On Rejection Sampling Algorithms for Centered Discrete Gaussian Distribution over Integers

Yusong Du and Baodian Wei
School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510006, China
duyusong@mail.sysu.edu.cn

Abstract

Lattice-based cryptography has been accepted as a promising candidate for public key cryptography in the age of quantum computing. Discrete Gaussian sampling is one of fundamental operations in many lattice-based cryptosystems. In this paper, we discuss a sub-problem of discrete Gaussian sampling, which is to sample from a centered discrete Gaussian distribution $D_{\mathbb{Z},\sigma,c}$ over the integers $\mathbb{Z}$ with parameter $\sigma > 0$ and center $c = 0$. We propose three alternative rejection sampling algorithms for centered discrete Gaussian distributions with parameter σ in two specific forms. The first algorithm is designed for the case where σ is an positive integer, and it requires neither pre-computation storage nor floating-point arithmetic. While the other two algorithms are fit for parameter $\sigma = k \cdot \sqrt{1/(2 \cdot \ln 2)}$ with a positive integer k, and they require fixed look-up tables of very small size (e.g. 128 bits and 320 bits respectively) but are much more efficient than the first algorithm. The experimental results show that our algorithms have better performance than that of two rejection sampling algorithms proposed by Karney in 2016 and by Ducas et al. in 2013 respectively. The expected numbers of random bits used in our algorithms are significantly smaller than that of random bits used in Karney's rejection sampling algorithm.

1 Introduction

Lattice based cryptography has gained much popularity in recent years. Many classical cryptographic primitive can be realized efficiently using lattice, providing conjectured security against quantum computers. Some advanced schemes that go beyond classical public key cryptography can also be built up based on lattices, like fully homomorphic encryption (FHE) [5, 13], identity based encryption (IBE) [2, 1], attribute based encryption (ABE) [4] and (some forms of) multi-linear maps [10, 12]. Lattice-based cryptography has been considered as a promising candidate for public key cryptography in the age of quantum computing.

Discrete Gaussian sampling, which is to sample from a discrete Gaussian distribution $D_{\Lambda,\sigma,\mathbf{c}}$ with parameter $\sigma > 0$ and center $\mathbf{c} \in \mathbb{R}^n$ over an n-dimensional lattice Λ, plays a fundamental role in lattice-based cryptography. Discrete Gaussian sampling is not only one of fundamental operations in many lattice-based cryptosystems but also at the core of security proofs of these cryptosystems [22, 14, 27, 24, 19]. It has been considered by the cryptography research community as one of the fundamental building blocks of lattice-based cryptography [25, 21, 7, 20, 11, 23].

An important sub-problem of discrete Gaussian sampling is to sample from a discrete Gaussian distribution $D_{\mathbb{Z},\sigma,c}$ over the integers $\mathbb{Z}$ with parameter $\sigma > 0$ and center $c \in \mathbb{R}$. Sampling from a discrete Gaussian distribution over the integers $\mathbb{Z}$, denoted by Sample$\mathbb{Z}$, is usually a kernel subroutine in discrete Gaussian sampling algorithms for a distribution over a general n-dimensional lattice. Furthermore, since Sample$\mathbb{Z}$ is far more efficient and simpler than discrete Gaussian sampling over a general lattice, in some lattice-based cryptosystems, including some encryption and signature schemes [18, 16, 19, 3], the operations involving discrete Gaussian sampling are just Sample$\mathbb{Z}$. In recent years, therefore, some efforts have been made to improve the performance of Sample$\mathbb{Z}$.

The commonly used methods (techniques) for sampling from a discrete Gaussian distribution over the integers $\mathbb{Z}$ are the inversion method (using a cumulative distribution table), the rejection sampling method [14, 7, 15], the Knuth-Yao method (using a discrete distribution generating (DDG) tree) [6, 28, 9], and the 'reduction and recombination' technique [26, 23], which was developed from the convolution properties of (discrete) Gaussian distributions [25]. From an implementation standpoint, a good sampling algorithm for a discrete Gaussian distribution (not necessarily over the integers $\mathbb{Z}$) should not only be fast, but also have a negligible statistical distance to the target distribution. From the perspective of complexity, a good sampling algorithm should have low entropy consumption, i.e., it uses a smaller number of random bits on average to generate a sample from the target distribution. The methods of sampling from a continuous Gaussian distribution are not trivially applicable for the discrete case. A good sampling algorithm that is not only fast but also accurate for discrete Gaussian distributions is very important for the implementations of lattice-based cryptosystems.

1.1 Related Work

The first Sample$\mathbb{Z}$ algorithm, which was given by Gentry et al. in [14], uses rejection sampling method. This algorithm is not very efficient, because it uses (high-precision) floating-point arithmetic and requires at least about 10 trials on average before outputting an integer in order to get a negligible statistical distance to the target discrete Gaussian distribution over the integers $\mathbb{Z}$. In 2013, Ducas et al. present an efficient sampling algorithm for centered discrete Gaussian distributions (center $c = 0$) [7]. It uses rejection sampling from a specific Gaussian distribution, called the binary discrete Gaussian distribution, with (relative) probability density 2^{-x^2} for $x \geq 0$. The average number of rejections is smaller than 1.47, and hence its performance is much better than that of Sample$\mathbb{Z}$ algorithm in [14].

In [25], C. Peikert suggested to use a cumulative distribution table (CDT) to sample $D_{\mathbb{Z},\sigma,c}$ more efficiently when $c \in \mathbb{R}$ is known in advanced. One tabulates the CDT of the desired distribution, and generates a random deviate in $[0, 1)$ at sampling time, then performs a binary search through the table to locate the output. This method is also called the inversion method and it can lead to the best sampling efficiency. In order to satisfy a negligible statistical distance, however, it requires a large pre-computation storage varied with the parameter of the distribution.

Knuth-Yao method is a technical framework, which can simulate any discrete distribution [6]. A sampling algorithm based on Knuth-Yao method also require pre-computation storage varied with the parameter of the distribution, though the storage size can be smaller than that of an algorithm based on the inversion method [9, 28]. The main advantage of Knuth-Yao method is that it consumes a smaller number of random bits, which may lead to better performance especially when random bits are generated by a hardware device.

In 2016, C. Karney proposed an algorithm for sampling exactly from a discrete distribution over the integers $\mathbb{Z}$ [15]. This algorithm also uses rejection sampling method and it is a discretization of his algorithm for sampling exactly from the normal distribution. Karney's algorithm is generic, i.e., it is fit for any discrete Gaussian distribution $D_{\mathbb{Z},\sigma,c}$ with arbitrary and varying (rational) parameters. Meanwhile, it uses no floating point arithmetic and does not need any pre-computation storage. The experimental results show that Karney's algorithm also has considerable sampling efficiency.

Very recently, D. Micciancio et al. extended and generalized the 'reduction and recombination' technique [23], which was developed by T.Pöppelmann et al. from Peikert's (discrete) Gaussian convolution lemma [26, 25]. The main idea is to reduce the general sampling problem to the recombination of a relatively small number of samples coming from a Gaussian distribution with a fixed and rather small value of $\sigma > 0$. Their sampling algorithms are fit for discrete Gaussian distributions with arbitrary and varying parameters, and have good performance according to their experimental results. A theoretical advantage of Karney's algorithm is that it exactly samples from the target distributions, while the algorithms given by D.Micciancio et al. are approximately close to the target distributions.

Compared to the rejection sampling, an algorithm based on the inversion method or the Knuth-Yao method may be faster and consumes a smaller number of random bits, but it has a larger pre-computation storage, especially for attaining a closer statistical distance to the target distribution or sampling from distributions with large σ or varying c. Whereas, the main advantage of the rejection sampling is that it may produce samples exactly from the target distribution with much less, or even no, pre-computation storage. For examples, Ducas's algorithm only needs a look-up table of size 4kb for sampling a centered discrete Gaussian distribution with $\sigma \approx 271$ [7], and Karney's algorithm does not need any pre-computation storage for an approximate distribution [15]. From this point of view, algorithms based on the rejection sampling may be more appropriate for use in resource-constrained devices. It is interesting to improve the sampling efficiency of the algorithms based on the rejection sampling.

In this paper, we discuss rejection sampling algorithms for a centered discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ over the integers $\mathbb{Z}$ with parameter $\sigma > 0$ and (omitted) center $c = 0$. We consider centered discrete Gaussian distributions because some lattice-based cryptosystems, such as [18, 16, 19, 3], only require sampling from centered discrete Gaussian distributions. Also, we note that the performance of existing rejection sampling algorithms may be further improved exclusively for a centered discrete Gaussian distribution.

1.2 Our Contribution

We propose three alternative rejection sampling algorithms for centered discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ over the integers $\mathbb{Z}$ with parameter σ in two specific forms. The first one does not need any pre-computation storage, while the other two require fixed look-up tables of very small size. All of them do not use floating-point arithmetic. We demonstrate the correctness and efficiency of our proposed algorithms both through theoretical analysis and practical software experimentation.

(1) For the parameter σ that is a positive integer, we present a rejection sampling algorithm without pre-computation storage (Algorithm 4), which consumes a smaller number of random bits and has better sampling efficiency compared to Karney's algorithm.

