High Precision Discrete Gaussian Sampling on FPGAs

Sujoy Sinha Roy, Frederik Vercauteren and Ingrid Verbauwhede

ESAT/SCD-COSIC and iMinds, KU Leuven
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
Email: {firstname.lastname}@esat.kuleuven.be

Abstract. Lattice-based public key cryptography often requires sampling from discrete Gaussian distributions. In this paper we present an efficient hardware implementation of a discrete Gaussian sampler with high precision and large tail-bound based on the Knuth-Yao algorithm. The Knuth-Yao algorithm is chosen since it requires a minimal number of random bits and is well suited for high precision sampling. We propose a novel implementation of this algorithm based on an efficient traversal of the discrete distribution generating (DDG) tree. Furthermore, we propose optimization techniques to store the probabilities of the sample points in near-optimal space. Our implementation targets the Gaussian distribution parameters typically used in LWE encryption schemes and has maximum statistical distance of 2^{-90} to a true discrete Gaussian distribution. For these parameters, our implementation on the Xilinx Virtex V platform results in a sampler architecture that only consumes 47 slices and has a delay of $3ns$.

Keywords. Lattice-based cryptography, Discrete Gaussian Sampler, Hardware implementation, Knuth-Yao algorithm, Discrete distribution generating (DDG) tree

1 Introduction

Lattice-based cryptography has become one of the main research tracks in cryptography due to its wide applicability (see [19] for some applications) and the fact that its security is based on worst-case computational assumptions that remain hard even for quantum computers. The significant advancements in theoretical lattice-based cryptography [14, 15, 17] have more recently been complemented with practical implementations [16, 9, 5] both in software and hardware.

Lattice-based cryptosystems often require sampling from discrete Gaussian distributions. The implementation of such a discrete Gaussian sampler for cryptographic applications faces several challenges [7]. Firstly, most existing sampling algorithms require a large number of random bits, which could become a limitation for lattice-based cryptosystems on a computationally weak platform. Secondly, the sampling should be of high precision, i.e. the statistical distance to the true distribution should be negligible for the provable security results to hold [4]. Sampling with negligible statistical distance however either requires high precision floating arithmetic or large precomputed tables.

There are various methods for sampling from a non-uniform distribution [1]. Rejection sampling and inversion sampling are the best known algorithms. In practice, rejection sampling for a discrete Gaussian distribution is slow due to the high rejection rate for the sampled values which are far from the center of the distribution [9]. Moreover, for each trial, many random bits are required which is very time consuming on a constrained platform.

The inversion method first generates a random probability and then selects a sample value such that the cumulative distribution up to that sample point is just larger than the randomly generated probability. Since the random probability should be of high precision, this method also requires a large number of random bits. Additionally, the size of the comparator circuit increases with the precision of the probabilities used. A recent work [3] shows that the number of random bits can be reduced by performing table lookup in a lazy fashion.

In [11], Knuth and Yao proposed a random walk model for sampling from any non-uniform distribution. They showed that the number of random bits required by the sampling algorithm is close to the entropy of the distribution and thus near-optimal. However the method requires the probabilities of the sample points to be stored in a table. In case of a discrete Gaussian distribution, the binary representations of the probabilities are infinitely long. For security reasons, the probability expansions should be stored with high precision to keep the statistical distance between the true Gaussian distribution and its approximation small [4]. Hence the storage required for the probabilities becomes an issue on constrained platforms. In [7], Galbraith and Dwarkanath observed that the probability expansions for a discrete Gaussian distribution contain a large number of leading zeros which can be compressed to save space. The authors proposed a block variant of the Knuth-Yao algorithm which partitions the probabilities in different blocks having roughly the same number of leading zero digits. The paper however does not report on an actual implementation.

Although there are several hardware implementations of continuous Gaussian samplers [6, 10], these implementations have low precisions and are not suitable for sampling from discrete Gaussian distributions. To the best of our knowledge, the only reported hardware implementation of a discrete Gaussian sampler can be found in [9]. The hardware architecture uses a Gaussian distributed array and an LFSR as a pseudo-random bit generator to generate a random index of the array. However the sampler has rather low precision and samples up to a tail bound which is small ($2s$). This results in a large statistical distance to the true discrete Gaussian distribution which invalidates worst case security proofs [4].

Our contributions In this paper we propose a hardware implementation of a discrete Gaussian sampler based on the Knuth-Yao algorithm [11]. To the best of our knowledge, this is the first hardware implementation of Knuth-Yao sampling. The implementation targets sampling from discrete Gaussian distributions with small standard deviations that are typically used in LWE encryption systems [18, 12]. The proposed hardware architecture for the sampler has high precision and large tail-bound to keep the statistical distance below 2^{-90} to the true Gaussian distribution for the LWE cryptosystem parameter set [9]. Furthermore, this paper proposes the following optimizations which are novel:

1. An implementation of the discrete distribution generating (DDG) tree [11] data structure at run time in hardware is challenging and costly. We use specific properties of the DDG tree to devise a simpler approach to traverse the DDG tree at run time using only the relative distance between the intermediate nodes.
2. The Knuth-Yao sampling algorithm assembles the binary expansions of the probabilities of the sample points in a bit matrix. How this bit matrix is stored in ROM greatly influences the performance of the sampling operation. Unlike the conventional row-wise approach, we propose a column-wise method resulting in much faster sampling.
3. Unlike the block variant of the Knuth-Yao method in [7], we perform column-wise compression of the zeros present in the probability matrix due to the ROM specific storage style. A *one-step* compression method is proposed which results in a near-optimal space requirement for the probabilities.

The remainder of the paper is organized as follows: Section 2 has a brief mathematical background. Implementation strategies for the Knuth-Yao sampler architecture are described in Section 3. The hardware architecture for the discrete Gaussian sampler is presented in Section 4 and experimental results are given in Section 5. Finally, Section 6 has the conclusion.

