Memory-Constrained Implementation of Lattice-based Encryption Scheme on the Standard Java Card Platform[⋆]

Ye Yuan[1⋆⋆], Kazuhide Fukushima[2] Junting Xiao[1], Shinsaku Kiyomoto[2], and Tsuyoshi Takagi[3,4]

[1] Graduate School of Mathematics, Kyushu University
[2] KDDI Research, Inc.
[3] Department of Mathematical Informatics, The University of Tokyo
[4] CREST, Japan Science and Technology Agency

Abstract. Memory-constrained devices, including widely used smart cards, require resisting attacks by the quantum computers. Lattice-based encryption scheme possesses high efficiency and reliability which could run on small devices with limited storage capacity and computation resources such as IoT sensor nodes or smart cards. We present the first implementation of a lattice-based encryption scheme on the ***standard Java Card platform*** by combining number theoretic transform and improved Montgomery modular multiplication. The running time of decryption is nearly optimal (about 7 seconds for 128-bit security level). We also optimize discrete Ziggurat algorithm and Knuth-Yao algorithm to sample from prescribed probability distributions on the Java Card platform. More importantly, we indicate that polynomial multiplication can be performed on Java Card efficiently even if the long integers are not supported, which makes running more lattice-based cryptosystems on smart cards achievable.

Keywords: Post-Quantum Cryptography, Lattice-based Encryption Scheme, Java Card, Discrete Gaussian Sampling, Montgomery Modular Multiplication, Number Theoretic Transform

[⋆] A preliminary version of this paper appeared with the title ”Memory-constrained implementation of lattice-based encryption scheme on standard Java Card” in *Proceedings of 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* [43].

[⋆⋆] Email: y-yuan@math.kyushu-u.ac.jp

1 Introduction

Java Card is a mainstream applet development platform which allows developers to program in a subset of the Java language or run applets on a deployed smart card virtual machine. It is now widely used in telecommunications, finance, and banking, and aims to define a standard smart card computing and developing environment [20]. The card vendors and issuers can develop and test applets on smart cards quickly due to its interoperability and standard development interface; hence, the costs of applet development, employment, and maintenance are reduced significantly. Accordingly, Java Card has rapidly become the most popular smart card platform.

In the IoT era, tons of devices will be connected to the Internet and need cryptographic schemes for the security and privacy. As an essential part of IoT ecosystems, embedded devices like smart cards also require safe and reliable data protection features. A significant unique benefit for Java Card vendors and issuers is inherent security, including secure memory/data protection and cryptographic support. Typically, standard Java Card platform has an insufficient computational capability (see [17,40]). To make up for the processing capacity, almost all Java card manufacturers install dedicated cryptographic co-processors to speed up the operations of encryption algorithms such as 2048-bit RSA. However, Shor's algorithm that was introduced in [39] for factorization and computation of discrete logarithms on quantum computers can efficiently break the traditional RSA and elliptic curve cryptography (ECC). Since National Security Agency (NSA) announced that they planned to transition from RSA and the ECC to a new set of quantum resistant algorithms, post-quantum cryptography (PQC) has attracted more attention from academic and industries.

Lattice-based cryptography [30] is known to be secure against attacks by quantum computers. As one of the variants of lattice-based cryptography, ring-LWE based encryption schemes (see [26]) possess high efficiency and reliability. In recent years, the interest of implementation of ring-LWE based cryptography on various devices has emerged. Some previous works such as [4,16,27,34,35] have reported practical implementations on hardware or small devices. However, very limited literatures provided the implementation of ring-LWE based cryptography on individual smart cards, e.g., [5,6]. Considering the extensive usage of smart card markets, PQC needs to be implemented on the standard Java Card platform, imperatively.

Recently, more instantiations of ring-LWE based encryption schemes and key encapsulation mechanisms such as [1,7] had been submitted as the candidates of the PQC standardization project launched by National Institute of Standards and Technology (NIST) (see [31]); hence, it is necessary to investigate the performance of ring-LWE based cryptography on memory-constrained devices such as widely used smart cards. We also have implemented several famous ring-LWE based encryption schemes around 128-bit security on IoT devices successfully (see [42]). We noticed that polynomial multiplication is a challenge as well as the most expensive operation on memory-constrained devices due to the fact that there is insufficient computing capacity. Some literature provides polynomial operations optimized for specific parameters or architecture used in experiments such as [1,6], but we would like to find a general solution for all possible parameter sets and see the performance on the Java Card platform.

On the other hand, concerning the implementation of ring-LWE encryption scheme, the function of sampling from a target discrete Gaussian distribution is an indispensable component. However, sampling random values from a discrete Gaussian distribution on standard Java Card platform is still a quite tricky task. Studies were performed to deal with discrete Gaussian sampling in some literature, e.g., rejection sampling methods in [12,15,18], inversion sampling method in [33], discrete Ziggurat method in [3], and Knuth-Yao method in [23]. Rejection sampling does not require too much memory consumption but needs lots of time on high-precision computing. Inversion algorithm pre-computes and stores the cumulative distribution function (CDF) of the sampled distribution. When a probability is generated uniformly at random, it is very convenient to use the CDF in determining the sampled value directly. However, when applied to the standard Java Card platform, it is quite time-consuming to generate high-precision decimals. These two discrete Gaussian sampling methods are relatively straightforward but inefficient on such memory-constrained device. Therefore, we apply discrete Ziggurat algorithm [3] and Knuth-Yao algorithm [23] to sample random values from a target probability distribution. Different from rejection sampling and inversion algorithm, discrete Ziggurat algorithm and Knuth-Yao algorithm are aimed at getting a speed-memory trade-off. However, due to the limited computational capacity of Java Card, there is not enough space for a large look-up table storage, and it is inefficient to simulate floating-point number arithmetic using small integers, which is essential for many discrete Gaussian sampling algorithms.

1.1 Paper contribution

Our focus is to consequently implement the original ring-LWE encryption scheme [26] on the standard Java Card platform (or, simply, "Java Card" for short) with good compatibility, portability, and expansibility. In this paper, firstly, we implement and optimize discrete Ziggurat method and Knuth-Yao method on Java Card by reducing the storage consumption. The results show that even slight optimization of storage space can decrease those running time significantly. Secondly, we revisit the approach of the polynomial multiplication using more memory efficient algorithms and then improved the implementation of Java Card. In our previous work, we have shown polynomial multiplication is a challenge as well as the most expensive operation because of the insufficient computing capacity of Java Card. Compared to the previous work, further improvements of the implementation are achieved by reducing local method invocation and the use of temporary arrays. Thirdly, we propose an improved Montgomery modular multiplication from [2], which can be carried out for big integer arithmetic using only small integer operations without loss of accuracy and integer overflow. By combining efficient multiplication approach such as number theoretic transform (NTT) with the improved modular multiplication, we are the first to perform efficient polynomial multiplication with arbitrary large coefficients or moduli on Java Card, that means it is possible to apply ring-LWE based cryptography on more memory-constrained devices. Our solution obtains the nearly optimal running time (***about 7 seconds*** of decryption, and it is ***more than 10 times faster*** than our previous result) for an approximate security level of AES-128. With the performance boost over upgrading hardware, apparently, there is much room for improvement in our future work.

1.2 Differences

The major differences between this extended version and the previous conference paper [43] are as follows:

- We refactor our implementation of modular multiplication (see [2]), NTT, and inverse NTT methods (see [24,35]) on Java Card. We reduce the memory overhead of creating temporary instances and the number of function calls, and vastly improve the efficiency of modular multiplication and polynomial operations.
- We present our optimized approaches for discrete Gaussian sampling on Java Card and report the experiment results. We perform optimized binary number generating method and optimize the look-up tables to improve the performance of discrete Gaussian sampling on Java Card. The former aims at accelerating the discrete Ziggurat algorithm, whereas the latter optimizes the probability array tables for the Knuth-Yao algorithm by adding additional sign bits. Our optimizations are the result of a trade-off between cost, performance, and security.

– We rewrite the descriptive portions and explain the motivation and significance of our work in detail. We also report our experimental results and compare with our previous work, e.g., decryption is more 10 times faster than that of our previous work, and only costs about 7 seconds using 128-bit security parameter set. The results show that this combination method requires far less memory capacity and significantly improves computational speed.

1.3 Outline

The rest of this paper is organized as follows. We give a brief mathematical background of discrete Gaussian sampling and the ring-LWE based encryption scheme in Section 2. We introduce the specification of Java Card in Section 3 and describe our implementation techniques in Section 4. We then give the experiment reports of our implementation on a standard Java Card in Section 6. Finally, we conclude this paper in Section 6.

2 Preliminaries

In this section, we introduce the relevant mathematical background for discrete Gaussian sampling and the ring-LWE based encryption scheme [26].