(2) For the parameter σ that is a real number in the form of $k \cdot \sqrt{1/(2 \cdot \ln 2)}$ with positive integer k, we give a rejection sampling algorithm (Algorithm 5) using a fixed look-up table of very small size (e.g. 128 bits), which consumes a smaller number of random bits and much better performance compared to our first proposed algorithm (Algorithm 4).

(3) We also provide a time-memory trade-off version of Algorithm 5 by using the Knuth-Yao method, which has better sampling efficiency at the cost of storing a bigger (but still very small) fixed look-up table.

The experimental results show that they all have better sampling efficiency. In our implementation environment (the g++ compiler, enabling -Os optimization option, on a laptop computer with Intel i7-6820hq and 8GB RAM), our most efficient algorithm for sampling $D_{\mathbb{Z},\sigma}$ with $\sigma = 254 \cdot \sqrt{1/(2 \cdot \ln 2)} \approx 215$ generates about 12.67×10^6 samples per second, i.e., about 0.0789μs per sample.

1.3 Techniques

The 'half exponential Bernoulli trial' algorithm given by Karney is one of basic techniques in our proposed algorithms [15]. It is adapted from Von Neummann's algorithm for sampling from the exponential distribution e^{-x} for real $x > 0$. It aims at generating a Bernoulli random value b without using floating-point exponential computation, where b is true with probability $1/\sqrt{e}$. In this paper, we improve on this technique by giving a algorithm (Algorithm 6) for generating a Bernoulli random value b without using floating-point exponential computation, where b is true with probability $e^{-(\ln 2)(z/k^2)} = 2^{-z/k^2}$ and k, z are two positive integers such that $z < k^2$. It requires a fixed look-up table that stores the binary expansion of $\ln 2$ with adequate precision (e.g. 128 bits).

In addition, we present an efficient implementation of the Knuth-Yao method for the binary discrete Gaussian distribution $D_{\mathbb{Z},\sigma_2}$ with $\sigma_2 = \sqrt{1/(2 \cdot \ln 2)}$. It requires a fixed look-up table of very small size (192 bits) instead of the whole probability matrix (even a trimmed one), which is usually involved in a generic implementation of the Knuth-Yao method.

2 Preliminaries

In this section, we recall the basic notion of rejection sampling and remark three typical rejection sampling algorithms for discrete Gaussian distributions over the integers $\mathbb{Z}$.

Trough the paper, we denote the set of real numbers by $\mathbb{R}$, the set of integers by $\mathbb{Z}$, and the set of non-positive integers by $\mathbb{Z}^+$ respectively. We extend any real function $f(\cdot)$ to a countable set A by defining $f(A) = \sum_{x \in A} f(x)$. The Gaussian function on $\mathbb{R}$ with parameter $\sigma > 0$ and center $c \in \mathbb{R}$ evaluated at $x \in \mathbb{R}$ can be defined by $\rho_{\sigma,c}(x) = \exp\left(-\frac{(x-c)^2}{2\sigma^2}\right)$. The subscripts σ and c are taken to be 1 and 0 respectively when omitted. For $c \in \mathbb{R}$ and real $\sigma > 0$, the discrete Gaussian distribution over the integers $\mathbb{Z}$ is defined by $D_{\mathbb{Z},\sigma,c} = \rho_{\sigma,c}(x)/\rho_{\sigma,c}(\mathbb{Z})$ for $x \in \mathbb{Z}$, where parameter σ can also be called the standard deviation of the discrete Gaussian distribution.

2.1 Basic Rejection Sampling

Rejection sampling is a basic technique used to generate observations from a distribution. It is a type of Monte Carlo method. The rejection sampling method generates sampling values from a target distribution X with arbitrary probability density function $f(x)$ by using a proposal distribution Y with probability density function $g(x)$. The idea is that one can generate a sample value from X by instead sampling from Y and accepting the sample from Y with probability $f(x)/Mg(x)$, repeating the draws from Y until a value is accepted, where M is a constant with $M \geq 1$ such that $f(x) \leq Mg(x)$ for all values of x (or for almost all values of x). This requires that the support of Y includes (almost includes) that of X. It is not hard to prove that if $f(x) \leq Mg(x)$ for all x then the rejection sampling procedure produces exactly the distribution of X, otherwise, if $f(x) \leq Mg(x)$ for almost all x then it produces a distribution within a negligible statistical distance of the distribution of X (one may refer to [18]).

The first Sample$\mathbb{Z}$ algorithm, which was given by Gentry et al. in [14], uses rejection sampling from the uniform distribution over $[c - \tau\sigma, c + \tau\sigma]$ by outputting a uniform integer x with probability $\rho(x) = \exp(-(x-c)^2/(2\sigma^2))$. Ducas and Nguyen suggested to use lazy floating-point arithmetic to compute the probability efficiently and speedup the whole sampling procedure [8]. However, this algorithm requires about $2\tau/\sqrt{2\pi}$ trails on average, where τ is determined so that the probability that $|x - c| \geq \tau \cdot \sigma$ is negligible. More precisely, according to the inequality given by W. Banaszczyk, we need an integer τ such that $2 \cdot \exp(-\tau^2/2)$ is negligible [14, 18]. We set $\tau = 12$, then $2 \cdot \exp(-\tau^2/2) \approx 1.0 \times 10^{-31}$, which is acceptably negligible value. Then $2\tau/\sqrt{2\pi} \approx 10$, which means that this algorithm requires about 10 trails on average, and so it is not very efficient even using the lazy floating-point arithmetic.

2.2 Binary Method

The Sample$\mathbb{Z}$ algorithm proposed by Ducas et al. in [7] is far more efficient than the algorithm given by Gentry et al. in [14], though it was designed exclusive for a centered discrete Gaussian distribution over the integers, namely $D_{\mathbb{Z},\sigma,c}$ with $\sigma = k\sigma_2$ and $c = 0$, where k is a positive integer and $\sigma_2 = \sqrt{1/(2 \cdot \ln 2)}$.

This algorithm uses rejection sampling from the binary discrete Gaussian distribution, denoted by $D_{\mathbb{Z}^+,\sigma_2}$, instead of the uniform distribution in Sample$\mathbb{Z}$ algorithm in [14]. Firstly, it samples an integer $x \in \mathbb{Z}^+$ from $D_{\mathbb{Z}^+,\sigma_2}$, then samples $y \in \mathbb{Z}$ uniformly in $\{0, 1, 2, \cdots, k\}$ and accepts $z = kx + y$ with probability $\exp(-(y^2 + 2kxy)/2k^2\sigma_2^2)$, finally, it outputs a signed integer, z or $-z$, with equal probabilities because of the symmetry of the target distribution (see Algorithm 11 and Algorithm 12 in [7]).

On one hand, Sampling from $D_{\mathbb{Z}^+,\sigma_2}$ can be very efficient using only unbiased random bits (see Algorithm 10 in [7]). In fact, this procedure is a combination of the rejection sampling and the inversion method. In our implementation environment, one can get at least about 60.06×10^6 samples per second from $D_{\mathbb{Z}^+,\sigma_2}$, and each sample costs only about 2.556 random bits on average.

On the other hand, the target and the proposed probability density function are $f(x) = \rho_\sigma(x)/\rho_\sigma(\mathbb{Z}^+)$ and $g(x) = \rho_{\sigma_2}(x)/\rho_{\sigma_2}(\mathbb{Z}^+)$ respectively. It was shown that $f(x) \leq Mg(x)$ with $M \leq 1.47$ for all $x \in \mathbb{Z}^+$. This implies that the rejection sampling procedure itself will not lead to any statistical discrepancy. Its statistical distance to the target distribution only depends on the precision of computing the exponential function $\exp(-(y^2+2kxy)/2\sigma^2)$ with $\sigma = k\sigma_2$, and its average number of rejections is not more than 1.47. A pre-computed look-up table is required for computing the exponential function $\exp(-(y^2+2kxy)/2\sigma^2)$. The size of the table varies with the parameter σ. For examples, the size is 2.3kb for $\sigma \approx 107$ and 4kb for $\sigma \approx 271$ according to the estimations given by Ducas et al. in [7], which is not very small but acceptable.

In addition, one may note that Lumbroso's algorithm [17], which is designed for drawing discrete uniform random variables within a given range, can be applied to speed up the procedure of sampling $y \in \mathbb{Z}$ uniformly in $\{0, 1, 2, \cdots, k\}$ (for not very large k). In our implementation environment, for instance, one can get about 4.386×10^6 samples per second from $D_{\mathbb{Z},\sigma}$ with $\sigma \approx 215$ by using the algorithms presented by Ducas et al. in [7] and Lumbroso's algorithm.

2.3 Karney's Algorithm

Karney's algorithm is described as Algorithm 1 in this paper. It samples an integer $k \in \mathbb{Z}^+$ with (relative) probability density $e^{-k^2/2}$ instead of sampling the binary discrete Gaussian distribution in [7]. Step 1 and 7 are two kernel subrotines in Algorithm 1. They can be realized without floating point arithmetic by repeatedly applying Algorithm 2 and Algorithm 3 respectively [15].

