \ldots, c_{k-2}]_n * [c'_{k-1}, c'_0, \ldots, c'_{k-2}]_n \\ &= [p_0(c_{k-1}, c_0, \ldots, c_{k-2}, c'_{k-1}, c'_0, \ldots, c'_{k-2}), \ldots, \\ &\quad\ p_{k-1}(c_{k-1}, c_0, \ldots, c_{k-2}, c'_{k-1}, c'_0, \ldots, c'_{k-2})]_n. \end{aligned} \tag{14}$$

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 (14). 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 $0 \le i, j \le k-1$. Moreover, we can modify formula for c_i, c_j, c'_i, c'_j to :

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

where $i \neq j$. Therefore, a multiplication over $\mathrm{GF}(q)$ requires $k(k-1)(k+2)/2$ additions and $k(k^2+1)/2$ multiplications over $\mathrm{GF}(p)$ 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.

6.3 Zech's logarithm

In this method, a multiplication over $\mathrm{GF}(q)$ 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 $\mathrm{GF}(p)$, 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 $\mathrm{GF}(q)$ to itself, Zech's method needs $2p^n \lceil \log_2 p^n \rceil$ bits of memory.

6.4 Multiplication tables

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

An addition over $\mathrm{GF}(q)$ can be computed using k additions over $\mathrm{GF}(p)$. On the other hand, a multiplication over $\mathrm{GF}(q)$ needs only one table look-up. Since the entire multiplication table needs to store every possible combination of multiplications over $\mathrm{GF}(q)$, this method requires $p^{2n} \lceil \log_2 p^n \rceil$ bits of memory.

6.5 Using intermediate fields

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

6.6 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,

Table 1: Cost of multiplication over GF(2^{32}).

Intermediate field GF(2^l)	Method		Computational cost					Memory space
	GF(2^l)/GF(2)	GF(2^k)/GF(2^l)	XOR	AND	LOOKUP	MOD	ADD	
Direct	-	Polynomial basis (Bitslicing)	1,096	1,024	0	0	0	4B
			34.25	32	0	0	0	128B
		Normal basis (Bitslicing)	16,864	16,400	0	0	0	125B
			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

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.

6.7 Costs of multiplications over GF(2^{32})

Table 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 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 1 shows an average computational cost of 1 multiplications of 32 bitsliced multiplications.

6.8 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: GF(2^{32})/GF(2^k)/GF(2) ($k = 1, 2, 4, 8$)
3. Zech's logarithm + polynomial basis method: GF(2^{32})/GF(2^{16})/GF(2)

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

Intermediate field GF(2^l)	Multiplication method GF(2^l)/GF(2)	Intel Core i7 875K		NVIDIA GTX TITAN		NVIDIA GeForce GTX 580 [29]	
		Polynomial basis	Normal basis	Polynomial basis	Normal basis	Polynomial basis	Normal basis
Direct (bitslicing)	-	338.077s	575.096s	0.0764s	8.657s	1.552s	25.064s
		18.484s	15.425s	0.00246	0.949s	N.A.	N.A.
GF(2^2)	Multiplication table	121.997s	159.989s	1.548s	5.099s	1.242s	3.813s
GF(2^4)		31.651s	38.281s	0.368s	0.485s	0.583s	0.776s
GF(2^8)		8.627s	9.121s	0.0479s	0.129s	0.0555s	0.0621s
GF(2^{16})	Zech's logarithm	3.510s	3.015s	0.153s	0.0764s	0.195s	0.153s

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

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

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

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

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

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

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

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

5. GF(2^{32})/GF(2^{16})/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.

6.8.1 Experimental result

Table 2 shows the result of implementations computational time for 67,108,864 multiplications. Table 3 shows throughputs of the result with our previous work [29]. In CPU implementations, the normal basis method using 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 [29], the polynomial 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 memor the tables in Zech's logarithm over GF(2^8), as these tables are too large to fit into the fast memory on GPU.

Table 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) [29]	
		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

Table 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

7 QUAD stream cipher on GPU

7.1 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 shows other parameters of these constructions. Security parameters in Table 4 are roughly evaluated with formula (6) and (7) by analysises of Yang, Chen, Bernstein, et. al. [33]. 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 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, $C_{XL} \simeq 27.4171 \times 10^{41} \times 0.03 \simeq 2.2251 \times 10^{40} \leq 2^{134}$.

7.2 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 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 3 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 seperated into bit-data chunks.

