Computer Security Symposium 2017
23 - 25 October 2017

Efficient Implementation of discrete Gaussian sampling for Lattice-based Cryptography using JavaScript

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

Abstract: For the sake of new cryptographic algorithms to resist attacks from the quantum computers, the researches of post-quantum cryptography (PQC) has attracted much more attentions. Lattice-based cryptography, including lattice-based encryption and digital signature, has become one of the most popular research field in PQC. However, a number of lattice-based cryptographic schemes suffer from large-scale numbers sampled from discrete Gaussian distribution which could affect their efficiency remarkably. By using appropriate sampling methods, the performances of lattice-based cryptographic schemes could be more efficient under multiple circumstances. Certainly, most discrete Gaussian sampling methods need computations of high precision floating-point numbers which could be unsupported in some cases such as running on JavaScript platforms. In this paper, we propose the approaches to implement several classical discrete Gaussian sampling methods efficiently and compare them on several JavaScript platforms. We also implemented certain lattice-based encryption schemes and digital signature schemes for the security level of 128 bits by JavaScript. The results show that our sampling methods run efficiently in both digital signature schemes and encryption schemes on JavaScript platforms. After comparisons, we indicate the sampling method which has the best performance to these lattice-based cryptographic schemes.

Keywords: Post-Quantum cryptography, Lattice-based cryptography, Discrete Gaussian sampling, Digital signature, JavaScript platform

1. Introduction

Universally used public-key cryptography schemes such as RSA or elliptic curve cryptography (ECC), could not meet the needs of resisting attacks from the quantum computers since Shor's algorithm [28] was proposed. National Institute of Standards and Technology (NIST) had released the plan for new algorithms and will publish the adoption results in the next few years. In practice, lattice-based cryptography, whose security is arguably based on the hardness of well-studied lattice problems [1], is one of the most promising candidates in PQC. Since a number of initial constructions of lattice-based cryptography have been proposed [1], [11], [14], more lattice-based encryption schemes [26], [17] and digital signatures [5], [12], [20] with probable security have been put forward over the past years.

It is well-recognized that discrete Gaussian sampling (DGS) plays an important role in lattice-based cryptography. Each sampling method has its own dominant position and the performances of different sampling methods might be quite different under certain condition. Among the various sampling methods, rejection sampling method [7], [10], [18], inversion method [24], discrete Ziggurat method [2], and Knuth-Yao method [15] are the most classical. In general, rejection sampling is slow while it doesn't need too many storages. On the contrary, inversion method is fast but need to store the cumulative distribution function (CDF) of the sampled distribution in a relatively large look-up table. Discrete Ziggurat method is aimed at getting a speed-memory trade-off. The storage and speed are alterable when different numbers of rectangles are used [2]. Knuth-Yao method also needs to precompute probabilities and store them in a look-up table. It requires a minimal number of random bits and is well suited for high precision sampling [27].

Under various circumstances, the implementation of discrete Gaussian sampling faces different challenges. For many sampling methods, it is unavoidable to do high precision floating-point arithmetic which could be very hard or complicated to compute in some cases such as running on JavaScript platforms. At the mention of JavaScript, it is a kind of scripting language which had been widely used in Web application development. JavaScript programs could execute on multi-platforms (such as Windows, Linux, Android, iOS, etc.) owning to its cross-platform feature. How-

[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

ever, with the limited computational capacity, varieties of lattice-based cryptographic schemes could not be applied to JavaScript platforms which also need to defend future post-quantum attacks.

In this work, we propose the general solutions of high precision floating-point arithmetic for discrete Gaussian sampling by JavaScript. Several JavaScript platforms were chosen to measure the performances of the sampling methods. We have also implemented two lattice-based encryption schemes and a digital signature scheme. The first one is Bimodal Lattice Signature Scheme (BLISS) [5], which is one of the lattice-based signature schemes. Moreover, two encryption schemes with provable security are a multi-bit version of Regev's LWE-based scheme [8], [21], [25], [26] and an encryption scheme which we refer to LP11 [16]. After comparisons of sampling methods on JavaScript platforms, we apply them to the lattice-based encryption schemes and digital signature scheme. To the best of our knowledge, it is the first time that these sampling methods are implemented and applied to both digital signatures and encryption schemes in JavaScript environment. We will give detailed introductions of these lattice-based cryptographic schemes in the next section and the performance results will be analysed in section 4. The parameters have been selected from Léo Ducas et al. [5], Tore Kasper Frederiksen [8], and Lindner and Peikert [16] to provide the 128-bit security.

2. Lattice-based cryptography

In this section, we present the relevant mathematical background for discrete Gaussian sampling, BLISS, Regev's LWE, and LP11. Throughout this paper, we denote $\mathbb{Z}_q$ as the set of integers $[-q/2, -q/2+1, ..., q/2)$. Polynomials are denoted by bold italic small letters such as $\boldsymbol{f}$. Matrices are in the form of bold large letters such as $\mathbf{M}$, while vectors are denoted by bold small letters such as $\mathbf{v}$.

2.1 Discrete Gaussian sampling

Given a real number $\sigma > 0$ and mean $c \in \mathbb{R}$, the discrete Gaussian distribution over integers is denoted by $D_{c,\sigma}$. For each $x \in \mathbb{Z}$, the probability is proportional to $\exp(-\pi x/s^2)$ with mean 0, where $s = \sigma\sqrt{2\pi}$. Different from the continuous Gaussian distribution, we need to choose a tail-cut factor $t > 0$, therefore the sampled values locate in the range of $\{-t\sigma, ..., t\sigma\}$.

In general, discrete Gaussian sampling needs to do floating-point arithmetic. In order to ensure the accuracy of sampled values, a high precision is always used [4]. However, dealing with high precision floating-point numbers is unsupported in some cases. When applied to JavaScript platforms, we convert floating-point arithmetic to binary arithmetic. A look-up table is made to store binary values with finite precision. A more detailed introduction for the discrete Gaussian sampling methods will be discussed in section 3.

