Low-area ROM-Centric Hardware Implementation of the FALCON Discrete Half-Gaussian Sampler

Alexandre Ortega, Brice Colombier and Lilian Bossuet
Université Jean Monnet Saint-Etienne, CNRS, Institut d Optique Graduate School, Laboratoire Hubert Curien UMR 5516, F-42023, SAINT-ETIENNE, France
{alexandre.ortega; b.colombier; lilian.bossuet}@univ-st-etienne.fr

***Abstract*—Preparing for the upcoming quantum era and the threat that quantum computers pose to classical cryptosystems, NIST has standardized FALCON, along with two other schemes, for post-quantum digital signature schemes. As part of its key generation and signature generation procedures, FALCON requires discrete Gaussian sampling. Constant-time cumulative distribution table (CDT) half-Gaussian sampling has been at the core of FALCON implementations since its adoption in the reference implementation. However, research has been conducted to explore alternatives to this sampler. Among them, constant-time Knuth-Yao sampling has been brought up as a potentially more efficient solution. In this article, we describe a novel method to implement the Knuth-Yao sampler in hardware, which is particularly suited for FPGA targets. The proposed RTL implementation is ROM-centric and, fitting in this cryptographic context, is constant-time. This work achieves the smallest reported area being at least 60 % smaller than the smallest implementation in the literature. It enables integration of multiple samplers alongside other operators such as NTT in area-constrained accelerators. The proposed design is open-source.**

***Index Terms*—Post-quantum cryptography, FALCON, Discrete Gaussian Sampling, Hardware, FPGA**

I. Introduction

The security of current public key cryptographic schemes relies mainly on integer factorization and discrete logarithm. While those problems are considered hard to solve with classic algorithms, Shor's quantum algorithm [1] can solve them efficiently but requires large-scale quantum computers. With the expectation for such computers to become available in the next decades, general computer-based quantum-proof cryptography, commonly referred to as post-quantum cryptography (PQC), is currently being developed. To prepare for the quantum era, the National Institute of Standards and Technology (NIST) announced a worldwide standardization process for PQC algorithms. As of today, NIST has standardized five algorithms: CRYSTALS-KYBER [2] as ML-KEM for key encapsulation, CRYSTALS-Dilithium [3] as ML-DSA, FALCON [4] as FN-DSA, and SPHINCS+ [5] as SLH-DSA for digital signatures, and HQC [6] as HQC-KEM for key encapsulation. HQC was added to the standard in 2025 whereas others were chosen at the end of the six-year competition in 2022.

The FALCON algorithm, being a cryptographic digital signature algorithm, consists of three procedures: the key generation, the signature generation, and the signature verification. The FALCON algorithm requires the use of discrete Gaussian sampling in both the key generation procedure and the signature generation procedure. The discrete Gaussian sampling includes a half-Gaussian sampler centered on zero. Hardware implementations of this half-Gaussian sampler [7]–[10] are typically done using cumulative distribution table (CDT) sampling [11], following the approach presented in the specification and used in the reference software implementation of the algorithm. This approach allows for constant-time implementations, small memory usage, simple logic due to the absence of complex arithmetic such as floating-point arithmetic, and high-precision discrete Gaussian sampling. In the current literature, the key generation procedure has yet to receive any full RTL implementation, and the signature generation only has one full RTL implementation tailored for high-throughput applications [10]. While high-throughput implementations of the FALCON Gaussian sampler have been studied, they are expensive in logic resources and need to cohabit in the accelerator with many other operators such as polynomial arithmetic operators and NTT operators. In area-constrained environments, often required in applications like IoT or embedded systems, such high-throughput designs may not be feasible, motivating ultra-compact sampler architectures.

When Prest *et al.* revised the CDT sampler [12] for the FALCON specification, during the NIST competition, they mentioned using Knuth-Yao sampling [13] instead of CDT sampling as one possible way to improve the efficiency of the sampler, and proposed parameters to use with it. Hardware implementations of the Knuth-Yao sampler rely on binary search in the probability matrix of the target distribution, with either the probability matrix being stored in a ROM [14]–[18], or with the binary paths being precomputed and the implementation being purely combinational Boolean functions [19]. When the distribution only has a small set of possible values but their probabilities need to be expressed with high precision, such as in the FALCON scheme, the resource utilization efficiency of FPGA implementations storing the matrix as ROM built in an FPGA memory block lowers. This is because not only the memory block capacity is heavily underused, but logic to perform the binary search over the probability matrix is also required, which increases the length of the critical path.