Algorithm 1 [15] Sampling $D_{\mathbb{Z},\sigma,c}$ for $\sigma, c \in \mathbb{Q}$

```
Input: rational number $\sigma$ and $c$
Output: an integer $z$ according to $D_{\mathbb{Z},\sigma,c}$
1: sample $k \in \mathbb{Z}^+$ with (relative) probability density $e^{-k^2/2}$
2: set $s \leftarrow \pm 1$ with equal probabilities
3: set $i_0 \leftarrow \lceil k\sigma + sc \rceil$ and set $x_0 \leftarrow (i_0 - (k\sigma + sc))/\sigma$
4: sample $j \in \mathbb{Z}$ uniformly in $\{0, 1, 2, \cdots, \lceil \sigma \rceil - 1\}$
5: set $x \leftarrow x_0 + j/\sigma$ and goto step 1 if $x \geq 1$
6: goto step 1 if $k = 0$ and $x = 0$ and $s < 0$
7: accept $x$ with probability $e^{-\frac{1}{2}x(2k+x)}$ otherwise goto step 1
8: return $s(i_0 + j)$
```

Algorithm 2 is adapted from Von Neummann's algorithm for sampling from the exponential distribution e^{-x} for real $x > 0$. More precisely, the probability that the length of the longest decreasing sequence is n is $x^n/n! - x^{n+1}/(n+1)!$, and the probability that n is even is exactly equal to

$$(1 - x) + \left(\frac{x^2}{2!} - \frac{x^3}{3!}\right) + \ldots + = \sum_{n=0}^{\infty} \left(\frac{x^n}{n!} - \frac{x^{n+1}}{(n+1)!}\right) = e^{-x}.$$

Algorithm 2 [15] Generating a Bernoulli random value b which is true with probability $1/\sqrt{e}$

```
Output: a Boolean value $b$ according to $\mathcal{B}_{1/\sqrt{e}}$
1: sample uniform deviates $u_1, u_2, \ldots$ with $u_i \in [0, 1)$ and determine the maximum value $n \geq 0$ such
   that $1/2 > u_1 > u_2 > \ldots > u_n$
2: return true if $n$ is even, or false if otherwise
```

Algorithm 3 [15] Generating a Bernoulli random value b which is true with probability $\exp(x\frac{2k+x}{2k+2})$ with integer $k > 0$ and real $x \in [0, 1)$

```
Output: a Boolean value $b$ according to $\exp(-x\frac{2k+x}{2k+2})$
1: set $y \leftarrow x$, $n \leftarrow 0$.
2: sample uniform deviates $z$ with $z \in [0, 1)$; go to step 6 unless $z < y$.
3: set $f \leftarrow C(2k + 2)$; if $f < 0$ go to step 6.
4: sample uniform deviates $r \in [0, 1]$ if $f = 0$, and go to step 6 unless $r < x$.
5: set $y \leftarrow z$, $n \leftarrow n + 1$; goto step 2.
6: return true if $n$ is even, or false if otherwise
```

Algorithm 2 provides a method of sampling an integer $k \in \mathbb{Z}^+$ with (relative) probability density $e^{-k^2/2}$. To end this, one can select an integer $k \geq 0$ with probability $\exp\big((-1/2)k\big) \cdot (1 - 1/\sqrt{e})$, then accept k as the sample with probability $\exp\big((-1/2)k(k-1)\big)$ by repeatedly applying Algorithm 2. Moreover, by replacing $1/2$ with p/q in Algorithm 2 we can get a Bernoulli random value b which is true with probability $e^{-p/q}$, where p, q are positive integers such that $p < q$.

The main idea behind Alg, 3 is to sample two sets of uniform deviates $u_1, u_2, \ldots$ and $v_1, v_2, \ldots$, and to determine the maximum value $n \geq 0$ such that $x > u_1 > u_2 > \ldots > u_n$ and $v_i < (2k + x)/(2k + 2)$. Then, n is even, which means it returns **true**, with the probability

$$1 - x\left(\frac{2k+x}{2k+2}\right) + \frac{x^2}{2!}\left(\frac{2k+x}{2k+2}\right)^2 - \frac{x^3}{3!}\left(\frac{2k+x}{2k+2}\right)^3 + \ldots = \exp\left(-x\frac{2k+x}{2k+2}\right).$$

It is clear that one can obtain a Bernoulli random value which is true with probability $\exp(-(1/2)x(2k + x))$ by applying Algorithm 3 $k + 1$ times at most for given k and x. Let $m = 2k + 2$. The function $C(m)$ is a random selector that outputs -1, 0 and 1 with probability $1/m$, $1/m$ and $1 - 2/m$ respectively. It is used for avoiding performing arithmetic on real x.

Finally, in Algorithm 1, step 5 ensures that x is in the allowed range for invoking Alg, 3 in step 7. If σ is an integer, it was pointed out that step 5 can be omitted since '$x \geq 1$' cannot happen in this case. Step 6 is designed to avoid double counting 0 when c is an integer.

Karney's algorithm (Algorithm 1) has at least two advantages over the algorithms given by Ducas et al. in [7]. It needs neither (high-precision) exponential computation nor pre-computation storage, and it is fit for any discrete Gaussian distribution $D_{\mathbb{Z},\sigma,c}$ with arbitrary and varying (rational) parameters (standard deviation σ and center c). Furthermore, it has a slightly better performance in our implementation environment. For instance, one can get about 4.751×10^6 samples per second from $D_{\mathbb{Z},\sigma,c}$ with $\sigma = 215$ and $c = 0$ by using Karney's C++ library 'RandomLib', in which the source code of Algorithm 1 is encapsulated as a .hpp file named 'DiscreteNormal.hpp'[1].

In this paper, the software implementation of our three proposed sampling algorithms is also based on the adaptions of 'DiscreteNormal.hpp' as well as the runtime environment provided by 'RandomLib'.

[1]'RandomLib' is available at http://randomlib.sourceforge.net/.

3 An Improved Implementation of Karney's Algortihm

In this section, we show that Karney's Algortihm (Algorithm 1) can be implemented more efficiently if one only needs samples from a centered discrete Gaussian distribution over the integers $D_{\mathbb{Z},\sigma}$ with an integer σ.

3.1 On Accepting x with Probability $e^{-\frac{1}{2}x(2k+x)}$

With notations in Algorithm 1, when σ is an integer and $c = 0$, we can see that j/σ will be assigned to x since $x_0 = 0$ in this case. Thus, in step 7, we need to generate a Bernoulli random value b which is true with probability

$$\exp\left(-\frac{1}{2}\left(\frac{j}{\sigma}\right)\left(2k+\left(\frac{j}{\sigma}\right)\right)\right) = \exp\left(-\frac{2jk\sigma+j^2}{2\sigma^2}\right) = \left(\frac{1}{\sqrt{e}}\right)^{\lfloor i/\sigma^2\rfloor}\exp(-\frac{r}{2}),$$

where $i = 2jk\sigma + j^2$ and $r = i/\sigma^2 - \lfloor i/\sigma^2\rfloor$, i.e., $r < 1$ is the fractional part of i/σ^2. This means that we only need to generate $\lfloor i/\sigma^2\rfloor$ Bernoulli deviates according to $\mathcal{B}_{1/\sqrt{e}}$ by repeatedly applying Algorithm 2 in step 7. If no false value is generated in this procedure, then the algorithm returns the result with probability $e^{-r/2}$, otherwise it restarts. Note that $r/2$ can also be written as p/q with $p = i \bmod (\sigma^2)$ and $q = 2\sigma^2$. As mentioned in Section 2.3, the Bernoulli random value which is true with probability $e^{-r/2} = e^{-p/q}$ can be obtained by applying the adapted version of Algorithm 2.

Therefore, for centered discrete Gaussian distribution over the integers with an integer standard deviation, we can avoid using Karney's implementation method (Algorithm 3) in step 7 of Algorithm 1 to get a Bernoulli random value which is true with probability $e^{-\frac{1}{2}x(2k+x)}$, where $x = j/\sigma$. Based on these observation, for centered discrete Gaussian distribution over the integers with an integer standard deviation σ, we give Algorithm 4 as an improved implementation of Karney's algorithm.

Algorithm 4 Sampling $D_{\mathbb{Z},\sigma}$ with integer σ

```
Input: an integer $\sigma$
Output: an integer $z$ according to $D_{\mathbb{Z},\sigma}$
1: sample $k \in \mathbb{Z}^+$ with (relative) probability density $e^{-k^2/2}$
2: set $s \leftarrow \pm 1$ with equal probabilities
3: sample $j \in \mathbb{Z}$ uniformly in $\{0, 1, 2, \cdots, \sigma - 1\}$ and set $j \leftarrow j + 1$ if $s = 1$
4: set $i \leftarrow 2jk\sigma + j^2$
5: sample $\lfloor i/2\sigma^2\rfloor$ Bernoulli deviates according to $\mathcal{B}_{1/\sqrt{e}}$ and goto step 1 unless all of them are 1.
6: set $r \leftarrow i/2\sigma^2 - \lfloor i/2\sigma^2\rfloor$
7: return $s(k\sigma + j)$ with probability $e^{-r/2}$ otherwise goto step 1
```

Moreover, in Algorithm 4 we set $j \leftarrow j + 1$ if $s = 1$ so that it returns 0 only in the case that $s = -1$ and $k = j = 0$, and we can avoid double counting 0 in advance.