2.2 Bimodal lattice signature scheme

There already have varieties of researches about the provably-secure lattice-based signature schemes [13], [19], [20], [22]. Léo Ducas et al. [5] proposed the Bimodal lattice signature scheme (BLISS) originally. Oder et al. [23] and Boorghany et al. [3] have implemented this scheme on some constrained devices respectively in recent years. These works implied that the implementation of BLISS on multiple environments is feasible by the proper ways.

Let $n, q, d \in \mathbb{Z}$, q is a prime number and n is a power of 2 such that $q = 1 \pmod{2n}$. $\mathbb{Z}_q$ denotes a ring with the interval $[-q/2, q/2) \cap \mathbb{Z}$ and we define a quotient ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$ (thus $R_{2q} = \mathbb{Z}_{2q}[x]/(x^n+1)$). For every integer x in the range $[-q, q]$, $\lfloor x \rceil_d$ means the value obtained by dropping d low-order bits from x. Set $p = \lfloor 2q/2^d \rfloor$ and we denote H to be a standard hash function. Given real numbers $\delta_1, \delta_2 \geq 0$ as densities such that $\delta_1 + \delta_2 \in [0, 1]$, compute $d_1 = \lceil \delta_1 n \rceil$ and $d_2 = \lceil \delta_2 n \rceil$. Define $\zeta \in \mathbb{Z}$ such that $\zeta \cdot (q-2) = 1 \bmod 2q$. We denote $\|\cdot\|_2$ for the ℓ_2-norm and $\|\cdot\|_\infty$ for the ℓ_∞-norm. For an integer $\kappa \leqslant n$, $\mathbb{B}_\kappa^n$ is the set of binary vectors of length n and Hamming weight κ. The Table 1 shows the summary of BLISS.

Table 1 Summary of the implemented BLISS

Key Generation	1. Generate polynomial $\boldsymbol{f}$ with d_1 coefficients in $\{\pm 1\}$ and d_2 coefficients in $\{\pm 2\}$, all other coefficients are set to 0. Polynomial $\boldsymbol{f}$ should be invertible in R_{2q}; 2. Generate polynomial $\boldsymbol{g}$ with d_1 coefficients in $\{\pm 1\}$ and d_2 coefficients in $\{\pm 2\}$, all other coefficients are set to 0; 3. Compute $\boldsymbol{a}_q = (2\boldsymbol{g}+1)/\boldsymbol{f} \in R_q$; 4. The public key $\mathbf{A} = (\boldsymbol{a}_1, \boldsymbol{a}_2) = (2\boldsymbol{a}_q, q-2)$; 5. The secret key $\mathbf{S} = (\boldsymbol{s}_1, \boldsymbol{s}_2) = (\boldsymbol{f}, 2\boldsymbol{g}+1)$.
Signature	1. Sample $\boldsymbol{y}_1, \boldsymbol{y}_2 \leftarrow D_{\mathbb{Z}^n,\sigma}$; 2. Compute $\boldsymbol{u} = \zeta \cdot \boldsymbol{a}_1 \cdot \boldsymbol{y}_1 + \boldsymbol{y}_2 \in R_{2q}$; 3. For a message μ, compute $\boldsymbol{c'} = H(\lfloor \boldsymbol{u} \rceil_d \bmod p, \mu)$ and generate $\boldsymbol{c} \in \mathbb{B}_\kappa^n$ from $\boldsymbol{c'}$ according to a random oracle; 4. Generate an integer $b \in \{0, 1\}$ uniformly at random, compute $\boldsymbol{z}_1 = \boldsymbol{y}_1 + (-1)^b \boldsymbol{s}_1 \boldsymbol{c}$ and $\boldsymbol{z}_2 = \boldsymbol{y}_2 + (-1)^b \boldsymbol{s}_2 \boldsymbol{c}$; 5. Compress $\boldsymbol{z}_2$ by computing $\boldsymbol{z}_2^\dagger = (\lfloor \boldsymbol{u} \rceil_d - \lfloor \boldsymbol{u} - \boldsymbol{z}_2 \rceil_d) \bmod p$; 6. The signature is $(\boldsymbol{z}_1, \boldsymbol{z}_2^\dagger, \boldsymbol{c})$.
Verification	1. Given $B_2, B_\infty \in \mathbb{Z}$ as acceptance bounds; 2. Reject the signature if either $\|(\boldsymbol{z}_1 2^d \cdot \boldsymbol{z}_2^\dagger)\|_2 > B_2$ or $\|(\boldsymbol{z}_1 2^d \cdot \boldsymbol{z}_2^\dagger)\|_\infty > B_\infty$ is satisfied.; 3. Only accept if $\boldsymbol{c} = H(\lfloor \zeta \cdot \boldsymbol{a}_1 \cdot \boldsymbol{z}_1 + \zeta \cdot q \cdot \boldsymbol{c} \rceil_d + \boldsymbol{z}_2^\dagger \bmod p, \mu)$.

When a message μ is chosen, it is inescapable to do discrete Gaussian sampling since each message has to be signed online. For instance, when $n = 512$, one needs to sample $\boldsymbol{y}_1 = (y_0, y_1, ..., y_{511})$ and $\boldsymbol{y}_2 = (y'_0, y'_1, ..., y'_{511})$ from $D_{\mathbb{Z}^n,\sigma}$ for the signature. When a new message is chosen, sampling operations need to be executed again. Therefore, if dimension n becomes larger, more values have to be sampled from

Table 2 Summary of the selected parameters that provide about 128-bit security

Scheme	Parameters									Bit-security
BLISS	n	q	δ_1	δ_2	σ	κ	Dropped bits d	B_2	B_∞	128
	512	11289	0.3	0	107	23	10	11074	1563	

target discrete Gaussian distribution and more time is cost. Hence, it becomes an imperative work to optimize discrete Gaussian sampling methods as efficient as possible. We will introduce our optimization techniques in section 3. We select the parameters [5] as shown in Table 2 which has the 128-bits security and will give the experimental results in section 4.