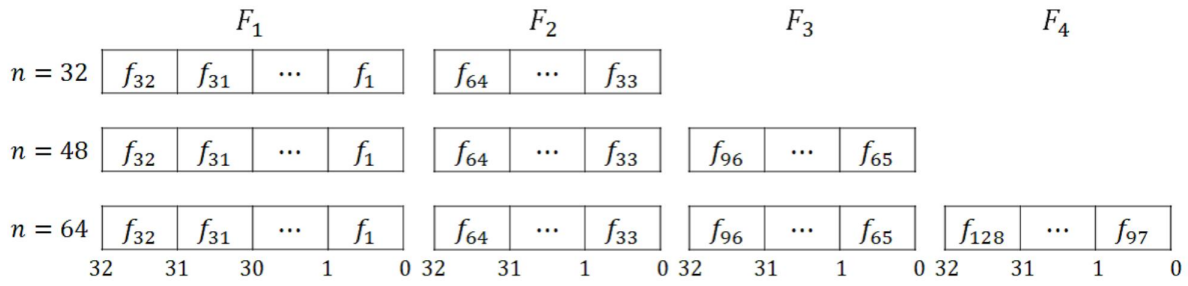

Figure 3: Placements of polynomials for each QUAD construction.

7.3 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 5.

Table 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 6 and 7. Table 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 less secure than QUAD($2^{32}, 64, 64$). Hence, our implementations seems to be a tradeoff point between speed and security. Table 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 [28]. Moreover, these factors show that QUAD stream cipher is suitable for parallel implementations.

Table 6: Comparison with other QUAD implementations

	Implementation environment	Constructions			Output (bit)	Key (KB)	Throughputs (Mbps)	Security (bit)
		q	n	m				
BGP06 [8]	Pentium 4	2	160	320	160	503.164	5.7	≤ 140
BBG067 [7]	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 [5]	FPGA	2	256	512	256	2056.063	2.0	≤ 140
HB13 [14]	Virtex-4, FPGA	2	128	256	128	262.031	374.7	≤ 118
TNS [28]	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

Table 7: Comparison with previous GPU implementations of QUAD.

	GPU	Algorithm	Throughputs	Speed up factor
TNS13 [28]	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

8 Conclusion

In this work, we discuss fast implementations of QUAD over GF(2^{32}). We discuss 3 approaches of accelerating QUAD, parallelization of evaulating 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 [29]. 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})).

Acknowledgement

This work is partly supported by "Study on Secure Cryptosystem using Multivariate Polynomials," no. 0159-0172, Strategic Information and Communications R&D Promotion Programme (SCOPE), the

Ministry of Internal Affairs and Communications, Japan. The authors are grateful to Xavier Dahan for his valuable comments on our proposal.

References

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

[2] NVidia developer zone. `https://developer.nvidia.com/category/zone/cuda-zone`, Accessed July 2014.

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

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

[5] D. Arditti, C. Berbain, O. Billet, and H. 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.

[6] G. V. Bard. *Algebraic Cryptanalysis*. Springer, 2009.

[7] C. Berbain, O. Billet, and H. Gilbert. Efficient implementations of multivariate quadratic systems. In *Proc. of the 13th International Workshop on Selected Areas in Cryptography (SAC'06), Concordia University, Montreal, Quebec, Canada, Revised Selected Papers, LNCS*, volume 4356, pages 174–187. Springer-Verlag, August 2006.

[8] C. Berbain, H. Gilbert, and J. 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'06), St. Petersburg, Russia, LNCS*, volume 4004, pages 109–128. Springer-Verlag, May-June 2006.

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

[10] J. W. Bos and D. 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'10), Santa Barbara, USA, LNCS*, volume 6225, pages 279–293. Springer-Verlag, August 2010.

[11] C. D. Cannière and B. 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.

[12] N. Courtois, A. Kilmov, J. Patarin, and A. 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'00), Bruges, Belgium, LNCS*, volume 1807, pages 392–407. Springer-Verlag, May 2000.

[13] J. Goodman and A. P. Chandrakasan. An energer-efficient reconfigurable public-key cryptography processor. In *Proc. of the 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'00), Worcester, Massachusetts, USA, LNCS*, volume 1965, pages 175–190. Springer-Verlag, August 2000.

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

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

[16] K. Imamura. A method for computing addition tables in $\mathrm{gf}(p^n)$. *Journal of IEEE Transactions on Information Theory*, 26(4):367–369, May 1980.

[17] K. Jang, S. Han, S. Han, S. Moon, and K. 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.

[18] M. 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.