Throughout this paper, the symbols $\mathbb{Z}$ and $\mathbb{Z}_q$ denote the ring of integers and ring of integers modulo a positive integer q, respectively, and $\mathbb{Z}_q[x]$, denotes the polynomials over $\mathbb{Z}_q$. Polynomials are denoted by bold italic letters such as $\boldsymbol{f}$ or $\boldsymbol{A}$, while vectors, bold small letters such as $\mathbf{v}$ and matrices, bold large letters such as $\mathbf{A}$.

2.1 Discrete Gaussian sampling

We define discrete Gaussian distribution $D_{L,\sigma}$ for given lattice L and deviation σ, and $\exp(-\pi\|\mathbf{x}\|/s^2)$ is the probability centered at zero of any $x \in L$, where $s = \sigma\sqrt{2\pi}$. In our case, we set $L = \mathbb{Z}$ so that each integer is sampled randomly from $D_{\mathbb{Z},\sigma}$.

The tail of the sampled discrete Gaussian distribution should have a finite length so that every sampling algorithm can cover it. Therefore, it is necessary to choose a suitable tail-cut factor $t > 0$ to determine the range of sampled values. Note that the sampled discrete Gaussian distribution differs from normal distribution because of the tail bound. The tail bound is closely related to the maximum statistical distance allowed by the security parameters discussed in [38].

Sampling from a target discrete Gaussian distribution on Java Card is a big challenge due to its specification (these features will be discussed in Section 3). Computing the probabilities is a resource-consuming operation that depends on the floating-point capabilities of the devices. In general, the high precision floating-point operation or large storage is required as explained in [9]. Therefore, setting the suitable precision is very significant. In our case, the random values are pre-computed and stored in particular ways to reduce computation cost and storage, and a detailed introduction of discrete Gaussian sampling will be discussed in Section 4.1.

2.2 Ring-LWE based encryption scheme

The definition of the ring-LWE problem in [26] is similar to the original LWE problem presented by Oded Regev in [36] firstly. Let a polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$, called R_q an ideal lattice if each polynomial in R_q has a bijective mapping to an ideal in $\mathbb{Z}_q^n$. Given a pair of polynomials $(\boldsymbol{a}, \boldsymbol{b}) \in R_q \times R_q$, the search version of the ring-LWE problem is to find the secret $\boldsymbol{s} \in R_q$, where $\boldsymbol{a}$ is chosen uniformly and $\boldsymbol{b} = \boldsymbol{as} + \boldsymbol{e}$ with an "error" $\boldsymbol{e}$ generated from a target probability distribution. The decision version can be defined as follows: given a pair $(\boldsymbol{a}, \boldsymbol{b}) \in R_q \times R_q$ where $\boldsymbol{a}$ is chosen uniformly, we distinguish whether $\boldsymbol{b}$ is also chosen uniformly, or there exists a polynomial $\boldsymbol{s} \in R_q$ such that $\boldsymbol{b} = \boldsymbol{as} + \boldsymbol{e}$. In the worst-case, the ring-LWE problem can be reduced to approximate shortest vector problem (α-SVP) on ideal lattices.

Here, we investigate the ring-LWE based encryption scheme [26] which is more efficient than the Regev's LWE schemes [30,36]. It uses constructed polynomials over R_q instead of the structured integer matrices, which is based on the hardness of solving the ring-LWE problem with IND-CPA (short for indistinguishability against chosen plaintext attacks) security by appropriately parameterized ideal lattices. In this paper, we refer to [26] as LPR-LWE.

Let Σ be a message alphabet, the message encoder and decoder are a pair of functions within a certain tolerance, that is, *encode*: $\Sigma^n \rightarrow R_q$ and *decode*: $R_q \rightarrow \Sigma^n$, such that $decode(encode(\boldsymbol{m}) + \boldsymbol{e}) = \boldsymbol{m}$ is an anti-operation of encoding for any "small enough" error $\boldsymbol{e} \in R_q$. $D_{L,\sigma}$ denotes a discrete Gaussian distribution as mentioned earlier. The following procedures define LPR-LWE encryption scheme:

Key generation:

Sample $\boldsymbol{e} \leftarrow D_{L,\sigma}$; choose a "small" random polynomial $\boldsymbol{s} \in R_q$ and a uniformly random polynomial $\boldsymbol{a} \in R_q$, then compute $\boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e} \in R_q$. The public key is the pair ($\boldsymbol{a}$, $\boldsymbol{b}$) and the secret key is $\boldsymbol{s}$.

Encryption:

Choose a "small" random polynomial $\boldsymbol{t} \in R_q$, sample $\boldsymbol{e}_1$, $\boldsymbol{e}_2 \leftarrow D_{L,\sigma}$. Given a plaintext $\boldsymbol{m} \in \{0,1\}^n$, let $\bar{\boldsymbol{m}} = \mathit{encode}(\boldsymbol{m}) \in R_q$; compute $\boldsymbol{c}_1 = \boldsymbol{a} \cdot \boldsymbol{t} + \boldsymbol{e}_1 \in R_q$ and $\boldsymbol{c}_2 = \boldsymbol{b} \cdot \boldsymbol{t} + \boldsymbol{e}_2 + \bar{\boldsymbol{m}} \in R_q$. The ciphertext is the pair ($\boldsymbol{c}_1$, $\boldsymbol{c}_2$).

Decryption:

Output $\mathit{decode}(\boldsymbol{c}_2 - \boldsymbol{c}_1 \cdot \boldsymbol{s}) \in \{0,1\}^n$.

Table 1: The selected Ring-LWE parameters works on Java Card

	n	q	s	$\lceil \log_2(q) \rceil$	bit sec
dimension 128	128	3329	8.62	12	≥ 80
dimension 256	256	7681	11.31	13	≥ 128

In our case, we select the parameters from [15] in Table 1, corresponding to medium-level security (AES-128) for $n = 256$ the least (see [15,25]). The primary reason is that those parameters are not too large, and suitable for implementation using NTT on Java Card. We could find that the size of the modulus q is far less than that of the 2048-bit. Note that the concrete quantum security levels of those parameters are still debatable, depending on the secure estimation algorithms (e.g., see [44,45,46,47]).

3 Java Card platform specification

In this section, we introduce some general information and features about the specification of the standard Java Card platform. Throughout the paper, we use the Java Card Platform Specification 2.2.2 [40] (Classic Edition) and GlobalPlatform Card Specification 2.1.1 [17] as our standard.

A typical Java Card has the same shape and size of a credit card, i.e., an integrated circuit is embedded in a plastic card. Java Card is a small device with limited capacity and processing power, and its physical characteristics are defined by `ISO/IEC 7816-1` standard [19]. Java Card applets execute on the Java Card virtual machine (JCVM) which is a subset of the Java virtual machine (JVM), thereby achieves hardware-independent compatibility and reduces costs.

There are three types of memories presented by Java Card, random access memory (RAM), read-only memory (ROM), and non-volatile memory (NVM),

Table 2: Features of Java Card Platform Specification 2.2.2

Supported	Unsupported
`boolean`, `byte`, `short` (optionally `int`)	`float`, `double`, `char`, `String`
one-dimensional arrays	multi-dimensional arrays
packages, interfaces, exceptions, access scopes, abstract methods, inheritance, overloading	object cloning, multithreading, GC, finalization, dynamic class loading, security manager

of which electrically erasable programmable read-only memory (EEPROM or E2PROM) and Flash are the two most common NVM. Only a section of RAM and NVM is available for the user. EEPROM allows code and data to be read, erased, and rewritten individually; hence it could be used to save the applets and data. RAM has a faster processing speed with unlimited use times in Java Card, especially writing data to RAM is about 1000 times faster than writing to EEPROM, that makes it suitable for allocating transient objects. Ordinarily, the available RAM (up to about 10 KBytes) is less than NVM, but it is better suited to temporary process data. In our implementation, the run-time data are all stored in RAM, rather than EEPROM, that means we can update those data quite fast in the process of responding.

To execute applets on smart cards, the programming language and JCVM are reduced to fit into the smart card configuration. Table 2 shows an overview of the supported and unsupported Java Card language features. There are two difficulties for implementation of LPR-LWE because of those features. Firstly, single and double-precision floating-point types (including `float` and `double`) and garbage collection (GC) are not available within JCVM. Floating-point arithmetic is required for the offline or online phase of discrete Gaussian sampling methods, but it is inefficient at simulating floating-point arithmetic using 8-bit or 16-bit signed integers on Java Card. In addition, without garbage collector feature, we must pay attention to the memory usage when creating look-up tables (LUT) or initializing objects. Secondly, the standard Java Card platform only supports small integer types such as `byte` or `short`. Even the 32-bit signed integer type `int` is optional, the vast majority of smart card manufacturers do not support this feature. In view of the size of the moduli we shown in Table 1, it is difficult to deal with efficient integer modular multiplication on the standard Java Card platform. Although the cryptographic co-processors of Java

Card perform integer calculations for RSA internally, they only offer limited execution functionality (see [41]) which does not support modular multiplication and polynomial operations used in the ring-LWE encryption scheme.

