A Post-Quantum Secure Discrete Gaussian Noise Sampler

Rashmi Agrawal, Lake Bu†, Michel A. Kinsy
Adaptive and Secure Computing Systems (ASCS) Laboratory, ECE Department, Boston University
{rashmi23, mkinsy}@bu.edu
†The Charles Stark Draper Laboratory, Inc., Cambridge, MA
†{lbu}@draper.com

***Abstract*—While the notion of achieving "quantum supremacy" may be debatable, rapid developments in the field of quantum computing are heading towards more realistic quantum computers. As practical quantum computers start becoming more feasible, the requirement to have quantum secure cryptosystems becomes more compelling. Due to its many advantages, lattice-based cryptography has become one of the key candidates for designing secure systems for the post-quantum era. The security of lattice-based cryptography is governed by the small error samples generated from a Gaussian distribution. Hence, the Gaussian distribution lies at the core of these cryptosystems. In this paper, we present the hardware design implementation of three different sampling algorithms including rejection, Box-Muller, and the Ziggurat method for the Gaussian Sampler. Our goal is to provide concrete recommendations for future use and adoption in various cryptosystems based on sampling efficiency, hardware cost and throughput. The key feature of our design implementation is that it performs high-precision sampling to meet the NIST's recommended security level of 112-bits or higher for the post-quantum era, which most existing hardware implementations fail to do. Furthermore, our design implementation is highly optimized for FPGA-based implementation and is also generic so that it can be seamlessly integrated into most cryptosystems. Synthesis results are obtained using Vivado design suite for a Xilinx Zynq-7010 CLG400ACX1341 FPGA board.**

***Index Terms*—Gaussian Noise Sampler, Lattice-based, R-LWE, Ziggurat, Rejection, Box-Muller**

I. Introduction

Recent breakthroughs in the field of quantum computing have led to two key milestone achievements. IBM launched a 53-qubit quantum computer [1] showcasing its advances in the field, and then Google pushed further toward quantum supremacy [2], by successfully programming its 53-qubit Sycamore quantum computer to generate random numbers. Although everyday use of quantum computers may seem distant, it is no longer regarded as a farfetched idea.

Quantum supremacy is defined as the point at which quantum computers can solve problems that are essentially unsolvable by classical computers within a reasonable time frame. Hence, with quantum supremacy, quantum computers will have the capability to break many existing encryption schemes whose security reductions are based on integer factorization and discrete logarithm problems. While these problems cannot be solved by existing classical computers in polynomial time, quantum computers will possess the computation power to do so.

With efforts from the National Institute of Standards and Technology (NIST) [3] to standardize new encryption schemes for the post-quantum era, lattice-based cryptography has gained popularity. This is because many classical cryptographic primitives can be realized very efficiently using lattices, providing strong security guarantees, including conjectured security against quantum computers. Furthermore, lattices allow the building of advanced schemes that go beyond classical public key encryption [4], like digital signatures [5], identity-based encryption [6], and even fully homomorphic encryption [7]–[9]. Lattice-based cryptography is also very alluring from an implementation standpoint, requiring only simple arithmetic operations on integers. Thus, it lends itself very well to implementations using reconfigurable hardware, such as field-programmable gate arrays (FPGAs), offering ample opportunity for optimizations and parallel designs.

At the heart of lattice-based cryptography is noise sampling, generally from discrete Gaussian distributions. These small noise samples play a crucial role in hiding information during both the key generation and encryption phases in lattice-based cryptosystems. Moreover, it has been shown that the bounded distance decoding problem underlying the Gaussian Sampling can be reduced to the Shortest Vector Problem (SVP) or Closest Vector Problem (CVP) [10]. The security reductions from these problems, i.e., SVP and CVP, form the theoretical foundation of lattice-based cryptography.

Although noise sampling could be performed using simpler distributions, e.g., uniform or binomial distributions, this would require increasing the noise levels significantly. Higher noise levels increase the complexity and lower the performance of these cryptosystems. Furthermore, many simpler distributions suffer from information leakage and weaker security guarantees. Hence, the Gaussian distribution represents a highly desirable choice for noise sampling with optimal performance and security trade-offs.

Obtaining discrete Gaussian samples with high precision is challenging and non-trivial. It represents one of the main hurdles in the effort to implement a secure scheme and poses a serious bottleneck to achieving good performance in practice. High-precision floating-point arithmetic operations are required to perform a high-precision Gaussian sampling with negligible statistical distance. Furthermore, achieving security against side-channel attacks [11], [12] has been recognized as an important problem. This is because developing a constant-time implementation of Gaussian sampling without incurring major performance penalties is still largely an unsolved problem. Moreover, an efficient implementation of a Gaussian

noise sampler without security losses remains elusive. It requires careful timing analysis of the underlying sampling algorithm so that the resulting implementation is constant-time and resistant to side-channel attacks.

Several algorithms exist for sampling from a discrete Gaussian distribution. Well-known algorithms include inversion sampling, rejection sampling, Knuth-Yao sampling [13], the Ziggurat method [14], Box-Muller [15] sampling, and Bernoulli sampling among others. Making an appropriate choice for the algorithm requires weighing the pros and cons of each algorithm, which can be overwhelming given the choices available. Two important parameters to consider are sampling efficiency and throughput.

Although a few hardware implementations of Gaussian distribution sampling exist, they are either low precision implementations that do not provide enough security or hardware-resource intensive. In this work, we present an FPGA-based efficient implementation with three different sampling algorithms and make usage recommendations based on the hardware cost, sampling efficiency and throughput. The key contributions of this work are:

1) **Implementation:** Highly-optimized FPGA-based implementation of Box-Muller, rejection, and Ziggurat sampling algorithms over a Gaussian probability distribution.
2) **High-Precision and Security:** The use of Max-log distance [16] along with statistical distance provides 128-bits of security, as recommended by NIST, with 64-bit floating-point precision. Moreover, all three implementations generate samples in constant-time and, hence, are resistant to side-channel attacks.
3) **Recommendation:** Evaluation of hardware utilization, sampling efficiency, and throughput to provide useful insights on the performance and make recommendations on which sampling algorithm is best to use for practical purposes.
4) **Parameterization:** A generic design implementation of all three sampling algorithms to enable easy plug in to a current or future cryptosystem's implementation.

The rest of the paper is organized as follows. Section II discusses some of the relevant related work. Section III will provide the required mathematical background. Sections IV, V, and VI will describe each of the three sampling algorithms along with their implementation details. Section VII will evaluate performance, Section VIII will discuss recommendations and future directions, and we then conclude the paper in Section IX.

