Discrete Gaussian Sampling for Low-Power Devices

Shruti More

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science and Systems

University of Washington

2015

Committee:

Raj Katti

Orlando Baiocchi

Anderson Nascimiento

Program Authorized to Offer Degree:

Computer Science Systems

©Copyright 2015

Shruti More, Raj Katti

University of Washington

Abstract

Discrete Gaussian Sampling for Low-Power Devices **Shruti More**

Chair of the Supervisory Committee:

Dr. Raj Katti

Institute of Technology

Abstract: Sampling from the discrete Gaussian probability distribution is used in latticebased cryptosystems. A need for faster and memory-efficient samplers has become a necessity for improving the performance of such cryptosystems. We propose a new algorithm for sampling from the Gaussian distribution that can efficiently change on-the-fly its speed/memory requirement. The Ziggurat algorithm that attempted to do this requires up to thousands of seconds of computation time to change memory requirements on-the-fly. Our algorithm eliminates this large computational overhead.

I. Introduction

Lattice based cryptosystems [1-3,5,6] have gained importance because of their ability to achieve homomorphic encryption. Sampling from a discrete Gaussian probability distribution over the integers is an important operation in lattice-based cryptography. For example, the sampling algorithm took more than 50% of the time in a lattice-based signing algorithm [5]. Finding efficient and accurate methods to sample from a discrete Gaussian distribution for low-power devices still

remains a challenge. To the best of our knowledge the only work that addresses this issue is [1]. The algorithm converts the Ziggurat algorithm to the discrete case.

The discrete Gaussian probability distribution, for parameter $\sigma > 0$ (σ = standard deviation) [1] assigns $x \in Z$ (Z is the set of integers) a probability proportional to $\rho(x) = \exp(-\frac{\frac{x^2}{2}}{\sigma^2})$. For lattice based cryptography we sample from the bounded subset $B = [-t\sigma, t\sigma] \cap Z$, where the tail-cut $t > 0$ is a parameter [6]. The discrete Ziggurat algorithm uses precomputed rectangles of equal area to cover the area under the Gaussian probability density function (pdf) (see Fig. 4 or 5). Since the rectangles have to be of the same area, changing the number of rectangles requires their recomputation. The advantage of the discrete Ziggurat algorithm is that when it uses more memory (for the precomputed rectangles) it runs faster than when it uses less memory. Resource constrained devices can therefore use the "low memory-low speed" option when running low on battery power. However, if the device wanted to switch on the fly from the "high memory-high speed" option to the "low memory-low speed" option, then it would have to re-compute and store the rectangles for the new option. This re-computation takes anywhere from tens of seconds to thousands of seconds, as shown in *Fig. 10*, and therefore requires the very resources that the device is short on. The only other option is to pre-compute the rectangles for each option and store them. This solution requires a large amount of memory and is not good for low-power devices.

We propose a modification of the discrete Ziggurat algorithm that does not require the recomputation of rectangles each time a device chooses a new "memory-speed" option. Our algorithm precomputes the rectangles only once and enables a device to choose different "memory-speed" options by storing a minimal number of extra points $(x_i, \rho(x_i))$. If a device wanted to switch from one option to another, it could do so on-the-fly by simply using the extra values stored. Our algorithm also runs faster by decreasing about more than 50% of the number of times the computationally intensive pdf function, $\rho(x_i)$ is computed.

The rest of the paper is organized as follows. Section II describes the Ziggurat Algorithm and the ideas behind our proposed modification of this algorithm. Section III describes prior work. Section IV describes the implementation of the proposed algorithm. Section V describes the timing results and Section VI concludes the paper.

II. THE ZIGGURAT ALGORITHM

For ease of understanding we first describe the Ziggurat algorithm for the discrete case.

In discrete case, the Ziggurat algorithm uses the Gaussian curve where x is positive. The curve is a scaled density function with $y_0 \geq 1$. The curve is then partitioned into equal area horizontal rectangles. The partitions are made based on the bottom right vertices of the rectangles on the curve.
The algorithm is described in more detail below.

The Ziggurat Algorithm:

1. Split the area under the Gaussian pdf curve into rectangles of equal area. See Fig. 4 that shows rectangles R_1 through R_{14} in the right half of the pdf curve. Let the bottom right hand corner of rectangle R_i have co-ordinates (x_i, y_i).

2. Select a rectangle R_i uniformly at random.
3. Sample an x-coordinate inside R_i by sampling x' from $[0, x_i]$ uniformly at random.
4. If $x' \leq x_{i-1}$ (that is x' is in the left rectangle of R_i (see rectangle R_3 in Fig. 4 that shows the left and right rectangle for R_3)), return x'.
5. Otherwise, $x' > x_{i-1}$, and x' is in the right rectangle of R_i. We set y' to $y_{i-1} - y_i$.
6. If $(\lfloor x_i \rfloor + 1 \leq \sigma)$, then draw line between$(\lfloor x_{i-1} \rfloor, y_{i-1})$, $(\lfloor x_i \rfloor, y_i)$. If the point lies below the line then return or else check if $y' \leq \rho(x')$ then returnx'. Otherwise, reject and restart the whole process by going to step 1.
7. If $(\sigma \leq \lfloor x_{i-1} \rfloor)$, then draw line between$(\lfloor x_{i-1} \rfloor, y_{i-1})$, $(\lfloor x_i \rfloor, y_i)$. If the point lies in the rejection area then return or else check if $y' < \rho(x')$ then returnx'. Otherwise, reject and restart the whole process by going to step 1.