This works aims at minimizing the hardware footprint of Knuth-Yao sampling implementations on FPGAs, for distributions such as the one used in FALCON. The proposed design stores, in a ROM, the precomputed discrete distribution

generating (DDG) tree, instead of the probability matrix, in a way where it still only uses a single memory block at most on modern AMD and Intel FPGA architectures, and replaces the binary search logic by a counter and light control logic to perform a fixed number of read operations in the memory. Hence the critical path is shortened to almost match the maximum frequency of a memory block, allowing the design to operate at high frequencies.

Contributions: Our contributions in this paper are summarized as follows:

1) We put forward a novel approach to the hardware implementation, targeting FPGAs, of the Knuth-Yao sampler based on pure memory readings of a precomputed DDG tree. This approach allows for minimal logic usage: only a single ROM, a counter, and light control logic are needed excluding the random number generator.
2) We describe an open-source, constant-time, and very low-area RTL implementation of the half-Gaussian sampler used in the FALCON post-quantum algorithm.
3) We present variants of the proposed architecture in order to show possible design trade-offs. As such, we show how to adapt the proposed architecture to use 1, 2, 3 and 4 random bits at each clock cycle, increasing the throughput by the same factor.

In this paper, Section II presents the state of the art regarding FALCON and more precisely the half-Gaussian sampler centered on zero used in it, as well as the Knuth-Yao sampler. Then, Section III details the design choices made for the proposed architecture. Furthermore, Section IV explores the design space further and shares alternatives to the proposed architecture for three different configurations. Afterwards, Section V gives the proposed implementations results and compares them with related works. Eventually, the conclusion is in Section VI.

Reproducibility: To allow the reproducibility of our results, the proposed design is made open-source[1].

II. State of the art

A. FALCON

FALCON is a lattice-based digital signature algorithm based on the GPV framework [20] for creating hash-and-sign lattice-based signature schemes that are secure in the classical and quantum oracle models [20], [21]. This framework requires a class of cryptographic lattices and a trapdoor sampler. For the FALCON algorithm, the chosen class of cryptographic lattices is NTRU lattices [22], and the chosen algorithm for the trapdoor sampler is the Fast Fourier Sampling algorithm [23]. Among FALCON three procedures, two of them, the key generation and the signature generation, require discrete Gaussian sampling [4].

As part of the key generation procedure, two polynomials, f and g are sampled using a unique discrete Gaussian distribution of parameters $(0, 1.17\sqrt{q/8192})$ with $q = 12289$ a fixed parameter of FALCON. Those two polynomials are then used to compute two other polynomials, F and G, such that the NTRU equation is satisfied [4]. The public key is a basis for a lattice of dimension $2n$, with $n = 512$ for the security level I or 1024 for the security level V, and is computed using f and g [4]. The private key is another basis for the same lattice made of the four generated polynomials [4]. Lastly, the private key is preprocessed into a compact format allowing fast signature generation on-the-go, the result is referred to as the FALCON tree and appended to the private key [4].

Regarding the signature generation procedure, the message to sign is first hashed with a random nonce, and then the private key is used to generate two short polynomials s_1 and s_2. The signature is s_2, and s_1 is used in the signature verification procedure [4]. Using a trapdoor sampler to generate this pair of polynomials is necessary for security, as it adds noise using discrete Gaussian distributions to (s_1, s_2), without which the signature would leak the private key [4]. Discrete Gaussian Sampling within the signature procedure uses the set of parameters (μ, σ). The mean μ is dynamically computed depending on both the message to sign and the previous values returned by the sampler [4]. The standard deviation σ is dynamically computed from the private key using the generated FALCON tree [4].