II. Related Work

The inversion sampling method samples a random number from any probability distribution given its cumulative distribution function. Hence, inversion sampling requires precomputing and storing the values from a given probability distribution function and then computing the cumulative distribution function (CDF) for the given sample point. These CDF computations are often expensive to perform. One of the optimized inversion sampling implementations has been done by Du and Bai [17]. Although their sampler has good efficiency, it does not generate samples in constant time.

Other implementations of inversion sampling have been done by Pöppelmann et al. [18] and Howe et al. [19]. While the latter implementation is a constant time implementation, both of these implementations are costly as they incur a high resource utilization and generate samples at a very low rate.

Another popular choice for Gaussian sampling has been the Knuth-Yao sampling algorithm. This algorithm was introduced by Knuth and Yao [13] who used a tree-based approach to sample non-uniform distributions by using a minimal number of random bits, close to the entropy of the probability distribution. A non-constant time Gaussian sampler based on the Knuth-Yao algorithm was proposed in [20]. It takes 17 clock cycles on average to generate a sample, which is order of magnitude slower than any other existing implementations.

A constant time optimized implementation was done by Howe et al. [19]. Even though the implementation is light-weight compared to the previous implementations, the throughput is still insufficient as it takes about 10 clock cycles to generate a sample. Based on these existing implementations, it is safe to conclude that the Knuth-Yao sampling algorithm is inherently slow and costly when implemented in hardware.

Another alternative explored by Lee et al. [21], is the Box-Muller sampling algorithm. Their implementation consists of $sine$, $cosine$, and square root computation units which results in, the hardware-cost being approximately $3\times$ higher than any other existing implementations. However, the sampling efficiency is quite high with a throughput of 466 million samples per second.

Use of rejection sampling algorithm for Gaussian noise sampling was initially proposed by Gentry et al. [6]. There exists at least one hardware and software implementation of rejection sampling algorithm, both of which are done by Göttert et al. [22]. In contrast to their implementation approach applied in the software variant, which would require floating point arithmetic, the Gaussian sampler in hardware has been implemented by means of a look-up table. They present limited details on their hardware-based implementation and also do not present the associated hardware cost or the performance data for the Gaussian Noise sampler in the paper.

To the best of our knowledge other hardware implementations of rejection sampling algorithm do not exist. This can be attributed to the fact that the computation of the probability density function for a sampled integer value x can be complex and expensive in hardware. Furthermore, rejection sampling is not considered a very efficient sampling method due to its high rejection rate and low sampling efficiency.

The Ziggurat method is one of the more efficient sampling methods. Hence, it has been a popular choice for random number generation. Therefore, quite a few software and hardware implementations based on Ziggurat method exist. One of the early software implementations was done by Marsaglia et al. [14]. Another efficient and optimized C++ based implementation using the NTL [23] library was done by Buchman

et al. [24]. Although the software-based implementation is efficient if the Ziggurat setup is done offline, the hardware implementation is highly efficient, cost effective and generates samples in constant time at high rate.

One of the first hardware implementations of this method was done by Zhang et al. [25]. In their implementation, computation of the rectangular region is heavily pipelined. Furthermore, a buffering scheme is used to allow the processing of rectangular regions to continue operations in parallel with the evaluation of the wedge and tail computation. Yet another hardware implementation of Ziggurat sampling was done by Edrees et al. [26]. They suggested a modified Ziggurat with trapezoids partitioning the distribution rather than rectangles. This is better than the rectangular setup but does not improve the efficiency since rejection sampling still needs to be performed in the wedge area of the trapezoid.

Although these two hardware implementations are good candidates for comparing efficiency and hardware cost, we will refrain from doing so because they were not implemented specifically for lattice-based cryptography, and the authors do not present any details on the sampling precision. Hence, we do not know with certainty the level of security these implementations provide.

A more recent implementation for lattice-based cryptographic applications was presented by Howe et al. [19]. Their implementation is not very hardware-cost effective and takes about 9 clock cycles to generate a sample. Another drawback of this implementation is that it is based on the security proof from [27] and thus uses 32-bits of precision for sampling. Shortcomings of the security proof made in [27] were shown in [16]. Therefore, an implementation based on this security proof cannot guarantee enough security for the post-quantum era.

To summarize, there is still scope for optimization, both in terms of reducing the hardware cost and increasing the sampling efficiency using various sampling methods. As a result, in this work, we further optimize and implement three different sampling algorithms and present a detailed evaluation of these with respect to hardware cost and sampling efficiency.

III. Discrete Gaussian Distribution

The Gaussian distribution with standard deviation $\sigma \in \mathbb{R}$ and center $c \in \mathbb{R}$ evaluated at $x \in \mathbb{R}$ is defined by

$$\rho_{c,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-c)^2}{2\sigma^2}} \tag{1}$$

Here, $\frac{1}{\sigma\sqrt{2\pi}}$ is the normalizing factor. It does not impact the sampling process, as the samples from a non-normalized Gaussian distribution are equally good and it is safe to ignore the normalizing factor. Also, while sampling, the mean or the center c of the Gaussian distribution will be considered as zero and we will omit it. Hence, the probability distribution function equation (1) reduces to following equation:

$$\rho_{\sigma}(x) = e^{\frac{-x^2}{2\sigma^2}} \tag{2}$$

A discrete Gaussian distribution over $\mathbb{Z}$ centered at 0 is defined by $D_{\sigma}(x) = \rho_{\sigma}(x)/\rho_{\sigma}(\mathbb{Z})$ [28]. We also require a lattice parameter known as *smoothing parameter* that defines the width of the discrete Gaussian distribution beyond which the discrete Gaussian distribution acts like a continuous distribution. For an n-dimensional lattice L and positive real $\varepsilon > 0$, the smoothing parameter is denoted as $\eta_{\varepsilon}(L)$. The two upper bounds on the smoothing parameter are defined in [29]. We do not list them here due to space constraints.

Another important factor to consider is the *tail cut* parameter τ. The tail cut parameter administers how much of the less-heavy tails can be excluded in the practical implementation, for a given security level. By tail-cutting the Gaussian distribution curve, we ignore the large values present in the infinitely long tail of a given Gaussian probability distribution. For a one-dimensional Gaussian, *tail cut* parameter is computed as follows to tail cut less than $2^{-\lambda}$:

$$\tau \approx \sqrt{\lambda . 2 \log_e 2} \tag{3}$$

Here, λ is the security parameter and for $\lambda = 128$, $\tau \approx 9$. Figure 1 shows an example of Gaussian Distribution curve with center $c = 0$ and standard deviation $\sigma = 1$.