Step 1 states that the curve is partitioned into equal size, S, rectangles. If m is the number of rectangles, $t\,\sigma$ is the tailcut, and $\rho(x)$ is the equation of the curve, then the following variables are initialized as described by Buchmann [1]:

$$y_m := 0, x_0 := 0, x_m := t\sigma$$

Then the following are iteratively computed from left to right for the partitions,

$$y_{m-1} = \frac{S}{1+\lfloor x_m \rfloor}, \qquad x_{m-1} = \rho_\sigma^{-1}(y_{m-1}),$$

$$\text{for } i = m-2, \ldots, 1:$$

$$y_i = \frac{S}{1+\lfloor x_{i+1} \rfloor} + y_{i+1}, \; x_i = \rho_\sigma^{-1}(y_i),$$

$$y_0 = \frac{S}{1+\lfloor x_1 \rfloor} + y_1$$

The curve becomes wide or narrow based on σ or the standard deviation. However it maintains $y_0 \geq 1$. Thus the partitions have to be made in such a way that this condition is satisfied. S or the size of the rectangles is given by the following equation, where constant c is initialized to 1:

$$S = \frac{\sigma}{m\sqrt{\frac{\pi}{2}}}.c$$

The partitions are computed using the above formula and increasing c until $y_0 \geq 1$. If in case a valid partition is not found then x_m is increased by 1 and the process of iteratively computing the partition parameters is repeated.

Step 2 is about sampling a rectangle at random. This is done by randomly picking a number i between 1 and m where m is the number of rectangles. The first rectangle is the top most rectangle on the curve as depicted by *Fig. 4.*

Once the rectangle R_i is chosen, next a variable x is chosen from $\{0, \dots, \lfloor x_i \rfloor\}$ which is the value of the point inside the rectangle and variable s is chosen from $\{-1, 1\}$ which is the sign of the $x$value. If the sign is negative, it simply means that the point sampled is on the left side of the $y$axis.

Each of the rectangles are divided into right and left as shown in *Fig. 4*. If a point x_i is in the left rectangle it is returned as the area of left rectangle is contained under the curve.

If the point falls in the right rectangle, a y-value is set to $y_{i-1} - y_i$. This is very fast as the x coordinates as well as the corresponding y co-ordinates of the discrete partition points are precomputed and stored in arrays.

The computationally intensive part of the algorithm is computing $\rho(x')$ in steps 6 and 7 above. The probability that step 6 is executed decreases as the number of rectangles increases. The computation of $\rho(x')$ can be avoided in half the cases by modifying step 6 as follows [1].

Step 6, 7: Draw a straight line between the upper-left corner (co-ordinates (x_{i-1}, y_{i-1})) and the lower-right corner (co-ordinates (x_i, y_i)) of the right rectangle of R_i. The equation of this line is, $y = f(x) = y_i + \frac{y_i - y_{i-1}}{x_i - x_{i-1}}(x - x_i)$. Fig. 1 and 2 show the right part of rectangle R_i with the pdf curve $\rho(x)$ and the straight line. The pdf curve is concave down in Fig. 1 and is concave up in Fig. 2. If $\rho(x)$ is concave down then if $y' \leq y_i + \frac{y_i - y_{i-1}}{x_i - x_{i-1}}(x' - x_i) = f(x')$ return x' else if $y' \leq \rho(x')$ then return x'. Otherwise, reject and restart the whole process by going to step 1. If $\rho(x)$ is concave up then if $y' \geq y_i + \frac{y_i - y_{i-1}}{x_i - x_{i-1}}(x' - x_i) = f(x')$ go to step 1 else if $y' < \rho(x')$ then return x'. Otherwise, reject and restart the whole process by going to step 1.

The steps 6 and 7 use the line to do rejection sampling and hence reduces the number of computations of $\rho(x)$. Our main contribution is to further modify step 6 and 7 by using several lines instead of one line. This idea is described in Fig. 3 which shows three lines instead of one line from point P (co-ordinates (x_{i-1}, y_{i-1})) to point Q (co-ordinates (x_i, y_i)). The figure shows two extra points A and B on the pdf curve $\rho(x)$. Note that the number of extra points is a parameter called "*factor*". We modify step 6 by checking if point (x', y') is under the segments $(\overline{PA}, \overline{AB}, \overline{BQ})$ or over these segments when $\rho(x)$ is concave down or concave up respectively (Fig. 3 only shows the concave down case). For the concave down case, if point (x', y') is under the segments $(\overline{PA}, \overline{AB},$

BQ) then return x' else if $y' \leq \rho(x')$ then return x'. Otherwise, reject and restart the whole process by going to step 1.

This reduces the number of times $\rho(x')$ is computed by more than the optimization of [1] that uses only one line. It also gives the same effect as having three rectangles instead of one without re-computing the rectangles. Another advantage of our method is that each right rectangle could have a different number of points added thereby giving us more control over how much memory (precomputed extra points) is needed by different parts of the pdf curve. We can also compute rectangle points P and Q only once and add points A and B on the fly as more rectangles are needed. Such an algorithm becomes useful for low-power devices which no longer have to re-compute rectangles for different options (low-memory, low-speed vs high-memory, high-speed). Moreover, the Ziggurat algorithm is known to be good for Gaussian distributions with a high standard deviation.