2.3 Lattice-based encryption scheme

Besides digital signature, we will also investigate two encryption schemes which belong to lattice-based cryptography. The first one was the seminal scheme proposed by Regev [26] which we refer to as Regev's LWE in this paper.

Given positive integers m, n, l, q, t, r and a real $\alpha > 0$. In key generation, a matrix $\mathbf{S} \in \mathbb{Z}_q^{n\times l}$ is generated uniformly at random as the secret key. Then a matrix $\mathbf{A} \in \mathbb{Z}_q^{m\times n}$ is generated uniformly at random and a matrix $\mathbf{E} \in \mathbb{Z}_q^{m\times l}$ is chosen randomly from discrete Gaussian sampling. Compute $\mathbf{B} = \mathbf{AS} + \mathbf{E}$ and set the pair $(\mathbf{A}, \mathbf{B}) \in \mathbb{Z}_q^{m\times n} \times \mathbb{Z}_q^{m\times l}$ as the public key. For encryption, a vector $\mathbf{a} \in \{-r, -r+1, ..., r\}^m$ is generated uniformly at random. For a message $\mathbf{m} \in \mathbb{Z}_t^l$, we output the ciphertext $(\mathbf{u}, \mathbf{c}) = (\mathbf{A}^T\mathbf{a}, \mathbf{B}^T\mathbf{a} + f(\mathbf{m})) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^l$ where $f : \mathbb{Z}_t^l \to \mathbb{Z}_q^l$ is the encode function. In decryption, the plaintext $\mathbf{m}$ is obtained by computing $f^{-1}(\mathbf{c} - \mathbf{S}^T\mathbf{u}) \in \mathbb{Z}_t^l$ where $f^{-1} : \mathbb{Z}_q^l \to \mathbb{Z}_t^l$ is the decode function.

The second one was proposed by Lindner and Peikert [16] and we refer to as LP11 in this paper. Similar as Regev's LWE, the functions for encoding and decoding are also integral parts in the scheme. Given positive integers n, q and a real $s > 0$. In key generation, matrices $\mathbf{R}_1, \mathbf{R}_2 \in \mathbb{Z}_q^{n\times n}$ are generated randomly from discrete Gaussian sampling and $\mathbf{R}_2$ is set to be the secret key. Then a matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$ is generated uniformly at random. Compute $\mathbf{B} = \mathbf{R}_1 - \mathbf{AR}_2 \in \mathbb{Z}_q^{n\times n}$. The pair $(\mathbf{A}, \mathbf{B}) \in \mathbb{Z}_q^{n\times n} \times \mathbb{Z}_q^{n\times n}$ is set to be the public key. For encryption, $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \in \mathbb{Z}_q^{n\times n}$ are chosen according to discrete Gaussian sampling. For the message $\mathbf{m} \in \{0,1\}^n$, we output the ciphertext $[\mathbf{c}_1^t\ \mathbf{c}_2^t] = [\mathbf{e}_1^t\ \mathbf{e}_2^t\ \mathbf{e}_3^t + f(\mathbf{m})^t] \cdot \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{I} & \\ & \mathbf{I} \end{bmatrix} \in \mathbb{Z}_q^{1\times 2n}$. The decryption operation outputs $f^{-1}(\mathbf{c}_1^t \cdot \mathbf{R}_2 + \mathbf{c}_2^t)^t \in \{0,1\}^n$.

Both Regev's LWE and LP11 need to sample a plenty of values from discrete Gaussian sampling (DGS). However, DGS is only required in key generation of Regev's LWE, while it exists in both key generation and encryption stages of LP11.

3. Efficient algorithms using JavaScript

In this section, we describe our implementation techniques for four discrete Gaussian sampling methods by JavaScript.

3.1 Rejection sampling algorithm

In practice, rejection sampling doesn't need to do a varieties of precomputations since it generates random values each time. However, if computing high precision floating-point number is beyond the capacity of some devices or programming languages, we need to compute some values and store them in a look-up table in advance.

Let $t > 0$ be the tail-cut factor, for a real number $\sigma > 0$, the sampled value is chosen uniformly at random from the range $\{-t\sigma, ..., t\sigma\}$. In our case, we precompute all the numbers in the range and convert them to their binary expansions. Both integer and decimal parts of the floating-point number should be stored in a look-up table. There is no doubt that the probability in floating-point form of any sampled value $x \in \mathbb{Z} \cap [-t\sigma, t\sigma]$ is greater than 0. However, the binary expansions with finite precision of some probabilities equal to 0, so that we do not need to store them in the look-up table.

Given $l, n \in \mathbb{Z}$, l is the precision of binary expansion of the probabilities in the range $\{-t\sigma, t\sigma\}$, while n means the number of binary expansions which are greater than 0. We store these probabilities $\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{n-1} \in \mathbb{Z}_2^{l+1}$ in a two-dimensional probability array $\mathbf{P} = (\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{n-1}) \in \mathbb{Z}_2^{n\times(l+1)}$ as our look-up table for algorithm 1. More memory could be saved by sampling an integer $x \in \mathbb{Z} \cap [0, t\sigma]$ since the target discrete Gaussian distribution is symmetric. If $x = 0$, we accept with probability $1/2$, otherwise a sign bit $s \in \{-1, 1\}$ is generated uniformly at random and return sx. Algorithm 1 shows our method of rejection sampling algorithm by JavaScript.

3.2 Inversion sampling algorithm

Different from rejection sampling method, inversion method precomputes and stores the cumulative distribution function (CDF) instead of dealing with the probability distribution function (PDF) of the sampled distribution. However, when applied to JavaScript platforms, we need to convert the precomputed values to their binary expansions with finite precision.