3.2 The Expected Bits Consumption of Algorithm 4

We estimate the expected number of random bits used in Algorithm 4 and in Algorithm 1 respectively. We try to show that average bits consumption of Algorithm 4 is lower than that of Algorithm 1. Since the front part of Algorithm 1 and that of Algorithm 4 consume the same number of random bits, we only need to compare the average bits consumption of step 7 in Algorithm 1 with that of steps 5-7 in Algorithm 4.

Hereinafter, we consider the binary representation (and implementation) of a uniform deviate $u \in [0,1)$. From an implementation standpoint, the expected number of random bits we need for determining the relation between u and another newly-generated uniform deviate $u' \in [0,1)$ (determine $u < u'$ or $u > u'$) is at most

$$1 \cdot \frac{1}{2} + 3 \cdot \frac{1}{2^2} + 5 \cdot \frac{1}{2^3} + \ldots = \sum_{i=1}^{\infty} \frac{2i-1}{2^i} = 3$$

if only the most significant bit of u is determined and all the other digits of u needs to be randomly generated on-the-fly if necessary, while the expected number is at least

$$1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{2^2} + 3 \cdot \frac{1}{2^3} + \ldots = \sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

if each binary digit of u is given in advance. Thus, if one wants to sample uniform deviates $u_1, u_2, \ldots$ with $u_i \in [0,1)$ and determine the maximum value $n \geq 0$ such that $x > u_1 > u_1 > \ldots > u_n$ for a given $x \in [0,1)$, the expected number of random bits she/he needs is at most

$$\begin{aligned} & 2\left(\frac{x^0}{0!} - \frac{x^1}{1!}\right) + 5\left(\frac{x^1}{1!} - \frac{x^2}{2!}\right) + 8\left(\frac{x^2}{2!} - \frac{x^3}{3!}\right) + \ldots \\ & = \sum_{n=1}^{\infty} (3n-1)\left(\frac{x^{n-1}}{(n-1)!} - \frac{x^n}{n!}\right) = -1 + 3e^x. \end{aligned}$$

This equation also implies that the larger x the greater number of random bits are used for getting a Bernoulli random value b which is true with probability e^{-x}. In particular, when $x = 1/2$, i.e., the expected number of random bits we need for applying Algorithm 2 is at most

$$\sum_{n=1}^{\infty} (3n-2)\left(\frac{(1/2)^{n-1}}{(n-1)!} - \frac{(1/2)^n}{n!}\right) = -2 + 3\sqrt{e} \approx 2.946.$$

This is because it costs only one random bit to determine the relation between $1/2$ and a uniform deviate $u' \in [0,1)$.

With the notations of Algorithm 4, j/σ can be roughly viewed as a random deviate in $[0,1]$ since j is uniformly taken from $\{0,1,2,\cdots,\sigma-1\}$ or $\{1,2,3,\cdots,\sigma\}$. For a given $k \in \mathbb{Z}^+$ the expectation of $i/\sigma^2 = (2jk\sigma + j^2)/\sigma^2$ is equal to

$$\sum_{j=0}^{\sigma-1} \frac{1}{\sigma}\left(2k\left(\frac{j}{\sigma}\right) + \left(\frac{j}{\sigma}\right)^2\right) \leq \int_0^1 (2ku + u^2)du = \frac{1}{3} + k \leq \sum_{j=1}^{\sigma} \frac{1}{\sigma}\left(2k\left(\frac{j}{\sigma}\right) + \left(\frac{j}{\sigma}\right)^2\right)$$

This implies that Algorithm 4 generates k Bernoulli deviates on average according to $\mathcal{B}_{1/\sqrt{e}}$ by repeatedly calling Algorithm 2, then it returns the output with probability $e^{-r/2}$, where $r = 1/3$ on average.

For a given $k \geq 0$ we denote by $\mathbf{E}_k$ the average bits consumption of steps 5–7 in Algorithm 4. It is not hard to see that

$$\mathbf{E}_k \approx \big(k(-2+3e^{1/2}) - 1 + 3e^{1/6}\big)\big(e^{-1/2}\big)^k + \sum_{i=1}^{k} i\big(-2+3e^{1/2}\big)\big(e^{-1/2}\big)^{i-1}\big(1-e^{-1/2}\big)$$

for $k \geq 0$. Meanwhile, the value of k is determined with probability $\mathbf{P}_k = e^{(-k^2/2)} \Big/ \sum_{j=0}^{\infty} e^{-j^2/2}$. Therefore, for a random variable $k \geq 0$, the average bits consumption of steps 5–7 in Algorithm 4 can be roughly upper-bounded by $\sum_{k=0}^{\infty} \mathbf{P}_k \mathbf{E}_k \approx \sum_{k=0}^{3} \mathbf{P}_k \mathbf{E}_k \approx 3.482$.

3.3 The Expected Bits Consumption of Algorithm 3

In this subsection, with a similar idea, we discuss the average bits consumption of Algorithm 3, i.e., step 7 of Algorithm 1.

For a given k, there are three cases in which the algorithm goes to step 6 and returns the result before it goes to step 2. (1) $z > y$ with probability $(1-x)$; (2) $f = -1$ with probability $x(1/(2k+2))$; (3) $f = 0$ and $r > x$ with probability $x(1/(2k+2))(1-x)$. For simplicity, we let $m = 2k+2$. Generally, after restarting $n-1$ times ($n \geq 1$), there are three cases in which the algorithm goes to step 6 and returns the result before it goes to step 2 again.
(1) $z > y$ with probability

$$\mathbf{p}_{k,x,n}(z > y) = \left(\frac{x}{m} + \frac{2k}{m}\right)^{n-1} \left(\frac{x^{n-1}}{(n-1)!} - \frac{x^n}{n!}\right);$$

(2) $f = -1$ with probability

$$\mathbf{p}_{k,x,n}(f = -1) = \left(\frac{x}{m} + \frac{2k}{m}\right)^{n-1} \left(\frac{x^n}{n!}\right) \left(\frac{1}{m}\right);$$

(3) $f = 0$ and $r > x$ with probability

$$\mathbf{p}_{k,x,n}(f = 0; r > x) = \mathbf{p}_{k,x,n}(f = -1)(1-x).$$

Here, it can be verified that

$$\sum_{n=1}^{\infty} \big(\mathbf{p}_{k,x,n}(z > y) + \mathbf{p}_{k,x,n}(f = -1) + \mathbf{p}_{k,x,n}(f = 0; r > x)\big) = 1.$$

The random selector $C(m)$ with $k \geq 1$ uses $\lceil \log_2(m-1) \rceil$ random bits if returns $f = -1$ or $f = 0$, and uses $1 + \sum_{i=0}^{l-2} i \cdot (1/2^i)$ random bits on average if returns $f = 1$. Based on our previous discussions, it costs at least 2 random bits on average to determine the relation between z and y. Hence, for a given $k \geq 1$ the average bits consumption of Algorithm 3, denoted by $\mathbf{e}_k(x)$, satisfies

$$\begin{aligned}\mathbf{e}_k(x) > &\sum_{n=1}^{\infty} \big(2n + (n-1)e_l\big) \cdot \mathbf{p}_{k,x,n}(z > y) \\ &+ \sum_{n=1}^{\infty} \big(2n + (n-1)e_l + l\big) \cdot \mathbf{p}_{k,x,n}(f = -1) \\ &+ \sum_{n=1}^{\infty} \big(2n + (n-1)e_l + l + 2\big) \cdot \mathbf{p}_{k,x,n}(f = 0; r > x),\end{aligned}$$

where $e_l = 1 + \sum_{i=0}^{l-2} i \cdot (1/2^i)$ and $l = \lceil \log_2(m-1) \rceil$.

In particular, when $k = 0$ the random selector $C(2k+2) = C(2)$ returns -1 or 0 using only one random bit, which means that we can switch the orders of step 2 and step 3 in Algorithm 3 to save bits consumption. Then, after restarting $n-1$ times ($n \geq 1$), the three cases in which the algorithm goes to step 6 and returns the result before it goes to step 2 again are as follows:
(1) $f = -1$ with probability

$$\mathbf{p}_{0,x,n}(f = -1) = \left(\frac{1}{2}\right)^n \left(\frac{x^{n-1}}{(n-1)!}\right) x^{n-1};$$

(2) $z > y$ with probability

$$\mathbf{p}_{0,x,n}(z > y) = \left(\frac{1}{2}\right)^n x^{n-1}\left(\frac{x^{n-1}}{(n-1)!} - \frac{x^n}{n!}\right);$$

(3) $f = 0$ and $r > x$ with probability

$$\mathbf{p}_{0,x,n}(f = 0; r > x) = \left(\frac{1}{2}\right)^n x^{n-1}\left(\frac{x^n}{n!}\right)(1 - x).$$

Thus, when $k = 0$ the average bits consumption of Algorithm 3, denoted by $\mathbf{e}_0(x)$, is at least

$$\sum_{n=1}^{\infty}(5n-4)\cdot\mathbf{p}_{0,x,n}(f=-1) + \sum_{n=1}^{\infty}(5n-2)\cdot\mathbf{p}_{0,x,n}(z>y) + \sum_{n=1}^{\infty}(5n)\cdot\mathbf{p}_{0,x,n}(f=0;r>x).$$

Let's go back to Algorithm 1. The average number of step 7 invoking Algorithm 3, denoted by $\mathbf{t}_k(x)$, is equal to

$$\sum_{i=1}^{k} i\cdot\left(\exp\left(-x\frac{2k+x}{2k+2}\right)\right)^{i-1}\left(1-\exp\left(-x\frac{2k+x}{2k+2}\right)\right)$$
$$+(k+1)\left(\exp\left(-x\frac{2k+x}{2k+2}\right)\right)^{k},$$

for a given $k \geq 1$. Then, in Algorithm 1, step 7 costs about $\mathbf{e}_k(x)\cdot\mathbf{t}_k(x)$ bits at least on average for $k \geq 1$ and $\mathbf{e}_0(x)$ bits at least for $k = 0$.

