Memory-Constrained Implementation of Lattice-based Encryption Scheme on Standard Java Card

Ye Yuan[1] Kazuhide Fukushima[2] Shinsaku Kiyomoto[2] Tsuyoshi Takagi[3,4]

Abstract:
Lattice-based cryptography, as one of the strongest candidates for post-quantum cryptography, has inspired and derived several variant cryptosystems which were proposed and attracted wide attention due to their applicability and operating efficiency in recent years. Currently, ring learning with errors (Ring-LWE) based encryption scheme possesses high efficient and reliable, that brings us a practical implementation which could run on the small devices with limited storage capacity and computational resource such as IoT senor nodes or smart cards. Meanwhile, Java Card is a Java-enabled platform and a kind of smart card-based hardware with very limited card memory. Computing multiplication of large integers is a challenge due to the memory constraints, and it had been considered that only a few lattice-based cryptosystems fitting into such constrained device. In this paper, we show the first implementation of a mix of two the iterative forward number theoretic transform with improved Montgomery modular multiplication for lattice-based cryptosystem on standard Java Card whose the running time is nearly optimal about 100 seconds. More important, we indicate that polynomial multiplication and over signed 16-bit integer arithmetic can be performed on Java Card platform even if the unsigned short type and integer type are not supported, which makes more lattice-based protocols running on Java Card probably.

Keywords: Lattice-based cryptography, Ring-LWE, Java Card, Montgomery modular multiplication, Fast Fourier transform

1. Introduction

Java Card is a Java-enabled platform with limited computational resource that can run applets on smart cards securely. Java Card applets execute on the Java Card virtual machine (JCVM) which is a subset of the Java virtual machine (JVM), to achieve hardware-independent compatibility. Several basic security and cryptography implementations such as DES, RSA or the elliptic curve cryptography (ECC) are supported on Java Card, however, Shor's algorithm for factoring on quantum computers is efficient to break traditional RSA and ECC. Because of its wide application in development, there is a tremendous need to research the implementation of post-quantum cryptography on Java Card.

Lattice-based cryptography has attracted attention because it is thought to be secure against attack by quantum computers. Since the short integer solution (SIS) problem introduced by Ajtai [Ajt96], lattice-based cryptography has been researched several years and many efficient constructions have been provided that leverage average-case hardness of SIS such as the learning with errors (LWE) problem [Reg05] etc. In recent years some derivative schemes of original Regev's LWE-based cryptosystem were presented [PVW08][MR08][LPR10][LP11]. Implementations of the schemes on hardware or small devices have been reported in some literatures [PG12][GLP12][GFS+12][CRVV15], however only the few were implemented on certain kinds of smart cards such as Java Card [BJ14][BSJ15]. Usually most of these implementations are optimized by their prototypical platforms or theoretical structures hence portabilities and compatibilities could be challenged profoundly from time to time when implementing them in reality. Due to limited processing power and memory of Java Card platform specification 2.2.2, only a few features of Java language are available, thus it is difficult to solve practical operation problems in dealing with lattice-based cryptosystems, e.g. being able to compute modular multiplication of common big integers turns out to be an important and complicated issue for polynomial multiplication in a Java Card case.

Basically, fast Fourier transform (FFT) is commonly used in implementation on constrained devices [OPG14][CRVV15][BSJ15][POG15]. There are several ways to compute polynomial multiplication such as Karatsuba algorithm, which is another approach of fast multiplication algorithm. It applies recursive invocations until the degrees

[1] Graduate School of Mathematics, Kyushu University
[2] KDDI Research, Inc.
[3] Institute of Mathematics for Industry, Kyushu University
[4] CREST, Japan Science and Technology Agency

of polynomials are small enough to be multiplied directly. FFT also has recursive implementation which is a classical divide and conquer algorithm [GT01]. Although these algorithms are available and usable for embedded devices, they are too heavy for the applets. Recursive algorithms create a lot of temporary variables during the calculation, usually trying to write non-recursive algorithms is tedious, hence it might be inappropriate to use those algorithms for memory constrained systems.

Iterative FFT works fine on smart cards, however there still is a challenge for integer multiplication. Due to the limited memory and processing capabilities, Java Card basically only support small integer arithmetic so that there is insufficient computing capacity for most lattice-based schemes. Some literatures provide optimal modular reduction operations for a couple of specific parameters such as [BSJ15], but does not offer any general solutions for all possible parameter sets, that is not enough for actual development activity. To solve the problem, In this paper we propose to apply an improved Montgomery modular multiplication (MMM) algorithm [AT06], it can be carried out for big integer arithmetic only using small integer operations without any loss of accuracy and integer overflow. By combining iterative FFT [CRVV15] with improved modular multiplication, it is possible to work out the efficient polynomial multiplication for any parameters available and achieving other lattice-based cryptosystems on standard Java Card.

In this paper, we have implemented several famous lattice-based encryption schemes and compared the performance of them [YCK+15]. We considered that ring-LWE based encryption schemes can achieve high run-time efficiency and good portability, thus we have selected the ring-LWE based scheme to implement in this paper. We expected to develop practical program with good compatibility, portability and expansibility, therefore we have chosen the improved MMM algorithm for the lower-cost version of Java Card specification, and have also implemented non-recursive FFT algorithm to speed up polynomial multiplication. Our implementation obtains the nearly optimal running time about 100 seconds in polynomial multiplication for an approximate security level of 128 bit AES [GFS+12].