Given a tail-cut factor $t > 0$ and a real number $\sigma > 0$, we use Ψ to denote the CDF of $D_{\mathbb{Z},\sigma}$. Precompute the values of $\Psi(x)$ where $x \in \mathbb{Z} \cap [-t\sigma, t\sigma]$ and change the ratio of each value by the same way to make sure the last value equals to 1. Let $l \in \mathbb{Z}$ be the precision of binary expansions of these values. Let $n \in \mathbb{Z}$, there are n binary expansions of the values $\boldsymbol{\varphi}_0, \boldsymbol{\varphi}_1, ..., \boldsymbol{\varphi}_{n-1} \in \mathbb{Z}_2^{l+1}$. We store these values in a two-dimensional array $\boldsymbol{\Phi} = (\boldsymbol{\varphi}_0, \boldsymbol{\varphi}_1, ..., \boldsymbol{\varphi}_{n-1}) \in \mathbb{Z}_2^{n\times(l+1)}$ as the look-up table. When a l-bit precision value u whose

Algorithm 1: Rejction sampling algorithm (RS)

Input: $n, l, t \in \mathbb{Z}, \sigma \in \mathbb{R}$,
$\mathbf{P} = (\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{n-1}) \in \mathbb{Z}_2^{n \times (l+1)}$
Output: Sample value $s \in \mathbb{Z} \cap [-t\sigma, t\sigma]$

1 Let $s = 0$
2 Let $\mathbf{a} = (a_0, a_1, ..., a_l) \in \mathbb{Z}^{l+1}$
3 Let $\mathbf{b} = (b_0, b_1, ..., b_l) \in \mathbb{Z}^{l+1}$
4 **while** true **do**
5 $s \leftarrow \{0, 1, ..., n-1\}$ uniformly at random
6 **if** $s = 0$ **then** $b \leftarrow \{0, 1\}$ uniformly at random
7 **if** $b = 0$ **then return** s
8 **else continue**
9 **for** $j = 0$ to l by 1 **do**
10 $a_j = \mathbf{P}[s][j]$
11 Let $b_0 = 0$
12 **for** $i = 1$ to l by 1 **do**
13 $b_i \leftarrow \{0, 1\}$ uniformly at random
14 **for** $m = 0$ to l by 1 **do**
15 **if** $b_m < a_m$ **then** $sign \leftarrow \{-1, 1\}$ uniformly at random
16 **return** $sign * s$
17 **if** $b_m > a_m$ **then break**

decimal expansion locates in the range $[0, 1)$ is generated uniformly at random, we use the binary search to scan the look-up table and get the sampled value from it eventually. Algorithm 2 shows the inversion sampling algorithm by JavaScript.

Algorithm 2: Inversion sampling algorithm (IS)

Input: $n, l, t \in \mathbb{Z}, \sigma \in \mathbb{R}$,
$\boldsymbol{\Phi} = (\boldsymbol{\varphi}_0, \boldsymbol{\varphi}_1, ..., \boldsymbol{\varphi}_{n-1}) \in \mathbb{Z}_2^{n \times (l+1)}$
Output: Sample value $s \in \mathbb{Z} \cap [-t\sigma, t\sigma]$

1 Let $s = 0, beginindex = 0, endindex = n - 1$
2 Let $\mathbf{a} = (a_0, a_1, ..., a_l) \in \mathbb{Z}^{l+1}$
3 Let $\mathbf{b} = (b_0, b_1, ..., b_l) \in \mathbb{Z}^{l+1}$
4 Let $\mathbf{c} = (c_0, c_1, ..., c_l) \in \mathbb{Z}^{l+1}$
5 **for** $i = 0$ to l by 1 **do**
6 $a_i \leftarrow \{0, 1\}$ uniformly at random
7 **while** $beginindex \leq endindex$ **do**
8 Let $temp = 0, midindex = (beginindex + endindex)/2$
9 **for** $j = 0$ to l by 1 **do**
10 $b_j = \boldsymbol{\Phi}[midindex][j]$
11 **for** $m = 0$ to l by 1 **do**
12 $c_m = \boldsymbol{\Phi}[midindex - 1][m]$
13 **for** $k = 0$ to l by 1 **do**
14 **if** $a_k < b_k$ **then**
15 **for** $q = 0$ to l by 1 **do**
16 **if** $a_k > c_k$ **then return** $midindex$
17 **else** $endindex = midindex - 1$
18 **else if** $a_k > b_k$ **then** $beginindex = midindex + 1$
19 **else if** $a_k = b_k$ **then** $temp = temp + 1$
20 **if** $temp = l + 1$ **then return** $midindex$

3.3 Discrete Ziggurat algorithm

Classical rejection sampling aspires for less memory but has to spend much more time, while inversion method aspires for higher speed with a larger look-up table, the discrete Ziggurat sampling method proposed by Buchmann et al. [2] allows for a flexible speed-memory trade-off. Some horizontal rectangles are settled to cover the target probability distribution. When there are more rectangles, more values need to be stored in the look-up table and therefore the efficiency is improved. However, it is not a wise choice to set too many rectangles because an oversized look-up table is also a notable problem. Hence, a balance between memory and speed is essential.

Same as rejection sampling and inversion sampling method, the sampled value is chosen uniformly at random from the range $\{-t\sigma, ..., t\sigma\}$. Let $m \in \mathbb{Z}$ be the number of horizontal rectangles. Given $l, n \in \mathbb{Z}$, l is the precision of binary expansion of the probabilities while n means the number of binary expansions which are greater than 0. Both integer and decimal parts of the floating-point number have to be stored in a look-up table.