2 Background

Here we recall the mathematical background required to understand the paper.

2.1 Discrete Gaussian Distribution

The continuous Gaussian distribution with standard deviation $\sigma > 0$ and mean $c \in \mathbb{R}$ is defined as follows: let E be a random variable on $\mathbb{R}$, then for $x \in \mathbb{R}$ we have $Pr(E = x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-c)^2/2\sigma^2}$. The discrete version of the Gaussian distribution over $\mathbb{Z}$ with mean 0 and standard deviation $\sigma > 0$ is denoted by $D_{\mathbb{Z},\sigma}$ and is defined as follows: let E be a random variable on $\mathbb{Z}$, then

$$Pr(E = z) = \frac{1}{S} e^{-z^2/2\sigma^2} \quad \text{where } S = 1 + 2\sum_{z=1}^{\infty} e^{-z^2/2\sigma^2}$$

Here S is a normalization factor and is approximately $\sigma\sqrt{2\pi}$. Some authors use a slightly different normalization and define $Pr(E = z)$ proportional to $e^{-\pi z^2/s^2}$. Here $s > 0$ is called the parameter of the distribution and is related to the standard deviation σ by $s = \sigma\sqrt{2\pi}$.

The discrete Gaussian distribution $D_{L,\sigma}$ over a lattice L with standard deviation $\sigma > 0$ assigns a probability proportional to $e^{-|\mathbf{x}|^2/2\sigma^2}$ to each element $\mathbf{x} \in L$. Specifically when $L = \mathbb{Z}^m$, the discrete Gaussian distribution is the product distribution of m independent copies of $D_{\mathbb{Z},\sigma}$.

2.2 Tail Bound of the Discrete Gaussian Distribution

The tail of the Gaussian distribution is infinitely long and cannot be covered by any sampling algorithm. Indeed we need to sample up to a bound known as the *tail bound*. 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. As per Lemma 4.4 in [13], for any $c > 1$ the probability of sampling $\mathbf{v}$ from $D_{\mathbb{Z}^m,\sigma}$ satisfies the following inequality.

$$Pr(|\mathbf{v}| > c\sigma\sqrt{m}) < c^m e^{\frac{m}{2}(1-c^2)} \tag{1}$$

2.3 Precision Bound of the Discrete Gaussian Distribution

The probabilities in a discrete Gaussian distribution have infinitely long binary representations and hence no algorithm can sample according to a true discrete Gaussian distribution. Secure applications require sampling with high precision to maintain negligible statistical distance from actual distribution. Let ρ_z denote the true probability of sampling $z \in \mathbb{Z}$ according to the distribution $D_{\mathbb{Z},\sigma}$. Assume that the sampler selects z with probability p_z where $|p_z - \rho_z| < \epsilon$ for some error-constant $\epsilon > 0$. Let $\tilde{D}_{\mathbb{Z},\sigma}$ denote the approximate discrete Gaussian distribution corresponding to the finite-precision probabilities p_z. The approximate distribution $\tilde{D}_{\mathbb{Z}^m,\sigma}$ corresponding to m independent samples from $\tilde{D}_{\mathbb{Z},\sigma}$ has the following statistical distance Δ to the true distribution $D_{\mathbb{Z}^m,\sigma}$ [7]:

$$\Delta(\tilde{D}_{\mathbb{Z}^m,\sigma}, D_{\mathbb{Z}^m,\sigma}) < 2^{-k} + 2mz_t\epsilon\,. \tag{2}$$

Here $Pr(|\mathbf{v}| > z_t : \mathbf{v} \leftarrow D_{\mathbb{Z}^m,\sigma}) < 2^{-k}$ represents the tail bound.

2.4 Sampling Methods and the Knuth-Yao Algorithm

Rejection and inversion sampling are the best known techniques to sample from a discrete Gaussian distribution [1]. However both sampling methods require a large number of random bits. On the other hand, the Knuth-Yao algorithm performs sampling from non-uniform distributions using a near-optimal number of random bits. A detailed comparative analysis of different sampling methods can be found in [7]. Since our proposed hardware architecture uses the Knuth-Yao algorithm, we mainly focus on the Knuth-Yao method.

The Knuth-Yao algorithm uses a random walk model to perform sampling using the probabilities of the sample space elements. The method is applicable for any non-uniform distribution. Let the sample space for a random variable X consist of n elements $0 \leq r \leq n-1$ with probabilities p_r. The binary expansions

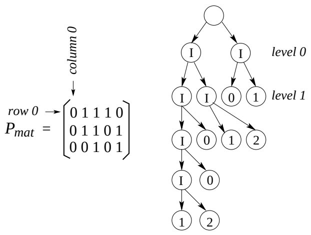


Fig. 1. Probability matrix and corresponding DDG-tree

of the probabilities are written as a matrix which we call the *probability matrix* P_{mat}. The r^{th} row of the probability matrix corresponds to the binary expansion of p_r. An example of the probability matrix for a sample space containing three sample points $\{0, 1, 2\}$ with probabilities $p_0 = 0.01110$, $p_1 = 0.01101$ and $p_2 = 0.00101$ is shown in Figure 1.

A rooted binary tree known as a discrete distribution generating (DDG) tree is constructed from the probability matrix. Each level of the DDG tree can have two types of nodes: intermediate nodes (I) and terminal nodes. The number of terminal nodes in the i^{th} level of the DDG tree is equal to the Hamming weight of i^{th} column in the probability matrix. Here we provide an example of the DDG tree construction for the given probability distribution in Figure 1. The root of the DDG tree has two children which form the 0^{th} level. Both the nodes in this level are marked with I since the 0^{th} column in P_{mat} does not contain any non-zero. These two intermediate nodes have four children in the 1^{st} level. To determine the type of the nodes, the 1^{st} column of P_{mat} is scanned from the bottom. In this column only the row numbers '1' and '0' are non-zero; hence the right-most two nodes in the 1^{st} level of the tree are marked with '1' and '0' respectively. The remaining two nodes in this level are thus marked as intermediate nodes. Similarly the next levels are also constructed. The DDG tree corresponding to P_{mat} is given in Figure 1. In any level of the DDG tree, the terminal nodes (if present) are always on the right hand side.