The rest of this paper is organized as follows. We will explain the notation, give a brief mathematical background and introduce the ring-LWE based encryption scheme in section 2. We will introduce the specification of Java Card platform in section 3. We will describe our implementation techniques in section 4. We will then give detailed performance reports on the standard Java Card in section 5. Finally, we conclude this paper in section 6.

2. Lattice-based cryptography

In this section, we present the relevant mathematical background for discrete Gaussian sampling, the learning with errors problem, and the ring-LWE based encryption scheme.

Throughout this paper, we denote $\mathbb{Z}_q$ as the set of integers $\{0, 1, \ldots, q-1\}$, and $\mathbb{Z}_q[x]$, polynomials whose coefficients are in $\mathbb{Z}_q$. Let R_q=$\mathbb{Z}_q[x]/f(x)$ be a quotient polynomial ring, where $f(x)$ is a degree n polynomial. In the meantime, polynomials are denoted by bold italic small letters such as $\boldsymbol{f}$, while vectors, bold small letters such as $\mathbf{v}$ and matrices, bold large letters such as $\mathbf{A}$.

2.1 Discrete Gaussian sampling

For a lattice L and a real $\sigma > 0$, we use $D_{L,\sigma}$ to denote the discrete Gaussian distribution over L, which is a probability distribution that assigns to each vector $\mathbf{x} \in L$ a probability proportional to $\exp(-\pi\|\mathbf{x}\|/s^2)$ (or its equivalent form $\exp(-\|\mathrm{x}\|^2/2\sigma^2)$) centered at the zero vector, and the Gaussian parameter s equals $\sigma\sqrt{2\pi}$. In our implementation, we are setting $L = \mathbb{Z}$, so that each integer is sampled randomly according to $D_{\mathbb{Z},\sigma}$. It limits the domain of the probability density function of a normal distribution to integer. We will describe the sampling algorithm of discrete Gaussian distribution in section 4.1.

2.2 Learning with errors (LWE)

Let m, n be positive integers, integer vectors $\{\mathbf{a}_1, \ldots, \mathbf{a}_n\} \in \mathbb{Z}^m$ are n linearly independent vectors, then put these vectors as columns in a matrix $\mathbf{A}$, i.e. $\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_n] \in \mathbb{Z}^{m\times n}$. The lattice L that generated by the basis matrix $\mathbf{A}$ is the set

$$L(\mathbf{A}) = \{\Sigma_{i=1}^{n} x_i \mathbf{a}_i \mid x_i \in \mathbb{Z}\}.$$

Regev proposed the original learning with errors (LWE) problem based cryptosystem [Reg05] which uses a special structure lattice call q-ary lattice whose all of operation are done modulo an positive integer q. The LWE-problem can be expressed as follows: input a pair of matrices ($\mathbf{A}$, $\mathbf{b}$), where $\mathbf{A}$ is randomly selected from $\mathbb{Z}_q^{m\times n}$ by uniform sampling and $\mathbf{b} = \mathbf{As} + \mathbf{e} \in \mathbb{Z}_q^m$, with the error $\mathbf{e} \in \mathbb{Z}_q^{m\times n}$ sampled from a probability distribution, the problem is to find the secret vector $\mathbf{s}$ from such given pair ($\mathbf{A}$, $\mathbf{b}$).

Given integers $n, q > 0$ and a polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, call R_q an ideal lattice if each polynomial in R_q has a bijective mapping to an ideal $\mathbb{Z}_q^n$. The ring-LWE problem is based on such ideal lattice R_q. Similar to LWE problem, given a pair of polynomials $\boldsymbol{a}$, $\boldsymbol{b} \in R_q$ where $\boldsymbol{a}$ is chosen uniformly at random, there exists a polynomial $\boldsymbol{s} \in R_q$ that $\boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e}$ and $\boldsymbol{e} \in R_q$ is sampled from a probability distribution, the problem is to recover the secret $\boldsymbol{s}$.

The solution of such LWE and ring-LWE problem can be reduced to the NP-hard shortest vector problem (SVP) which is required to output a non-zero vector $\mathbf{v} \in L(\mathbf{A})$ and it is generally believed that non-deterministic polynomial-time algorithm can approximate its equivalent problem [LPR10].

2.3 LWE-based encryption schemes

The security of ring-LWE based encryption scheme is based on the hardness of ring-LWE problem. Here we inves-

tigate the ring-LWE based encryption scheme which is more efficient than the original Regev's LWE cryptosystem using matrices. A famous ring-LWE scheme is proposed by Lyubashevsky, Peikert, and Regev using constructed polynomials over an ideal integer ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ instead of the structured integer matrices, which is based on the hardness of solving ring-LWE problem with IND-CPA security provably by appropriately parameterized ideal lattices. In this paper we refer to as LPR-LWE [LPR10]. Note that all operations of polynomial take place in R_q.