In FALCON, the discrete Gaussian sampler algorithm [4], [12] is the same for both the key generation and the signature generation procedures. To sample in an arbitrary discrete Gaussian distribution of parameters (μ, σ'), noted $D_{\mu,\sigma'}$, the following steps are performed:

1) The fractional part of μ, referred to as r, and the constant css are computed. css is computed using σ' and the constant σ_{min} which is equal to 1.277833967 and 1.298280334 for the security levels I and V respectively.
2) A discrete half-Gaussian sampler of fixed parameters $(0, \sigma_0)$, is used to sample an integer z_0. z_0 follows the distribution $D_{\mathbb{Z}^+,0,\sigma_0}$.
3) A bit b is sampled uniformly and the integer $z = (2b - 1)z_0 + b$ is computed. Hence, the integer z follows the bimodal discrete half-Gaussian distribution $G_{\mathbb{Z},0,\sigma_0}$.
4) Rejection sampling using a Bernoulli sampler is applied to verify that $P(z) \propto \frac{D_{\mu,\sigma'}(z)}{G_{\mathbb{Z},0,\sigma_0}(z)}$. If it is the case, then the sample $s = z + \lfloor \mu \rfloor$ is returned, else the entire sampling process is restarted.

The whole discrete Gaussian sampler algorithm is given in Fig. 1. The BaseSampler operation in this algorithm, at line 4 and which corresponds to the discrete half-Gaussian sampler, is given in Fig. 2. This sampler uses cumulative distribution table (CDT) sampling over the distribution χ, defined in Table I, with $\sigma_0 = 1.8205$ as the standard deviation and $\theta = 72$ as the bit-precision of the probabilities.

Having a bit-precision of θ, for the probabilities, means that the probability for each sampled value is scaled by a factor 2^θ. This is done in order to write the probabilities with integer values instead of fixed-point or floating-point values. Hence, each sampled value probability of the distribution χ is given as a 72-bit integer value in FALCON specification [4] as shown

[1]https://doi.org/10.5281/zenodo.20596529

Require: Floating-point values $\mu, \sigma' \in \mathcal{R}$ such that $\sigma' \in [\sigma_{min}, \sigma_{max}]$
Ensure: An integer $z \in \mathbb{Z}$ sampled from a distribution very close to $D_{\mathbb{Z},\mu,\sigma'}$

$r \leftarrow \mu - \lfloor \mu \rfloor$ ▷ r must be $[0, 1)$
$ccs \leftarrow \sigma_{min}/\sigma'$ ▷ css helps make the runtime independent of σ'
while (1) **do**
  $z_0 \leftarrow$ BaseSampler()
  $b \leftarrow$ UniformBits(8) & 0x1
  $z \leftarrow b + (2 \cdot b - 1)z_0$
  $x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{max}^2}$
  if (BerExp$(x, ccs) = 1$) **then**
    return $z + \lfloor \mu \rfloor$
  end if
end while

Fig. 1: SamplerZ algorithm [4]

Require: -
Ensure: An integer $z_0 \in \{0, \ldots, 18\}$ such that $z \sim \chi$ ▷ χ is a uniquely defined half-Gaussian distribution

$u \leftarrow$ UniformBits(72) ▷ Each bit is sampled uniformly
$z_0 \leftarrow 0$
for $i = 0$ to 17 **do**
  $z_0 \leftarrow z_0 + [\![u < \text{RCDT}[i]]\!]$
end for
return z_0

Fig. 2: BaseSampler algorithm [4]

TABLE I: Distribution χ: table of probabilities [4]

I	$P(X = I) \cdot 2^{72}$	I	$P(X = I) \cdot 2^{72}$
0	1,697,680,241,746,640,300,030	10	476,288,472,308,334
1	1,459,943,456,642,912,959,616	11	20,042,553,305,308
2	928,488,355,018,011,056,515	12	623,729,532,807
3	436,693,944,817,054,414,619	13	14,354,889,437
4	151,893,140,790,369,201,013	14	244,322,621
5	39,071,441,848,292,237,840	15	3,075,302
6	7,432,604,049,020,375,675	16	28,626
7	1,045,641,569,992,574,730	17	197
8	108,788,995,549,429,682	18	1
9	8,370,422,445,201,343		

in Table I.

An important property of this distribution is that $\sum_{i=0}^{18} P(i) = 1$, which ensures that sampling can be performed exactly without needing a rejection loop.

Hardware CDT implementations of the FALCON specification discrete half-Gaussian sampler have recently been reported in the literature [7]–[10]. To improve the efficiency of the discrete half-Gaussian sampler, replacing the CDT sampling by Knuth-Yao sampling, over the same distribution, has been proposed [12], [24]. Afterwards, a hardware implementation of a Knuth-Yao sampler with the proposed parameters [12], based on the use of Boolean functions to output each sample bit based on the input sequence [25], for FALCON has been proposed [19].