Discretization of the above algorithm is carried out by using a Gaussian distribution centered at 0 with a bounded support $B = [-t\sigma, t\sigma] \cap Z$, where the tail-cut t > 0 is a parameter (in our case t = 13 is sufficient [1]). The area of rectangle R_i in the discrete case is $(1 + \lfloor x_i \rfloor)(y_{i-1} - y_i)$. An algorithm for computing (x_i, y_i) for each rectangle R_i (i = 1 to m, where m is the number of rectangles and is a parameter) is given in [1]. Note that each rectangle must have the same area. We use the n-bit fixed-point representation to specify real numbers (x_i, y_i) and $\rho(x_i)$. A parameter $\omega = n + 1$ = 106 (see [1]) is used to represent the number of bits in the fixed-point representation of real numbers.

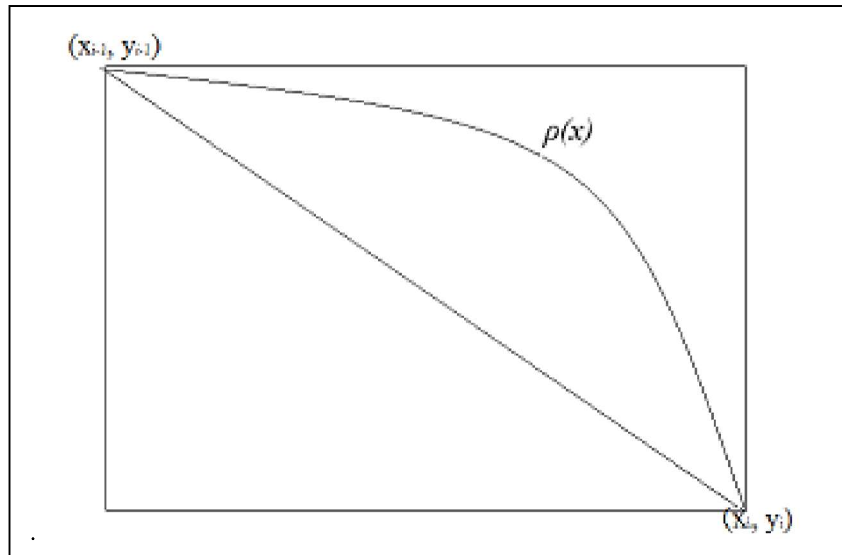

Fig. 1. Right rectangle Ri with $\rho(x)$ concave down.

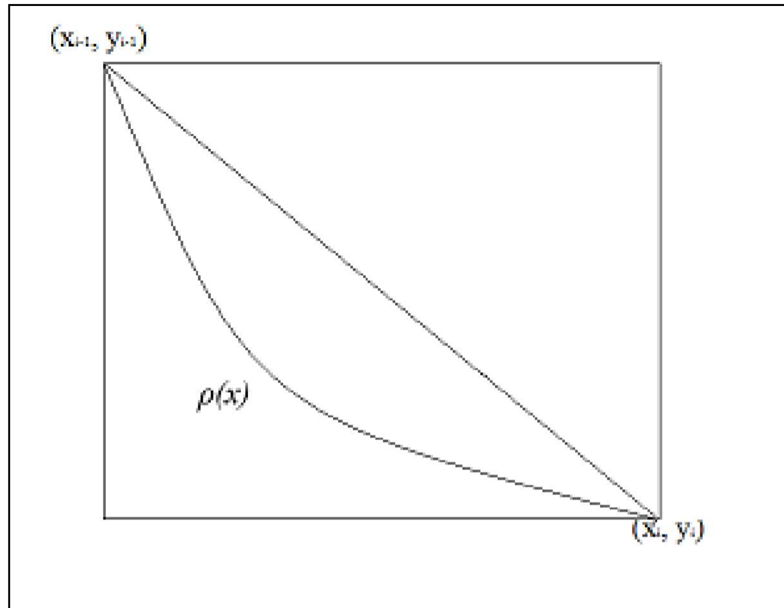

Fig. 2. Right rectangle *Ri* with $\rho(x)$ concave up.