Let R_q be the polynomial ring $\mathbb{Z}_q[x]/(x^n + 1)$ with n a power of 2. 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($\boldsymbol{m}$) + $\boldsymbol{e}$ (mod $\boldsymbol{q}$)) = $\boldsymbol{m}$ is an anti-operation of encoding for any 'small enough' error $\boldsymbol{e} \in R_q$, e.g, one whose coefficients are sampled through discrete Gaussian sampling where each coefficient is sampled independently. $D_{L,\sigma}$ denotes a discrete Gaussian distribution over a lattice L with a positive real σ 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 $\boldsymbol{e} \in R_q$ 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 $\bar{\boldsymbol{m}}$ = 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 cipher text is the pair ($\boldsymbol{c}_1$, $\boldsymbol{c}_2$).

Decryption:

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

Table 1 shows the parameter sets which are used in our implementation. We select the parameters suggested by Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin Huss [GFS+12] which are not too large to supported by Java Card platform. For $n = 256$, the bit-security matches about that of AES-128.

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

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

3. Java Card specification

In this section, we give some general information and features about the standard Java Card. In this paper, the standard we referred is based on Java Card Platform Specification 2.2.2 [Sun06] and Global Platform 2.1.1 [GP03].

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, its physical characteristics are defined by `ISO/IEC 7816-1` standard [ISO7816-1]. There are three types of memory presented by Java Card platform, random access memory (RAM), read-only memory (ROM), and non-volatile memory (NVM), 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, thus it could be used to hold the applets and data. RAM in Java Card has a faster processing speed with limitless use times, the transient objects are allocated in RAM. As opposed to EEPROM, RAM is better suited to process temporary data.

The base package `javacard.framework` which is at the core of the Java Card framework defines the classes, interface, constants, and exceptions. Java Card uses a request-response protocol called application protocol data unit (APDU) to communicate with the off-card applications. `ISO/IEC 7816-4` defines the APDU protocol which contains the command part (C-APDU) and the response part (R-APDU) [ISO7816-4]. Table 2 and Table 3 show the structures of C/R-APDU and explanation of these fields.

Table 2 The structure of C-APDU and R-APDU

Header (mandatory)				Body (optional)		
CLA	INS	P1	P2	Lc	Data field	Le

Body (optional)	Trailer (mandatory)	
Data field	SW1	SW2

Table 3 The explanation of each field of APDU

CLA	Class byte
INS	Instruction byte
P1/2	Parameter bytes
Lc	The number of bytes in the data field of a C-APDU
Le	The number of bytes in the data field of a R-APDU
SW1/2	Status words of a R-APDU
Date field	The data field in APDU

Java Card reduces some Java language features due to the constrained resource that the JCVM specification defined a subset of Java programming language standard. Table 4 shows an overview of the language features between Java Card and Java platform. Note that the multi-dimensional array is not supported, this is one reason we didn't choose the encryption schemes using matrices such as [MR08][Fre10][LP11].

The Java Card runtime environment (JCRE) is consisted of the JCVM, the Java Card framework and APIs, which is to get the applets residing or running on the card. As a subset of the Java runtime environment (JRE), the JCRE is implemented as two separate parts, the Java Card converter and the Java Card bytecode interpreter. The Java Card converter is responsible for the execution on PC via a card reader to undertake some time-consuming task before uploading an applet or a `.CAP` file to the card. The Java Card bytecode interpreter is running on the card and responsible for memory allocations, execution of the bytecode

Table 4 Supported and unsupported features of Java Card specification 2.2.2

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

instructions, and so on.

One of the advantages of Java Card is the security architecture so that it suits many domains which requiring highly secure. Java Card performs several classic cryptographic algorithms for encryptions and digital signatures. They are implemented by two packages of the Java Card API, `javacard.security` and `javacardx.crypto`, which contain the classes and interfaces for various types of the fast symmetric (such as 256 bit AES, 56/112/168 bit DES) and the slow asymmetric (such as 2048 bit RSA, 320 bit ECC) cryptographic algorithms.

In a Java Card environment, a message is represented as bit-stream or binary data. We use a binary array to express a plaintext in our implementation, this allows ring-LWE scheme to perform a bit-wise text encryption, therefore ring-LWE scheme is fit for authentication and digital signature on Java Card.

4. Efficient algorithms for Java Card implementation

In this section, we describe our implementation techniques for discrete Gaussian sampling, polynomial multiplication, and modular multiplication.

4.1 Random numbers generation

Java Card platform provides `javacard.security.RandomData` as the random number generator that returns a pseudo-random number by specifying `ALG_PSEUDO_RANDOM` algorithm, or a cryptographically secure random number using `ALG_SECURE_RANDOM` algorithm. However, in the new release such as Java Card 3.0.5, these random number generation algorithms have already been deprecated. In our cases, we pre-generated all needed random number arrays.

Ring-LWE based encryption scheme needs many random number samples from discrete Gaussian distribution. Sampling values from a target discrete Gaussian distribution can be a time and resource consuming activity [CWB14]. We used rejection sampling [GPV08] which is one of these available approaches with a negligible memory footprint and implemented the inverse-transform sampling to reduce runtime computation by considering time and memory trade-offs in our implementation.