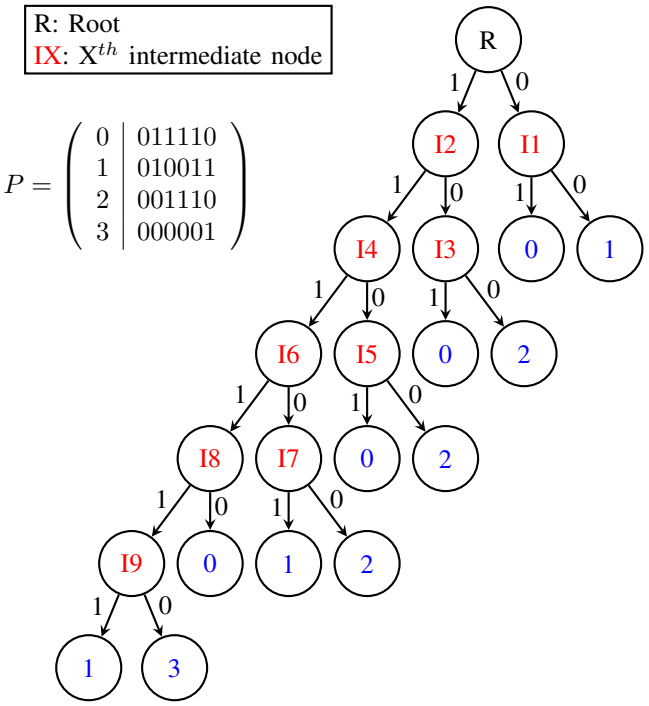

Fig. 3: An example of DDG tree with its probability matrix

B. The Knuth-Yao algorithm

The Knuth-Yao sampling algorithm [13] is a discrete sampling algorithm, for known distributions, based on random walk through a binary tree. To construct the binary tree, known as the distribution generating tree (DDG), the distribution samples probability matrix is used. The i_{th} row of the probability matrix is the probability of the i_{th} sampled value, and the probabilities are stored in binary formats after being scaled by a factor 2^{θ} in order to be represented by θ-bit integer values. Hence the probability matrix is a matrix of dimension $N \times \theta$ with each row storing the θ-bit probabilities $P(i) = 2^{\theta} \times p_i$, with $i \in \{0, \ldots, N-1\}$ and $p_i \in [0, 1)$ the probability of sampling the i_{th} value. For each level of the tree, two types of nodes are found: intermediate nodes and leaves. Starting from the root of the tree, a random bit is used at each level to traverse the tree until a leaf is reached, at which point a sample is obtained. An example of DDG tree, and the associated probability matrix, are shown in Fig. 3.

Roy *et al.* proposed using the Knuth-Yao sampling to perform sampling over discrete Gaussian distributions [14]. In their paper, they detailed a novel hardware architecture to implement the Knuth-Yao sampling algorithm. In their architecture, the binary search is performed over the probability matrix columns, as shown in Fig. 4, instead of the DDG tree itself, which is equivalent to computing the DDG tree on-the-fly. In a following paper [16], Roy *et al.* proposed optimizations to their previous hardware implementation [14].

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

Fig. 4: Knuth-Yao sampling algorithm [14]

The first optimization makes use of the fact that a leaf is reached upon encountering a one in the probability matrix. Hence, the number of leaves per level of the DDG tree is equal to its Hamming weight. Furthermore, in a given level of the DDG tree, nodes with a distance to the rightmost node of the level inferior to the Hamming Weight, of the associated probability matrix column, are leaves while others are intermediate nodes. Hence, columns with a Hamming weight inferior to the current node distance can be skipped. Another proposed optimization is to scan several columns in parallel, meaning to go through the DDG tree with several bits, instead of just one. Additionally, they also proposed using a LUT to store precomputed levels of the tree which would either return a sampled value or an intermediate node position to start the binary search from, instead of starting from the root. Dwarakanath *et al.* demonstrated in [15] that it is possible to only sample in discrete half-Gaussian instead of the full discrete Gaussian, and to use a uniformly random bit to determine the sign. They also showed that it is possible to compress the probability matrix by using the zero positions in the probability matrix. They proposed a corresponding hardware design. Finally, they theorized a new approach for implementations of the Knuth-Yao algorithm where the probability matrix is divided in blocks. Afterwards, Howe *et al.* proposed a constant-time implementation of the Knuth-Yao sampler [17]. In [25], constant-time Boolean functions were proposed to perform Knuth-Yao sampling, by making use of the unique mapping between input and output bits, to sample bits from the input bit sequence. Then, a constant-time discrete half-Gaussian sampler, targeting the FALCON scheme, was proposed by Karmakar *et al.* [24]. Based on [25], Lyons and Gaj [19] proposed a fully combinational hardware implementation of a Knuth-Yao sampler through Boolean functions, which includes results with the recommended [12] parameters for FALCON. A recent work by Baidya *et al.* [18] improved the hardware implementation of Knuth-Yao by not only reducing the LUT count by almost 29% but also achieving an almost $17\times$ speed-up.