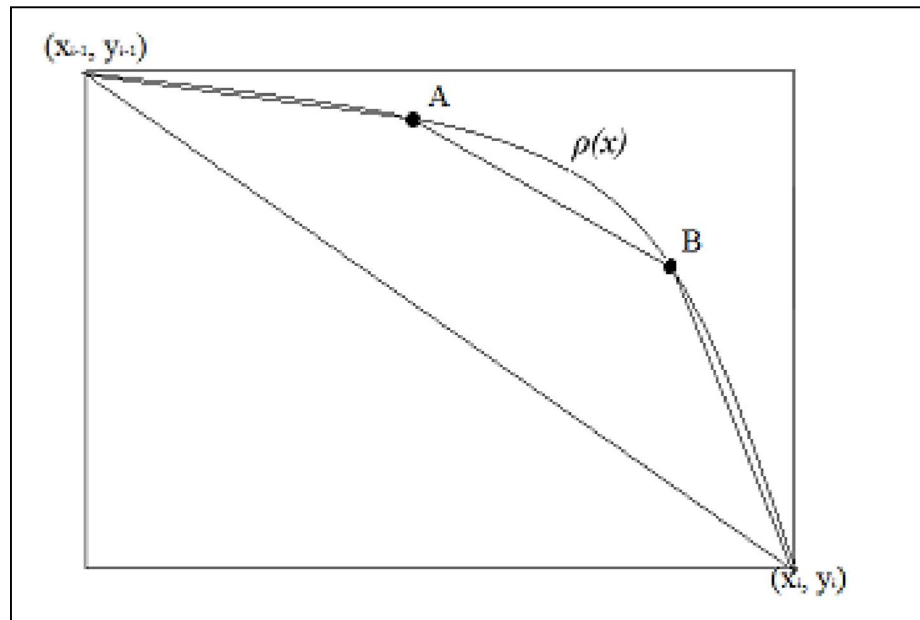

Fig. 3. Right rectangle *Ri* with $\rho(x)$ concave down and extra points A and B.

III. RELATED WORK

We briefly survey the existing methods for discrete Gaussian sampling and compare these with our algorithm.

The Discrete Ziggurat algorithm [1] by Buchmann, et al. describes an algorithm with timememory trade-offs. The algorithm uses partitioning of the Gaussian pdf curve. Our algorithm is based on the Ziggurat algorithm using which we try to improve how time-memory constraints are handled. Like Ziggurat, our algorithm is also customizable based on memory- availability on a device. We can change the parameter *factor* to accommodate memory constraints.

The Knuth Yao algorithm described by Galbraith and Dwarkanath [3] performs better than Ziggurat and hence our algorithm. It uses a tree to store the values of the Gaussian pdf curve in a table. However, this algorithm uses more than 400 times the memory used by the Ziggurat algorithm [1].

IV. Algorithm Implementation

This section describes our algorithm which is an improvement over the Ziggurat algorithm. We first describe how to decide how many extra points need to be added on the pdf curve in a rectangle. Fig. 4 shows a scenario ($\sigma = 32$) where there are 14 rectangles of equal area with two extra points on the pdf curve in each rectangle (these are like points A and B in Fig. 3). The parameter *factor* in this case equals 2 for every rectangle. The number of extra points can be changed according to the memory available on the device or the microcontroller being used. Instead of using *factor = 2* as shown in *Fig. 5*, we can use a larger number to increase speed. Let the two points added on the pdf curve in rectangle R_i be A and B. Then the x-coordinates of A and B can be calculated as follows.

$$x_A = x_{i-1} + (x_i - x_{i-1})/(factor + 1)$$

$$x_B = x_A + (x_i - x_{i-1})/(factor + 1)$$

Around the inflection point of the pdf curve (at $x = \sigma$), adding extra points in a rectangle makes no difference. This is because the line segment between the upper-left corner and lower-right corner of a right-rectangle almost coincides with line segments drawn similar to segments $\overline{PA}, \overline{AB}, \overline{BQ}$ in Fig. 3. Therefore, in our algorithm we do not add extra points on the pdf curve in rectangles around the inflection point. The condition for a rectangle R_i not to add any extra points is as follows. Let the slope of the pdf curve at the inflection point be $S_1 = \frac{d\rho(x)}{dx}$ evaluated at $x = \sigma, y = \rho(\sigma)$. Let the coordinates of the lower right corner of rectangle R_i be (x_i, y_i). Let the slope of the pdf curve at (x_i, y_i) be $S_2 = \frac{d\rho(x)}{dx}$ evaluated at (x_i, y_i). If $\frac{S_1}{S_2} \leq 2$ then no extra points are added on the pdf
curve for rectangle R_i. In Fig. 5 this results in rectangles R_3 through R_6 having no extra points. All other rectangles have two extra points added. Parameter *count* represents the number of rectangles that do not have any extra points (for Fig. 5, *count* = 4). The total number of points, T, stored by our algorithm is the points (x_i, y_i) and the extra points in rectangles not around the inflection point. Thus, T = (factor + 1) * (m+1) – (factor * count). The number of points needed to store *m* rectangles are 1 more than *m*, which is the number of rectangles. Thus when we add 1 to *m* when multiplying with *factor*.