In rejection sampling, τ is the tail-cut factor with a suitable size that determines where to drop the ignored probability of far samples, select a random number uniformly from $[-\tau\sigma, \tau\sigma]$, then accept the chosen value if it has probability proportional to $\exp(-\|\mathrm{x}\|^2/2\sigma^2)$, or otherwise rejected and then repeats this process until success. Algorithm 1 shows the basic rejection sampling method for the target discrete Gaussian distribution [GPV08][DN12]. *RandomInt* in line 2 is a function to sample a random integer in the range $[a, b]$ uniformly, where $a, b \in \mathbb{Z}$. *RandomFloat* in line 4 is to generate a random floating-point number in the range (0, 1] uniformly.

Algorithm 1: Rejection sampling on $\mathbb{Z}$

Input: Floating-point numbers c, r and τ
Output: A sample value $x \in \mathbb{Z}$

1 Let $h = -\pi/r^2$; $x_{min} = \lfloor c - \tau r \rfloor \in \mathbb{Z}$, $x_{max} = \lceil c + \tau r \rceil \in \mathbb{Z}$;
2 let $x = RandomInt(x_{min}, x_{max})$;
3 let $p = \exp(h \cdot (x - c)^2)$;
4 let $r = RandomFloat()$;
5 **if** $r < p$ **then return** x; **else goto** step 2

Usually rejection sampling only stores a few parameters and fits into a constrained device that has a small memory footprint. However, it requires a lot of computation and easily becomes a bottleneck in runtime when it comes to actual implementation. Inverse transform sampling generates random numbers from the cumulative distribution function (CDF) of the probability distribution, since the CDF depends on the parameters of rejection sampling. Therefore, we precompute all possible values of $\exp(-\|\mathrm{x}\|^2/2\sigma^2)$ via Algorithm 1, add up the individual probabilities for these values and store the probabilities in a table, then we generate a uniform random number and use inverse transform sampling to perform a table lookup at runtime to get sample from the target discrete Gaussian distribution.

4.2 Montgomery modular multiplication (MMM)

According to the specification of Java Card, only signed small integer types (`short, byte`) are supported that are represented in 2's complement format. Signed `short` type can range from -2^{15} to $2^{15} - 1$ which is the maximum positive value of Java Card supported. However, selecting a modular q from the Table 1 and then multiplying two positive integers a and $b \in \mathbb{Z}_q$, the product will probably out of bounds for signed `short` type. Therefore, considering the limits of Java Card platform, the better way is to compute the modular multiplication i.e. $a * b \bmod q$. Montgomery modular multiplication (MMM) algorithm is quick way to perform fast modular multiplication. It optimizes the complexity of division, that is, needs only to perform right shift operation where radix is a power of 2. Given a n-bit modulo M, integers $X, Y \in \mathbb{Z}_q$, $R = 2^n$, MMM returns the $X * Y * R^{-1} \bmod M$. Therefore, to compute modular multiplication $X * Y \bmod M$, we need to perform MMM twice: first input X and R^2:

$$A' = MMM(X, R^2) = XR^2 * R^{-1} \bmod M = XR \bmod M,$$

then call this function again and input A' and Y:

$$A = MMM(A', Y) = XRY * R^{-1} \bmod M = XY \bmod M.$$

Optimization of MMM have been reported in some literatures, such as [FSV07], which proposed a modified algorithm called finely integrated operand scanning (FIOS). However, according to the algorithm, we could see that it still have to compute integer multiplication before performing division (i.e., shifting to the right), which means the products will probably have out of the signed `short` type boundary first as the modular size increases, therefore that algorithm cannot be used in the device with limited storage capacity and computational resource. In order to fit Montgomery modular multiplication into Java Card platform, we adopted an improved algorithm, which can compute 2^ω-bit MMM using s-bit multiplier only [AT06].

Algorithm 2: 2^ω-bit Montgomery modular multiplication using s-bit multiplier

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 $0 \le X, Y \le M$, $b = 2^\omega$, $s = \lceil n/\omega \rceil$, $R = b^s$ with $gcd(M, b) = 1$ and $M' = -M_0^{-1} \bmod b$

Output: Non-negative integer $A = X * Y * R^{-1} \bmod M$

1 Let $A = 0 = (A_s, ..., A_0)_b$;
2 **for** $i = 0$ **to** $s - 1$ **do**
3 $temp = 0$;
4 **for** $j = 0$ **to** $s - 1$ **do**
5 $(temp, A_j) = X_j * Y_i + A_j + temp$;
6 $A_s = 0$, $temp = 0$, $\mu_i = A_0 * M' \bmod b$;
7 **for** $j = 0$ **to** s **do**
8 $(temp, A_j) = M_j * \mu_i + A_j + temp$;
9 $A = A/b$;
10 **if** $A \ge M$ **then** $A = A - M$;
11 **return** A

According to Algorithm 2, we need to select a suitable size for multipliers to make sure the product would not overflow. We have seen that the standard Java Card supports up to the maximum positive value of signed `short` type. In the step 5 and 8, it computes the product of two b-bit multipliers, which means the actually available size of each multiplier is only about half the size of `short`, that is, signed `byte` type. However, all integers are signed and stored in 2's complement format for Java language, the maximum value of `byte` is only $2^7 - 1$. Therefore we split an arbitrary-length integer into an array of `short` numbers and the effective length of each `short` number is 8-bit because the modulus q we chosen is less than 2^{15}, c.f. Table 1. When a signed `short` type integer splits into two signed `byte` integers, the low 8 bits might be a negative `byte` integer. If this is the case, we have to convert the 8-bit negative integer to a 16-bit positive `short` number e.g. by calculating M_0 `& 0xff`. Even if the modular M have increased, we just need to add more multipliers. Therefore Algorithm 2 guarantees its correctness throughout the entire process, none of the variables ever overflowed.