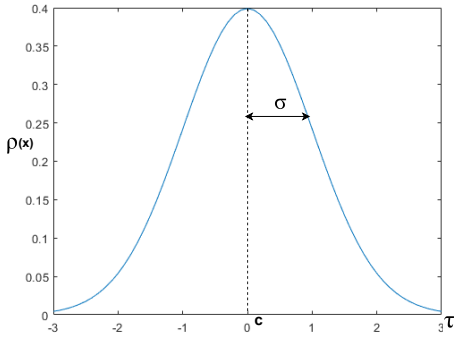

Fig. 1. Illustrative Gaussian Distribution Plot

A. Sampling Precision and Security

As we tail-cut the Gaussian distribution, a finite tail-bound introduces a statistical difference with the true Gaussian distribution. The tail-bound depends on the maximum statistical distance allowed by the security parameters. This is what is represented by equation 3, which includes the security parameter.

Statistical distance also determines the precision of sampling. This is true because secure applications require sampling with high precision to maintain a negligible statistical distance from the actual distribution. If ρ_z is the true probability of sampling $z \in \mathbb{Z}$ according to the distribution $D_{\mathbb{Z},\sigma}$ then our sampler will select z with a probability p_z where $|p_z - \rho_z| < \epsilon$ for some error-constant $\epsilon > 0$. For finite precision probabilities p_z, we can denote the approximate discrete Gaussian distribution with $\tilde{D}_{\mathbb{Z},\sigma}$. In order to preserve λ bits of security, the statistical distance Δ between the actual distribution $D_{\mathbb{Z},\sigma}$ and the approximate distribution $\tilde{D}_{\mathbb{Z},\sigma}$ is defined as follows:

$$\Delta(\tilde{D}_{\mathbb{Z},\sigma}, D_{\mathbb{Z},\sigma}) = sum|p(x) - \rho(x)| < 2^{-\lambda} \tag{4}$$

Hence to keep the statistical distance negligible, estimating the true probabilities will require either λ-bit fixed point or floating point approximations. So to achieve 128-bits of security, 128-bits of precision will be required, which means that using statistical distance is not cryptographically efficient.

To obtain sharper security bounds with lower precision, a new measure of closeness metric, max-log distance [16], between the probability distributions is more useful. The max-log distance Δ_{ML} between the two distributions $\tilde{D}_{\mathbb{Z},\sigma}$ and $D_{\mathbb{Z},\sigma}$ is defined as follows:

$$\Delta_{ML}(\tilde{D}_{\mathbb{Z},\sigma}, D_{\mathbb{Z},\sigma}) = max|ln\ p(x) - ln\ \rho(x)| < 2^{-\lambda/2} \quad (5)$$

Using max-log distance, it is possible to achieve more than 128 bits of security using just 64-bits of precision. This is not only cryptographically efficient but also leads to a highly optimized hardware implementation.

While statistical distance is convenient and easy to use, 64-bit security offered by 64-bit floating-point precision is not sufficient. Hence, in our implementations, we will first use statistical distance to perform tail-cut on the Gaussian distribution and then use max-log distance metric to define sampling precision so as to achieve 128-bits security with 64-bit floating-point precision.

IV. Box-Muller Sampling

The Box-Muller sampling method is based on the Box-Muller transform proposed by Box and Muller [15]. A basic form of Box-Muller transform takes two samples from the uniform distribution on the interval $[0, 1]$ and maps them to two standard Gaussian distributed samples. Algorithm 1 shows how the Box-Muller sampling works. Input to the algorithm is the standard deviation σ for desired probability distribution. Steps 1 and 2 generate two samples uniformly at random from the interval $[0, 1]$. In step 4 and 5, the required computations are performed to map the samples from the uniform distribution to the required Gaussian distribution. At every iteration, the algorithm generates two samples x and y as an output.

A. Hardware Implementation

To the best of our knowledge, there is just one existing hardware implementation of Box-Muller sampling in the literature. Due to the fact that it is expensive to compute cos, sin and square-root values on the fly in hardware. We faced similar challenge and thus, we precompute the cos, sin and square-root values. The precomputation stage is carried out offline and accordingly, the modified Box-Muller algorithm for the hardware implementation is as shown in algorithm 2.

Using the modified Box-Muller algorithm for hardware implementation simplifies the circuit as shown in Figure 2. We need three storage elements to store the precomputed values of $u1_{store}$, $u2_{store}$, and $u3_{store}$. Here, $u1_{store}$ is the precomputation of square-root component, $u2_{store}$ is the cos component's precomputation, and $u3_{store}$ is the precomputation os sin component. We utilize Block RAMs(BRAMs) on the FPGA board for this storage purpose.

Algorithm 1 Box-Muller Sampling Algorithm

Input: σ
Output: x, y
Repeat
1: choose $u_1 \leftarrow R = \mathbb{R} \cap [0, 1]$ uniformly at random
2: choose $u_2 \leftarrow R = \mathbb{R} \cap [0, 1]$ uniformly at random
3: **if** $u_1 \neq 0$ **then**
4: compute $x = \sigma\sqrt{-2 \ln u_1} \cos(2\pi u_2)$
5: compute $y = \sigma\sqrt{-2 \ln u_1} \sin(2\pi u_2)$
6: return x, y
7: **end if**

Algorithm 2 Modified Box-Muller Sampling Algorithm for Hardware Implementation

Input: σ
Output: x, y
1: Precompute:
2: choose $u_1 \leftarrow R = \mathbb{R} \cap [0, 1]$ uniformly at random
3: choose $u_2 \leftarrow R = \mathbb{R} \cap [0, 1]$ uniformly at random
4: **if** $u_1 \neq 0$ **then**
5: compute $u1_{store} = \sqrt{-2 \ln u_1}$
6: compute $u2_{store} = \cos(2\pi u_2)$
7: compute $u3_{store} = \sin(2\pi u_2)$
8: **end if**
Repeat
9: compute $x = \sigma \times u1_{store} \times u2_{store}$
10: compute $y = \sigma \times u1_{store} \times u3_{store}$
11: return x, y

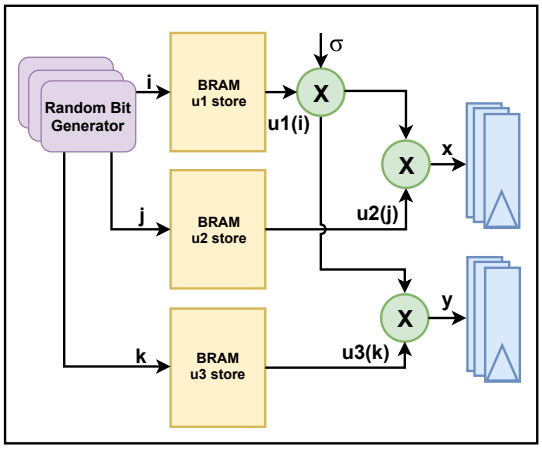

Fig. 2. Box-Muller Sampling Circuit

To generate the addresses for these storage elements so as to randomly access one value from each BRAM per iteration, we use a random bit generator. This random bit generator follows the design implementation proposed by Majzoobi et al. [30]. But we will not get into the design implementation details for

the random bit generator as it is out of scope for this work. Moreover, it can be replaced by any other random bit generator implementation as desired.