[19] S. A. Manavski. Cuda compatible GPU as an efficient hardware accelerator for AES cryptography. In *Proc. of the 2007 IEEE International Conference on Signal Processing and Communications (ICSPC'07), Dubai, United Arab Emirates*, pages 65–68. IEEE, November 2007.

[20] T. Matsumoto and H. Imai. Public quadratic polynomial-tuples for efficient signature-varification and message-encryption. In *Proc. of the 7th Annual International Conference on the Theory and Applications of Cryptographic Techniques Advances (EUROCRYPT'88), Davos, Switzerland, LNCS*, volume 330, pages 419–453. Springer-Verlag, May 1988.

[21] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[22] A. Moss, D. Page, and N. 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.

[23] C. Rebeiro, D. Selvakumar, and A. Devi. Bitslice implementation of AES. In *Proc. of the 5th International Conference on Cryptology and Network Security, (CANS'06), Suzhou, China, LNCS*, volume 4301, pages 203–212. Springer-Verlag, December 2006.

[24] A. Reyhani-Masoleh and M. A. 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.

[25] H. N. Rudolf Lidl. *Introduction to finite fields and their applications, Revised edition*. Cambridge University Press, 1994.

[26] S. Tanaka, T. Chou, B.-Y. Yang, C.-M. Cheng, and K. Sakurai. Efficient parallel evaluation of multivariate quadratic polynomials on GPUs. In *Proc. of the 13th International Workshop on Information Security Applications (WISA'12), Jeju Island, Korea, LNCS*, volume 7690, pages 28–42. Springer-Verlag, August 2012.

[27] S. Tanaka, T. Nishide, and K. Sakurai. Efficient implementation of evaluating multivariate quadratic system with gpus. In *Proc. of the 6th International Conference on Innovate Mobile and Internet Services in Ubiquitous Computing (IMIS'12), Palermo, Italy*, pages 660–664. IEEE, July 2012.

[28] S. Tanaka, T. Nishide, and K. Sakurai. Efficient implementation for QUAD stream cipher with GPUs. *Journal of Computer Science and Information Systems*, 10(2, special issue):897–911, April 2013.

[29] S. Tanaka, T. Yasuda, and K. 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'14), Bali, Indonesia, LNCS*, volume 8407, pages 602–611. Springer-Verlag, April 2014.

[30] S. Tanaka, T. Yasuda, B.-Y. Yang, C.-M. Cheng, and K. 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'13), Taichung, Taiwan*, pages 843–846. IEEE, July 2013.

[31] H. Wu. A new stream cipher HC-256. In *Proc. of the 11th International Workshop on Fast Software Encryption (FSE'04), Delhi, India, Revised Papers, LNCS*, volume 3017, pages 226–244. Springer-Verlag, February 2004.

[32] H. 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.

[33] B.-Y. Yang, D. J. B. Owen Chia-Hsin Cheng, and J.-M. Chen. Analysis of QUAD. In *Proc. of the 14th International Workshop on Fast Software Encryption (FSE'07), Luxembourg, Luxembourg, Revised Selected Papers, LNCS*, volume 4593, pages 290–308. Springer-Verlag, March 2007.

Author Biography

Satoshi Tanaka received a B.E. degree from National Institution for Academic Degrees and University Evaluation, Japan in 2010, and an M.E. degree from Kyushu University, Japan in 2012. Currently he is a candidate for the Ph.D. in Kyushu University. His primary research is in the areas of cryptography and information security.

Takanori Yasuda received the PhD. degrees in mathematics from Kyushu University in 2007. He was a postdoctal fellow in Osaka City University from 2007 through 2008, in Kyushu University from 2008 through 2011. He is currently a researcher in Institute of Systems, Information Technologies and Nanotechnologies. His current research interests are pairing cryptography, multivariate public-key cryptosystem, and automorphic representations.

Kouichi Sakurai received the B.S. degree in mathematics from the Faculty of Science, Kyushu University in 1986. He received the M.S. degree in applied science in 1988, and the Doctorate in engineering in 1993 from the Faculty of Engineering, Kyushu University. He was engaged in research and development on cryptography and information security at the Computer and Information Systems Laboratory at Mitsubishi Electric Corporation from 1988 to 1994. From 1994, he worked for the Dept. of Computer Science of Kyushu University in the capacity of associate professor, and became a full professor there in 2002. He is concurrently working also with the Institute of Systems & Information Technologies and Nanotechnologies, as the chief of Information Security laboratory, for promoting research co-oporations among the industry, university and government under the theme “Enhancing IT-security in social systems”.