For some lattice-based cryptography in the matrix form such as [13,25,30,36], although the matrices can be converted to one-dimensional arrays for multiplication, the sizes of those matrices are too large to hold in RAM or EEPROM when the security level reaches a predetermined level, e.g., 128-bit security. Polynomials, by contrast, can be stored in memory using one-dimensional coefficient arrays, and the required key sizes are also small (e.g., see [42]). Therefore, ring-LWE based cryptography is suited to implement on memory-constrained devices such as Java Card.

For the standard Java Card platform, a message can be represented as bitstream or binary data. Hence, we use a binary array to express a plaintext in our implementation, and this allows the ring-LWE scheme to perform bit-wise text encryption.

4 Implementation techniques of ring-LWE scheme on Java Card

In this section, we describe our implementation techniques for discrete Gaussian sampling, improved modular multiplication, and efficient polynomial multiplication on Java Card.

4.1 Discrete Gaussian sampling

High precision floating-point arithmetic is essential for discrete Gaussian sampling to ensure the security and accuracy. However, computations of values with high precision are time-consuming because of the specification of Java Card. A lower precision in sampling process provides a higher efficiency but not secure and accurate enough. It is a challenge to balance the security and efficiency at the same time on Java Card. A number of techniques for discrete Gaussian sampling, such as the discrete Ziggurat algorithm in [3], the Knuth-Yao algorithm in [11,23], and Karney's algorithm [22] have been proposed and adopted to some different environments or devices. Since its simplicity and without floating-point arithmetic, it is likely to apply Karney's algorithm on Java Card in the future. In this subsection, we introduce the way of dealing with high precision floating-point numbers for discrete Ziggurat algorithm and Knuth-Yao algorithm.

Algorithm 1: Discrete Ziggurat sampling algorithm on Java Card (DZ)

```
Input: $m, n, l, t \in \mathbb{Z}, \sigma \in \mathbb{R}$, a probability array $\mathbf{p} = (\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{n-1})$
    $\in \mathbb{Z}_2^{n(l+1)}$, a $y$-coordinate value array $\mathbf{y} = (\mathbf{y}_0, \mathbf{y}_1, ..., \mathbf{y}_m)$
    $\in \mathbb{Z}_2^{(m+1)(l+1)}$, a $x$-coordinate value array $\mathbf{x} = (x_0, x_1, ..., x_m)$
    $\in \mathbb{Z}^{m+1}$, and an array $\mathbf{r} = (\mathbf{r}_0, \mathbf{r}_1, ..., \mathbf{r}_{m-1}) \in \mathbb{Z}_2^{m(l+1)}$.
Output: Sample value $s \in \mathbb{Z} \cap [-t\sigma, t\sigma]$
Let $\mathbf{a} = (a_0, a_1, ..., a_l) \in \mathbb{Z}^{l+1}$, $\mathbf{b} = (b_0, b_1, ..., b_l) \in \mathbb{Z}^{l+1}$, and
$\mathbf{c} = (c_0, c_1, ..., c_l) \in \mathbb{Z}^{l+1}$;
while true do
  $i \leftarrow \{1, 2, ..., m\}$ uniformly at random; $sign \leftarrow \{-1, 1\}$ uniformly at
  random; $x \leftarrow \{0, ..., x_i\}$ uniformly at random;
  if $0 < x \leq x_{i-1}$ then return $s = sign * x$
  else
    if $x = 0$ then $d \leftarrow \{0, 1\}$ uniformly at random;
    else
      for $j = 0$ to $l$ by 1 do
        $a_j = \mathbf{y}_i[j]$;
      for $j = 0$ to $l$ by 1 do
        $b_j = \mathbf{r}_{(i-1)}[j]$;
      $\mathbf{c} = generate(\mathbf{b})$;
      for $j = 0$ to $l$ by 1 do
        $b_j += a_j + c_j$;
        if $b_j > 1$ then
          $b_j -= 2; b_{j-1} += 1$;
      for $j = 0$ to $l$ by 1 do
        $a_j = \mathbf{p}_x[j]$;
      for $j = 0$ to $l$ by 1 do
        if $b_j > a_j$ then return $s = sign * x$
        else continue
    if $d = 0$ then return $s = sign * x$;
    else continue
```

Algorithm 2: Optimized binary number generator

```
Input: $l \in \mathbb{Z}$, an array $\mathbf{c}' = (c'_0, c'_1, ..., c'_l) \in \mathbb{Z}_2^{l+1}$
Output: An array $\mathbf{a}' = (a'_0, a'_1, ..., a'_l) \in \mathbb{Z}_2^{l+1}$
for $t = 0$ to $c'_0 - 1$ by 1 do
    $a'_t = 0$;
while true do
    for $i = c'_0$ to $l$ by 1 do
        $a'_i \leftarrow \{0, 1\}$ uniformly at random;
    for $j = 0$ to $l$ by 1 do
        if $a'_j < c'_j$ then
            return $\mathbf{a}'$;
        else if $a'_j > c'_j$ then
            break;
    if $a'_l = c'_l$ then return $\mathbf{a}'$
```

Discrete Ziggurat algorithm Some sampling methods, with large look-up tables or floating-point calculations, are less feasible to be implemented in memory-constrained platforms. Nevertheless, the discrete Ziggurat sampling method proposed in [3] allows for a flexible time-memory trade-off.

Let $t > 0$ be the tail-cut factor, for a real $\sigma > 0$, the sampled value is chosen uniformly at random from $\{-t\sigma, ..., t\sigma\}$. Let $m \in \mathbb{Z}$ be the number of horizontal rectangles which cover the probability density function (PDF) of the target probability distribution. More memory could be saved by sampling an integer $x \in \mathbb{Z} \cap [-t\sigma, t\sigma]$ since the target discrete Gaussian distribution is symmetric. If $x = 0$, we accept with probability $1/2$, otherwise a sign $s \in \{-1, 1\}$ is generated uniformly at random and return sx. Firstly, discrete Ziggurat method chooses a rectangle, then a point (x, y) with $x \in \mathbb{Z}, y \in \mathbb{R}$ is sampled inside the chosen rectangle [3,9]. According to the location of that point inside the rectangle, either x is accepted as a sampled value, or be rejected.

Floating-point arithmetic is unsupported because of the features of the Java Card Platform Specification 2.2.2 (see Section 3), hence the floating-point number needs to be converted into its binary expansion with a finite precision. Let $l \in \mathbb{Z}^+$ be the precision of the binary expansion of the floating-point number. Both integral and fractional part of the floating-point number has to be stored into a look-up table.

Apparently, the probability in floating-point form of any sampled value $x \in \mathbb{Z} \cap [0, t\sigma]$ is greater than 0. However, the binary expansions with a finite precision of some probabilities equal to 0, so that it's not necessary to store them in a look-up table. Let $n \in \mathbb{Z}$, there are n binary expansions of the probabilities $\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{n-1} \in \mathbb{Z}^{l+1}$. $x_0, x_1, ..., x_m \in \mathbb{Z}$ are the values of x-coordinates, and $\mathbf{y}_0, \mathbf{y}_1, ..., \mathbf{y}_m \in \mathbb{Z}^{l+1}$ are the values y-coordinates with $(l+1)$-bit precision of m rectangles. For $i \in \mathbb{Z} \cap [0, m)$, let $\mathbf{r}_i = \mathbf{y}_i - \mathbf{y}_{i+1} \in \mathbb{Z}^{l+1}$ be the difference of neighboring y-coordinates. Note that the value of y-coordinate of the vertex is not less than one in order to cover the PDF.

Floating-point arithmetic is infeasible on Java Card due to limited memory and computing power. In addition, the multi-dimensional array cannot adapt to this environment. Therefore, the floating-point number is converted to its binary expansion and a multi-dimensional array is transformed to a one-dimensional array in our case. We modified the discrete Ziggurat sampling on Java Card as written in Algorithm 1. In Step 12 of Algorithm 1, the supported method *generate*() generates a $(l+1)$-bit precision binary number less than or equal to the input value. In consideration with the problem of time-consuming in this process, we could improve the efficiency by optimizing the leading zeros for every $\mathbf{r}_i$ as shown in Algorithm 2.

When an array is inputted, Algorithm 2 searches the numbers of leading zeros and inserts them directly into the generated array. This method will increase the speed of discrete Ziggurat algorithm remarkably. At the beginning of each $\mathbf{r}_i$, only a 1-bit number is required to represent the numbers of leading zeros.