We store only 16 values for $u1$, $u2$, and $u3$ each. Using these 16 values we will be able to generate 4096 samples with different permutations. There are two advantages of doing this. First, the random bit generator needs to generate only 4 random bits in each iteration to index into the BRAMs. Second, the BRAM size will be small as it needs to store only limited number of values. Each value will be a standard double-precision float requiring 64-bits. Hence, for storing 16 such values in BRAM, we need only 1024-bits of storage space within each BRAM.

The remaining circuit is fairly straight forward, requiring only three multipliers to obtain the final samples. It is worth noting that we perform an additional multiplication operation to multiply $u1$ with σ. This trade-off is to provide generic implementation of the algorithm. Thus, different values of σ can be plugged-in to obtain the samples from any desired Gaussian distribution curve. Each iteration of the algorithm generates two samples and therefore, we will need only, say, 500 iterations to generate 1000 samples. Hence, the Box-Muller sampling algorithm is a very efficient constant-time sampling algorithm.

V. Rejection Sampling

Rejection sampling is a basic technique used to generate samples from a given probability distribution. It is also called an "acceptance-rejection" method or "accept-reject algorithm". The principle behind rejection sampling is based on the observation that to sample a random variable in one dimension, a uniformly random sampling from the interval $[0, 1]$ can be performed and the samples that fall under the required density function graph can be retained. Rejection sampling algorithm works as shown in algorithm 3.

Algorithm 3 Rejection Sampling Algorithm

Input: σ, τ
Output: x
Repeat
1: choose $x \leftarrow Z = \mathbb{Z} \cap [0, \tau]$ uniformly at random
2: choose $y \leftarrow R = \mathbb{R} \cap [0, 1]$ uniformly at random
3: **if** $y < \rho(x)$ **then**
4: return x
5: **else**
6: reject x
7: **end if**

The rejection sampling algorithm starts by generating an integer value x in the specified range. As lattice-based cryptography mostly requires positive samples only, so we sample in the range 0 to τ where τ is the tail cut parameter. If an application requires negative samples as well, then the range can be extended from $-\tau$ to $+\tau$. The algorithm then generates a real number y in the interval $[0, 1]$. The next step in the algorithm is to check if y lies under the curve of the required Gaussian distribution. For this purpose, we need to compute the value of probability density function of x and then compare y against $\rho(x)$. If y is smaller then the sample x is accepted. Otherwise, x has to be rejected as it does not lie within the required Gaussian distribution curve.

A. Hardware Implementation

The hardware implementation for rejection sampling is straight forward and its circuit is as shown in figure 3.

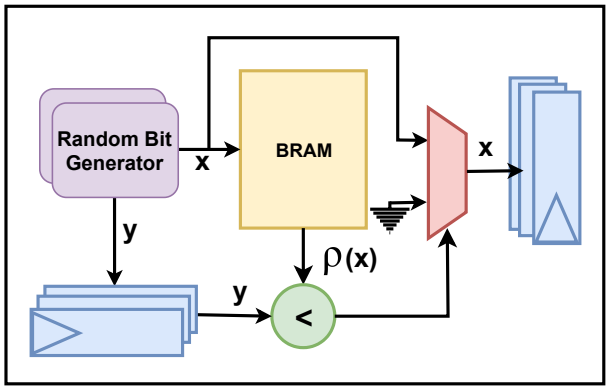

Fig. 3. Rejection Sampling Circuit

We again leverage the same random bit generator as in the Box-Muller sampling algorithm. Here, we will use it first to generate a random number x between a given range, and then we will use it to generate another random number y in the interval [0,1]. We will precompute and store corresponding $log(\rho(x))$ values in the BRAM. Computing the $\rho(x)$ values is expensive due to the exponentiation operation and so, we do not want to do it in hardware. The memory required to store the precomputed probability density function values will be $O(\tau)$. we need to store 9 values as value of τ approximately equals 9 for 128-bit security, with each value requiring only 64 bits. Therefore, storing these values do not add to the hardware cost as much.

The value of x will require only $log_2(\tau)$ bits and is also used to index into the BRAM. This substantially reduces the hardware cost as we do not need additional address generation logic to index into BRAM. Once both the $\rho(x)$ and y values are available, a 64-bit comparator is used to compare the values. If the comparator generates a one after the comparison operation then the value of x is accepted as a sample under the desired Gaussian distribution. Otherwise, the algorithm needs to perform another iteration to generate more samples.

This hardware implementation is generic and can be reused to generate samples from any Gaussian distribution. To do so what needs to be modified is just the probability density function values stored in BRAM, corresponding to another desired Gaussian distribution having a different standard deviation σ. Additionally, the value of tail cut parameter τ is parameterized to generate samples from a different range as per the security requirements of a given cryptographic application.

The hardware implementation of rejection sampling is simple and inexpensive. The efficiency of rejection sampling

algorithm is given by $2\tau/\sqrt{2\pi}$. The poor efficiency can be a real bottleneck and slow down the entire cryptographic system. Thus, to achieve a better sampling efficiency, we discuss an improved version of rejection sampling in the next section.

VI. Ziggurat Sampling

Marsaglia and Tsang developed the Ziggurat transform method [14] for sampling from decreasing densities. The method is based on covering the target density with a set of horizontal equal-area rectangles, a cap and a tail. The Ziggurat title came from the appearance of the layered rectangles. Ziggurats are ancient Mesopotamian terraced temple mounds that, mathematically, are two-dimensional step functions. A one-dimensional ziggurat underlies Marsaglia's algorithm.