Moreover, according to the step 6, we also need to select an appropriate parameter b. In our case we define that every multiplier is a 8-bit integer, therefore let $\omega = 8$ so that $b = 2^8$, $R = b^2 = 2^{16}$, then we can precompute M' and the inverse R^{-1}.

4.3 Fast Fourier Transform (FFT)

To compute the polynomial multiplication on Java Card, a polynomial $\boldsymbol{f} = \sum_{i=0}^{n-1} f_i x^i \in R_q$ could be represented as an integer array object $[f_0, \ldots, f_{n-1}]$ that the elements of the array are the coefficients, therefore the indices of the created array object are the degrees of the corresponding coefficients. In our case, we stored the coefficients in an ascending order. The array object provides all coefficients from the lowest to the highest degree such that the position of each coefficient corresponds to the degree of the term to which it belongs.

In ring-LWE based encryption scheme, polynomial multiplication is both the major step and the most expensive operation. There are several efficient approached such as Karatsuba algorithm and fast Fourier transforms (FFT). When the selected parameter set of ring-LWE scheme is small, runtime efficiency is not varied much between Karatsuba algorithm and FFT. For example, we know that Karatsuba algorithm, it recursively calls itself while executing, each recursion of Karatsuba algorithm will create a couple of new coefficient arrays.

Multiple recursive calls are not a big problem issue for other platforms or programming languages. However, on memory-constrained, small embedded devices such as Java Card, more temporary arrays means more required memory capacity that we might not have enough memory to perform the operations.

We have also rewritten Karatsuba algorithm into the non-recursive form, however its computing speed was still slow and did not reduce the number of arrays. In other words, Karatsuba algorithm does not fit for the implementation of polynomial multiplication with larger degrees. Therefore we follow the state-of-the-art and use the iterative forward number theoretic transform (NTT) [RVM+14][CRVV15] which has lower asymptotic complexity $O(n \log n)$ for multiplying polynomials with higher degrees.

NTT is one of generalization of FFT by replacing complex numbers with an primitive n-th root of unity in $\mathbb{Z}_q$ where q is a prime number, and there is no need for any floating-point and complex number arithmetic. In fact, to perform non-recursive FFT, we have to compute a primitive roots modulo q which is a prime number [GT01]. If the chosen q equals $cn + 1$ where c is a reasonably small positive integer and n is a power of 2, then we can find an integer x with order $q - 1$ that is a primitive root of $\mathbb{Z}_q^\times$. That is, $x^i \bmod q$ is different for $i \in \mathbb{Z}_q$. Because $x \bmod q \neq 0$, by Fermat's Little Theorem, we have

$$(x^c)^n \bmod q = x^{cn} \bmod q = x^{q-1} \bmod q = 1,$$

thus $\omega \equiv x^c \bmod q$ is a primitive n-th roots modulo q.

Algorithm 3 [*1] shows the iterative forward number theoretic transform that we have implemented for ring-LWE encryption schemes. *BitReverse* at line 1 is a permutation of a sequence of n items so that the new index of each element is the reverse of the bit string of its old index, where n is a power of 2. To compute polynomial multiplication $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b} \in R_q$, it is required to compute the inverse of NTT (NTT^{-1}), so that $NTT^{-1}(NTT(\boldsymbol{f})) = \boldsymbol{f}$ for all $\boldsymbol{f} \in R_q$, thus the product $\boldsymbol{c} = NTT^{-1}(NTT(\boldsymbol{a}) \otimes NTT(\boldsymbol{b}))$, where $\otimes$ is entry-wise multiplication.

Algorithm 3: Iterative forward number theoretic transform (NTT)

Input: Polynomial $\boldsymbol{a} \in R_q$ of degree $n-1$ and n-th primitive root $\omega_n \in \mathbb{Z}_q$ of unity
Output: Polynomial $\boldsymbol{A} = NTT(\boldsymbol{a})$

1 Let $\boldsymbol{A} = BitReverse(\boldsymbol{a})$;
2 **for** $m = 2$ **to** n by $m = 2m$ **do**
3 $\omega_m = \omega_n^{n/m}$, $\omega = 1$;
4 **for** $j = 0$ **to** $m/2 - 1$ **do**
5 **for** $k = 0$ **to** $n - 1$ by m **do**
6 $t = \omega \cdot \boldsymbol{A}[k + j + m/2] \bmod q$;
7 $u = \boldsymbol{A}[k + j]$;
8 $\boldsymbol{A}[k + j] = u + t \bmod q$;
9 $\boldsymbol{A}[k + j + m/2] = u - t \bmod q$;
10 $\omega = \omega \cdot \omega_m$;
11 **return** $\boldsymbol{A}$

We can find that NTT computes the modular multiplication at the step 6 from Algorithm 3. According to the sizes of the parameter sets, we have to invoke Algorithm 2 in step 6 when overflow occurs. In addition, the speed of Algorithm 2 is slower than direct modular multiplication (c.f. Table 5), hence each cycle we need to judge whether the product is out of the range of signed `short` type before performing modular multiplication and choose which approach is more suitable for the target platform.