Knuth-Yao algorithm Donald Ervin Knuth and Andrew Chi-Chih Yao proposed an algorithm which aims to sample values from the non-uniform distribution in [23]. Precomputation has to be executed in Knuth-Yao algorithm. Similar to discrete Ziggurat algorithm, floating-point number of the probabilities should be converted to its binary expansion with finite precision. Note that multi-dimensional array is unsupported on Java Card. Therefore, we stored these probabilities in a one-dimensional array as the look-up table.

Let $l \in \mathbb{Z}$ be the precision of the binary expansion of the probabilities, and $n \in \mathbb{Z}$, there are n binary probabilities $\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{n-1} \in \mathbb{Z}^l$. A probability matrix $\mathbf{P}_{mat}$ is composed of all the probabilities, and each row stores one probability. Whereas numbers could only be stored in one-dimensional arrays on Java Card, while Knuth-Yao algorithm samples value by scanning numbers from the bottom of one column at each time as shown in [38]. When applied to Java Card, the probability matrix $\mathbf{P}_{mat}$ is transposed and divided into l blocks. Each block is one of the columns of $\mathbf{P}_{mat}$. Let $\mathbf{k}_0, \mathbf{k}_1, ..., \mathbf{k}_{l-1} \in \mathbb{Z}_2^n$ be all of the blocks. An array $\mathbf{k} = (\mathbf{k}_0, \mathbf{k}_1, ..., \mathbf{k}_{l-1}) \in \mathbb{Z}_2^{ln}$ is stored in a look-up table for Algorithm 3. Similar to discrete Ziggurat algorithm, it uses less memory if the algorithm only stores the probabilities of sampled value $x \in \mathbb{Z} \cap [0, t\sigma]$ (see Figure 1).

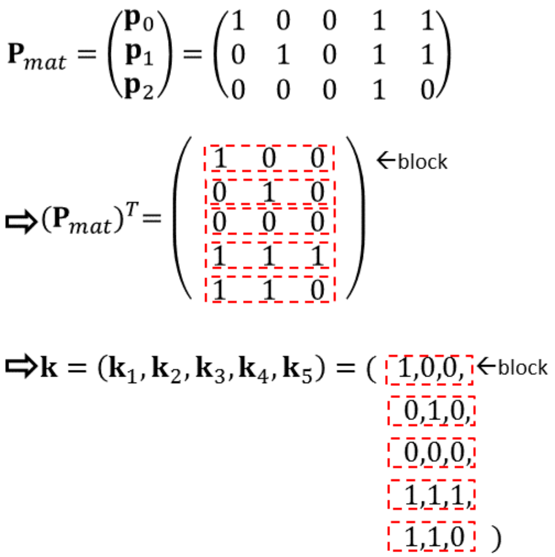
$\mathbf{P}_{mat} = \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$
$\Rightarrow (\mathbf{P}_{mat})^T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$
←block
$\Rightarrow \mathbf{k} = (\mathbf{k}_1, \mathbf{k}_2, \mathbf{k}_3, \mathbf{k}_4, \mathbf{k}_5) = ($ 1,0,0, 0,1,0, 0,0,0, 1,1,1, 1,1,0 $)$
←block

Fig. 1: Probability array $\mathbf{k}$ including l blocks

Note that the scale of the look-up table should be as less as possible because of limited memory in Java Card. It is clear that there are many zeros in probability array $\mathbf{k}$ and these zeros can be compressed as in [11].

Figure 2 shows that by "deleting" zeros in grey areas, less memory can be used. Similar to Algorithm 3, $\mathbf{P}_{mat}$ is divided into l blocks $\mathbf{k}'_0, \mathbf{k}'_1, ..., \mathbf{k}'_{l-1}$. Let $q \in \mathbb{Z}$ represents the numbers of the first several blocks whose elements are all zeros. After the optimization of zeros, the length differences between two consecutive blocks can be marked with 1 or 0 [38]. A 0 is inserted as a sign bit at the front of the 0-th block firstly. Next, the length of 0-th block with its next neighboring block is compared. Then a sign bit 0 for no-increment or 1 for an increment by one is inserted. Hence, all sign bits for every block can be obtained by this method. As shown in Algorithm 4, $h \in \mathbb{Z}$ is the total length of all blocks which have been scanned, and $g \in \mathbb{Z}$ is the length of probability array $\mathbf{k}' = (\mathbf{k}'_0, \mathbf{k}'_1, ..., \mathbf{k}'_{l-1}) = (k'_0, k'_1, ..., k'_{g-1})$. This method consumes less time because these zeros can be skipped when scanning the look-up table.

In Step 1, 19, and 20 of Algorithm 4, $FirstBlockLength$ means the length of the first block in the optimized probability array.

Algorithm 3: Knuth-Yao Sampling algorithm on Java Card (`KY`)

```
Input: $l, n \in \mathbb{Z}$, a probability array $\mathbf{k} = (\mathbf{k}_0, \mathbf{k}_1, ..., \mathbf{k}_{l-1}) \in \mathbb{Z}_2^{ln}$
Output: Sample value $s \in \mathbb{Z} \cap [-t\sigma, t\sigma]$
Let $d = 0$, $x = 0$, $sign = 0$ ;
  while true do      $r \leftarrow \{0, 1\}$ uniformly at random;
    $d = 2d + r$;
    for $i = n$ down to 0 by 1 do        $d = d - \mathbf{k}_x[i]$;
      if $d = -1$ then
        if $i = 0$ then $sign \leftarrow \{0, 1\}$ uniformly at random;
        else
          $sign \leftarrow \{-1, 1\}$ uniformly at random;
          return $s = sign * row$;
        if $sign = 1$ then return $s = i$;
        else
          $d = 0$;
          $r \leftarrow \{0, 1\}$ uniformly at random;
          $d = 2d + r$;
          $x = 0$;
          continue
    $x += 1$;
```

4.2 Montgomery modular multiplication

Implementation of multiplications on small devices, especially on Java Card, is a difficult task that not only due to its constrained memory, but also because of its limited computational capacity. Basically, according to the Java Card Platform Specification 2.2.2, only signed small integer types (`short, byte`) are supported that are represented in 2's complement form in the most common cards, which means the signed integer larger than 15 bits is unavailable. However, choosing a modulus q from the Table 1 and then multiplying two positive coefficients a and $b \in \mathbb{Z}_q$, the multiplication will be probably out of bounds of the signed `short` integer. Even worse, the running time of integer operations on Java Card is quite slow, for instance, it takes about half of a second for 10000 times `byte` integer multiplication on our experimental Java card. Therefore, we need to find a way to achieve polynomial multiplications which support small integer computation.

Montgomery modular multiplication (MMM) is a quick way to compute modular product, replacing that division with an exact multiplication. Koç et al. [21] analyzed several different MMM methods including Separated Operand Scanning method (SOS), Coarsely Integrated Operand Scanning method (CIOS), Finely

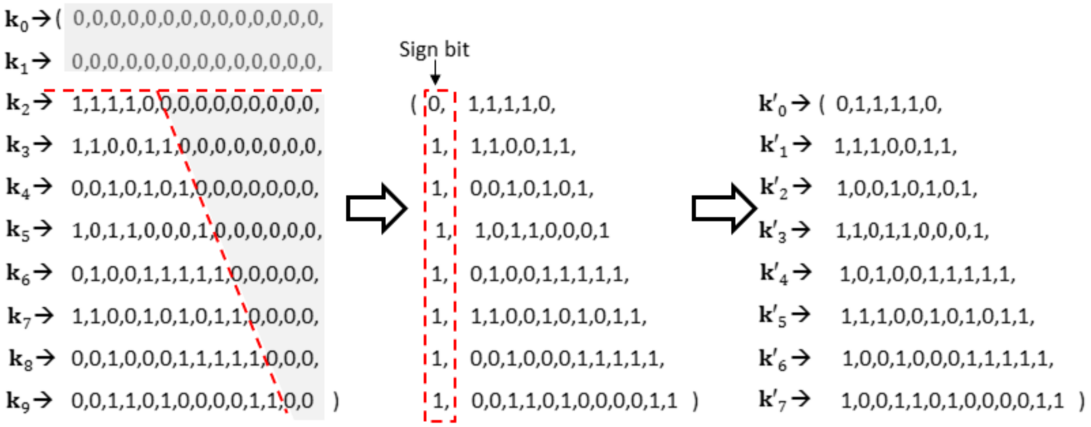
k0→(0,0,0,0,0,0,0,0,0,0,0,0,0,0,
k1→ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,
k2→ 1,1,1,1,0,0,0,0,0,0,0,0,0,0,
k3→ 1,1,0,0,1,1,0,0,0,0,0,0,0,0,
k4→ 0,0,1,0,1,0,1,0,0,0,0,0,0,0,
k5→ 1,0,1,1,0,0,0,1,0,0,0,0,0,0,
k6→ 0,1,0,0,1,1,1,1,1,0,0,0,0,0,
k7→ 1,1,0,0,1,0,1,0,1,1,0,0,0,0,
k8→ 0,0,1,0,0,0,1,1,1,1,1,0,0,0,
k9→ 0,0,1,1,0,1,0,0,0,0,1,1,0,0)
Sign bit
(0, 1,1,1,1,0,
1, 1,1,0,0,1,1,
1, 0,0,1,0,1,0,1,
1, 1,0,1,1,0,0,0,1
1, 0,1,0,0,1,1,1,1,1,
1, 1,1,0,0,1,0,1,0,1,1,
1, 0,0,1,0,0,0,1,1,1,1,1,
1, 0,0,1,1,0,1,0,0,0,0,1,1)
k'0→(0,1,1,1,1,0,
k'1→ 1,1,1,0,0,1,1,
k'2→ 1,0,0,1,0,1,0,1,
k'3→ 1,1,0,1,1,0,0,0,1,
k'4→ 1,0,1,0,0,1,1,1,1,1,
k'5→ 1,1,1,0,0,1,0,1,0,1,1,
k'6→ 1,0,0,1,0,0,0,1,1,1,1,1,
k'7→ 1,0,0,1,1,0,1,0,0,0,0,1,1)