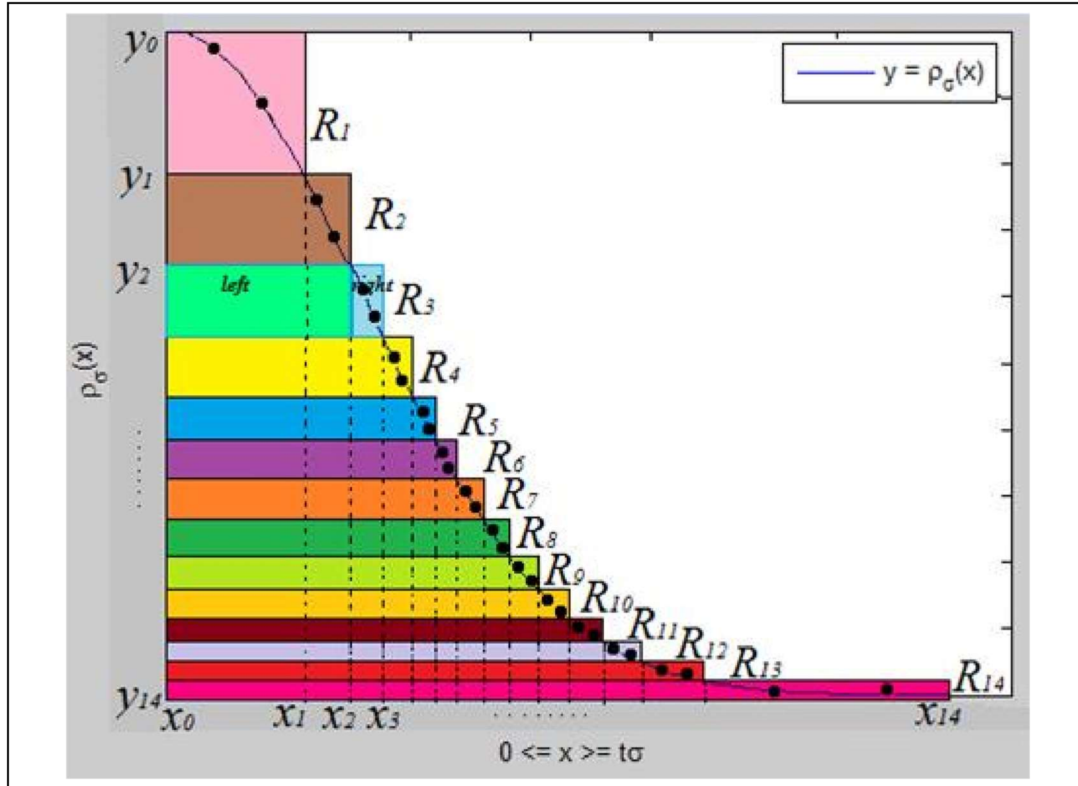

Fig. 4. The extra points are caused by partitioning the rectangles further. In this case each rectangle has 2 extra points.

A. *Our Algorithm*

The Ziggurat algorithm starts with partitioning the area under the pdf curve into equal-area rectangles. One of the rectangles, R_i (with lower right-corner co-ordinates (x_i, y_i)) is chosen uniformly at random. Values $x' \leq x_i$ and $y' \in [y_i, y_{i-1}]$ are sampled uniformly at random and $x'$is returned only if (x', y') lies under the pdf curve. This happens if it lies in the left rectangle of R_i or in the part of the right rectangle that is under the pdf curve. To check if a point lies under the pdf curve efficiently, the curve is approximated by a straight line. In this work we approximate the pdf curve by multiple straight lines. This gives us the following advantages for resource constrained devices:

1. Adding extra points is equivalent to having more rectangles without the re-computing rectangles. The re-computation of rectangles is a time consuming operation that takes from a tens of seconds to thousands of seconds.
2. Points do not have to be added near the inflection point of the pdf curve. This reduces the memory requirement of our algorithm.

In summary, if we let (x_i, y_i) to represent the lower right corners of rectangles computed by the original Ziggurat algorithm and(x_i', y_i') to represent the extra points within the rectangles, then the steps of our algorithm are shown below and described in detail in Algorithm 2. The algorithm checks if a point (x', y') is below or above the pdf curve depending on whether the pdf curve is concave-down or concave-up respectively. Also function sLine( ), described in Algorithm 3 below, simply draws a line between two points.