The sampling operation is a random walk which starts from the root; visits a left-child or a right-child of an intermediate node depending on the random input bit. The sampling process completes when the random walk hits a terminal node and the output of the sampling operation is the value of the terminal node. By construction, the Knuth-Yao random walk samples accurately from the distribution defined by the probability matrix.

The DDG tree requires $O(nk)$ space where k is the number of columns in the probability matrix [7]. This can be reduced by constructing the DDG tree at run time during the sampling process. As shown in Figure 1, the i^{th} level of the DDG tree is completely determined by the $(i-1)^{th}$ level and the i^{th} column of the probability matrix. Hence it is sufficient to store only one level of the DDG tree during the sampling operation and construct the next level (if required) using the probability matrix [11]. In fact, in the next section we introduce a novel method to traverse the DDG tree that only requires the current node and the i^{th} column of the probability matrix to derive the next node in the tree traversal.

3 Efficient Implementation of the Knuth-Yao Algorithm

In this section we propose an efficient hardware-implementation of the Knuth-Yao based discrete Gaussian sampler which samples with high precision and large tail-bound. We describe how the DDG tree can be traversed efficiently in hardware and then propose an efficient way to store the probability matrix such that it can be scanned efficiently and also requires near-optimal space. Before we describe the implementation of the sampler, we first recall the parameter set for the discrete Gaussian sampler from the LWE implementation in [9].

3.1 Parameter Sets for the Discrete Gaussian Sampler

Table 1 shows the tail bound $|z_t|$ and precision ϵ required to obtain a statistical distance of less than 2^{-90} for the Gaussian distribution parameters in Table 1 of [9]. The standard deviation σ in Table 1 is derived from the parameter s using the equation $s = \sigma\sqrt{2\pi}$. The tail bound $|z_t|$ is calculated from Equation 1 for the right-hand upper bound 2^{-100}. For a maximum statistical distance of 2^{-90} and a tail bound $|z_t|$, the required precision ϵ is calculated using Equation 2.

m	s	σ	$\lvert z_t\rvert$	ϵ
256	8.35	3.33	84	106
320	8.00	3.192	86	106
512	8.01	3.195	101	107

Table 1. Parameter sets and precisions to achieve statistical distance less than 2^{-90}

However in practice the tail bounds are quite loose for the precision values in Table 1. The probabilities are zero (upto the mentioned precision) for the sample points greater than 39 for all three distributions. Given a probability distribution, the Knuth-Yao random walk always hits a sample point when the sum of the probabilities is one [11]. However if the sum is less than one, then the random walk may not hit a terminal node in the corresponding DDG tree. Due to finite range and precision in Table 1, the sum of the discrete Gaussian probability expansions (say P_{sum}) is less than one. We take an difference $(1 - P_{sum})$ as another sample point which indicates *out of range* event. If the Knuth-Yao random walk hits this sample point, the sample value is discarded. This *out of range* event has probability less than 2^{-100} for all three distribution sets.

3.2 Construction of the DDG Tree During Sampling

During the Knuth-Yao random walk, the DDG tree is constructed at run time. The implementation of DDG tree as a binary tree data structure is an easy

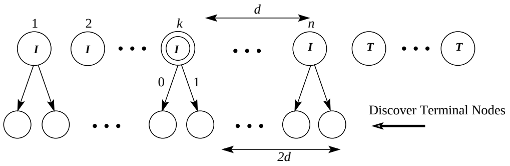


Fig. 2. DDG Tree Construction

problem [21] in software, but challenging on hardware platforms. As described in Section 2.4, the implementation of the DDG tree requires only one level of the DDG tree to be stored. However the i^{th} level of a DDG tree may contain as many as 2^i nodes (where $2^i < nk$). On software platforms, dynamic memory allocation can be used at run time to allocate sufficient memory required to store a level of the DDG tree. But in hardware, we need to design the sampler architecture for the worst case storage requirement which makes the implementation costly.

We propose a hardware implementation friendly traversal based on specific properties of the DDG tree. We observe that in a DDG tree, all the intermediate nodes are on the left hand side; while all the terminal nodes are on the right hand side. This observation is used to derive a simple algorithm which identifies the nodes in the DDG tree traversal path instead of constructing each level during the random walk. Figure 2 describes the $(i-1)^{th}$ level of the DDG tree. The intermediate nodes are I, while the terminal nodes are T. The node visited at this level during the sampling process is highlighted by the double circle in the figure. Assume that the visited node is not a terminal node. This assumption is obvious because if the visited node is a terminal node, then we do not need to construct the i^{th} level of the DDG tree. At this level, let there be n intermediate nodes and the visited node is the k^{th} node from the left. Let $d = n - k$ denote the distance of the right most intermediate node from the visited node.

In the next step, the sampling algorithm reads a random bit and visits a child node on the i^{th} level of the DDG tree. If the visited node is a left child, then it has $2d+1$ nodes to its right side. Otherwise, it will have $2d$ nodes to its right side (as shown in the figure). To determine whether the visited node is a terminal node or an intermediate node, the i^{th} column of the probability matrix is scanned. The scanning process detects the terminal nodes from the right side of the i^{th} level and the number of terminal nodes is equal to the Hamming weight h of the i^{th} column of the probability matrix. The left child is a terminal node if $h > (2d+1)$ and the right child is a terminal node if $h > 2d$. If the visited node is a terminal node, we output the corresponding row number in the probability matrix as the result of sampling process. When the visited node in the i^{th} level is internal, its visited-child in the $(i+1)^{th}$ level is checked in a similar way.