5. Performance of lattice-based encryption scheme on Java Card

In this section, we report the performance results of running LPR-LWE scheme on the standard Java Card [*2]. Performing MMM (Algorithm 2) on Java Card is a time consuming task. Table 5 shows the speed for two calculations of modular multiplication of signed `short` integers, therefore performing MMM is relatively slow compared to the speed of calculating modular multiplication directly.

Table 5 The total running time (ms) of modular multiplication executed 10,000 times with different approaches

	The total running time (ms)
MMM	160239
Direct calculation	750.85

We pre-sample all the random values on PC[*3] from a target discrete Gaussian distribution for key generation and encryption. A more detailed introduction of discrete Gaussian sampling is out of the scope of this paper, but may be handled in our future work.

5.1 Performance results on Java Card

We implemented LPR-LWE scheme and generated key pairs by choosing parameters in Table 1. Table 6 shows the performance results of LPR-LWE executed on the standard Java Card. Compared with these performance, 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. Furthermore, the running time ratio for key generation, encryption and decryption is almost exactly $1 : 2 : 1$. We have already known that polynomial multiplication, or in this case, NTT (Algorithm 3) is performed once in key generation and decryption of LPR-LWE, and twice in encryption. The results indicate that the execution time of LPR-LWE on Java Card is almost equivalent to the running time of polynomial multiplication that we'll be looking at more details in the next section. In our implementation, the run-time data are all stored in RAM of the applet, rather than EEPROM, which means we can compute and update these data quite fast in the process of responding and performing an incoming APDU command. According to the specification of Java Card and the chosen parameters, our implementation design is almost optimal for general versatility and portability though the operating speed is somewhat slow.

On the recent papers, [BJ14][BSJ15] present the performance of ring-LWE based encryption scheme and some authentication protocols on several smart card platforms including Java Card 2.2.2. Their results shows that the performance of lattice-based cryptography on smart card could be increased with good optimization and better hardware. However, there still have some issues not so explicitly mentioned. First, for ring-LWE based encryption scheme, we have already known that it is necessary to check whether the product of integer multiplication is in bounds, and it is more complicated in actual development. However, in

[*1] This algorithm is also being used to compute the inverse of NTT (NTT^{-1}).

[*2] **JCOP v2.4.1 NXP J3A081 Dual Interface Card**:
Java Card Version: 2.2.2;
Global Platform: 2.1.1;
SCP Secure Channel Protocol: SCP01, SCP02;
ISO/IEC 7816, T=0, T=1 (communication speed, kbit/s): 223.2;
ISO/IEC 14443 T=CL (communication speed, kbit/s): 848;
Available EEPROM Options KByte: 80;
ROM (free for Applets, up to KBytes): 76;
APDU Buffer (RAM/Bytes): 1462;
Cryptography: DES/TDES [bit] 56/112/168, AES [bit] 256, RSA [bit] 2048, ECC GF(p) [bit] 320, SHA-1/2.

[*3] The test PC has the following specifications:
CPU: Intel(R) Core(TM) i7-4710MQ @ 2.5GHz;
Memory: 8GB DDR3 RAM;
Hard Disk: 1TB 5400rpm;
OS: Windows 10 build 10.0.10586.494 Pro x64;
Java(JDK) Version: jdk1.7.0_67.

Table 6 Performance results (second) of LPR-LWE encryption scheme on Java Card

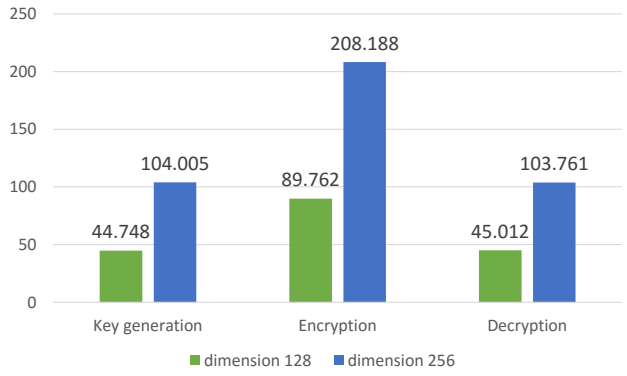
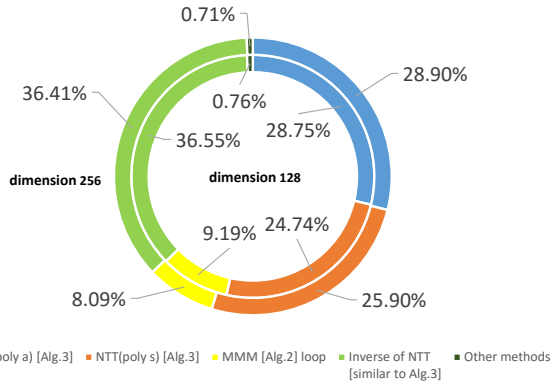