A. Ziggurat Setup

Figure 4 shows the Ziggurat formation over a Gaussian distribution curve. The Ziggurat algorithm covers the area under the probability density function by a slightly larger area with $n-1$ sections. To sample x's from our desired normal distribution, we need to generate random points (x, y), uniformly distributed in the plane, and retain those that fall under the curve while rejecting points that do not fall under this curve. The Figure 4 has $n = 8$; a practical implementation of Ziggurat algorithm will typically take values for n between 128 and 512. The choice of n affects the speed, but does not affect the accuracy of the algorithm.

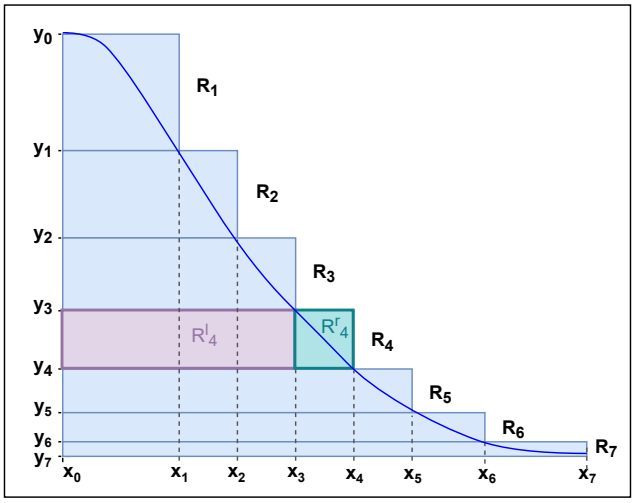

Fig. 4. *Ziggurat(with $n = 8$) over a Gaussian Distribution Curve*

The $n-1$ sections of the ziggurat are represented using rectangles. The bottom section is a rectangle along with a bounded tail under the curve of $\rho(x)$. The right-hand edges of the rectangles are at the point's x_k shown with the points where dotted lines intersect the distribution curve on the figure 4. With $\rho(x_0) = 1$ and $\rho(x_{n+1}) = 0$, the height of the k^{th} section is given by $\rho(x_k) - \rho(x_{k+1})$. It is important to choose the x_k's so that all $n-1$ sections, including the one on the bottom with a tail-cut, have the same area. This is the key distinguishing feature of Marsaglia's algorithm; the rectangles are horizontal and have equal areas.

For a specified number, n, of sections, it is possible to solve a transcendental equation to find corresponding x_n. This will be the point where the infinite tail meets the last rectangular section. The following equation can be used to compute the next x value:

$$x_{k-1} = -\sqrt{2\sigma^2 log(\rho(x_{k-1})\sigma\sqrt{2\pi})} \tag{6}$$

Hence, once x_n is known, it is easy to compute the common area of the sections and the other right-hand endpoints x_k. It is also possible to compute $\rho_k = x_{k-l}/x_k$, which is the fraction of each section that lies underneath the section above it. These fractional sections form the core of the Ziggurat. The right-hand edge of the core is the dotted line in the figure. The remaining portions of the rectangles, to the right of the dotted lines in the area covered by the graph of $\rho(x)$, are the tips. The computation of the x_k's and ρ_k's is done only once, and the values can be precomputed and stored. In table I, we present the x_n and v i.e. the area of the rectangles corresponding to various n values. The table is crucial to ziggurat setup and serves as a quick starting point.

TABLE I
x_n AND RECTANGLE AREAS CORRESPONDING TO DIFFERENT n VALUES

n	x_n	v (area of rectangles)
8	2.3383716982472524	1.7617364011877759e-1
16	2.6755367657376135	8.3989463747827300e-2
32	2.9613001212640193	4.0758744432219871e-2
64	3.2136576271588955	2.0024457157351700e-2
128	3.4426198558966519	9.9125630353364726e-3
256	3.6541528853610088	4.9286732339746571e-3
512	3.8520461503683916	2.4567663515413529e-3

B. Ziggurat Sampling Algorithm

After the Ziggurat setup is done, Gaussian distributed random numbers can be computed very quickly. The key portion of the code computes a single random integer, j, between 0 and n and a single uniformly distributed random number, u, between 0 and 1. Then, we compute $z = u \cdot x_j$. After computing z, first we check to see if z falls in the tail area of the curve. If it does, then we return z from the tail. If not, a check is made to see if z falls in the core of the j^{th} section. If it does, then we know that z is the x-coordinate of a point under the probability distribution function, and this value can be returned as one sample from the normal distribution. If this is also not the case, then the point z lies in the smaller rectangular area. This small rectangular area is shown by the green rectangle marked as R_4^r in the figure 4. If z is in this region, we need to check if it lies under or above the curve. For this check, we can perform rejection sampling and if z lies under the curve it is accepted otherwise, it is rejected. The Ziggurat algorithm's formal description is as shown in algorithm 4.

Algorithm 4 Ziggurat Sampling Algorithm

Input: σ, core, n, x
Output: z
Repeat
1: choose $j \leftarrow Z = \mathbb{Z} \cap [0, n]$ uniformly at random
2: choose $u \leftarrow R = \mathbb{R} \cap [0, 1]$ uniformly at random
3: compute $z = u * x_j$
4: **if** j = 0 **then**
5: return z from tail
6: **else if** $z < core_j$ **then**
7: return z
8: **else**
9: perform rejection sampling on z
10: **end if**

$$e = \frac{\sqrt{2\pi}/2n}{v} \tag{7}$$

The efficiency of the rejection procedure in this algorithm is given by the equation 7. As the value of n increases, the efficiency of the sampling improves.

C. Hardware Implementation

For an optimized and efficient hardware implementation, we propose a modified Ziggurat sampling algorithm. Before we introduce the modified algorithm, we present a quick analysis on why the modification is feasible. We were interested in learning what percent of the area of the rectangles will fall into the core area. The core area for a sample rectangle is highlighted in pink in Figure 4. For this purpose, we performed the Ziggurat setup with various n values ranging from $n = 8$ to $n = 512$ rectangles and computed the average core area for each setup.

TABLE II
AVERAGE CORE AREA(IN PERCENT) FOR DIFFERENT n VALUES

n	Avg. core area
8	77.5
16	81.2
32	84.7
64	89.8
128	91.3
256	94.8
512	96.9

An interesting observation is that as we divide the Gaussian distribution curve using a larger number of rectangles while setting up the Ziggurat, the core area increases as well. Table II affirms this fact furthermore. The advantage is that this leads to increasing the probability of a sample lying in the core area and hence we can either just accept or reject the sample. This will, in turn, save us from performing the additional rejection sampling check for the remaining part of the rectangle. Moreover, as mentioned earlier, the Ziggurat setup can be done offline and x values precomputed and stored. So, the only added overhead will be in terms of the memory required to store large number of x values. For our implementation, we will use an optimal value of $n = 64$ so that we do not increase the memory requirements significantly and at the same time we still have almost 90% of the core area covered.