From the analysis of DDG tree construction, we see the following points :

1. The sampling process is independent of the internal nodes that are to the left of the visited node.

Algorithm 1: *Knuth-Yao Sampling*

```
    Input: Probability matrix $P$
    Output: Sample value $S$
 1  begin
 2      $d \leftarrow 0$; /* Distance between the visited and the rightmost internal node */
 3      $Hit \leftarrow 0$; /* This is 1 when the sampling process hits a terminal node */
 4      $col \leftarrow 0$;     /* Column number of the probability matrix */
 5      while $Hit = 0$ do
 6          $r \leftarrow RandomBit()$ ;
 7          $d \leftarrow 2d + \bar{r}$ ;
 8          for $row = MAXROW$ down to 0 do
 9              $d \leftarrow d - P[row][col]$ ;
10              if $d = -1$ then
11                  $S \leftarrow row$ ;
12                  $Hit \leftarrow 1$ ;
13                  $ExitForLoop()$ ;
14              end
15          end
16          $col \leftarrow col + 1$ ;
17      end
18  end
```

2. The terminal nodes on the $(i-1)^{th}$ level have no influence on the construction of the i^{th} level of the DDG tree.
3. The distance d between the right most internal node and the visited node on the $(i-1)^{th}$ level of the DDG tree is sufficient (along with the Hamming weight of the i^{th} column of the probability matrix) to determine whether the visited node on the i^{th} level is an internal node or a terminal node.

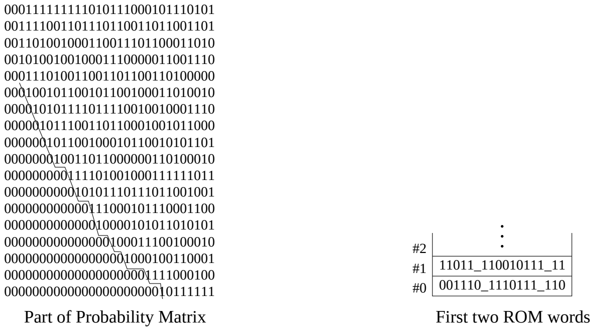
During the Knuth-Yao sampling we do not store an entire level of the DDG tree. Instead, the difference d between the visited node and the right-most intermediate node is used to construct the visited node on the next level. The steps of the Knuth-Yao sampling operation are described in Algorithm 1. In Line 6, a random bit r is used to jump to the next level of the DDG tree. On this new level, the distance between the visited node and the rightmost node is initialized to either $2d$ or $2d+1$ depending on the random bit r. In Line 8, the for-loop scans a column of the probability matrix to detect the terminal nodes. Whenever the algorithm finds a 1 in the column, it detects a terminal node. Hence, the relative distance between the visited node and the right most internal node is decreased by one (Line 9). When d is reduced to -1, the sampling algorithm hits a terminal node. Hence, in this case the sampling algorithm stops and returns the corresponding row number as the output. In the other case, when d is positive after completing the scanning of an entire column of the probability matrix, the sampling algorithm jumps to the next level of the DDG tree.

3.3 Storing the Probability Matrix Efficiently

The Knuth-Yao algorithm reads the probability matrix of the discrete Gaussian distribution during formation of the DDG tree. A probability matrix having r rows and c columns requires rc bits of storage. This storage could be significant when both r (depends on the tail-bound) and c (depends on the precision) are large. Figure 3 shows a portion of the probability matrix for the probabilities of $0 \leq |z| \leq 17$ with 30-bits precision according to the discrete Gaussian distribution with parameter $s = 8.01$. In [7] the authors observed that the leading zeros in the probability matrix can be compressed. The authors partitioned the probability matrix in different blocks having equal (or near-equal) number of leading zeros. Now for any row of the probability matrix, the conditional probability with respect to the block it belongs to is calculated and stored. In this case the conditional probability expansions do not contain a long sequence of leading zeros. The precision of the conditional probabilities is less than the precision of the absolute probabilities by roughly the number of leading zeros present in the absolute probability expansions. The sampling of [7] then applies two rounds of the Knuth-Yao algorithm: first to select a block and then to select a sample value according to the conditional probability expansions within the block.

However the authors of [7] do not give any actual implementation details. In hardware, ROM is ideal for storing a large amount of fixed data. To minimize computation time, data fetching from ROM should be minimized as much as possible. The pattern in which the probability expansions are stored in ROM determines the number of ROM accesses (thus performance) during the sampling process. During the sampling process the probability matrix is scanned column by column. Hence to ease the scanning operation, the probability expansions should be stored in a column-wise manner in ROM.

In Figure 3, the probability matrix for a discrete Gaussian distribution contains large chunks of zeros near the bottom of the columns. Since we store the probability matrix in a column-wise manner in ROM, we perform compression of zeros present in the columns. The *column length* is the length of the top portion after which the chunk of bottom zeros start. We target to optimize the storage requirement by storing only the upper portions of the columns in ROM. Since the columns have different lengths, we also store the lengths of the columns. The number of bits required to represent the length of a column can be reduced by storing only the difference in column length with respect to the previous column. In this case, the number of bits required to represent the differential



```
000111111111101011100010111010 1
001111001101110110011011001101
001101001000110011101100011010
001010010010001110000011001110
000111010011001101100110100000
000100101100101100100011010010
000010101111011110010010001110
000001011100110110001001011000
000000101100100010110010101101
000000010011011000000110100010
000000001111010010001111111011
000000000101011101110110010 01
000000000000111000101110001100
000000000000010000101011010101
000000000000000100011100100010
000000000000000001000100110001
000000000000000000001111000100
000000000000000000000010111111
```

Part of Probability Matrix

#2	⋮
#1	11011_110010111_11
#0	001110_1110111_110

First two ROM words

Fig. 3. Storing Probability Matrix

column length is the number of bits in the maximum deviation and a sign bit. For the discrete Gaussian distribution matrix shown in Figure 3, the maximum deviation is three and hence three bits are required to represent the differential column lengths. Hence the total number of bits required to store the differential column lengths of the matrix (Figure 3) is 86 (ignoring the first two columns).