In our case, four look-up tables are necessary. A two-dimensional array $\mathbf{P} = (\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{n-1}) \in \mathbb{Z}_2^{n \times (l+1)}$ which stores the probabilities of all sampled values is set to be the first look-up table. The second look-up table stores all the $x-$coordinates of rectangles, denoted by $\mathbf{x} = (x_0, x_1, ..., x_m) \in \mathbb{Z}^{m+1}$ and the third look-up table stores all the y-coordinates of rectangles, denoted by $\mathbf{Y} = (\mathbf{y}_0, \mathbf{y}_1, ..., \mathbf{y}_m) \in \mathbb{Z}_2^{(m+1) \times (l+1)}$. In addition, we compute the difference between neighboring y-coordinates $\mathbf{r}_i = \mathbf{y}_i - \mathbf{y}_{i+1} \in \mathbb{Z}_2^{l+1}$ for $i \in \mathbb{Z} \cap [0, m)$ and store them in the last look-up table. Since there are some leading zeros in each $\mathbf{r}_i$ for $i \in \mathbb{Z} \cap [0, m)$, a sign bit is set to represnent the numbers of leading zeros. When the look-up table is scanned, we could skip these leading zeros directly which the efficiency could be improved a lot. Figure 1 gives an example of four values with 4-bit precision, the first column of $\mathbf{R}$ shows the number of leading zeros in each row.

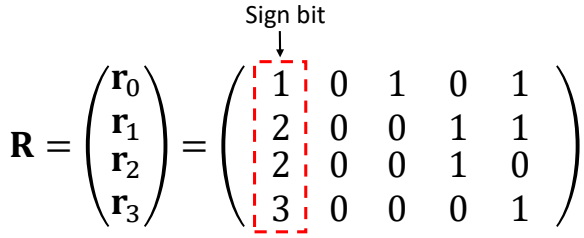

Fig. 1 The optimized array $\mathbf{R}$

Algorithm 3 is our method of discrete Ziggurat sampling by JavaScript. In step 12 of Algorithm 3, the method $generate()$ could generate a $(l+1)$-bit precision binary number less than or equal to the inputted value.

3.4 Knuth-Yao algorithm

Donald E. Knuth and Andrew C. Yao proposed an algorithm which aims to sample values from the non-uniform distribution [15]. Knuth-Yao algorithm precomputes the probabilities of sampled values and uses a special way to construct the look-up table [9], [27].

Given a tail-cut factor $t > 0$ and a real number $\sigma > 0$. Let $l, n \in \mathbb{Z}$, we make a probability matrix $\mathbf{P}_{mat} =$

Algorithm 3: Discrete Ziggurat sampling algorithm (DZ)

Input: $n, l, t, m \in \mathbb{Z}, \sigma \in \mathbb{R}$, $\mathbf{P} = (\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{n-1}) \in \mathbb{Z}_2^{n\times(l+1)}$, $\mathbf{Y} = (\mathbf{y}_0, \mathbf{y}_1, ..., \mathbf{y}_m) \in \mathbb{Z}_2^{(m+1)\times(l+1)}$, $\mathbf{x} = (x_0, x_1, ..., x_m) \in \mathbb{Z}^{m+1}$, $\mathbf{R} = (\mathbf{r}_0, \mathbf{r}_1, ..., \mathbf{r}_{m-1}) \in \mathbb{Z}_2^{m\times(l+1)}$.
Output: Sample value $s \in \mathbb{Z} \cap [-t\sigma, t\sigma]$

Let $\mathbf{a} = (a_0, a_1, ..., a_l) \in \mathbb{Z}^{l+1}$, $\mathbf{b} = (b_0, b_1, ..., b_l) \in \mathbb{Z}^{l+1}$, $\mathbf{c} = (c_0, c_1, ..., c_l) \in \mathbb{Z}^{l+1}$
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 $0 < x \le x_{i-1}$ **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 4: Optimized generate algorithm

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** $\mathbf{a}'$

$(\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_{n-1}) \in \mathbb{Z}_2^{n\times l}$ where each $\mathbf{p}_i \in \mathbb{Z}_2^l$ is the binary expansion of the corresponding probability. In other words, each row of the matrix stores one probability. When applied to JavaScript platforms, the probability matrix $\mathbf{P}_{mat}$ is divided into l blocks $\mathbf{k}_0, \mathbf{k}_1, ..., \mathbf{k}_{l-1} \in \mathbb{Z}_2^n$. Each block is one of the columns of $\mathbf{P}_{mat}$. A one-dimensional array $\mathbf{k} = (\mathbf{k}_0, \mathbf{k}_1, ..., \mathbf{k}_{l-1}) \in \mathbb{Z}_2^{ln}$ is stored as our look-up table. Figure 2 shows an example of four probabilities with 5-bit precision. We store the probabilities in a probability matrix $\mathbf{P}_{mat}$, then it is divided into 5 blocks. An array $\mathbf{k}$ stores each block in sequence as the look-up table. Similarly, it uses less memory if the algorithm only stores the probabilities of sampled value $x \in \mathbb{Z} \cap [0, t\sigma]$.

The look-up table could be optimized since there are many

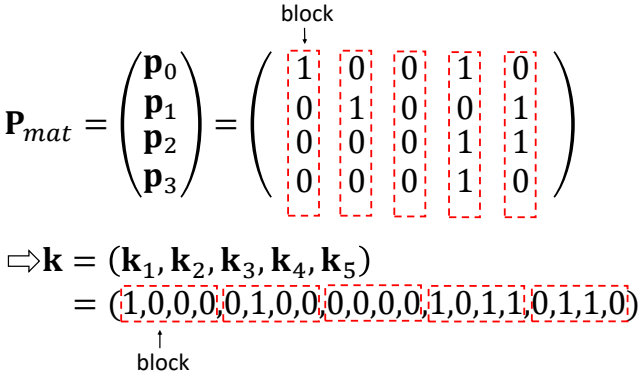

Fig. 2 Probability array $\mathbf{k}$ including l blocks

leading zeros in each block of the probability array $\mathbf{k}$. These zeros can be compressed [6] to reduce the size of our look-up table. Figure 3 shows that the new array $\mathbf{k}'$ is obtained by storing the values at the left side of dotted line and a sign bit which represents the difference between neighboring block is added to each $\mathbf{k}'_i$ for $i \in \mathbb{Z} \cap [0, n-1]$. This method also accelerates the speed remarkablely since Knuth-Yao algorithm can skip the leading zeros when it scans the look-up table. Note that if all values in a block are zeros, it is impossible to sample value from the block, therefore we will delete the whole block. However, even if these blocks are not stored, we still need to generate numbers for each block, in other words, these blocks still have meanings to sampling. By deleting them from the array, we save a lot of time to scan these blocks.