1) Modified Ziggurat Sampling Algorithm: The modified Ziggurat algorithm is given in algorithm 5. We get rid of sampling from the tail as we have a tail cut parameter in place. Thus, in step 1 on the modified algorithm, we sample j from the interval $[1, n]$. Also, we get rid of step 9 of the original algorithm, because we just reject the sample if it is not in the core area. This also has the added advantage that the implementation generates samples in constant-time as it does not need to spend extra time performing rejection sampling. Constant-time sampling provides resistance to side-channel attacks. The steps to check if the sample lies in the core area are similar to those of the original algorithm. We will discuss and evaluate the impact on sampling efficiency for the modified Ziggurat algorithm in Section VII.

Algorithm 5 Modified Ziggurat Sampling Algorithm for Hardware Implementation

Input: σ, core, n, x
Output: z
Repeat
1: choose $j \leftarrow Z = \mathbb{Z} \cap [1, n]$ uniformly at random
2: choose $u \leftarrow R = \mathbb{R} \cap [0, 1]$ uniformly at random
3: compute $z = u * x_j$
4: **if** $z < core_j$ **then**
5: return j
6: **else**
7: reject j
8: **end if**

Figure 5 shows our design implementation circuit for the Ziggurat sampling algorithm. We again leverage the same random bit generator to generate two random samples i.e. j in the range $[1, n]$ and u in the interval $[0, 1]$. To represent j, we will require $log_2(n)$ bits. In our implementation, we used $n = 64$ and hence j will require $log_2(64) = 6$ bits. The implementation considers n as a parameter and it is possible to replace it by a different value. But, as the value of n will change, the corresponding values stored for core area and x must also be updated in the BRAM. Each value, in both the BRAMs, will be 64 bits, and we will need to store 64 such values. Therefore, each BRAM will require 4096-bits of storage. If memory is a constraint, then a smaller value of n can be chosen while trading off the sampling efficiency.

To perform a multiplication between u and x_j we will need a 64-bit multiplier. Additionally, a 64-bit comparator will be required for performing the comparison between z and corresponding $core_j$ values. To get the required integer sample, we store the value of j if it satisfies the comparison results. Thus, the resulting implementation is compact, fast and

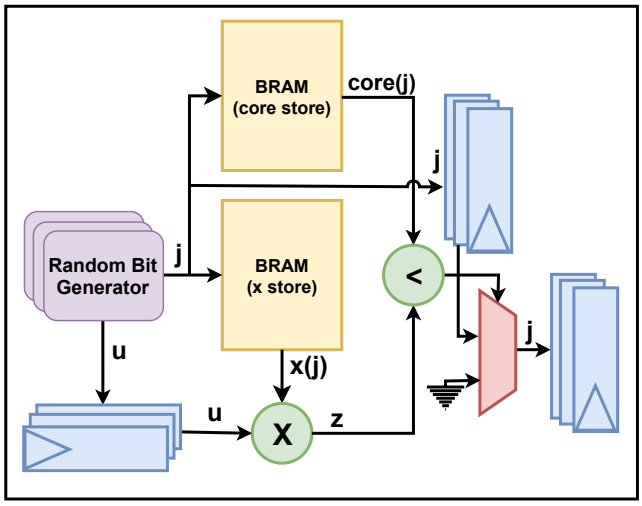

Fig. 5. Ziggurat Sampling Circuit

generates high quality Gaussian random numbers with correct distribution in constant time.

VII. Performance Evaluation

In this section, we present the hardware resource utilization for each of our sampler implementations. We will compare and evaluate the efficiency of these samplers as well. All of our implementations are done using Verilog, with Xilinx Zync-7010 CLG400ACX1341 FPGA board as the target device for synthesis.

TABLE III
Hardware cost for different samplers

Sampling Algorithm	Slice LUTs	BRAM	DSP	Freq. (MHz)
Box-Muller Sampling	146	1	11	204.6
Rejection Sampling	89	1	0	76.4
Ziggurat Sampling	114	1.5	9	103.5

In table III, we present the synthesis results for generating samples from the Gaussian distribution with $\sigma = 3.33$, $\tau = 9$, and a sampling precision of 64-bits. We observe that the hardware resource utilization for rejection sampling is the lowest, while the hardware cost of the Box-Muller sampling method is the highest. The hardware cost for the random bit generator is not included in the hardware resource utilization of the samplers because it can be seamlessly replaced by any efficient random bit generator implementation that meets the application's requirement.

TABLE IV
Hardware Cost Comparison for Box-Muller Sampling

Implementation	Precision	LUT	BRAM	DSP	Freq. (MHz)
[21]	16-bit	1528	12	3	233
Our work	64-bit	717	6	18	270.9

Table IV presents the comparison of hardware cost for Box-Muller sampling. The data represent the synthesis results of implementation done by Lee et al. [21] for $\sigma = 8.2$. The target device used by the authors is a Xilinx Virtex-4 XC4VLX100-12 FPGA board. We synthesized our implementation on the same FPGA board using similar parameters but still maintaining a sampling precision of 64 bits. The results thus obtained are also presented in the table IV. When comparing the hardware resource utilization for both the implementations, we found that LUT and BRAM utilization is about $2\times$ higher in [21] whereas our implementation has $6\times$ more DSP utilization. However, the achieved operating frequencies in both the implementations are almost comparable. Also we would like to highlight that the security provided by the implementation in [21] is not clearly defined.

Next, we compare the hardware cost associated with our Ziggurat sampling implementation to that in [19]. Again, to keep the comparison fair, we synthesized our Ziggurat sampling implementation using the Xilinx ISE design tool on a XC6SLX25-3 Spartan-6 FPGA board, as this is the target device used in [19]. Table V presents the result of this comparison for $\sigma = 3.33$.