For the discrete Gaussian distribution matrix, we observe that the difference between two consecutive column lengths is one for most of the columns. This observation is used to store the distribution matrix more efficiently in ROM. We consider only non-negative differences between consecutive column lengths; the length of a column either increases or remains the same with respect to its left column. When there is a decrement in the column length, the extra zeros are also considered to be part of the column to keep the column length the same as its left neighbor. In Figure 3 the dotted line is used to indicate the lengths of the columns. It can be seen that the maximum increment in the column length happens to be one between any two consecutive columns (except the initial few columns). In this representation only one bit per column is needed to indicate the difference with respect to the left neighboring column: 0 for no-increment and 1 for an increment by one. With such a representation, 28 bits are required to represent the increment of the column lengths for the matrix in Figure 3. Additionally, 8 redundant zeros are stored at the bottom of the columns due to the decrease in column length in a few columns. Thus, a total of 36 bits are stored in addition to the pruned probability matrix. There is one more advantage

Algorithm 2: *Knuth-Yao Sampling in Hardware Platform*

```
Input: Probability matrix $P$
Output: Sample value $S$
begin
    $d \leftarrow 0$;   /* Distance between the visited and the rightmost internal node */
    $Hit \leftarrow 0$;   /* This is 1 when the sampling process hits a terminal node */
    $ColLen \leftarrow INITIAL$;   /* Column length is set to the length of first column */
    $address \leftarrow 0$;   /* This variable is the address of a ROM word */
    $i \leftarrow 0$;   /* This variable points the bits in a ROM word */
    while $Hit = 0$ do
        $r \leftarrow RandomBit()$ ;
        $d \leftarrow 2d + \bar{r}$ ;
        $ColLen \leftarrow ColLen + ROM[address][i]$ ;
        for $row = ColLen - 1$ down to 0 do
            $i \leftarrow i + 1$ ;
            if $i = w$ then
                $address \leftarrow address + 1$ ;
                $i \leftarrow 0$ ;
            end
            $d \leftarrow d - ROM[row][i]$ ;
            if $d = -1$ then
                $S \leftarrow row$ ;
                $Hit \leftarrow 1$ ;
                $ExitForLoop()$ ;
            end
        end
    end
    return $(S)$
end
```

of storing the probability matrix in this way in that we can use a simple binary counter to represent the length of the columns. The binary counter increments by one or remains the same depending on the column-length increment bit.

In ROM, we only store the portion of a column above the partition-line in Figure 3 along with the column length difference bit. The column-length difference bit is kept at the beginning and then the column is kept in reverse order (bottom-to-top). As the Knuth-Yao algorithm scans a column from bottom to top, the column is stored in reverse order. Figure 3 shows how the columns are stored in the first two ROM words (word size 16 bits). During the sampling process, a variable is used to keep track of the column-lengths. This variable is initialized to the length of the first non-zero column. For the probability matrix in Figure 3, the initialization value is 5 instead of 4 as the length of the next column is 6. Whilst scanning a new column, this variable is either incremented (starting bit 1) or kept the same (starting bit 0). Algorithm 2 summarizes the steps when a ROM of word size w is used as a storage for the probability matrix.

4 Hardware Architecture

Figure 4 shows the different components of the hardware architecture for the Knuth-Yao sampling. The ROM block has word size 32 bits and is used to store the probability matrix as described in Section 3.3. Addressing of the ROM words is done using a ROM-Address counter. Initially the counter is cleared and later incremented by one to fetch data from higher ROM locations.

The scanning operation is performed using the 32-bit register present in the *Scan ROM Word* block. First a word is fetched from the ROM and then stored in the scan register. The scan register is a left-shift register and the MSB of the register is used by the *Control Unit*. A 5-bit counter (Word-Bit) is used to count the number of bits scanned from a ROM word. When all 32 bits are read from a ROM word, the counter reaches the value 31. This event triggers data reloading from next ROM word into the scan register.

Random (or pseudo random) bits are required during the traversal of the DDG tree. We have used a true random bit generator based on the approach by Golic [8]. The quality of the random bit generator is not the main focus of this paper; the random bit generator can be replaced by any other true random bit generator [2, 20] or pseudo-random bit generators based on LFSRs. The random bit generator can be slow since only five random bits are required on average during sampling from the distributions in Table 1.

An up-counter *Column Length* is used to store the lengths of the different columns of the probability matrix. This counter is first initialized to the length of the first non-zero column of the probability matrix. During the random walk, the counter increments or remains the same depending on the column-length bit. To count the number of rows during a column scanning operation, one down counter *Row Number* is used. At the start of the column-scanning, this counter is initialized to the length of that column; later the counter decrements. A column scanning is completed when the *Row Number* counter reaches zero.