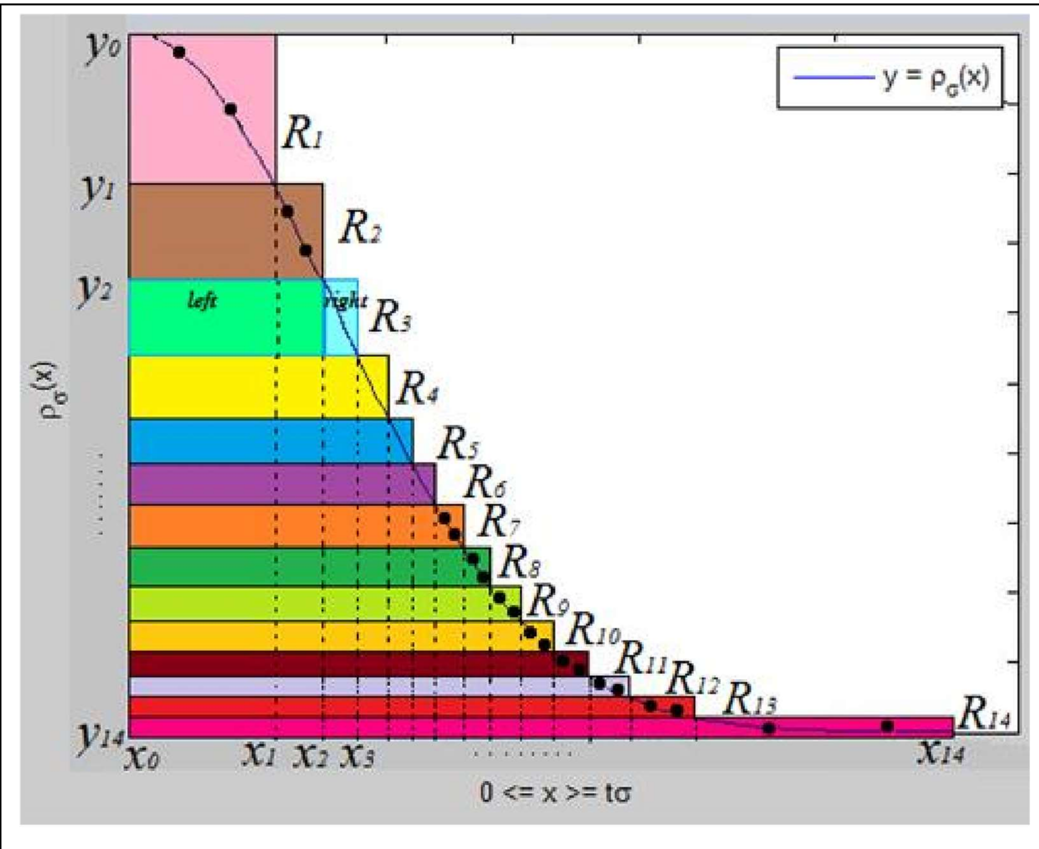

Fig. 5. The extra points are caused by partitioning the rectangles further. In this case each rectangle has 2 extra points.

B. *Steps of our algorithm:*

select rectangle R_i

sample a point x_i in R_i

if (x_i, y_i) is on concave down curve:

if $(x_i, y_i) \leq sLine(x_i, x_{i+1}, y_i, y_{i+1})$ then accept (x_i, y_i)

else

for(k between 0 and count):

if $(x_i, y_i) \leq sLine(x'_{i+k}, x'_{i+k+1}, y'_{i+k}, y'_{i+k+1})$

then accept (x_i, y_i)

else if (x_i, y_i) is on concave up curve:

t = false

if $(x_i, y_i) \leq sLine(x_i, x_{i+1}, y_i, y_{i+1})$ then continue

else

for(k between 0 and count):

if $(x_i, y_i) \leq sLine(x'_{i+k}, x'_{i+k+1}, y'_{i+k}, y'_{i+k+1})$

t = true

then continue

if(t = false) then accept (x_i, y_i)

else if $(x_i, y_i) \leq \rho(x_i)$ then accept (x_i, y_i)

Next we give the pseudo code for three algorithms. Algorithm 1 computes the extra points in rectangles. Note that the rectangles themselves are determined by the algorithm in [1]. Algorithm 1 takes as input, *m* (the number of rectangles), *σ* (the standard deviation of the Gaussian distribution), and *factor* (the number of extra points in a rectangle). Algorithm 1 produces as output, *start* (the index of the first rectangle which is not further partitioned), *count* (the number of rectangles which have no extra points), the y-coordinates of the lower right corners of the rectangles, and the x and y coordinates of the extra points in the rectangles. Algorithm 2 runs our new modified Ziggurat algorithm. Algorithm 3 determines the line segment between two points.

Algorithm 1: Improved Partitions

Input: m, σ, $\lfloor x_1 \rfloor, \ldots, \lfloor x_m \rfloor$, factor, ω

Output: m, σ, $\lfloor x_1 \rfloor, \ldots, \lfloor x_m \rfloor$, $\bar{y}_0, \bar{y}_1, \ldots, \bar{y}_m$, $\lfloor x'_1 \rfloor, \ldots, \lfloor x'_{m'} \rfloor$, $\bar{y}'_0, \bar{y}'_1, \ldots, \bar{y}'_{m'}$, ω, start, count

1 while true do

2 $i \leftarrow \{1,\ldots,m\}$, $x \leftarrow \{0,\ldots, \lfloor x_i \rfloor\}$

3 if $(\rho'_\sigma(\sigma) / \rho'_\sigma(x_i)) \geq 2$ then

// count x_i's when slope(ρ (σ)) / slope(ρ (x_i)) ≥ 2

count++

4 end

5 m' ← (factor+1)*(m+1)-(factor*count)

```
6        while true do
7        i ←{1,...,m}, x ←{0,..., ⌊xi⌋}
8        diff ← (xi - xi-1)/factor
9        x'i = xi ,   ȳ 'i =  ρ( x'i)
10       if (ρ′σ(σ) / ρ′σ(xi)) ≥ 2 then
     // find the first i in ignored area
11       start ← first  xi in the ignored area
12       for I from 0 to m-1
13       if i < start
14       index = i * (factor+1)
15       else if i > start + count
16       index = i * (factor+1) – count – 1
         17       else
18       index = i * (factor+1) + (i – start)

19       for j from 1 to factor
         //add additional points in rectangles
20       x'index + j  = x'index + j - 1 + diff
21       ȳ 'index+j =  ρ( x'index + j)
22       end
23       end
24       end
25       end
```