Algorithm 5 shows the optimized Knuth-Yao algorithm.

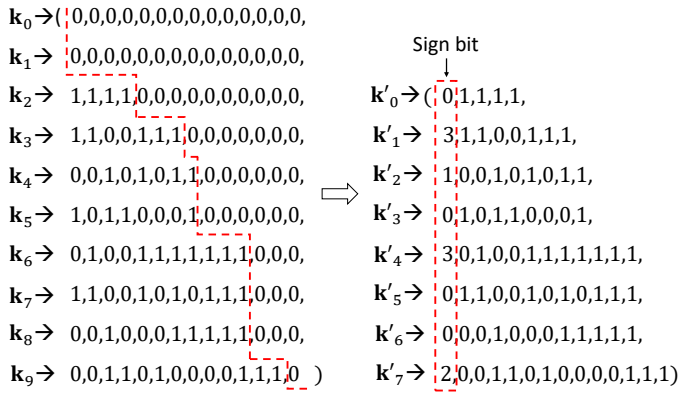

Fig. 3 Optimized probability array by deleting zeros at the right side of dotted line, the sign bit indicates the difference between two consecutive block lengths

Let $q \in \mathbb{Z}$ represents the numbers of the first several blocks whose elements are all zeros. Let $h \in \mathbb{Z}$ be the total length of all blocks which have been scanned, and $g \in \mathbb{Z}$ be the length of probability array $\mathbf{k}' = (\mathbf{k}'_0, \mathbf{k}'_1, ..., \mathbf{k}'_{l-1}) = (k'_0, k'_1, ..., k'_{g-1})$. In step 1, 21, and 22 of Algorithm 5, $FirstBlockLength$ means the length of the first block in the optimized probability array $\mathbf{k}'$.

Algorithm 5: Optimized Knuth-Yao algorithm (KYO)

Input: $g, q \in \mathbb{Z}$, $\mathbf{k}' = (k'_0, k'_1, ..., k'_{g-1}) \in \mathbb{Z}_2^g$
Output: Sample value $s \in \mathbb{Z} \cap [-t\sigma, t\sigma]$

1 Let $d = 0$, $len = FirstBlockLength$, $sign = 0$, $h = 0$
2 **for** $j = 0$ up to $q - 1$ by 1 **do**
3 $r \leftarrow \{0, 1\}$ uniformly at random
4 $d = 2d + r$
5 **while** true **do**
6 $r \leftarrow \{0, 1\}$ uniformly at random
7 $d = 2d + r$
8 **for** $i = len - 1$ down to 0 by 1 **do**
9 $d = d - k'_{i+1+h}$
10 **if** $d = -1$ **then**
11 **if** $i = 0$ **then** $sign \leftarrow \{0, 1\}$ uniformly at random
12 **else**
13 $sign \leftarrow \{-1, 1\}$ uniformly at random
14 **return** $s = sign * i$
15 **if** $sign = 1$ **then return** $s = i$
16 **else**
17 $d = 0$
18 **for** $j = 0$ up to $q - 1$ by 1 **do**
19 $r \leftarrow \{0, 1\}$ uniformly at random
20 $d = 2d + r$
21 $len = FirstBlockLength$
22 $h = -FirstBlockLength - 1$
23 **continue**
24 $h = h + len + 1$
25 **if** $k'_h > 0$ **then**
26 $len += k'_h$

4. Performance on Web browsers

In this section, we introduce the implementation platforms of PC[*1] Web browsers and report the performance results of running discrete Gaussian sampling methods and lattice-based cryptographic schemes on certain Web browsers.

4.1 Implementation platforms

Note that even the same JavaScript program has the different performance results when applied to different Web browsers. In our implementations, we have chosen four widely used PC browsers as our benchmark platforms, namely, Google Chrome, Firefox, Opera and Microsoft Edge. We will describe the more detailed performance results in the next section.

4.2 Performance results of discrete Gaussian sampling on Web browsers

The discrete Gaussian sampling methods were carried out on chosen Web browsers. With precision $l = 128$ bits, we selected the standard deviation $\sigma = 3.3311, 55.5649, 107$, [16], [21], [5], respectively. The tail-cut factor t is set to 13 to meet the minimum statistical distance. The number of rectangles is 63 for discrete Ziggurat algorithm [2]. As shown in Table 3, we compared the performance of four sampling methods: rejection sampling algorithm (RS, algorithm 1), inversion sampling algorithm (IS, algorithm 2), discrete Ziggurat algorithm with optimization by using algorithm 4 (DZO), and Knuth-Yao algorithm with optimization (KYO, algorithm 5) executing on Oprea. The speed of four methods is measured by the number of values sampled in each millisecond and the storage is measured by Kbytes.

Apparently, KYO has the better performance than other methods for both speed and storage. For instance, the speed of KYO is about 67.51 times, 22.6 times, and 10.48 times faster than RS, IS, DZO for $\sigma = 3.3311$, respectively. In addition, with $\sigma = 107$, the storage of KYO is 13.75 Kbytes, which is a few less than 19064 bytes ($\approx$ 18.62 Kbytes) from [23]. Figure 4 shows the total experimental results for DZO, RS , IS and KYO with $\sigma = 3.3311, 55.5469, 107$ running on Google Chrome, Opera, Microsoft Edge and Firefox, respectively. It implies that KYO has the best performance on whichever of four Web browsers.