Fig. 2: Optimized probability array by deleting zeros in grey areas, the sign bit indicates the difference between two consecutive block lengths

Integrated Operand Scanning method (FIOS), Finely Integrated Product Scanning method (FIPS) and Coarsely Integrated Hybrid Scanning method (CIHS). Some literature such as [14,28,29,32] reported the implementation results for MMM methods on different platforms. However, according to their implementation, some local variables in calculation might be still out of the signed `short` integer boundary using the parameters in Table 1; hence, those methods cannot be used in such memory-constrained device. To fit modular multiplication into the standard Java Card platform, we adopted an improved algorithm in [2] that we refer to as `R-MMM`, which can compute radix-2^{ω} MMM by only using s blocks for n-bit multipliers.

The key factor of Algorithm 5 to note here is that we compute the inverse M_0' of the block M_0 instead of the modulus M. Hence, the advantage of our implementation is that only the products of two ω-bit blocks will be computed, so that we can select the suitable size for blocks to make sure the products would not overflow. In Step 5 and 8, the maximum size of the product is 2ω bits. When a signed `short` integer splits into two signed `byte` integers, the low 8 bits might be a negative `byte` integer. For the convenience of calculations, we convert the negative `byte` integer to a positive `short` integer. Even if the modulus M has increased, we only need to add additional blocks. Therefore, Algorithm 5 guarantees its correctness throughout the entire process without overflowing any variables. We define that every multiplier is an 8-bit integer,

Algorithm 4: Optimized Knuth-Yao Sampling algorithm on Java Card (`KYO`)

```
Input: $g, q \in \mathbb{Z}$, an optimized probability array $\mathbf{k}' = (k'_0, k'_1, ..., k'_{g-1}) \in \mathbb{Z}_2^g$
Output: Sample value $s \in \mathbb{Z} \cap [-t\sigma, t\sigma]$
Let $d = 0$, $len = FirstBlockLength$, $sign = 0$, $h = 0$ for $j = 0$ up to $q - 1$ by 1 do
    $r \leftarrow \{0, 1\}$ uniformly at random;
    $d = 2d + r$;
    while true do
        $r \leftarrow \{0, 1\}$ uniformly at random;
        $d = 2d + r$;
        for $i = len - 1$ down to 0 by 1 do
            $d = d - k'_{i+1+sum}$;
            if $d = -1$ then
                if $i = 0$ then $sign \leftarrow \{0, 1\}$ uniformly at random;
                else
                    $sign \leftarrow \{-1, 1\}$ uniformly at random; return $s = sign * i$;
                if $sign = 1$ then return $s = i$;
                else
                    $d = 0$;
                    $r \leftarrow \{0, 1\}$ uniformly at random;
                    $d = 2d + r$;
                    $len = FirstBlockLength$;
                    $sum = -FirstBlockLength - 1$;
                    continue
        $h = h + len + 1$ ;
        if $k'_h = 1$ then
            $len += 1$;
```

i.e., $\omega = 8$ so that $b = 2^8$, and $R = b^2 = 2^{16}$, then we can pre-compute M'_0 and the inverse R^{-1}.

4.3 Number theoretic transform

We investigated several efficient approaches including widely used Karatsuba algorithm and number theoretic transform for implementation of polynomial multiplication on Java Card. Karatsuba algorithm is an efficient approach for polynomial multiplication with lower asymptotic complexity $O(n \log n)$. Multiplying recursively in some programming languages is not a big issue, but at the same time, too many temporary arrays will be created during the recursion

Algorithm 5: Radix-2^{ω} Montgomery modular multiplication (R-MMM)

```
Input: n-bit integers $M = (M_s, ..., M_0)_b$, $X = (X_{s-1}, ..., X_0)_b$,
       $Y = (Y_{s-1}, ..., Y_0)_b$ where $0 \leq X, Y \leq M$, $b = 2^{\omega}$, $s = \lceil n/\omega \rceil$, $R = b^s$
       with $gcd(M, b) = 1$ and $M'_0 = -M_0^{-1} \bmod b$
Output: Non-negative integer $A = X * Y * R^{-1} \bmod M$
Let $A = 0 = (A_s, ..., A_0)_b$;        // in our case, we set $n = 16, \omega = 8$;
for $i = 0$ to $s - 1$ do
    $temp = 0$;
    for $j = 0$ to $s - 1$ do
        $(temp, A_j) = X_j * Y_i + A_j + temp$;
    $A_s = temp$, $temp = 0$, $\mu_i = A_0 * M'_0 \bmod b$;
    for $j = 0$ to $s$ do
        $(temp, A_j) = M_j * \mu_i + A_j + temp$;
    $A = A/b$;
if $A \geq M$ then $A = A - M$;
return $A$.
```

process. For example, we have tried Karatsuba algorithm on Java Card using the parameters in Table 1, and the running time of decryption is more than 500 seconds for $n = 64$. We also have rewritten Karatsuba algorithm into the non-recursive form. However, its computing speed is still slow, and the number of temporary arrays could not be reduced greatly.

Therefore, we follow the state-of-the-art and use the NTT which has asymptotic complexity $O(n \log n)$ for multiplying polynomials with higher degrees. The NTT is similar to the fast Fourier transform (FFT), but transforms over $\mathbb{Z}_q$ instead of the complex numbers $\mathbb{C}$. In our previous work [42,43], we implemented a non-recursive forward number theoretic transform (NTT, see [8,37]) for ring-LWE based encryption schemes. For polynomial multiplication $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b} \in R_q$, it is required to compute the inverse ($\mathtt{NTT}^{-1}$), so that $\mathtt{NTT}^{-1}(\mathtt{NTT}(\boldsymbol{f})) = \boldsymbol{f}$ for all $\boldsymbol{f} \in R_q$, thus the multiplication $\boldsymbol{c} = \mathtt{NTT}^{-1}(\mathtt{NTT}(\boldsymbol{a}) \otimes \mathtt{NTT}(\boldsymbol{b}))$, where $\otimes$ denotes component-wise multiplication.

More efficient approaches of NTT are presented in [24,35], which focus on the optimization of NTT's bit-reverse computation, as shown in Algorithm 6 and Algorithm 7. Compared with our previous NTT implementation, Algorithm 6 and Algorithm 7 do not increase the length of the polynomial multipliers. We can pre-compute a table of the bit-reversed order to reduce the running time and the cost of RAM.

For a prime integer $q \equiv 1 \pmod{2n}$, where n is a power of 2, let ω be the n-th primitive root of unity over $\mathbb{Z}_q$, and $\psi = \sqrt{\omega}$ such that ψ is the $2n$-th

Algorithm 6: Cooley-Tukey forward number theoretic transform (`CT-NTT`)

```
   Input: Polynomial $\boldsymbol{a} \in R_q = \mathbb{Z}_q[x]/(x^n + 1)$, and a LUT $\mathit{\Psi}_{rev} \in \mathbb{Z}_q^n$ in
          bit-reversed order
   Output: Polynomial $\boldsymbol{a}' = $ CT-NTT $(\boldsymbol{a}) \in R_q$
 1 $t = n$;
 2 for $m = 1$ to $n - 1$ by $m = 2m$ do
 3     $t = t/2$;
 4     for $i = 0$ to $m - 1$ do
 5         $j_1 = 2 * i * t$;
 6         $j_2 = j_1 + t - 1$;
 7         $S = \mathit{\Psi}_{rev}[m + i]$;
 8         for $j = j_1$ to $j_2$ do
 9             $U = \boldsymbol{a}[j]$;
10             $V = $ R-MMM $(q, \boldsymbol{a}[j + t], S)$;
11             $\boldsymbol{a}[j] = U + V \bmod q$;
12             $\boldsymbol{a}[j + t] = U - V \bmod q$;
13 return $\boldsymbol{a}$.
```

primitive root of unity over $\mathbb{Z}_q$. We write two polynomials $\boldsymbol{f} = (f_0, f_1, ..., f_{n-1})$ and $\bar{\boldsymbol{f}} = (f_0, \psi f_1, ..., \psi^{n-1} f_{n-1}) \in R_q$. First, we compute all $2n$ powers of ψ and ψ^{-1}, and then store n powers of ψ and ψ^{-1} with the bit-reversed order in look-up tables $\mathit{\Psi}_{rev}$, $\mathit{\Psi}_{rev}^{-1} \in \mathbb{Z}_q^n$, respectively. All the coefficients of input polynomials are in the standard order for Algorithm 6, and in bit-reversed order for Algorithm 7, so the bit-reverse operation for input polynomial can be merged into pre-computation. Then one gets the output the negative wrapped convolution $\boldsymbol{c} = (1, \psi^{-1}, ..., \psi^{-(n-1)}) \otimes \texttt{GS-INTT}(\texttt{CT-NTT}(\bar{\boldsymbol{a}}) \otimes \texttt{CT-NTT}(\bar{\boldsymbol{b}}))$, and $\boldsymbol{c}$ is the polynomial multiplication.