Finally, since it is supposed that σ is an integer and $c = 0$, the random variable $x = j/\sigma$ can be roughly viewed as a random deviate in $[0, 1]$. Therefore, in Algorithm 1, the average bits consumption of step 7 with varied k and x is roughly lower-bounded by

$$\sum_{k=0}^{\infty}\mathbf{P}_k\left(\int_0^1 \mathbf{e}_k(x)\mathbf{t}_k(x)dx\right) \approx \sum_{k=0}^{3}\mathbf{P}_k\left(\int_0^1 \mathbf{e}_k(x)\mathbf{t}_k(x)dx\right),$$

where $\mathbf{P}_k$ is the probability density of the random variable k and $\mathbf{t}_0(x) = 1$.

After some routine (numerical) calculations, we have the value is approximately equal to 4.777. To verify our estimations, we also test the practical average number of random bits used in step 7 of Algorithm 1 and that of random bits used in steps 5–7 of Algorithm 4 respectively. They are about 5.075 and 3.20 random bits respectively. To a conclusion, the expected number of random bits used in Algorithm 4 is significantly smaller than that of random bits used in Algorithm 1,

4 Removing Pre-computed Tables in Binary Method

In Algorithm 4 as well as in Algorithm 1, we note that it accounts for a large part of the running time of the whole algorithm to sample $k \in \mathbb{Z}^+$ with (relative) probability density $e^{-k^2/2}$ (to execute step 1). In the algorithms given by Ducas et al. in [7], which are called the binary method in section 2.2 in this paper, on the contrary, it is very fast to sample $x \in \mathbb{Z}^+$ with the (relative) probability density $\exp(-x^2/(2\sigma_2^2)) = 2^{-x^2}$, where $\sigma_2 = \sqrt{1/(2\cdot\ln 2)}$. The main drawback of the binary method is that it needs a pre-computed table to generate a Bernoulli random value b which is true with probability $\exp(-(y^2 + 2kxy)/2\sigma^2)$. Although the table is not big (e.g. about 4kb for $\sigma = 271$), it has to be pre-computed before sampling for every different $\sigma = k\sigma_2$. Therefore, it is interesting to sample $D_{\mathbb{Z},k\sigma_2}$ without a pre-computed table.

In this section, we propose an alternative algorithm, which samples $D_{\mathbb{Z},k\sigma_2}$ for a given $k \geq 1$ and does not require a pre-computed table except a fixed look-up table of very small size. Compared to Algorithm 4, the experimental results show that our proposed algorithm in this section has a significant increase in the sampling efficiency. Furthermore, from the perspective of complexity, we will show that the average bits consumption of our proposed algorithm in this section is smaller than that of Algorithm 4.

4.1 The Proposed Algorithm

In this section, we use the notations of section 2.2. Since $\sigma_2 = \sqrt{1/(2 \cdot \ln 2)}$ and $\sigma = k\sigma_2$, it follows that $\exp(-(y^2 + 2kxy)/2\sigma^2)$ is equal to

$$\exp\left(-\ln 2 \cdot \left(\left\lfloor \frac{y^2+2kxy}{k^2} \right\rfloor + \left\{ \frac{y^2+2kxy}{k^2} \right\}\right)\right)$$
$$= \exp\left(-\ln 2 \cdot \left\lfloor \frac{y^2+2kxy}{k^2} \right\rfloor\right) \cdot \exp\left(-\ln 2 \cdot \left\{ \frac{y^2+2kxy}{k^2} \right\}\right)$$

where

$$\left\{ \frac{y^2+2kxy}{k^2} \right\} = \frac{y^2+2kxy}{k^2} - \left\lfloor \frac{y^2+2kxy}{k^2} \right\rfloor = \left((y^2+2kxy) \bmod k^2\right)/k^2$$

is the fractional part of $(y^2 + 2kxy)/k^2$. Then the Bernoulli value b can be obtained by generating $\lfloor y^2 + 2kxy/k^2 \rfloor$ Bernoulli deviates according to $\mathcal{B}_{1/2}$ and one Bernoulli random value which is true with probability

$$\exp(-(\ln 2)\{y^2 + 2kxy/k^2\}).$$

If no false value is generated in this procedure, then we have $b = 1$, otherwise $b = 0$. Based on this observation, we get Algorithm 5.

Algorithm 5 Sampling $D_{\mathbb{Z},k\sigma_2}$ for $k \in \mathbb{Z}^+$

```
Input: a positive integer $k$
Output: an integer $z$ according to $D_{\mathbb{Z},k\sigma_2}$
1: sample $x \in \mathbb{Z}$ according to $D_{\mathbb{Z}^+,\sigma_2}$
2: sample $y \in \mathbb{Z}$ uniformly in $\{0, 1, 2, \cdots, k\}$
3: set $s \leftarrow \pm 1$ with equal probabilities and set $j \leftarrow j + 1$ if $s = -1$
4: set $z \leftarrow kx + y$
5: sample $\lfloor (y^2 + 2kxy)/k^2 \rfloor$ Boolean values according to $\mathcal{B}_{1/2}$ and restart unless all of them are true
6: accept $z$ with probability $\exp(-(\ln 2)\{(y^2 + 2kxy)/k^2\})$ and goto step 1 if otherwise
7: return $s \cdot z$
```

It is clear that a Bernoulli deviate according to $\mathcal{B}_{1/2}$ is equivalent to a uniform random bit. Thus, the only remaining problem is to generate a Bernoulli random value which is true with probability $\exp(-(\ln 2)\{(y^2 + 2kxy)/k^2\})$. We address this problem via Algorithm 6.

One can see that Algorithm 6 is adapted from Algorithm 2. More precisely, we can take any bit in the binary expansion of p/q since p/q is a rational number, where p, q are positive integers such that $p < q$. This means that we can handle with the probability $e^{-p/q}$ by applying the technique in Algorithm 2. For the probability $\exp(-(\ln 2)\{(y^2 + 2kxy)/k^2\})$, however, we are not able to determine the bits in the binary expansion of $(\ln 2)\{(y^2 + 2kxy)/k^2\}$ for different k, x and y without (high-precision) floating-point operations. Thus, we can say that Algorithm 6 provides a technique for determining the relation between $(\ln 2)(z/k^2)$ and a uniform deviate $u_1 \in [0, 1)$. It does not need floating-point operations during runtime, if the binary expansion of $\ln 2$, that is long enough (e.g. 128 bits), has been stored as a fixed look-up

Algorithm 6 Generate a Bernoulli random value b which is true with probability $\exp(-(\ln 2)(z/k^2))$, where k, z are positive integers such that $z < k^2$.

```
Input: positve integers $k, z, t$ such that $z < k^2$ and $t \geq 1$
Output: a Bernoulli random value $b$ which is true with probability $\exp(-(\ln 2)(z/k^2))$
1: set $u_1 \leftarrow 0$, $i \leftarrow 0$ and $p \leftarrow (\ln 2) \cdot 2^t$
2: sample a uniform integer $v \in [0, 2^t)$
3: set $i \leftarrow i + 1$, $u_1 \leftarrow u_1 + v/2^{it}$ and $q \leftarrow vk^2$
4: return true if $\lfloor p \rfloor > \lfloor q/z \rfloor$ or goto step 8 if $\lfloor p \rfloor < \lfloor q/z \rfloor$
5: set $p \leftarrow (p - \lfloor p \rfloor) \cdot 2^t$ and $q \leftarrow q (\mathrm{mod}\, z)$
6: update the uniform integer $v \in [0, 2^t)$
7: set $q \leftarrow (q \cdot 2^t + vk^2)$ and goto step 3
8: sample uniform deviates $u_2, u_3 \ldots$ with $u_i \in [0, 1]$ and determine the maximum value $n \geq 1$ such that
   $u_1 > u_2 > u_3 > \ldots > u_n$
9: return true if $n$ is even, or false if otherwise
```

table. In addition, the expected number of random bits Algorithm 6 consumes is at most $-1 + 3e^x$ with $x = (\ln 2)(z/k^2)$ according to our estimation in section 3.2.

4.2 The Correctness of Algorithm 6

In step 1, p is assigned to be the t most significant bits of the binary expansion of $\ln 2$. In step 4, if $\lfloor p \rfloor > \lfloor q/z \rfloor = \lfloor (vk^2)/z \rfloor$, then we have $2^t(\ln 2) > (vk^2)/z$, which follows that $(\ln 2)(z/k^2) > v/2^t$. It is equivalent to obtaining a uniform deviate $u_1 = (v/2^t)$ that is strictly less than the value of $(\ln 2)(z/k^2)$. On the contrary, if $\lfloor p \rfloor < \lfloor q/z \rfloor$, then it is equivalent to obtaining a uniform deviate u_1 that is strictly more than the value of $(\ln 2)(z/k^2)$. Otherwise, if $\lfloor p \rfloor = \lfloor q/z \rfloor$, i.e., $\lfloor 2^t(\ln 2) \rfloor = \lfloor (vk^2)/z \rfloor$. This means that we get a uniform deviate u_1, but in this moment we cannot determine whether it is less than the value of $(\ln 2)(z/k^2)$ or not.