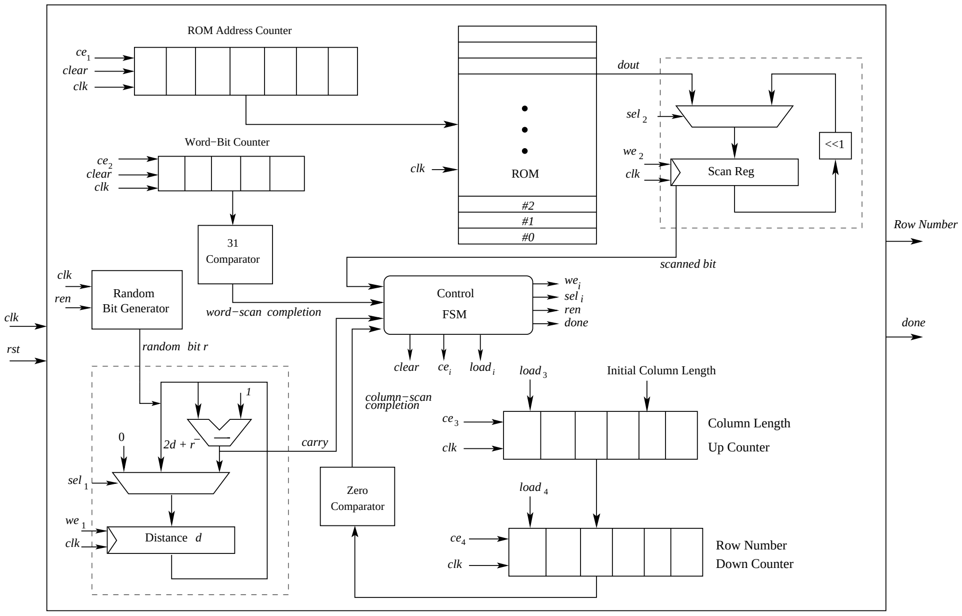


Fig. 4. Hardware Architecture for Knuth-Yao Sampler

During construction of any level of the DDG tree, the relative position d of the right most intermediate node with respect to the visited node is kept in the register *Distance*. During the Knuth-Yao random walk, the *Distance* register is first updated to $2d$ or $2d + 1$ according to the input random bit. Later each detection of a terminal node by the scanning operation decrements the register by one. A subtracter is used for this purpose. The carry from the subtracter is an input to the control FSM. When the carry flag is set ($d < 0$), the control FSM stops the random walk and indicates completion of the sampling operation. After completion, the value present in *Row Number* is the magnitude of the sample output. One random bit is used as a sign of the value of the sample output.

The hardware architecture is driven by the control FSM. The FSM generates the selection signals for the multiplexers, the write enable signals for the registers and the enable/clear/load signals for the counters present in the architecture.

Speeding up the sampling operation The sampling operation spends most time in scanning columns for which we propose two possible improvements.

1. Skipping Unnecessary Column Scanning The sampling operation hits a terminal node when the initial value of the distance d in that level is smaller than the Hamming weight of the respective column in the probability matrix (Algorithm 1). The initial columns which have smaller Hamming weight than d can thus be skipped; scanning is performed only for the first column that has larger Hamming weight than d. As such, unnecessary column scanning can be avoided by storing the Hamming weights of the columns.

There are two issues that make this strategy costly in terms of area. Firstly, extra memory is required to store the Hamming weights of the columns. Secondly, the shifting mechanism (to skip a column) for the *Scan Reg* (Figure 4) becomes complicated due to the varying lengths of the columns. This also increases the size of the multiplexer for the *Scan Reg*. Since the scan register is 32 bits wide, the area overhead is significant with respect to the overall area.

2. Window-based Scanning of Columns In hardware we can scan and process several bits of a column in a single clock cycle. Using a window-based scanning block, we can therefore reduce the computation time nearly by a factor equal to the size of the window. We can implement the window-based scanning operation by performing a minor modification in the sampler architecture shown in Figure 4. The modifications required for window size four are shown in Figure 5. The first four bits (bit[3] to bit[0]) near the MSB of the *Scan Reg* are scanned simultaneously and shift operations are always performed by four bits. During a column scanning operation, the register *Distance* is decremented by the sum of the bits. The register *Row Number* is decremented by four. However fast-scanning is affected when the following events occur: 1) $carry\#d$ is set, 2) $carry\#row$ is set, and 3) the wire *rowin* is zero. When only event 1 occurs, the sampling algorithm hits a terminal node and generates a sample value. Event 2 occurs when the four bits are from two columns: the end bits of the present column and the starting bits of the next column. For such events (1 or 2 or both), the control FSM suspends any register write operation and jumps to a *slow-scan* state. Now the FSM scans the bits $b[i]$ sequentially (similar to the architecture in Figure 4) using the selection signal sel_4. Operations in this phase are similar to the basic bit-by-bit scanning method described previously. Event 3 indicates that the scanned four bits are actually the last four bits of the column. In this case, the FSM updates the registers and then performs slow-scanning for the next four bits (the first bit is the column length change bit for the new column).

The window-based scanning requires a few extra multiplexers as shown in Figure 5 compared to the bit-by-bit method in Figure 4. However the *Word-Bit* counter size reduces to three as one scanning operation for 32 bits requires eight shifting operations. The respective comparator circuit size also reduces. Since the multiplexers are small in size (1 and 3 bits wide), the strategy has very small area overhead and is thus more cost effective compared to the Skip-Column method.

5 Experimental Results

We have evaluated the proposed discrete Gaussian sampler architectures on Xilinx Virtex V FPGAs for the distribution parameter sets given in Table 1. The results are obtained from the ISEv11.1 tool after place and route analysis. Since

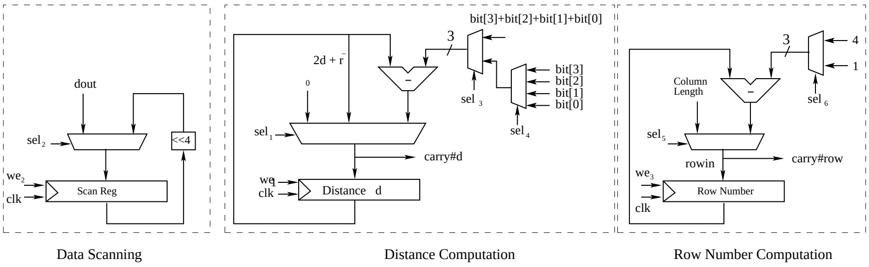


Fig. 5. Modifications required to perform 4-window scanning operation

the parameter sets in Table 1 have similar standard deviations, the same architecture is used to implement all the samplers; only the ROM contents are different. In case of a given Gaussian distribution parameter, the width of the counters, registers and arithmetic circuits can be pre-determined. Table 2 shows the width of the registers present in the proposed sampler architectures.