Algorithm 2: Modified Ziggurat

```
1   while tru do
2   choose i from {1 to m}, s from {-1, 1}, and x from {0 to ⌊xi⌋};
3   if 0 < x ≤⌊xi−1⌋ then return sx;
```

```
4     else
5     if x = 0 then
6     b ←{0,1};
7     if b = 0 then return sx;
8     else continue;
9     else
          // in rejection area now
10    y′ ←{0,...,2^ω −1}, ȳ = y′·( ȳ_{i−1} − ȳ_i);
11    if ⌊x_i⌋+ 1 ≤ σ then
              // in concave-down case
12    if ȳ≤2^ω·sLine(⌊x_{i−1}⌋,⌊x_i⌋, ȳ_{i−1}, ȳ_i ;x)  ∨          ȳ ≤2^ω ·(ρ_σ(x)− ȳ_i)
then return sx;
13    else
14    if i < start
15    index = i * (factor+1)
16    else if i > start + count
17    index = i * (factor+1) – count – 1 18                      else
19                       index = i * (factor+1) + (i – start)
20                       for k between 0 and factor
21                       if
                          ȳ≤2^ω·sLine(⌊x'_{index+k}⌋,
⌊x'_{index+k+1}⌋,                          ȳ '_{index+k},  ȳ
'_{index+k+1} ;x−1)  ∨                      ȳ ≤ 2^ω·(ρ_σ(x)− ȳ_i)
22                                return sx;
23          else if σ ≤⌊x_{i−1}⌋ then
24          t = false;
              // in concave-up case
25          if
             ȳ ≥2^ω·sLine(⌊x_{i−1}⌋, ⌊x_i⌋, ȳ_{i−1}, ȳ_i ;x−1)  ∨
ȳ > 2^ω·(ρ_σ(x)− ȳ_i)
26          if i < start
```

27 index = i * (factor+1)

28 else if i > start + count

29 index = i * (factor+1) – count – 1 30 else

31 index = i * (factor+1) + (i – start)

32 for k between 0 and factor

33 if

$\bar{y} \geq 2^{\omega} \cdot$ sLine($\lfloor x'_{index+k} \rfloor$, $\lfloor x'_{index+k+1} \rfloor$, $\bar{y}'_{index+k}$, $\bar{y}'_{index+k+1}$;x−1) 34 t = true;

35 if t is false ^ $\bar{y} \leq 2^{\omega} \cdot (\rho_{\sigma}(x) - \bar{y}_i)$ 36 return sx;

37 else if ^ $\bar{y} \leq 2^{\omega} \cdot (\rho_{\sigma}(x) - \bar{y}_i)$ then return sx; 38 end

39 end

40 end

41 end

Algorithm 3: sLine($\lfloor x_{i-1} \rfloor$, $\lfloor x_i \rfloor$, $\bar{y}_{i-1}$, $\bar{y}_i$;x)

1 if $\lfloor x_i \rfloor = \lfloor x_{i-1} \rfloor$ then return −1;

2 Set $y_i = \bar{y}_i$ and $y_{i-1} = \bar{y}_{i-1}$ for i > 1

1 for i = 1

3 return $(x - \lfloor x_i \rfloor) \cdot (y_i - y_{i-1}) / (\lfloor x_i \rfloor - \lfloor x_{i-1} \rfloor)$

V. RESULTS

The platform used to implement our algorithms was the Windows 8.1 64-bit operating system running on the i7 CPU at 2.00GHz. The computer used 8.00 GB RAM. We implemented all algorithms using the Microsoft Visual C++ 2008 Express Edition. We used the Number Theory Library (NTL) [4] to help in our implementations.

We refer to our algorithm as the improved partitions algorithm. Tables I through IV below compare the time for Ziggurat and our algorithm. Each table depicts for a standard deviation, the time it takes to run the Ziggurat algorithm vs. our algorithm for different values of *factor* (*factor* varies from 1 through 7). The tables below use precision or ω = 106. Each timing value is the amount of time it takes to generate 1 million Gaussian samples.

TABLE I. TIME COMPARISON FOR STD DEV 10

m	Timings (s)							
	Zigg	*Improved Partitions with factor:*						
		1	2	3	4	5	6	7
63	23.7	16.1	17.3	16.9	18.3	20.2	19.2	20.5
499	9.61	9.19	8.98	9.06	9.25	9.25	9.36	9.34
999	8.34	7.8	8.2	7.97	8.12	7.97	8.03	8.11
1999	7.94	7.78	8.14	7.94	8.08	8.56	7.7	8.05
3999	7.53	7.25	7.27	7.28	7.25	7.3	7.31	7.33
7999	7.36	7.16	7.23	7.19	7.24	7.19	7.22	7.22

Fig. 6. Time comparison for Ziggurat and our algorithm given *m* and σ = 10.

TABLE II. TIME COMPARISON FOR STD DEV 32

m	Timings (s)							
	Zigg	*Improved Partitions with factor:*						
		1	2	3	4	5	6	7
63	21.26	15.9	17.6	19.1	18.5	19.8	19.0	20.4
499	9.6	8.59	8.61	8.81	8.84	9.03	9.03	9.25
999	8.27	7.91	7.81	7.86	8.03	8.06	8.12	8.17
1999	8.11	7.61	7.61	7.66	7.69	7.7	7.78	7.74
3999	7.49	7.25	7.25	7.28	7.29	7.31	7.3	7.67
7999	7.36	7.29	7.27	7.34	7.31	7.31	7.31	7.34

Fig. 7. Time comparison for Ziggurat and our algorithm given *m* and σ = 32.