We combine `R-MMM` and number theoretic transform, which computes the modular multiplication in Step 10 of Algorithm 6 and Step 11, 15 of Algorithm 7. Modular multiplication is the most time-consuming operation in NTT executed on Java Card. Therefore, we should check whether the multiplication is in bounds firstly, and then determine to use the suitable modular reduction method. In our implementation, the performance of integer and polynomial multiplication is significantly improved on Java Card, e.g., the running time of decryption ($n = 256$) is more than 10 times faster than our previous result, and costs only about 7 seconds in decryption for 128-bit security level. We show the performance result in the next section.

Algorithm 7: Gentleman-Sande inverse of number theoretic transform (`GS-INTT`)

```
Input: Polynomial $\boldsymbol{b} \in R_q = \mathbb{Z}_q[x]/(x^n + 1)$, and a LUT $\Psi_{rev}^{-1} \in \mathbb{Z}_q^n$ in
       bit-reversed order
Output: Polynomial $\boldsymbol{b}' =$ GS-INTT $(\boldsymbol{b}) \in R_q$
$t = 1$;
for $m = n$ to 2 by $m = m/2$ do
    $h = m/2, j_1 = 0$;
    for $i = 0$ to $h - 1$ do
        $j_2 = j_1 + t - 1$;
        $S = \Psi_{rev}^{-1}[h + i]$;
        for $j = j_1$ to $j_2$ do
            $U = \boldsymbol{b}[j]$;
            $V = \boldsymbol{b}[j + t]$;
            $\boldsymbol{b}[j] = U + V \bmod q$;
            $\boldsymbol{b}[j + t] =$ R-MMM $(q, (U - V), S)$;
        $j_1 = j_1 + 2t$;
    $t = 2t$;
for $i = 0$ to $n - 1$ do
    $\boldsymbol{b}[i] =$ R-MMM $(q, \boldsymbol{b}[i], n^{-1})$;
return $\boldsymbol{b}$.
```

5 Performance of ring-LWE scheme on Java Card

In this section, we report the experimental results of running discrete Gaussian sampling methods and LPR-LWE scheme on a standard Java Card (**JCOP v2.4.1 NXP J3A081 Dual Interface Card**: Java Card Version: 2.2.2; Global Platform: 2.1.1; ISO/IEC 7816, T=0, T=1 (kbit/s): 223.2; ISO/IEC 14443 T=CL (kbit/s): 848; available EEPROM Options KByte: 80; ROM (free for Applets, up to KBytes): 76; APDU Buffer (RAM/Bytes): 1462). In our case, the random values are generated by pre-computing using our improved discrete Gaussian sampling methods.

5.1 Performance results of discrete Gaussian sampling on Java Card

We implement the Knuth-Yao algorithm and discrete Ziggurat algorithm with and without optimization on Java Card. With the precision $l = 128$ bits, it is infeasible to use large parameters because of the memory limitation. We choose the parameter sets $(n, s) \in \{(128, 8.62), (256, 11.31)\}$ same as in [15] to ensure

Table 3: Experimental comparison of discrete Gaussian sampling methods on Java Card for different parameter sets

Sampling method	Running time (s)		Storage (KBytes)	
	$(n, s) = (128, 8.62)$	$(n, s) = (256, 11.31)$	$(n, s) = (128, 8.62)$	$(n, s) = (256, 11.31)$
DZ [Alg.1]	11.01	24.49	22.40	24.29
DZO [Alg.2]	2.25	5.01	22.51	24.39
DZO-SP [Alg.2]	1.92	3.89	18.16	19.67
KY [Alg.3]	4.87	12.37	6.19	7.95
KYO [Alg.4]	1.15	2.47	4.32	5.48
KYO-SP [Alg.4]	1.11	2.41	3.29	4.24

the statistical distances for the discrete Ziggurat algorithm and the Knuth-Yao algorithm are at most 2^{-100} as in [5]. The number of rectangles is 63 for discrete Ziggurat algorithm [3]. To ensure the security and accuracy of discrete Gaussian sampling, the precision $l = 128$ bits for these algorithms is sufficient in our case. However, the memory size and computing power are limited according to the specification of Java Card. Under this circumstance, it is necessary to use a relatively less precision. Moreover, when a smaller look-up table comes true, larger parameters can be used on Java Card. According to Lemma 2 in [11], a more suitable precision can be used. For each algorithm with $n = 128$ (resp.256), we choose the precision $l = 104$ (resp.105) bits. As shown in Table 3, we compare the following sampling methods on Java Card: the original discrete Ziggurat (DZ, Algorithm 1)/Knuth-Yao algorithm (KY, Algorithm 3), the optimized discrete Ziggurat algorithm by using Algorithm 2 (DZO), and our optimized Knuth-Yao algorithm (KYO, Algorithm 4). Moreover, DZO-SP denotes the discrete Ziggurat algorithm with optimization using a suitable precision, and KYO-SP denotes the Knuth-Yao algorithm with optimization using a suitable precision. For each algorithm with $n = 128$ and 256, Table 3 shows the running time in seconds (s) and storage in kilobytes (KBytes).

Apparently, there are notable improvements by means of the appropriate optimizations for not only Knuth-Yao algorithm, but also discrete Ziggurat algorithm from Table 3.

Compared with KY, KYO takes less running time and memory consumption that is about 5 times faster than KY for any selected parameter sets. Compared with DZ, DZO shows better performance that is more than 4 times faster than DZ for any chosen parameter sets. It is clear that KYO-SP uses much less storage than KYO and DZO-SP has higher speed with smaller look-up tables than DZO.

Note that KY, KYO, and KYO-SP have better performances than DZ, DZO, and DZO-SP for both running time and memory consumption. The calculation of discrete Ziggurat algorithm can be sped up by two different ways. The first way is to optimize the time-consuming operation of generating random l-bit binary numbers such as Algorithm 2. It is clear that this technique improves the efficiency of discrete Ziggurat algorithm significantly. However, for any random l-bit binary number, there are at most $2^l - 1$ numbers which are not greater than it. It is difficult to store such scale of numbers on Java Card. Therefore, these numbers cannot be pre-determined. Another method could be increasing the number of rectangles. If there are more rectangles, the area covered by them will tighter enclose the area under the probability density function [3]. However, if a larger m is chosen, the additional values of coordinates have to be stored, increasing the memory requirements. Hence, the promotion of discrete Ziggurat algorithm is limited to a certain level. For these reasons, we noticed that DZ and DZO are much slower than KY and KYO on standard Java Card. On the other hand, Knuth-Yao algorithm has less precomputation and a smaller look-up table than discrete Ziggurat algorithm, because it only pre-computes the probabilities of all values in the specified range and stores their binary expansions into a probability array. Apart from this, discrete Ziggurat algorithm has to pre-compute the coordinates of all rectangles as well. Hence, Knuth-Yao algorithm is better than discrete Ziggurat algorithm for a memory-constrained device such as the standard Java Card platform.

5.2 Performance results of LPR-LWE on Java Card

In this subsection, we only report the performance results of key generation, encryption, and decryption of LPR-LWE without discrete Gaussian sampling, since we have generated all random values by pre-computing. We implement LPR-LWE scheme and generate key pairs by using parameters in Table 1.

Table 4 shows the performance results of LPR-LWE executed on a standard Java Card without using any additional co-processor. Compared to these performances, the running time of LPR-LWE with dimension 128 is less than half of that with dimension 256 due to the smaller size and modulus. The running time ratio for key generation, encryption and decryption is almost $1 : 2 : 1$. Furthermore, we can see that the current execution speed of LPR-LWE is much improved with performance acceleration of more than 10 times compared with our previous work [43]. The reason is that we refactor our code, pre-computing the bit-reversed permutation and keeping the number of temporary objects to a minimum in R-MMM and NTT computations.