ROM address	Word-Bit	Scan-Reg	Column Length	Row Number	Distance
7	5 or 3	32	6	6	6

Table 2. Width of the components in Figure 4 for the distributions in Table 1

The area, delay and average case clock cycle requirements for the two sampler architectures (Section 4) are shown in Table 3. The results do not include the random bit generator. The core of the bit-by-bit scanning sampler (excluding the ROM and the random bit generator in Figure 4) consumes only 30 slices; while the core of the 4-window based fast architecture requires 6 extra slices.

On Xilinx FPGAs, ROM can be generated using LUTs or block-RAM slices. For the parameter set in Table 1, a block ROM requires only one RAMB slice. If LUT-based ROM is used, then a 32-by-96 ROM is sufficient for the distribution in Table 1. On Virtex V FPGAs, the distributed ROM consumes 17 slices.

Architecture	FFs	Slices		LUTs		Delay (ns)	Clock
		Core	ROM	Core	ROM		Cycles
Figure 4	66	30	17	76	64	3	17
Figure 5	69	36	17	85	64	3.3	16

Table 3. Performance of the discrete Gaussian sampler on xc5vlx30

Time spent in a sampling operation is mainly the time involved in scanning the column bits of the probability matrix. The number of bits scanned during a sampling operation depends on the number of levels jumped (equal to the number of random bits consumed) by the Knuth-Yao random walk along the

DDG tree. Hence the number of bits scanned in a sampling operation increases with the number of levels jumped by random walk. However the probability of a jump from a level to its next level reduces exponentially with increase in the number of levels. Knuth and Yao showed that the expected number of random bits required (i.e the number of levels jumped) is at most two more than the entropy of the given distribution [11]. For the LWE parameter sets in Table 1, the entropy of the distributions is less than three (for $\sigma = 3.33$ the entropy is 2.9) and thus the average number of random bits required per sampling is at most five. When the Knuth-Yao random walk hits a terminal node during scanning of the fifth column of the probability matrix (Appendix A), then total number of bits scanned (column bits + column length bits) is in the range of 14 to 21.

In Appendix B, we performed a software simulation to know the average number of bits scanned. As per experimental results, on average 4.3 random bits are required and 13.5 memory-bits are scanned to generate a sample point. This experimental values support the Knuth-Yao *upper bound* for the average case. To scan the first 14 memory-bits, the bit-by-bit scanning architecture (Figure 4) consumes 17 clock cycles, while the 4-window based fast scanning architecture (Figure 5) spends 16 clock cycles. These average case clock cycle requirements for the two samplers are shown in Table 3.

The number of clock cycles saved by the fast architecture compared to the basic architecture is only 6% in average case. This is due to frequent *slow-scanning* operations for the initial columns which have small lengths. The savings of the fast architecture increases with the number of levels jumped by the Knuth-Yao random walk increases. For example, when sample point is found during scanning of the 7^{th} column of the probability matrix, the fast architecture takes only 24 cycles compared to 43 cycles (44% saving) required by the basic architecture. Thus the fast architecture provides drastic speedup when the Knuth-Yao random walk goes beyond the average case.

Performance of the Sampler in ring-LWE Here we present an estimated performance analysis for the proposed sampling architectures in a ring-LWE encryption system [12]. In ring-LWE encryption, the major computations are: 1) two polynomial multiplications and 2) construction of three error polynomials using discrete Gaussian sampling. A feasible solution to implement a high-performance ring-LWE cryptosystem is to keep the multiplier and the sampler in a pipeline; the sampler stores sampled values in a buffer and the multiplier reads the buffer. Due to small delay, the sampler architecture can be integrated easily with a high-frequency polynomial multiplier.

The proposed 4-window based sampling architecture requires around 12,300 and 24,600 clock cycles on average to compute the three error polynomials [12] for the LWE parameters $m = 256$ and $m = 512$ (Table 1) respectively. The two polynomial multiplications using the NTT-based multipliers [16] require 4,800 and 10,000 clock cycles for $m = 256$ and $m = 512$ respectively. Thus the sampler architecture is slower compared to the polynomial multipliers in [16]. However, we note that our implementation is optimized for area and not speed. Since the the sampler has a very small area (compared to the multiplier) and requires random bits only occasionally, we can simply parallelize the sampling operations. In such a parallel implementation, the ROM and the random bit generator will be shared by the parallel sampling cores.

6 Conclusion

In this paper we showed that for small standard deviation, high precision discrete Gaussian samplers can be implemented in hardware using an adaptation of the Knuth-Yao algorithm. We introduced a hardware implementation friendly strategy to traverse the DDG tree in the Knuth-Yao sampling operation and proposed an optimization technique to reduce the space required to store the probabilities of the sample points in the discrete Gaussian distribution. Finally, we presented efficient hardware architectures for the discrete Gaussian distribution samplers used in LWE encryption systems. The proposed sampler architectures are small, fast and have very high precision to obtain a negligible statistical distance to a true discrete Gaussian distributions.

Acknowledgment

This work was supported in part by the Research Council KU Leuven: TENSE (GOA/11/007), by iMinds, by the Flemish Government, FWO G.0550.12N and by the Hercules Foundation AKUL/11/19. We are thankful to Thomas Pöppelmann and Tim Güneysu for their suggestions to improve the quality of our paper.