TABLE III. TIME COMPARISON FOR STD DEV 1000

m	Timings (s)							
	Zigg	*Improved Partitions with factor:*						
		1	2	3	4	5	6	7
63	20.99	15.6	15.2	16.8	16.8	18.1	19.6	19.6
499	8.94	8.62	8.61	8.67	8.74	8.78	8.92	9.09
999	8.19	7.94	7.95	8.14	8.05	8.08	8.0	8.75
1999	7.86	7.61	7.67	7.52	7.58	7.64	7.64	7.67
3999	7.53	7.41	7.45	7.44	7.88	7.48	7.48	7.52
7999	7.39	7.3	7.31	7.34	7.34	7.34	7.38	7.39

Fig. 8. Time comparison for Ziggurat and our algorithm given *m* and σ = 1000

TABLE IV. TIME COMPARISON FOR STD DEV 1.6E5

m	Timings (s)							
	Zigg	*Improved Partitions with factor:*						
		1	2	3	4	5	6	7
63	20.14	15.2	16.7 1	16.2	17.6	17.4	18.7	20.0
499	9.36	8.72	8.84	8.86	9.13	9.08	9.05	9.3
999	8.06	7.99	8.20	8.31	8.3	8.45	8.19	8.69
1999	7.8	7.74	7.62	7.64	8.0	7.72	7.77	7.7
3999	7.42	7.39	7.38	7.39	7.41	7.42	7.44	7.44
7999	7.42	7.34	7.39	7.39	7.45	7.31	7.47	7.49

Fig. 9. Time comparison for Ziggurat and our algorithm given *m* and σ = 1.6e5

The tables above show that our algorithm runs faster for smaller *m*. When *m = 63*, the improved partitions algorithm performs well for all standard deviations. When *m = 7999*, there are already a lot of partitions uniformly spread throughout the pdf curve, hence the improved partitions algorithm does not perform better here.

The results show that our algorithm performs as well as the Ziggurat in terms of speed. However, our algorithm has the added advantage that it does not have to re-compute rectangles to change memory requirements on-the-fly. To compare the speed consider Table I for *m = 999* and *factor = 1*. Since *factor = 1*, it implies that there is one extra point in every one of the 999 rectangles. This creates a situation similar to having two rectangles in every rectangle (or having 999 * 2 = 1998 rectangles). The original Ziggurat for *m = 1999* rectangles executes in 7.94 seconds whereas our algorithm with *factor =1* and *m = 999* executes in 7.8 seconds. Therefore, our modified algorithm has the same time complexity as the Ziggurat while using less memory and having the flexibility to change memory requirements on-the-fly.

TABLE V. TIME COMPARISON FOR PARTITIONING

m	**Timings (s)**			
	Sigma:			
	10	32	1000	1.6e5
63	8.011	8.456	8.060	8.012
499	72.810	64.439	62.560	66.497
999	187.144	148.078	124.828	119.119
1999	424.449	297.494	248.006	269.375
3999	1630.223	1086.901	648.892	535.595
7999	3979.862	1954.301	1041.865	800.012

Fig. 10. Time comparison for Ziggurat's create partition algorithm for different sigmas.

The table in *Fig. 10* shows the timings taken to partition the Gaussian curve into *m* different rectangles by the create partitions algorithm in Ziggurat. It can be seen that partitioning the curve into 7999 rectangles takes a very long time (about an hour) which is very inefficient. However, going from 3999 partitions to about 7999 in just a few seconds using our algorithm by setting *factor* equal to 1 improves the performance. Our algorithm takes the two partitions created by Ziggurat and multiplies by *factor*+1 to get the extra points. As seen in *Fig. 6*, *Fig. 7*, *Fig. 8*, and *Fig. 9*, that the time it takes to generate a discrete random number using 3999 partitions and *factor* = 1 is almost the same or even better than the time it takes using 7999 Ziggurat partitions. Thus we can run the discrete random sampling generator algorithm along with the partitioning in about half

an hour for m = 3999 and *factor* = 1 which is better than an hour to run the Ziggurat algorithm with m = 7999.

Another observation from the tables is that *factor = 1* or *2* suffices for good performance for all m's and standard deviations. This is because as the number of extra points increases, the sLine algorithm is executed more frequently which takes up CPU cycles thus we increase the speed by avoiding the computations.

VI. CONCLUSION

We have proposed a new algorithm for sampling from the discrete Gaussian probability distribution that is a modification of the Ziggurat algorithm. Our algorithm is efficient and provides more control to low-power devices that need to use lesser amounts of memory when low on battery. Our algorithm, therefore, is an excellent alternative for use in lattice-based cryptography.

References

[1] Buchmann, Johannes, et al. "Discrete Ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers." *Selected Areas in Cryptography--SAC 2013*. Springer Berlin Heidelberg, 2014. 402-417.

[2] Lyubashevsky, Vadim, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings." *Journal of the ACM (JACM)* 60.6 (2013): 43.

[3] Dwarakanath, Nagarjun C., 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 (2014): 159-180. [4] Shoup, Victor. "Number Theory C++ Library (NTL) version 5.4. 1." (2003).

[5] Weiden, P., Hulsing, A., Cabarcas, D., and Buchmann, J., "Instantiating treeless signature schemes," Cryptology ePrint Archive, Report 2013/065, 2013. http://eprint.iacr.org/

[6] Gentry, C., Peikert, C., and Vaikuntanathan, V., "Trapdoors for hard lattices and new cryptographic constructions," 40th Annual ACM Symposium on Theory of Computing, pp 197-206, 2008.