Table 8 The portions of computation time in key generation



those papers, the authors try to solve the integer overflow problem only for very few limited parameters but not for all usable parameters. In addition, some of the parameters of their chosen protocols are considerably larger than or close to the maximum value of type signed `short` integer, and we can find that in their source code they used the signed `int` type and `int` arrays that a lot of cards may not support, which means their environment may supports higher-bit integer operations, and it would significantly increase the performance and reduce the difficulty of development, but it also means the backward compatibility and portability of the programs are probably dismal. Last, compared with their implementations, the authentication protocols are significantly slower than ring-LWE encryption scheme even the protocol such as DJ-LWE which is a derivative of LWE-based scheme selected very small parameters. The authors didn't discuss the differences in those papers.

5.2 Detailed running time on Java Card

Table 7 shows the decomposition of computation time of the main methods in key generation. Including many MMM loop in polynomial multiplication (c.f. the step 6, Algorithm 3.), we tested the execution time of NTT and the inverse of NTT (NTT^{-1}) as mentioned in section 4.3.

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 using Algorithm 3 is the single bottleneck computation that accounting for about 90% by summing both running time of NTT and NTT^{-1} because the sizes of moduli are close to 2^{15}. In fact, most time is spent on MMM in NTT and NTT^{-1}, and as the parameters increases, the running time of MMM loop within NTT/NTT^{-1} increases too.

As we have expected, in Table 7, the computing time required for polynomial multiplication is nearly equal to total running time of key generation, as well as encryption and decryption.

Table 8 shows the portions of the methods in polynomial multiplication that the data used are compiled from Table 7. We see that though the running time of dimension 128 only accounts for less than 50% of the running time of dimension 256, the proportion of time each important methods spent are approximately equal. Overall, as moduli diminish in size, the amount of modular multiplication within NTT would be reduced.

6. Conclusions

We have implemented ring-LWE based encryption scheme for normal arguments and successfully run the applets on standard Java Card platform, the performance of different parameter sets was compared. We have come up with an approach to solve the problem of big integer multiplication and provide nearly optimal running time (about 100 seconds) of polynomial operation on standard Java Card. We have used fast Fourier transform to improve polynomial multiplication speed and reduce calculation amount, the result shows that polynomial multiplication is the primary performance bottleneck. Furthermore, in order to achieve compatibility and platform-independent for Java Card, we have implemented the improved Montgomery modular multiplication algorithm, which shows that even if only small primitive data types can be supported by the lower-specification machine, we can still perform arithmetic of arbitrary signed large integers on a lattice. Our results indicate that it is possible to implement more lattice-based cryptosystems on such memory-constrained device by using these two algorithms in combination. With the performance boost over upgrading hardware, obviously there is much room for improvement in our future work, and we can further optimize the implementations as long as we sustain extensibility and portability of our programs.

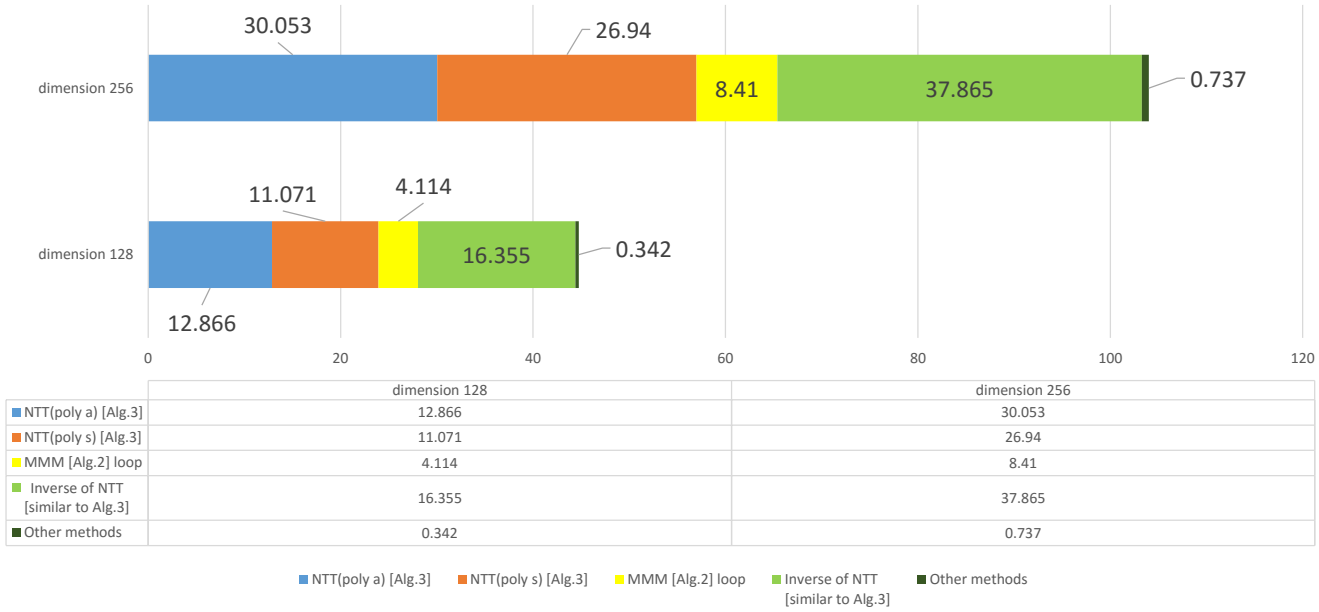
References

[Ajt96] Miklos Ajtai. "Generating hard instances of lattice problems." In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pp. 99–108, 1996.