In steps 5–7, the algorithm is designed to further compare the uniform deviate with the value of $(\ln 2)(z/k^2)$. The correctness is based on the following observation. Let $v_1, v_2, \ldots, v_n$ be n integers such that $v_i \in [0, 2^t)$ for each $i \geq 1$. For the simplicity, we define $G_t(i)$ and $H_t(k, z; v_1, v_2, \ldots, v_i)$ as follows:

$$G_t(i) = 2^t\{G_t(i-1)\}$$

and

$$H_t(k, z; v_1, \ldots, v_i) = 2^t\{H_t(k, z; v_1, \ldots, v_{i-1})\} + \frac{v_i k^2}{z},$$

where $i \geq 2$. For $i = 1$, let $G_t(1) = 2^t(\ln 2)$ and $H_t(k, z; v_1) = v_1k^2/z$. More explicitly, we have

$$G_t(i) = \underbrace{2^t\{\cdots 2^t\{2^t}_{i}(\ln 2)\}\cdots\}$$

and

$$H_t(k, z; v_1, v_2, \ldots, v_i) = \underbrace{2^t\{\cdots 2^t\{2^t}_{i-1}\{\frac{v_1k^2}{z}\} + \frac{v_2k^2}{z}\}\cdots\} + \frac{v_ik^2}{z}.$$

Suppose that $\lfloor G_t(i) \rfloor = \lfloor H_t(k, z; v_1, v_2, \ldots, v_i) \rfloor$ for each integer $i = 1, 2, \ldots, n-1$, but $\lfloor G_t(n) \rfloor > \lfloor H_t(k, z; v_1, v_2, \ldots, v_n) \rfloor$. In this case, we have

$$G_t(n) > H_t(k, z; v_1, v_2, \ldots, v_n),$$

which implies that $\{G_t(n-1)\} = G_t(n)/2^t$ and

$$\frac{G_t(n)}{2^t} > \frac{H_t(k,z;v_1,v_2,\ldots,v_n)}{2^t} = \{H_t(k,z;v_1,v_2,\ldots,v_{n-1})\} + \frac{v_n k^2}{2^t z}.$$

They follows that

$$\begin{aligned}&\{G_t(n-1)\} + \lfloor G_t(n-1)\rfloor \\ &\quad > \{H_t(k,z;v_1,v_2,\ldots,v_n)\} + \lfloor H_t(k,z;v_1,v_2,\ldots,v_{n-1})\rfloor + \frac{v_n k^2}{2^t z},\end{aligned}$$

as $\lfloor G_t(n-1)\rfloor = \lfloor H_t(k,z;v_1,v_2,\ldots,v_{n-1})\rfloor$. Thus, we have

$$G_t(n-1) > H_t(k,z;v_1,v_2,\ldots,v_{n-1}) + \frac{v_n k^2}{2^t z}.$$

If $n-1>1$, with similar arguments, applying $\lfloor G_t(i)\rfloor = \lfloor H_t(k,z;v_1,v_2,\ldots,v_i)\rfloor$ with $i=n-2$, we get

$$G_t(n-2) > H_t(k,z;v_1,v_2,\ldots,v_{n-2}) + \frac{v_{n-1}k^2}{2^t z} + \frac{v_n k^2}{2^{2t} z}.$$

We continue working backwards until finally we obtain

$$G_t(1) > H_t(k,z;v_1) + \frac{v_2 k^2}{2^t z} + \frac{v_3 k^2}{2^{2t} z} + \ldots + \frac{v_n k^2}{2^{(n-1)t} z},$$

which means that

$$2^t(\ln 2) > \frac{v_1 k^2}{z} + \frac{v_2 k^2}{2^t z} + \frac{v_3 k^2}{2^{2t} z} + \ldots + \frac{v_n k^2}{2^{(n-1)t} z}.$$

It follows that

$$\frac{(\ln 2)z}{k^2} > \frac{v_1}{2^t} + \frac{v_2}{2^{2t}} + \frac{v_3}{2^{3t}} + \ldots + \frac{v_n}{2^{nt}}.$$

This is equivalent to obtaining a uniform deviate

$$u_1 = \frac{v_1}{2^t} + \frac{v_2}{2^{2t}} + \frac{v_3}{2^{3t}} + \ldots + \frac{v_n}{2^{nt}} \in [0,1],$$

which is strictly less than the value of $(\ln 2)(z/k^2)$.

Furthermore, if

$$\lfloor G_t(i)\rfloor = \lfloor H_t(k,z;v_1,v_2,\ldots,v_i)\rfloor$$

for each integer $i=1,2,\ldots,n-1$, but

$$\lfloor G_t(n)\rfloor < \lfloor H_t(k,z;v_1,v_2,\ldots,v_n)\rfloor.$$

Then, we essentially get a uniform deviate u_1 that is strictly more than the value of $(\ln 2)(z/k^2)$. So the algorithm should return **true** in this case.

Note that $G_t(i)$ with $1 \le i \le n$ are corresponding to the binary expansion of $\ln 2$ of length nt bits, i.e.,

$$\ln 2 = G_t(1)/2^t + G_t(2)/2^{2t} + \ldots + G_t(n)/2^{nt} + \ldots.$$

When implementing the algorithm, we do not need to compute the exact value of p, since it is corresponding to $G_t(i)$, and can be always taken from the binary expansion of $\ln 2$.

In addition, $H_t(k, z; v_1, v_2, \ldots, v_i)$ can be computed iteratively. More precisely, for $i \geq 2$ we always have

$$\{H_t(k, z; v_1, \ldots, v_i)\} = \frac{(q2^t + v_i k^2) \bmod z}{z},$$

where q is an integer such that $0 \leq q < z$ and $\{H_t(k, z; v_1, \ldots, v_{i-1})\} = q/z$.

Consequently, in the algorithm, we iteratively compute $H_t(k, z; v_1, v_2, \ldots, v_i)$, and compare it with $G_t(i)$, until we have some integer $n \geq 2$ such that

$$\lfloor G_t(i) \rfloor = \lfloor H_t(k, z; v_1, v_2, \ldots, v_i) \rfloor$$

for each integer $i = 1, 2, \ldots, n-1$, but

$$\lfloor G_t(n) \rfloor > \lfloor H_t(k, z; v_1, v_2, \ldots, v_n) \rfloor \quad \text{or} \quad \lfloor G_t(n) \rfloor < \lfloor H_t(k, z; v_1, v_2, \ldots, v_n) \rfloor.$$

The rest of two steps of the algorithm follows from Algorithm 2.

4.3 The Expected Bits Consumption of Algorithm 5

In this subsection, we compare the expected bits consumption of Algorithm 5 with that of Algorithm 4. To this end, we discuss the expected number of random bits which are used for step 1 of Algorithm 4, i.e., sampling $k \in \mathbb{Z}^+$ with (relative) probability density $e^{-k^2/2}$.

We assume that $k \geq 0$ is an integer which is generated with probability $\exp\big(-(1/2)k\big)\big(1 - 1/\sqrt{e}\big)$. The probability of accepting k as the sample is $\exp\big(-(1/2)k(k-1)\big)$. In particular, if $k = 0$ or $k = 1$, then k is output as the sample directly. Let $n = k(k-1)$. By abuse of notation, let $\mathbf{t}_k$ be the expected number of applying Algorithm 2 that we need for accepting or rejecting a given $k \geq 2$. We have

$$\mathbf{t}_k = n \left(\frac{1}{\sqrt{e}}\right)^{n-1} + \sum_{i=1}^{n-1} i \left(\frac{1}{\sqrt{e}}\right)^{i-1} \left(1 - \frac{1}{\sqrt{e}}\right).$$

Based on our previous discussions in section 3.2, it costs at least 2 random bits on average to determine the relation between two random deviates in $[0, 1)$. Then the expected number of random bits that we need for applying Algorithm 2 one time is at least

$$\sum_{n=1}^{\infty} (2n-1) \left(\frac{(1/2)^{n-1}}{(n-1)!} - \frac{(1/2)^n}{n!}\right) = -1 + 2\sqrt{e} \approx 2.2974.$$

Therefore, the expected number of random bits that we need for accepting or rejecting a given $k \geq 2$ as sample in Algorithm 4 is at least $\mathbf{t}_k(-1+2\sqrt{e})$. For instance, we have $\mathbf{t}_2(-1+2\sqrt{e}) = (1 + 1/\sqrt{e})\,(-1 + 2\sqrt{e})$ when $k = 2$ and $\mathbf{t}_k(-1+2\sqrt{e}) \approx (\sqrt{e} - 2e)\,/\,(1 - \sqrt{e})$ when $k \geq 3$.