References

1. L. Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, New York, 1986.
2. M. Dichtl and J. D. Golic. High-Speed True Random Number Generation with Logic Gates Only. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *LNCS*, pages 45–62. Springer Berlin, 2007.
3. L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice Signatures and Bimodal Gaussians. Cryptology ePrint Archive, Report 2013/383, 2013. http://eprint.iacr.org/.
4. L. Ducas and P. Q. Nguyen. Faster Gaussian Lattice Sampling Using Lazy Floating-Point Arithmetic. In *Advances in Cryptology ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 415–432. Springer Berlin, 2012.
5. T. Frederiksen. A Practical Implementation of Regev's LWE-based Cryptosystem. In *http://daimi.au.dk/ jot2re/lwe/resources/*, 2010.
6. G. Zhang, P. Leong et al. Ziggurat-based Hardware Gaussian Random Number Generator. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2005)*, pages 275–280, 2005.
7. S. D. Galbraith and N. C. Dwarakanath. Effcient Sampling from Discrete Gaussians for Lattice-based Cryptography on a Constrained Device. *Preprint.*

8. J. D. Golic. New Methods for Digital Generation and Postprocessing of Random Data. *Computers, IEEE Transactions on*, 55(10):1217–1229, 2006.
9. 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 *Cryptographic Hardware and Embedded Systems CHES 2012*, volume 7428 of *LNCS*, pages 512–529. Springer Berlin, 2012.
10. H. M. Edrees, B. Cheung, M. Sandora et al. Hardware-Optimized Ziggurat Algorithm for High-Speed Gaussian Random Number Generators. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 254–260, 2009.
11. D. E. Knuth and A. C. Yao. The Complexity of Non-Uniform Random Number Generation. *Algorithms and Complexity*, pages 357–428, 1976.
12. R. Lindner and C. Peikert. Better Key Sizes (and Attacks) for LWE-based Encryption. *CT-RSA 2011*, pages 319–339, 2011.
13. V. Lyubashevsky. Lattice Signatures without Trapdoors. In *Proceedings of the 31st Annual international conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'12, pages 738–755, Berlin, 2012. Springer-Verlag.
14. D. Micciancio. *Lattices in Cryptography and Cryptanalysis.* 2002.
15. P. Q. Nguyen and J. Stern. The Two Faces of Lattices in Cryptology. In *Cryptography and Lattices, International Conference (CaLC 2001)*, volume 2146 of *LNCS*, pages 146–180. Springer-Verlag, Berlin, 2001.
16. T. Pöppelmann and T. Güneysu. Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware. In A. Hevia and G. Neven, editors, *Progress in Cryptology LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 139–158. Springer Berlin, 2012.
17. O. Regev. Quantum Computation and Lattice Problems. *SIAM J. Comput.*, 33(3):738–760, Mar. 2004.
18. O. Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.
19. O. Regev. Lattice-Based Cryptography. In C. Dwork, editor, *Advances in Cryptology - CRYPTO 2006*, volume 4117 of *LNCS*, pages 131–141. Springer Berlin, 2006.
20. D. Schellekens, B. Preneel, and I. Verbauwhede. FPGA Vendor Agnostic True Random Number Generator. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, 2006.
21. R. L. R. Thomas H. Cormen, Charles E. Leiserson and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press, Cambridge, Massachusetts, 2001.

Appendix A

Probability matrix for the discrete Gaussian distribution with parameter $s = 8.01$ used in the LWE crypto system of dimension $n = 512$ is shown in Figure 6. The portion above the partition line is stored in ROM.

Appendix B

To know the average number of random bits required per sampling operation, we have performed a C-program simulation of the Knuth-Yao random walk for the

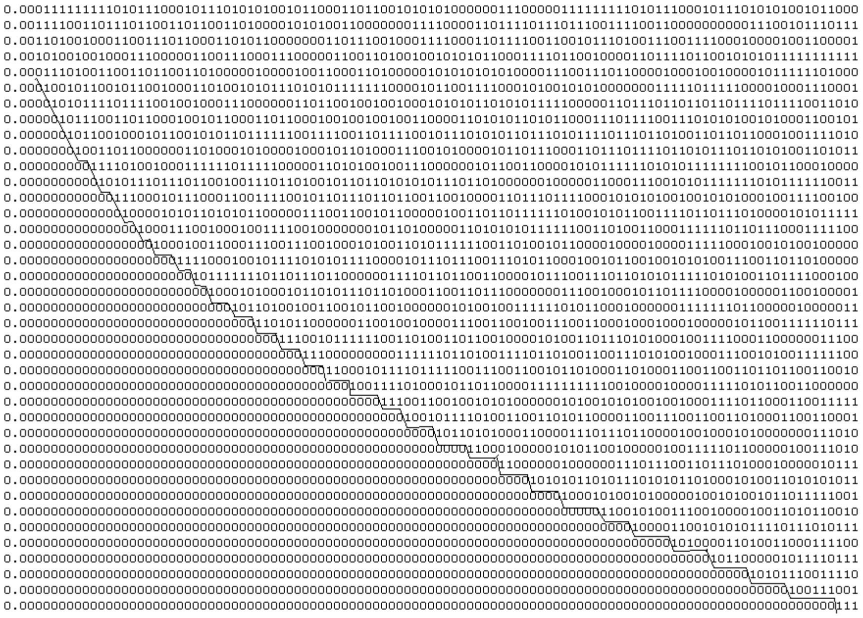


Fig. 6. Probability Matrix for the discrete Gaussian distribution with $s = 8.01$

distribution parameter $s = 8.01$. Column 1 and 2 in Table 4 shows the number of random bits required per sampling operation and the corresponding number of events in total 10^6 runs of the random walk. As per the experimental data in Table 4, on average 4.3 random bits are consumed and 13.5 bits are scanned from memory to sample a value from the discrete Gaussian distribution.

Table 4. Number of random bits required per sampling operation in 10^6 runs

Number of Random Bits	Occurrences
3	375097
4	312928
5	156405
6	77969
7	31093
8	19367
9	13510
10	6827
11	3380
12	1445
13	990
14	495
15	242
16	149
17	59
18	21
19	12
20	5
21	3
22	2
23	1