Table 4: Performance results (seconds) of LPR-LWE on Java Card for different parameter sets (comparing with our previous work in [43])

	Key-Gen		Enc		Dec	
dimension 128	Previous [43]	Now	Previous [43]	Now	Previous [43]	Now
	44.748	4.887	89.762	10.685	45.012	3.682
dimension 256	Previous [43]	Now	Previous [43]	Now	Previous [43]	Now
	104.005	10.358	208.188	16.050	103.761	7.398
233-bit ECC [10]	--		≈ 11.618		≈ 5.845	
2048-bit RSA [10]	--		≈ 0.123		≈ 622.009	

Comparing the performance between ring-LWE scheme and other cryptosystems on the standard Java Card platform is quite difficult. Firstly, we had tried to choose the prevailing public-key encryption algorithms such as RSA or ECC for comparison with the ring-LWE scheme at the approximate security level of 128 bits. However, Java Card needs some additional cryptographic co-processors that support such public encryption algorithms as RSA. Hence, because of the limited hardware resources, Java Card only offers asymmetric cryptographic support for RSA up to 2048 bits (112-bit security) and ECC up to 320 bits (160-bit security). Therefore, we can only compare our implementation of ring-LWE scheme achieving 128-bit security to 2048-bit RSA and 233-bit ECC that achieve same security level (112-bit security). According to the implementation results in [10], encryption of 233-bit ECC (Menezes-Vanstone ECC) takes about 12 seconds, and decryption takes less than 6 seconds. Encryption of 2048-bit RSA is also faster that only takes about 0.1 seconds. However, its decryption is quite slower, which running time is more than 600 seconds, i.e., over 80 times slower than that of LPR-LWE scheme ($n = 256$). Even considering the differences of the test platform specifications and bit-security, decryption of LPR-LWE is still comparable to that of ECC, and much more efficient than that of RSA.

Secondly, no result of ring-LWE scheme executed on the standard Java Card platform is found in current literature. Authors in [5,6] presented the performance of the ring-LWE scheme and some authentication protocols on several smart cards including one kind of Java Card which is not standard. Their results show that the performance of lattice-based cryptography on smart card could be increased with good optimization and better hardware. However, in those papers, the authors solve modular reduction operations only for several special moduli but not for arbitrary parameters. In addition, some of the mod-

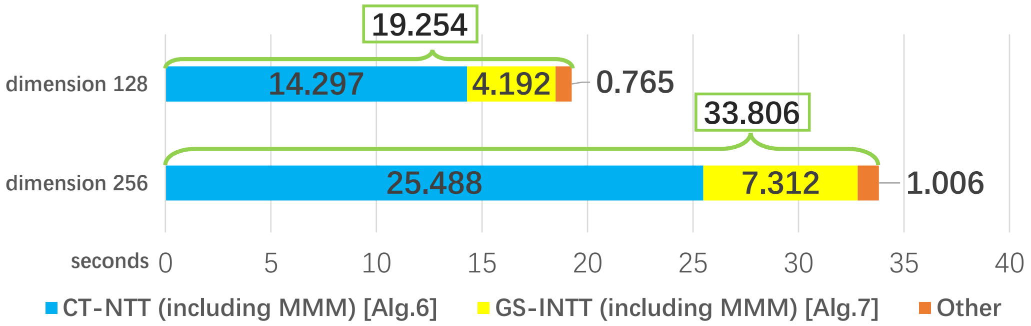

Fig. 3: Decomposition of the total running time (seconds) for different parameter sets

uli selected by them are much larger than 2^{15}, and we can find that in their source code they used the signed `int` instances and `int` arrays. That indicates their environment should support big integer operations, but it also means the backward compatibility and portability of the programs are probably dismal. The authors did not discuss these in their papers. Therefore, the performance of implementations executed on different smart card platforms are hard to be compared.

5.3 Detailed running time of LPR-LWE on Java Card

Figure 3 shows the decomposition of the total running time including key generation, encryption and decryption time for different parameter sets. Including many `R-MMM` loops in polynomial multiplication, we tested the execution time of `CT-NTT` and `GS-INTT`.

The running time of LPR-LWE with dimension 128 is less than half of that with dimension 256 due to the smaller parameter sizes. Compared with those methods, we noticed that polynomial multiplication is the single bottleneck computation that accounting for about 90% by summing both running time of `CT-NTT` and `GS-INTT`. In fact, most time is spent on `R-MMM` in `CT-NTT`/`GS-INTT`, and as the parameters increases, the running time of `R-MMM` loop within NTT increases too. As we have expected, in Figure 3, the running time required for polynomial multiplication is nearly equal to the total running time.

Figure 4 shows the proportions of computation time for different parameter sets. We see that though the running time using dimension 128 parameters only accounts for less than 50% by using dimension 256, the proportions of running time of each method are approximately equal. The results indicate that the execution time of LPR-LWE on Java Card is almost equivalent to the running time

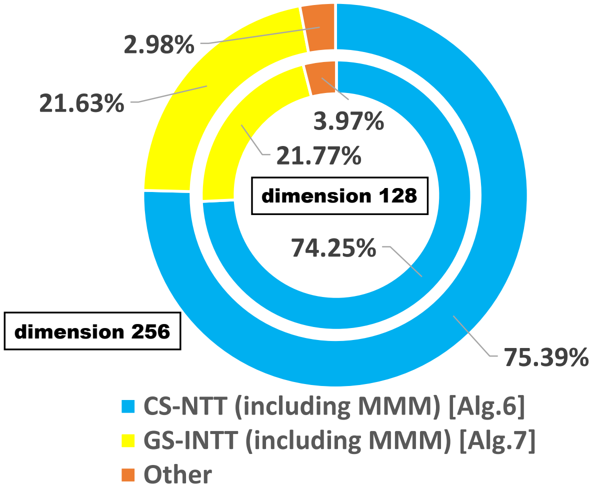

Fig. 4: The proportions of computation time for different parameter sets

of polynomial multiplication. Overall, as moduli diminish in size, the amount of modular multiplication within NTT will be reduced.

6 Conclusion

We implemented ring-LWE based encryption scheme for general arguments and successfully ran the applets on a standard Java Card, the performance of different parameter sets was compared. First, we came up with an approach to solve the problem of big integer multiplication and provided nearly optimal running time (about 7 seconds in decryption for 128-bit security) on the standard Java Card [43]. Then, we proposed the improving methods for discrete Ziggurat algorithm and Knuth-Yao algorithm implementing on Java Card, and the performance of our approaches are compared. We also have used number theoretic transform to improve polynomial multiplication speed and reduce the calculated amount, and the result shows that polynomial multiplication is the primary performance bottleneck. Furthermore, by combining NTT and improved MMM, our results indicate that it is possible to implement more lattice-based cryptosystems on such a memory-constrained device.

References

[1] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. "Post-quantum key exchange - a new hope." In *Proceedings of the 25th USENIX Security Symposium*, pp. 327–343, 2016.

[2] Toru Akishita and Tsuyoshi Takagi. "Power analysis to ECC using differential power between multiplication and squaring." In *Proceedings of the 7th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications*, LNCS, Vol. 3928, pp. 151–164, 2006.

[3] Johannes Buchmann, Daniel Cabarcas, Florian Göpfert, Andreas Hülsing, and Patrick Weiden. "Discrete Ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers." In *Proceeding of the 20th International Conference on Selected Areas in Cryptography – SAC 2013*, LNCS, Vol. 8282, pp. 402–417, 2013.

[4] Johannes Buchmann, Florian Göpfert, Tim Güneysu, Tobias Oder, and Thomas Pöppelmann. "High-performance and lightweight lattice- based public-key encryption." In *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security – IoTPTS '16*, pp. 2–9, 2016.

[5] Ahmad Boorghany and Rasool Jalili. "Implementation and comparison of lattice-based identification protocols on smart cards and microcontrollers." In IACR Cryptology ePrint Archive, Report 2014/078, 2014.

[6] Ahmad Boorghany, Siavash Bayat Sarmadi, and Rasool Jalili. "On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards." In *ACM Transactions on Embedded Computing Systems (TECS) – Special Issue on Embedded Platforms for Crypto and Regular Papers*, Vol. 14, Issue 3, No. 42, pp. 1–25, 2015.

[7] Jung Hee Cheon, Duhyeong Kim, Joohee Lee, and Yongsoo Song. "Lizard: Cut off the tail! Practical post-quantum public-key encryption from LWE and LWR." In IACR Cryptology ePrint Archive, Report 2016/1126, 2016.