[AT06] Toru Akishita and Tsuyoshi Takagi. "Power analysis to ECC using differential power between multiplication and squaring." In *Domingo-Ferrer, J., Posegga, J., Schreckling , D.(eds) CARDIS 2006*, LNCS, vol. 3928, pp. 151–164, Springer, Heidelberg, 2006.

[BJ14] Ahmad Boorghany and Rasool Jalili. "Implementation and comparison of lattice-based identification protocols

Table 7 Decomposition of computation time (second) in key generation



	dimension 128	dimension 256
NTT(poly a) [Alg.3]	12.866	30.053
NTT(poly s) [Alg.3]	11.071	26.94
MMM [Alg.2] loop	4.114	8.41
Inverse of NTT [similar to Alg.3]	16.355	37.865
Other methods	0.342	0.737

on smart cards and microcontrollers." Cryptology ePrint Archive, Report 2014/078, 2014.

[BSJ15] 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*, volume 14 issue 3, May 2015, article No. 42, pp. 42:1–42:25, 2015.

[CRVV15] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. "Efficient software implementation of ring-LWE encryption." In *DATE '15 Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 339–344, 2015.

[CWB14] Daniel Cabarcas, Patrick Weiden, and Johannes Buchmann. "On the efficiency of provably secure NTRU." In *Proceedings of PQCrypto 2014*, pp. 22–39, 2014.

[DN12] Leo Ducas and Phong Q. Nguyen. "Faster Gaussian lattice sampling using lazy floating-point arithmetic." In *Proceedings of ASIACRYPT 2012*, pp. 415–432, 2012.

[Fre10] Tore Kasper Frederiksen. "A practical implementation of Regev's LWE-based cryptosystem." Technical report, 2010. `http://daimi.au.dk/~jot2re/lwe/`

[FSV07] Junfeng Fan, Kazuo Sakiyama, and Ingrid Verbauwhede. "Montgomery modular multiplication algorithm for multi-core systems." In *Workshop on Signal Processing Systems: Design and Implementation* (SIPS), pp. 261–266, 2007.

[GFS+12] 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 CHES 2012*, pp. 512–529, 2012.

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. "Practical lattice-based cryptography: A signature scheme for embedded systems." In *Proceedings of CHES 2012*, LNCS, vol. 7428, pp. 530–547, Springer, 2012.

[GP03] GlobalPlatform. "GlobalPlatform card specification 2.1.1." 2003. `http://www.win.tue.nl/pinpasjc/docs/Card%20Spec%20v2.1.1%20v0303.pdf`

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. "Trapdoors for hard lattices and new cryptographic constructions." In *Proceedings of STOC 2008*, pp. 197–206, 2008.

[GT01] Michael T. Goodrich and Roberto Tamassia. "Algorithm design: Foundations, analysis, and internet examples." Wiley 2001.

[ISO7816-1] "ISO 7816 part 1: Physical characteristics of integrated circuit cards." Overview: `http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-1.aspx`.

[ISO7816-4] "ISO 7816 Part 4: Interindustry commands for interchange." Overview: `http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-4.aspx`.

[LP11] Richard Lindner and Chris Peikert. "Better key sizes (and attacks) for LWE-based encryption." In *Proceedings of CT-RSA 2011*, pp. 319–339, 2011.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings." In *Proceedings of EUROCRYPT 2010*, pp. 1–23, 2010.

[MR08] Daniele Micciancio and Oded Regev. "Lattice-based cryptography." In *Post-Quantum Cryptography*, pp. 147–191. Springer, 2008.

[OPG14] Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. "Beyond ECDSA and RSA: Lattice-based digital signatures on constrained devices." In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2014.

[PG12] Thomas Pöppelmann and Tim Güneysu. "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware." In *Cryptology - LATINCRYPT 2012*, LNCS, vol. 7533, pp. 139–158, Springer, 2012.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. "High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers." In *Cryptology - LATINCRYPT 2015*, LNCS, vol. 9230, pp. 346–365, Springer, 2015.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. "A framework for efficient and composable oblivious transfer." In *Proceedings of CRYPTO 2008*, pp. 554–571, 2008.

[Reg05] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography." *Journal of the ACM*, 56(6):34, pp. 1–40, 2009.

[RVM+14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. "Compact ring-LWE cryptoprocessor." In *Proceedings of CHES 2014*, pp. 371–391, 2014.

[Sun06] Sun Microsystems, Inc.. "Java Card platform specification 2.2.2." 2006. `http://www.oracle.com/technetwork/java/javacard/specs-138637.html`

[YCK+15] Ye Yuan, Chen-Mou Cheng, Shinsaku Kiyomoto, Yutaka Miyake, 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.