4.3 Performance results of lattice-based cryptographic schemes on Opera

After the comparisons of discrete Gaussian sampling

[*1] The test PC has the following specifications:
CPU: Intel(R) Core(TM) i7-6500U @ 2.5-3.1GHz;
Memory: 8GB DDR3L RAM;
Hard disk: 1TB 5400rpm;
OS: Windows 10 Pro x64;
Java(JDK) version: jdk1.8.0_131;
JavaScript version: 1.3;
Web browser: Google Chrome 60.0.3112.90; Firefox 54.0.1; Opera 47.0.2631.55; Microsoft Edge 38.14393.1066.0.

Table 3 Experimental Results for discrete Ziggurat, rejection sampling , inversion sampling and Knuth-Yao

	Rejection sampling		Inversion sampling		discrete Ziggurat		Knuth-Yao	
	Speed	Storage	Speed	Storage	Speed	Storage	Speed	Storage
$\sigma = 3.3311$	25.5493	0.71	179.2115	0.71	657.8947	2.74	5882.3529	0.48
$\sigma = 55.5649$	24.9813	11.75	83.4028	11.75	602.4096	13.75	1428.5714	7.19
$\sigma = 107$	24.6063	22.63	53.9374	22.63	544.7126	24.63	862.069	13.75

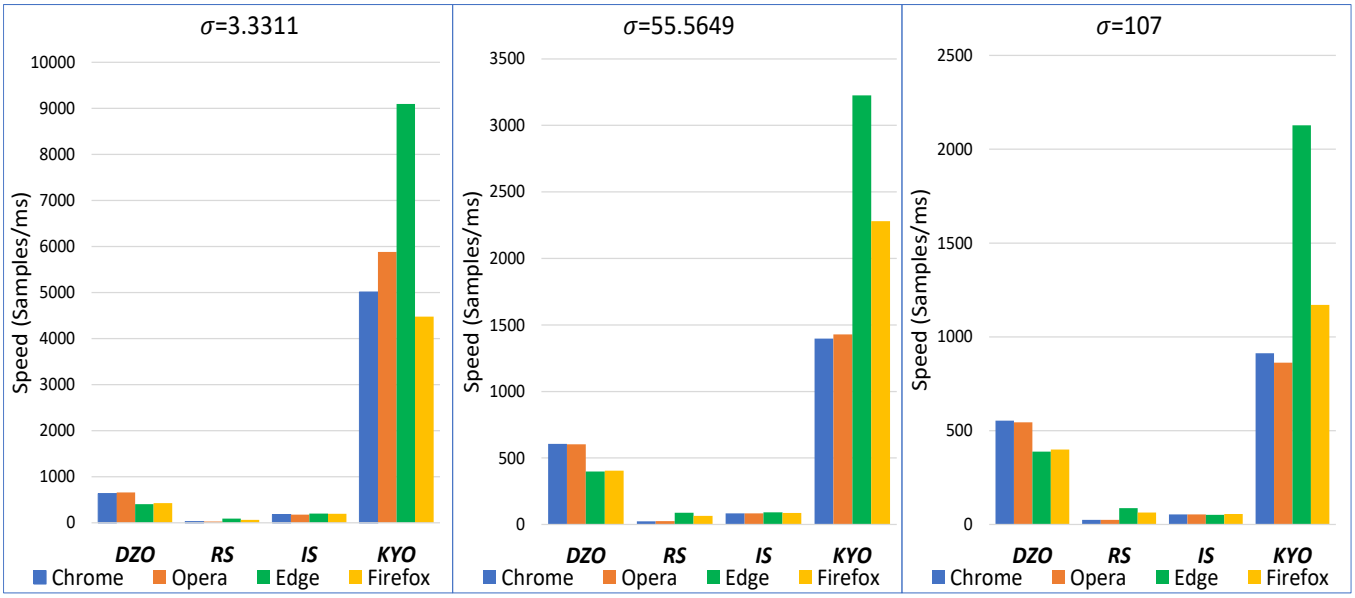

Fig. 4 Results for DZO, RS, IS , and KYO for $\sigma = 3.3311, 55.5649, 107$ respectively on four Web browsers.

(DGS) on several Web browsers, we apply KYO to BLISS, Regev's LWE, and LP11, respectively. We also run these lattice-based cryptographic schemes on Google Chrome, Opera, Microsoft Edge and Firefox and list the performance results of them executing on Opera in Table 5. Note that BLISS needs to sample values from DGS when in sign operation, while Regev's LWE generates random values from DGS for key generation and LP11 generates random values from DGS in the stages of key generation and encryption. Table 4 shows the performance result of BLISS on Opera. The results of Regev's LWE and LP11 are listed in Table 4 of [29]. Table 5 shows the performance results of KYO in BLISS, Regev's LWE and LP11 on Opera respectively. Compare to [29] for Regev's LWE and LP11, our method has better performance running in LP11, the speed is about 1 time faster.

Certainly, both speed and storage of discrete Gaussian sampling methods still have large optimize space. Not only the standard deviation and dimension could effect the performance remarkably, but also there are some factors which are always ignored. Under the condition of satisfying the minimum statistical distance, when the precision of floating-point numbers becomes larger, the look-up table will then become larger. In the meantime, more time is spent to scan the look-up table. Accordingly, the speed of discrete Gaussian sampling will decrease. Moreover, the way of storing and searching the look-up table could have a notable effection on efficiency, especially for binary arithmetic.

Table 4 Performance results on Opera

Scheme	Average running time (ms)			Security
	Key generation	Signature	Verification	
BLISS	130.33	3.196	0.211	128 bits

Table 5 Results of KYO in BLISS, Regev's LWE and LP11 on Opera

Scheme	Average running time of DGS (ms)		
	Signature	Key Generation	Encryption
BLISS	1.18	-	-
Regev's LWE	-	910.59	-
LP11	-	3.57	0.065

5. Conclusions

We have implemented and optimized four well-known discrete Gaussian sampling methods: rejection sampling method, inversion method, discrete Ziggurat method and Knuth-Yao method by JavaScript. Using different parameter sets, we compared their performances on multiple JavaScript platforms. According to the performance results, we have found that our proposed Knuth-Yao algorithm with optimization (KYO, algorithm 5) is the most efficient method. In addition, several lattice-based encryption schemes and digital signature schemes have been implemented using JavaScript. We applied the proposed Knuth-Yao

algorithm to these schemes and tested their performance on several JavaScript platforms. However, there still have a lot of work to do, we expect to do further optimization of discrete Gaussian sampling and apply the sampling methods to more lattice-based cryptographhic schemes on various platforms and devices in the future.