[8] Ruan De Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. "Efficient software implementation of ring-LWE encryption." In *Proceedings of 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 339–344, 2015.

[9] Daniel Cabarcas, Patrick Weiden, and Johannes Buchmann. "On the efficiency of provably secure NTRU." In *Proceedings of 6th International Work-*

shop on Post-Quantum Cryptography – PQCrypto 2014, LNCS, Vol. 8772, pp. 22–39, 2014.

[10] Aaradhana A. Deshmukh, Manali Dubal, TR Mahesh, and CR Chauhan. "Data security analysis and security extension for smart cards using Java Card." In *International Journal of Advanced Information Technology (IJAIT)*, Vol. 2, No. 2, pp. 41–57, 2012.

[11] Nagarjun C. Dwarakanath and Steven D. Galbraith. "Sampling from discrete Gaussians for lattice-based cryptography on a constrained device." In *Applicable Algebra in Engineering, Communication and Computing*, Vol. 25, Issue 3, pp. 159–180, 2014.

[12] Léo Ducas and Phong Quang Nguyen. "Faster Gaussian lattice sampling using lazy floating-point arithmetic." In *Proceedings of the 18th International Conference on the Theory and Application of Cryptology and Information Security – ASIACRYPT 2012*, LNCS, Vol. 7658, pp. 415–432, 2012.

[13] Tore Kasper Frederiksen. "A practical implementation of Regev's LWE-based cryptosystem." Technical report, 2010. `http://daimi.au.dk/~jot2re/lwe/`

[14] Junfeng Fan, Kazuo Sakiyama, and Ingrid Verbauwhede. "Montgomery modular multiplication algorithm for multi-core systems." In *Proceedings of 2007 IEEE Workshop on Signal Processing Systems (SIPS)*, pp. 261–266, 2007.

[15] Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin Huss. "On the design of hardware building blocks for modern lattice-based encryption schemes." In *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems – CHES 2012*, LNCS, Vol. 7428, pp. 512–529, 2012.

[16] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. "Practical lattice-based cryptography: A signature scheme for embedded systems." In *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems – CHES 2012*, LNCS, Vol. 7428, pp. 530–547, 2012.

[17] GlobalPlatform. "GlobalPlatform Card Specification 2.1.1." 2003. `http://www.win.tue.nl/pinpasjc/docs/Card%20Spec%20v2.1.1%20v0303.pdf`

[18] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. "Trapdoors for hard lattices and new cryptographic constructions." In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing – STOC '08*, pp. 197–206, 2008.

[19] ISO/IEC 7816–1:2011. "ISO/IEC 7816–1: Cards with contacts – Physical characteristics." Updated in 2011. Preview: https://webstore.iec.ch/preview/info_isoiec7816-1{ed2.0}en.pdf

[20] Java Card Forum. "Java Card Platform vs. Native Cards (White Paper)." 2013. https://javacardforum.files.wordpress.com/2013/11/jcf-java-vs-native-final.pdf

[21] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski Jr.. "Analyzing and comparing Montgomery multiplication algorithms." In *IEEE Micro*, Vol. 16, Issue 3, pp. 26–33, 1996.

[22] Charles F. F. Karney. "Sampling exactly from the normal distribution." In *ACM Transactions on Mathematical Software (TOMS)*, Vol. 42, Issue 1, No. 3, pp. 1–14, 2016.

[23] Donald Ervin Knuth and Andrew Chi-Chih Yao. "The complexity of non uniform random number generation." In *Algorithms and complexity: New directions and recent results*, Academic Press, pp. 357–428, 1976.

[24] Patrick Longa and Michael Naehrig. "Speeding up the number theoretic transform for faster ideal lattice-based cryptography." In *Proceedings of the 15th International Conference on Cryptology and Network Security – CANS 2016*, LNCS, Vol. 10052, pp. 124–139, 2016.

[25] Richard Lindner and Chris Peikert. "Better key sizes (and attacks) for LWE-based encryption." In *Proceedings of the 11th International Conference on Topics in Cryptology – CT-RSA 2011*, LNCS, Vol. 6558, pp. 319–339, 2011.

[26] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings." In *Proceedings of the 29th Annual international conference on Theory and Applications of Cryptographic Techniques – EUROCRYPT 2010*, LNCS, Vol. 6110, pp. 1–23, 2010.

[27] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. "Efficient ring-LWE encryption on 8-bit AVR processors." In *Proceedings of the 17th International Conference on Cryptographic Hardware and Embedded Systems – CHES 2015*, LNCS, Vol. 9293, pp. 663–682, 2015.

[28] Pedro Maat C. Massolino, Lejla Batina, Ricardo Chaves, and Nele Mentens. "Low power Montgomery modular multiplication on reconfigurable systems." In IACR Cryptology ePrint Archive, Report 2016/280, 2016.

[29] Ciaran McIvor, Máire McLoone, and John Vincent McCanny. "FPGA Montgomery multiplier architectures - a comparison." In *Proceedings of 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 279–282, 2004.

[30] Daniele Micciancio and Oded Regev. "Lattice-based cryptography." In *Post-Quantum Cryptography*, pp. 147–191, Springer-Verlag, 2009.

[31] National Institute of Standards and Technology. "Post-Quantum Cryptography." 2017. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/`

[32] Sıddıka Berna Örs, Lejla Batina, Bart Preneel, Joos Vandewalle. "Hardware implementation of a Montgomery modular multiplier in a systolic array." In *Proceedings of 2003 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, p. 8, 2003.

[33] Chris Peikert. "An effcient and parallel Gaussian sampler for lattices." In *Proceedings of the 30th Annual Conference on Advances in Cryptology – CRYPTO 2010*, LNCS, Vol. 6223, pp. 80–97, 2010.

[34] Thomas Pöppelmann and Tim Güneysu. "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware." In *Proceedings of the 2nd International Conference on Cryptology and Information Security in Latin America – LATINCRYPT 2012*, LNCS, vol. 7533, pp. 139–158, 2012.

[35] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. "High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers." In *Proceedings of the 4nd International Conference on Cryptology and Information Security in Latin America – LATINCRYPT 2015*, LNCS, Vol. 9230, pp. 346–365, 2015.

[36] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography." In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing – STOC '05*, pp. 84–93, 2005.

[37] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. "Compact ring-LWE cryptoprocessor." In *Proceedings of the 16th International Conference on Cryptographic Hardware and Embedded Systems – CHES 2014*, LNCS, Vol. 8731, pp. 371–391, 2014.

[38] Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. "High precision discrete Gaussian sampling on FPGAs." In *Proceeding of the 20th*

International Conference on Selected Areas in Cryptography – SAC 2013, LNCS, Vol. 8282, pp. 383–401, 2013.

[39] Peter Williston Shor. "Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer." In *SIAM Journal on Computing*, Vol. 26, Issue 5, pp. 1484–1509, 1997.

[40] Sun Microsystems, Inc.. "Java Card Platform Specification 2.2.2." 2006. http://www.oracle.com/technetwork/java/javacard/specs-138637.html

[41] Hendrik Tews and Bart Jacobs. "Performance issues of selective disclosure and blinded issuing protocols on Java Card." In *Proceeding of the Third IFIP WG 11.2 International Workshop on Information Security Theory and Practice. Smart Devices, Pervasive Systems, and Ubiquitous Networks – WISTP 2009*, LNCS, Vol. 5746, pp. 95–111, 2009.

[42] Ye Yuan, Chen-Mou Cheng, Shinsaku Kiyomoto, Yutaka Miyake, and Tsuyoshi Takagi. "Portable implementation of lattice-based cryptography using JavaScript." In *International Journal of Networking and Computing*, Vol. 6, No. 2, pp. 309–327, 2016.

[43] Ye Yuan, Kazuhide Fukushima, Shinsaku Kiyomoto, and Tsuyoshi Takagi. "Memory-constrained implementation of lattice-based encryption scheme on standard Java Card." In *Proceedings of 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 47–50, 2017.

[44] Robert Primas, Peter Pessl, and Stefan Mangard. "Single-trace side-channel attacks on masked lattice-based encryption." In *Proceedings of the 19th International Conference on Cryptographic Hardware and Embedded Systems – CHES 2017*, LNCS, Vol. 10529, pp. 513–533, 2017.

[45] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the concrete hardness of learning with errors." In *Journal of Mathematical Cryptology*, Vol. 9, Issue 3, pp. 169–203, 2015.

[46] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. "Practical CCA2-secure and masked ring-LWE implementation." In *IACR Transactions on Cryptographic Hardware and Embedded Systems*, Vol. 2018, Issue 1, pp. 142–174, 2018.

[47] Florian Gp̈fert, Christine van Vredendaal, and Thomas Wunderer. "A hybrid lattice basis reduction and quantum search attack on LWE." In IACR Cryptology ePrint Archive, Report 2017/221, 2017.