TABLE V
Hardware Cost Comparison for Ziggurat Sampling

Implementation	λ	LUT	BRAM	DSP	Freq. (MHz)
[19]	64	785	0	26	60.3
Our work	128	143	1.5	16	114.1

For similar sampling parameters with double the precision, our implementation utilizes approximately $5\times$ fewer LUTs and $2\times$ fewer DSPs than the implementation in [19]. In addition, we achieved almost twice the operating frequency as compared to the other implementation. It is worth noting that their Ziggurat setup involves dividing the Gaussian distribution curve by only $n = 8$ rectangles, while our Ziggurat setup involves dividing the curve into $n = 64$ rectangles. Hence, we have a 1.5 BRAM utilization while the other implementation has 0 BRAM utilization. To summarize, using the implementation in [19] one will have to spend about 3 times the hardware resources to obtain lower precision samples.

We do not compare the hardware cost for rejection sampling. Rejection sampling has been implemented in hardware by Göttert et al. [22], along with their public key encryption scheme. However, the authors do not present the hardware cost details explicitly for the Gaussian sampler in the paper.

TABLE VI
Approximate Latency(in clock cycles) of different Samplers to generate n samples

Sampling Algorithm	Latency
Box-Muller Sampling	0.61n
Rejection Sampling	7.30n
Ziggurat Sampling	3.13n

In Table VI, we present the generic latency computation equations for each of sampling methods. The lower latency

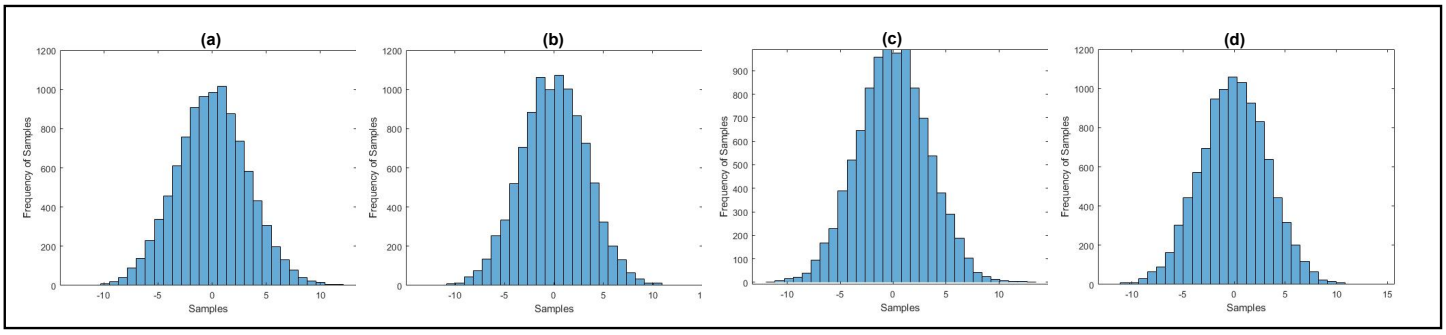

Fig. 6. Samples generated from the three samplers: *(a) and (b) Box-Muller Sampling*, *(c) Rejection Sampling*, and *(d) Ziggurat Sampling*.

in Box-Muller sampling can be attributed to the fact that the algorithm generates about two samples every clock cycle. With Ziggurat sampling, we observe a sub-optimal latency of close to three clock cycles per sample. And rejection sampling has poor latency, with a requirement of about 7 clock cycles per sample. This is expected for rejection sampling, as the area under the curve in figure 1 is less than the area outside the curve.

We generated about $10,000$ samples from each of the samplers. In figure 6, we present the histogram plots for $\sigma = 3.33$. The plots show the quality of samples obtained from all the three samplers. It is worth recalling that the Box-Muller sampler generates two samples at any given point of time, which yields two different plots for the Box-Muller sampling method.

The final parameters to be evaluated are the sampling efficiency and the throughput. The results are presented in the table VII. Box-Muller's sampling efficiency is almost 100%, with a throughput close to 408M samples per second. The efficiency of rejection sampling is governed by the equation $2\tau/\sqrt{2\pi}$. After evaluation of our implementation, we observed that rejection sampling's efficiency closely follows this equation and exhibits an efficiency as low as 15%. Additionally, the throughput for rejection sampling is about 152M samples per second. For Ziggurat sampling, the efficiency is close to 90%, as the Ziggurat setup has been done with $n = 64$. However, we observed that the Ziggurat sampler is capable of generating about 205M samples per second.

TABLE VII
APPROXIMATE EFFICIENCY(IN PERCENTAGE) AND THROUGHPUT(SAMPLES/SEC) OF DIFFERENT SAMPLERS

Sampling Algorithm	Efficiency	Throughput
Box-Muller Sampling	100	408M
Rejection Sampling	15	152M
Ziggurat Sampling	90	205M

Finally, we compare the latency and throughput of our work, [21] and [19]. The results shown in table VIII are obtained while running the implementations for similar FPGA boards. Our implementation of Ziggurat sampling is almost $3\times$ faster than the implementation in [19]. Their implementation is slower, because their algorithm is not optimized for hardware implementation and the authors perform rejection sampling to check if the sample lies in the smaller rectangular area near the curve. The operating frequency in our implementation is almost twice that of their implementation.

When we compare the latency and throughput of our Box-Muller implementation with the implementation in [21], we observe a speedup of about $1.15\times$. Thus, our implementation is capable of generating about 80M more samples per second. It is worth noting that the sampling precision in their implementation is just 16 bits, which is not sufficient to provide enough post-quantum security.

TABLE VIII
LATENCY(IN CLOCK CYCLES) AND THROUGHPUT(SAMPLES/SEC) COMPARISON FOR ZIGGURAT AND BOX-MULLER SAMPLING METHOD

Implementation	Sampling	Freq.(MHz)	Latency	Throughput
[19]	Ziggurat	60.3	9n	67M
Our work	Ziggurat	114.1	3.13n	115M
[21]	Box-Muller	233	0.5n	466M
Our work	Box-Muller	270.9	0.61n	540M

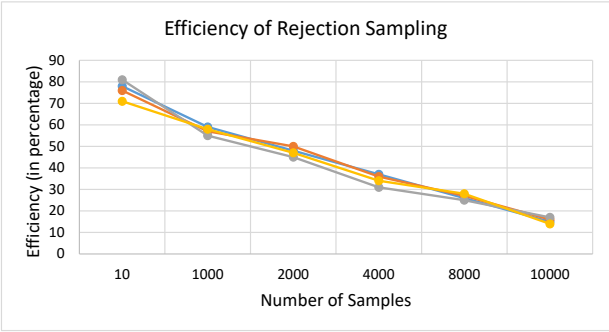

Fig. 7. Effect of increasing number of samples on efficiency in rejection sampling

Another interesting trend we observed with rejection sampling is that as the number of samples to be generated increases, the efficiency starts to drop. We generated between 10 to $10,000$ samples multiple times to confirm the trend. Figure 7 shows the plot of decrease in efficiency with the increase in the number of samples.