References

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

[2] J. Buchmann, D. Cabarcas, F. Göpfert, A. Hülsing, and P. Weiden. "Discrete Ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers." In *Selected Areas in Cryptography - SAC '13*, Burnaby, BC, Canada, (P. Lisonek et al., eds.), LNCS, vol. 8282, pp. 402–417, 2013.

[3] A. Boorghany, S. B. Sarmadi, and R. Jalili. "On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards." In *ACM TECS*, volume 14 issue 3, pp. 42:1–42:25, 2015.

[4] D. Cabarcas, P. Weiden, and J. Buchmann. "On the efficiency of provably secure NTRU." In *Proceedings of PQCrypto 2014*, pp. 22–39, 2014.

[5] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. "Lattice Signatures and Bimodal Gaussians." In *IACR Cryptology ePrint Archive*, 2013:383, 2013. To appear at CRYPTO 2013.

[6] N. Dwarakanath and S. Galbraith. "Sampling from Discrete Gaussians for Lattice-based Cryptography on a Constrained Device." In *Applicable Algebra in Engineering, Communication and Computing*, vol. 25, no. 3, pp. 159–180, 2014.

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

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

[9] S. D. Galbraith and N. C. Dwarakanath. "Efficient sampling from discrete Gaussians for lattice-based cryptography on a constrained device." *Preprint*, 2012.

[10] N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss. "On the design of hardware building blocks for modern lattice-based encryption schemes." In *Proceedings of CHES 2012*, pp. 512–529, 2012.

[11] O. Goldreich, S. Goldwasser, and S. Halevi. "Public-key cryptosystems from lattice reduction problems." In *Advances in cryptology*, vol. 1294 of *Lecture Notes in Comput. Sci.*, pp. 112-131. Springer, 1997.

[12] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann. "Practical lattice-based cryptography: A signature scheme for embedded systems." In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012*, vol. 7428 of *Lecture Notes in Computer Science*, pp. 530–547, Leuven, Belgium, September, 2012. Springer, Berlin, Germany.

[13] C. Gentry, C. Peikert, and V. Vaikuntanathan. "Trapdoors for hard lattices and new cryptographic constructions." In *Proceedings of STOC 2008*, pp. 197–206, 2008.

[14] J. Hoffstein, J. Pipher, and J. H. Silverman. "NTRU: A ring-based public key cryptosystem." In Joe Buhler, editor, *ANTS*, vol. 1423 of *Lecture Notes in Computer Science*, pp. 267–288. Springer, 1998.

[15] Donald E. Knuth and Andrew C. Yao. "The complexity of non uniform random number generation." In *Algorithms and complexity: New directions and recent results*, pp. 357–428, 1976.

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

[17] V. Lyubashevsky, C. Peikert, and O. Regev. "On ideal lattices and learning with errors over rings." In *Proceedings of EUROCRYPT 2010*, pp. 1–23, 2010.

[18] V. Lyubashevsky. "Lattice-based identification schemes secure under active attacks." In Ronald Cramer, editor, *PKC 2008: 11th International Conference on Theory and Practice of Public Key Cryptography*, vol. 4939 of *Lecture Notes in Computer Science*, pp. 162-179, Barcelona, Spain, 2008. Springer, Berlin, Germany.

[19] V. Lyubashevsky. "Fiat-Shamir with Aborts: Applications to Lattice and Factoring – Based Signatures." In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, no. 5912 in Lecture Notes in Computer Science, pp. 598–616. Springer, 2009.

[20] V. Lyubashevsky. "Lattice Signatures without Trapdoors." In *Proceedings of the 31st Annual international conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPY'12, pp. 738–755, Berlin, 2012. Springer-Verlag.

[21] D. Micciancio and O. Regev. "Lattice-based cryptography." In *Post-Quantum Cryptography*, pp. 147–191. Springer, 2008.

[22] D. Micciancio, and C. Peikert. "Trapdoors for lattices: Simpler, tighter, faster, smaller." In *Advances in Cryptology-EUROCRYPT 2012*, vol. 7237 of *Lectures Notes in Computer Science*, pp. 700-718, Cambridge, UK, April, 2012. Springer, Berlin, Germany.

[23] T. Oder, T. Pöppelmann, and T. Güneysu. "Beyond ECDSA and RSA: Lattice-based Digital Signatures on Constrained Devices." In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pp. 110:1–110:6, New York, NY, USA, 2014. ACM.

[24] Chris Peikert. "An effcient and parallel Gaussian sampler for lattices." In Tal Rabin, editor, *Advances in Cryptology-CRYPTO 2010*, vol. 6223 of *Lecture Notes in Computer Science*, pp. 80–97, 2010.

[25] C. Peikert, V. Vaikuntanathan, and B. Waters. "A framework for efficient and composable oblivious transfer." In *Proceedings of CRYPTO 2008*, pp. 554–571, 2008.

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

[27] S. S. Roy, F. Vercauteren, and I. Verbauwhede. "High Precision Discrete Gaussian Sampling on FPGAs." In *Selected Areas in Cryptography - SAC 2013*, ser. *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 383–401.

[28] P. W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring." In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pp. 124–134. IEEE, 1994.

[29] Y. Yuan, C. M. Cheng, S. Kiyomoto, Y. Miyake, and T. Takagi, "Portable implementation of lattice-based cryptography using JavaScript." In *International Journal of Networking and Computing*, vol. 6, no. 2, pp. 309–327, 2016.