Finally, since the random variate k is selected as $k = 0, 1, \ldots, i, \ldots$ with probability $1 - 1/\sqrt{e}$, $(1/\sqrt{e})(1 - 1/\sqrt{e}), \ldots, (1/\sqrt{e})^i(1 - 1/\sqrt{e}), \ldots$ respectively, the number of random bits which are used for sampling $k \in \mathbb{Z}^+$ with (relative) probability density $e^{-k^2/2}$ is roughly lower-bounded by

$$\begin{aligned} &(-1+2\sqrt{e})\left(1 - \frac{1}{\sqrt{e}}\right) + 2(-1+2\sqrt{e})\frac{1}{\sqrt{e}}\left(1 - \frac{1}{\sqrt{e}}\right) \\ &+ \sum_{k=2}^{\infty} \big((k+1)(-1+2\sqrt{e}) + \mathbf{t}_k(-1+2\sqrt{e})\big)\, e^{-k/2} \left(1 - \frac{1}{\sqrt{e}}\right) \approx 7.676. \end{aligned}$$

The practical average number of random bits used for the sampling procedure we obtain from our experimental measurement is about 9.475. The estimated lower-bounded is significantly smaller than the practical value possibly because the expected number of random bits we need for applying Algorithm 2 one time is usually more than $-1+2\sqrt{e}$. As mentioned in section 2.2, it costs only about 2.556 random bits on average to get a sample from the binary discrete Gaussian distribution $D_{\mathbb{Z}^+,\sigma_2}$. Therefore, from the perspective of complexity, we say that Algorithm 5 is far more efficient than Algorithm 4.

5 A Faster Rejection Algortihm Using Knuth-Yao Method

As mentioned in Section 1, a sampling algorithm based on the Knuth-Yao method consumes a smaller number of random bits and thus is potentially faster than the algorithms based on other sampling methods. In this section, we describe an algorithm based on Knuth-Yao method for sampling the binary discrete Gaussian distribution $D_{\mathbb{Z},\sigma_2}$. It requires a fixed look-up table of size only 128 bits (or 192 bits). Then, applying this algorithm to Algorithm 5 we get a further increase in the sampling efficiency of Algorithm 5 at the cost of storing a fixed look-up table of size only 320 bits, which consists of the look-up table for sampling $D_{\mathbb{Z},\sigma_2}$ and the binary expansion of $\ln 2$.

5.1 Knuth-Yao Method for Binary Discrete Gaussian Distribution

The Knuth-Yao sampling method performs a random walk along with a binary tree called the discrete distribution generating (DDG) tree [6, 9]. A DDG tree can be determined by the probabilities of the sample points of the discrete random variate. A DDG tree consists of two types of nodes: intermediate nodes and terminal nodes. A terminal node has no child node and is labeled with sample point j. An intermediate node has two child nodes in the next level of the DDG tree. Let X be a random variate to generate with probability $p_0, p_1, \ldots, p_j, \ldots$. A DDG tree satisfies $\sum_{i\geq 0}(t_j^i/2^i) = p_j$, where t_j^i is the number of the terminal nodes labeled j on the ith level. Another way to formulate the equation given above for the DDG tree is that there is a terminal node labeled j on level i if and only if the ith binary digit of p_j is one.

During a sampling operation a random walk is performed starting from the root of the DDG tree. An edge is chosen at each level uniformly at random, according to a uniformly random bit. The sampling operation terminates when the random walk hits a terminal node, in which case it outputs the label of the node j.

In practice, a DDG tree does not need to be stored before a sampling operation. It can be constructed on-the-fly from a table called *probability matrix*. Consider the binary expansions of the probabilities of the sample points. They can be written in the form of binary matrix, which is called the probability matrix and denoted by $\mathbf{M}$. For $i, j \geq 0$, let $p_j = p_{j0} + p_{j1}2^{-1} + p_{j2}2^{-2} + \ldots + p_{ji}2^{-i} + \ldots$ be the binary expansion of the probability of the jth sample point, where $p_{ji} \in \{0,1\}$. In the probability matrix $\mathbf{M}$, the jth row $\mathbf{M}_j$ is equal to $(p_{j0}, p_{j1}, \ldots, p_{ji}, \ldots)$, and exactly corresponds to the binary expansion of the probability of the jth sample point. Based on the analysis of the relative distance of the internal nodes in the DDG tree, the Knuth-Yao sampling method can be described as Algorithm 7, which introduced by S. S. ROY et al. in [28].

For a discrete distribution with infinite support, the number of sample points as well as the binary expansions of their probabilities is potentially infinite, which results in a probability matrix of infinite size. In practice, one has to truncate the binary expansions of the probabilities but ensure adequate precision at the same time. For instance, the probability matrix $\mathbf{M}$ with $\lambda+1$ rows in Algorithm 7 is a truncated one such that the probability density of sample point $\lambda+1$ is a negligible value. The number of columns of matrix $\mathbf{M}$ is also finite such that the number of the algorithm going back to step 2 is never more than the number of columns.

Now we deal with binary discrete Gaussian distribution $D_{\mathbb{Z},\sigma_2}$ with $\sigma_2 = \sqrt{1/(2\cdot\ln 2)}$. Its probability density function is $2^{-x^2}/\mathbf{w}$, where $\mathbf{w} = \sum_{i=0}^{\infty} 2^{-i^2} > 1$ is a constant. Note that $2^{-x^2}/\mathbf{w}$ can be viewed as

Algorithm 7 [28] Knuth-Yao sampling method for a discrete distribution

```
Input: probability matrix $\mathbf{M}$ with $\lambda + 1$ rows
Output: sample value $x$
1: set $j \leftarrow 0$ and $d \leftarrow 0$
2: sample a uniformly random bit $r$
3: set $d \leftarrow 2d + 1 - r$
4: for $i \leftarrow \lambda$ down to 0 do
5:     set $d \leftarrow d - \mathbf{M}[i][j]$
6:     return $i$ if $d = -1$, or continue if otherwise
7: end for
8: set $j \leftarrow j + 1$ and goto step 2
```

a 'BitShiftRight' operation on $1/\mathbf{w}$, i.e., shifting the binary bits in the real $1/\mathbf{w}$ to the right by x^2 places and padding with zeros on the left. In other words, if $1/\mathbf{w} = w_1 2^{-1} + w_2 2^{-2} + \ldots + w_i 2^{-i} + \ldots$, then

$$2^{-x^2}/\mathbf{w} = w_1 2^{-1-x^2} + w_2 2^{-2-x^2} + \ldots + w_i 2^{-i-x^2} + \ldots.$$

This means that the probability matrix for $D_{\mathbb{Z},\sigma_2}$ can be constructed and invoked on-the-fly from the binary expansion of the real $1/\mathbf{w}$, which can be pre-computed offline. More specifically, let $\mathbf{m}$ be the binary expansion of the real $1/\mathbf{w}$. The entry of the probability matrix for $D_{\mathbb{Z},\sigma_2}$ on the i-th row and jth column is exactly equal to the $(j-i^2)$th bit of $\mathbf{m}$, denoted by $\mathbf{m}[j-i^2]$, and equal to 0 if $j - i^2 < 0$. Thus, from Algorithm 7 we have the following sampling algorithm for binary discrete Gaussian distribution $D_{\mathbb{Z},\sigma_2}$.

Algorithm 8 Sampling $D_{\mathbb{Z},\sigma_2}$ with $\sigma_2 = \sqrt{1/(2 \cdot \ln 2)}$

```
Input: binary vector $\mathbf{m}$ and positive integer $\lambda$ (e.g. $\lambda = 11$) such that $2^{-(\lambda+1)^2}/\mathbf{w} < 10^{-31}$ is negligible.
Output: an integer $i$ from $D_{\mathbb{Z},\sigma_2}$
1: set $j \leftarrow 0$ and $d \leftarrow 0$
2: sample a uniformly random bit $r$
3: set $d \leftarrow 2d + 1 - r$
4: for $i \leftarrow \lambda$ down to 0 do
5:     continue unless $j - i^2 \geq 0$
6:     set $d \leftarrow d - \mathbf{m}[j - i^2]$
7:     return $i$ if $d = -1$, or continue if otherwise
8: end for
9: set $j \leftarrow j + 1$ and goto step 2
```

Our experimental results shows that the average number of the algorithm going back to step 2, namely the average number of random bits the algorithm consumes for generating one sample, is about 2.031. This means that a binary expansion of the real $1/\mathbf{w}$ of size 128 bits is (more than) sufficient for Algorithm 8.

5.2 Using Look-up Tables

The expected number of the random bits Algorithm 8 used may not decide the final efficiency of the sampling operation, though it is the primary performance characteristic of a sampler analyzed in the random bit model. Hence, in this subsection, we try to further optimize the performance of Algorithm 8 for practical use.

Our overall idea fully follows from [28], i.e., to avoid the costly bit-scanning operation especially for the software implementation firstly, and then to use a small pre-computed table that directly maps the initial random bits into a sample value with large probability or into an intermediate node in the DGG tree (an intermediate state in Algorithm 8) with small probability.