VIII. Future Directions and Recommendations

Based on the evaluations and observations made in the previous section, we make the following recommendations:

1) Box-Muller sampling has the highest sampling efficiency and throughput but also has the highest resource utilization. Thus, use of Box-Muller sampling is the best choice when there is no resource constraint and sampling efficiency is the main criterion for an application.
2) Rejection sampling has the lowest resource utilization and can be used in applications where resource (memory/area) is a constraint and only a few samples need to generated. Although, a trade off on sampling efficiency and throughput will have to be made if large number of samples are required.
3) The Ziggurat sampling method is optimal in terms of both the resource utilization and sampling efficiency. The choice of using Ziggurat sampling can be made irrespective of the resource utilization or sampling efficiency trade-offs.

The research community, at large, can benefit from these useful insights and in turn will be able to make an appropriate choice of Gaussian Noise Sample for their own applications.

IX. Conclusion

In this paper, we presented a highly optimized and efficient FPGA-based implementation of Box-Muller, rejection, and Ziggurat sampling algorithms over a Gaussian probability distribution. The design implementation is constant-time with high-precision sampling making it post-quantum secure and resistant to side-channel attacks. A parameterized design implementation of all three sampling algorithms provides an opportunity to plug them in easily to any existing or future cryptosystems. Evaluation of hardware cost, sampling efficiency, and throughput provided useful insights on the best sampling algorithm to use for practical purposes.

References

[1] IBM Quantum Computing, "On quantum supremacy," https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy/, 2019.

[2] Google Quantum Supremacy, "Computing takes a quantum leap forward," https://www.blog.google/technology/ai/computing-takes-quantum-leap-forward/, 2019.

[3] NIST. (2018) Post-quantum cryptography. [Online]. Available: csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions

[4] R. Agrawal, L. Bu, A. Ehret, and M. Kinsy, "Open-source fpga implementation of post-quantum cryptographic hardware primitives," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 211–217.

[5] V. Lyubashevsky and D. Micciancio, "Asymptotically efficient lattice-based digital signatures," in *Theory of Cryptography Conference*. Springer, 2008, pp. 37–54.

[6] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions," in *Proceedings of the fortieth annual ACM symposium on Theory of computing*. ACM, 2008, pp. 197–206.

[7] C. Gentry *et al.*, "Fully homomorphic encryption using ideal lattices." in *Stoc*, vol. 9, no. 2009, 2009, pp. 169–178.

[8] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, p. 13, 2014.

[9] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-lwe and security for key dependent messages," in *Annual cryptology conference*. Springer, 2011, pp. 505–524.

[10] N. Stephens-Davidowitz, "Discrete gaussian sampling reduces to cvp and svp," in *SODA*, 2016. [Online]. Available: http://arxiv.org/abs/1506.07490

[11] J. Bootle, C. Delaplace, T. Espitau, P.-A. Fouque, and M. Tibouchi, "Lwe without modular reduction and improved side-channel attacks against bliss," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2018, pp. 494–524.

[12] L. G. Bruinderink, A. Hülsing, T. Lange, and Y. Yarom, "Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2016, pp. 323–345.

[13] D. Knuth, "The complexity of nonuniform random number generation," *Algorithm and Complexity, New Directions and Results*, pp. 357–428, 1976.

[14] G. Marsaglia, W. W. Tsang *et al.*, "The ziggurat method for generating random variables," *Journal of statistical software*, vol. 5, no. 8, pp. 1–7, 2000.

[15] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *Ann. Math. Statist.*, vol. 29, no. 2, pp. 610–611, 06 1958. [Online]. Available: https://doi.org/10.1214/aoms/1177706645

[16] D. Micciancio and M. Walter, "Gaussian sampling over the integers: Efficient, generic, constant-time," in *Annual International Cryptology Conference*. Springer, 2017, pp. 455–485.

[17] C. Du and G. Bai, "Towards efficient discrete gaussian sampling for lattice-based cryptography," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015, pp. 1–6.

[18] T. Pöppelmann, L. Ducas, and T. Güneysu, "Enhanced lattice-based signatures on reconfigurable hardware," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2014, pp. 353–370.

[19] J. Howe, A. Khalid, C. Rafferty, F. Regazzoni, and M. O'Neill, "On practical discrete gaussian samplers for lattice-based cryptography," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 322–334, 2016.

[20] S. S. Roy, F. Vercauteren, and I. Verbauwhede, "High precision discrete gaussian sampling on fpgas," in *International Conference on Selected Areas in Cryptography*. Springer, 2013, pp. 383–401.

[21] D.-U. Lee, J. D. Villasenor, W. Luk, and P. H. W. Leong, "A hardware gaussian noise generator using the box-muller method and its error analysis," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 659–671, 2006.

[22] 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 *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012, pp. 512–529.

[23] NTL: A Library for doing Number Theory, https://www.shoup.net/ntl/.

[24] 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 *International Conference on Selected Areas in Cryptography*. Springer, 2013, pp. 402–417.

[25] Guanglie Zhang, P. H. W. Leong, Dong-U Lee, J. D. Villasenor, R. C. C. Cheung, and W. Luk, "Ziggurat-based hardware gaussian random number generator," in *International Conference on Field Programmable Logic and Applications, 2005.*, Aug 2005, pp. 275–280.

[26] H. Edrees, B. Cheung, M. Sandora, D. B. Nummey, and D. Stefan, "Hardware-optimized ziggurat algorithm for high-speed gaussian random number generators." in *ERSA*, 2009, pp. 254–260.

[27] M.-J. O. Saarinen, "Gaussian sampling precision and information leakage in lattice cryptography." *IACR Cryptology ePrint Archive*, vol. 2015, p. 953, 2015.

[28] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky, "Lattice signatures and bimodal gaussians," in *Annual Cryptology Conference*. Springer, 2013, pp. 40–56.

[29] D. Micciancio and O. Regev, "Worst-case to average-case reductions based on gaussian measures," *SIAM Journal on Computing*, vol. 37, no. 1, pp. 267–302, 2007.

[30] M. Majzoobi, F. Koushanfar, and S. Devadas, "Fpga-based true random number generation using circuit metastability with adaptive feedback control," in *Cryptographic Hardware and Embedded Systems – CHES 2011*, B. Preneel and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 17–32.