When three bits $\{b_0, b_1, b_2\}$ are given as the bits pool of Algorithm 8, i.e., each random bit r in Algorithm 8 is taken in turn from $\{b_0, b_1, b_2\}$, we observe that it returns a sample value $i \in \{0, 1\}$ if $b_0 + 2b_1 + 2^2b_2 \neq 0$ and it falls into an intermediate state if otherwise. Further, when eight bits $\{0, 0, 0, b_3, b_4, \ldots, b_7\}$ are given as the bits pool of Algorithm 8, it returns a sample value $i \in \{0, 1, 2\}$ if $b_3 + 2b_4 + \ldots + 2^4b_7 \neq 0, 16$ and it falls into an intermediate state if otherwise. Based on these two facts, we use two look-up tables, *TabChar* of size 8 bits and *TabLong* of size 64 bits. The look-up table *TabChar* records 7 possible sample values corresponding to the first three bits b_0, b_1, b_2, while *TabLong* records 30 possible sample values corresponding to the following five bits $b_3, b_4, \ldots, b_7$. In other words, if the first three bits b_0, b_1, b_2 are not all zero, then we look-up *TabChar* to get a sample value, otherwise we may look-up *TabLong* according to the following five bits $b_3, b_4, \ldots, b_7$. Finally, if b_0, b_1, b_2 are all zero and $b_3 + 2b_4 + \ldots + 2^4b_7 = 0, 16$ respectively, then we set $j = 8$ and set $d = 1, 0$ respectively, and invoke Algorithm 8 starting from step 2.

The above procedure can be described as the following algorithm. For simplicity, let $TabChar[k]$ with $0 \leq k \leq 7$ be the kth bit in *TabChar* and $TabLong[k]$ with $0 \leq k \leq 31$ be the integer whose binary expansion are exactly the $2k$-th bit and the $(2k+1)$-th bit in *TabLong*.

Algorithm 9 Sampling $D_{\mathbb{Z},\sigma_2}$ with $\sigma_2 = \sqrt{1/(2 \cdot \ln 2)}$

```
Output: an integer $i$ from $D_{\mathbb{Z},\sigma_2}$
 1: sample a uniformly random integer $k \in [0, 3]$
 2: if $k = 0$ then
 3:     sample a uniformly random integer $k \in [0, 31]$
 4:     return $TabLong[k]$ unless $k = 0$ or $k = 16$
 5:     set $j \leftarrow 8$
 6:     set $d \leftarrow 1$ if $k = 0$ or $d \leftarrow 0$ if $k = 16$
 7:     invoke Algorithm 8 and start from step 2
 8: else
 9:     return $TabChar[k]$
10: end if
```

5.3 Experimental Results

We implement Algorithm 4, Algorithm 5 and Algorithm 9 respectively by adapting the 'DiscreteNormal.hpp' file in the 'RandomLib' library as mentioned in the end of section 3, and by using the runtime environment provided by 'RandomLib'.

Figure 1 shows the performance of our three sampling algorithms compared to Karney's algorithm (Algorithm 1). For centered discrete Gaussian distributions over the integers with parameter σ from 50 to 1000 with step size 50, [2] one can get about 5.9×10^6 samples per second by using Algorithm 4 and about 10.1×10^6 samples per second by using Algorithm 5. The performance gains from using Knuth-Yao method is also substantial. One can get about 12.7×10^6 samples per second by using Algorithm 5 in combination with Algorithm 9. Also, we test our proposed algorithms for σ from 1000 to 20000 with step size 1000. The experimental results show that our proposed algorithms have no significant performance change, and thus they allow for efficient constant time (software) implementations and provide resilience against timing side-channel attacks for not very large σ.

[2]The parameter σ is approximately (not exactly) from 50 to 1000 for Algorithm 5.

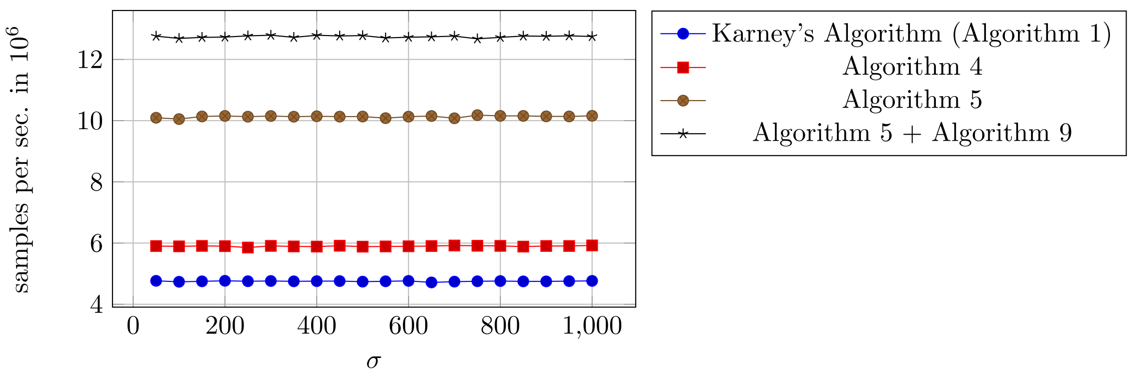

Figure 1: Performance of our sampling algorithms compared to Karney's algorithm

6 Limitations of Our Proposed Algorithms

A disadvantage of Algorithm 5 is that it only sample from a discrete Gaussian distribution with σ that is an integer multiple of $\sigma_2 = \sqrt{1/(2 \cdot \ln 2)}$. It seems more natural to use an integer σ as the standard deviation of a centered discrete Gaussian distribution, though there is no evidence to show that an integer σ must be better than a σ being an integer multiple of σ_2.

Another drawback of Algorithm 5 as well as Algorithm 4 is that a very large σ will cause a substantial decrease in performance. This is mainly because we need to compute $(y^2 + 2kxy)/k^2$ for each trial in Algorithm 5. When σ is very large, for example $\sigma \geq 2^{15}$, the value of $(y^2 + 2kxy)/k^2$ may exceed the range of the `int` type, and we have to use `long int` type to compute the value. Experimental results show that its performance decreases to about 8.62×10^6 samples per second for σ from 2^{15} to 2^{22}. While, the performance of Karney's algorithm remained basically stable at about 4.7×10^6 samples per second until $\sigma \geq 2^{23}$.

References

[1] Shweta Agrawal, Dan Boneh, and Xavier Boyen. Efficient lattice (H)IBE in the standard model. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 553–572. Springer, 2010.

[2] Shweta Agrawal, Dan Boneh, and Xavier Boyen. Lattice basis delegation in fixed dimension and shorter-ciphertext hierarchical IBE. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 98–115. Springer, 2010.

[3] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In Josh Benaloh, editor, *Topics in Cryptology - CT-RSA 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, volume 8366 of *Lecture Notes in Computer Science*, pages 28–47. Springer, 2014.

[4] Xavier Boyen. Attribute-based functional encryption on lattices. In Amit Sahai, editor, *Theory of Cryptography - TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, volume 7785 of *Lecture Notes in Computer Science*, pages 122–142. Springer, 2013.

[5] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[6] Luc Devroye. Discrete random variates. *Non-Uniform Random Variate Generation*, January 1986.

[7] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2013.

[8] Léo Ducas and Phong Q. Nguyen. Faster gaussian lattice sampling using lazy floating-point arithmetic. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *LNCS*, pages 415–432. Springer, 2012.

[9] Nagarjun C. Dwarakanath and Steven D. Galbraith. Sampling from discrete gaussians for lattice-based cryptography on a constrained device. *Applicable Algebra in Engineering, Communication and Computing*, 25(3):159, June 2014.

[10] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2013.

[11] Nicholas Genise and Daniele Micciancio. Faster gaussian sampling for trapdoor lattices with arbitrary modulus. *IACR Cryptology ePrint Archive*, 2017:308, 2017.

[12] Craig Gentry, Sergey Gorbunov, and Shai Halevi. Graph-induced multilinear maps from lattices. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *Theory of Cryptography - TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, volume 9015 of *Lecture Notes in Computer Science*, pages 498–527. Springer, 2015.

[13] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011.

[14] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Cynthia Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, STOC 2008, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 197–206. ACM, 2008.

[15] Charles F. F. Karney. Sampling exactly from the normal distribution. *ACM Trans. Math. Softw.*, 42(1):3:1–3:14, 2016.

[16] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In Aggelos Kiayias, editor, *Topics in Cryptology - CT-RSA 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, 2011.

[17] Jérémie Lumbroso. Optimal discrete uniform generation from coin flips, and applications. *CoRR*, abs/1304.1916, 2013.

[18] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755. Springer, 2012.

[19] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, 2013.

[20] Vadim Lyubashevsky and Thomas Prest. Quadratic time, linear space algorithms for gram-schmidt orthogonalization and gaussian sampling in structured lattices. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 789–815. Springer, 2015.

[21] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 700–718. Springer, 2012.

[22] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. *SIAM J. Comput.*, 37(1):267–302, 2007.

[23] Daniele Micciancio and Michael Walter. Gaussian sampling over the integers: Efficient, generic, constant-time. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017, CA, USA, August 20-24, 2017, Proceedings, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 455–485. Springer, 2017.

[24] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 333–342. ACM, 2009.

[25] Chris Peikert. An efficient and parallel gaussian sampler for lattices. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 80–97. Springer, 2010.

[26] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 353–370. Springer, 2014.

[27] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):34:1–34:40, 2009.

[28] Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. High precision discrete gaussian sampling on fpgas. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 383–401. Springer, 2013.