The Knuth-Yao algorithm is considered a great candidate for hardware implementation in cryptographic settings thanks to the following properties. The Knuth-Yao algorithm entropy is at most two bits higher than the distribution entropy, meaning that it has a near-optimal entropy. Thanks to this, the DDG tree has a near-optimal number of intermediate nodes. Also, the Knuth-Yao algorithm samples in the given distribution without bias. Furthermore, the Knuth-Yao algorithm does not require complex arithmetic such as floating-point operations but only shift, comparisons, counters, and bit traversal.

III. ROM-centric Knuth-Yao implementation

This work focuses on implementing the half-Gaussian discrete sampler using the Knuth-Yao method, instead of CDT, with the recommended [12] parameters $(0, 1.8205)$ and $\theta = 72$, which corresponds to the distribution χ given in Table I. Algorithmically, lines 2 to 5 in Fig. 2 are being replaced in this work. In this Section, the rationale leading to the proposed hardware architecture will be explained. Then, an example over a simple distribution will be detailed. Finally, the hardware architecture in the case of the FALCON algorithm will be given.

A. Concept

Using the Knuth-Yao algorithm, the DDG tree is generated from the probability matrix, and then stored in a ROM. To have a constant-time implementation, the naïve solution would be to expand the DDG tree in order to have paths of equal length for all sampleable values. However, in the case of the FALCON scheme, this would mean that 2^{72} nodes would have to be stored in ROM which is neither efficient nor feasible.

A more suitable option is to index the DDG tree and for each index associate two indices, one for the right child and one for the left child, and, once a leaf is reached, to perform dummy operations repeatedly until θ random bits have been used.

To index the DDG tree, the following method is used:

1) The leaves of the DDG tree take an index equal to the sampled value they represent.
2) The root of the DDG tree takes an index equal to the number of leaves.
3) Subsequent indices are given to the intermediate nodes starting from the top-right intermediate node to the bottom-left one.

In order to map a node with its left child and right child, the index is concatenated with the random bit, with the random bit becoming the MSB, and form an address for a ROM storing the nodes indices. For a node of index I, at the address $0\&I$ of the ROM the index of its right child is stored, with $\&$

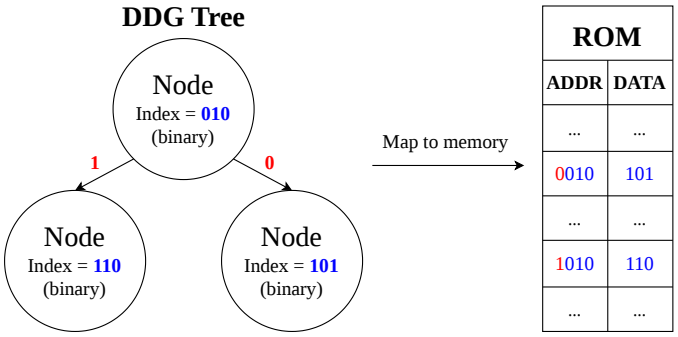

Fig. 5: Index to ROM mapping

representing a concatenation, and at the address $1\&I$ of the ROM the index of its left child is stored. This mapping process is illustrated in Fig. 5. The indices are written in binary format and colored in blue, while the random bits are colored in red. In this illustration, a DDG tree containing a parent node of index 010 with a right child of index 101 and a left child of index 110 is mapped to a ROM. At the ROM address 0010, corresponding to the concatenation of the random bit 0 and the parent node index, the data stored is 101, the index of the right child. The left child index 110 is stored at the ROM address 1010, corresponding to the concatenation between the random bit 1 and 010. A leaf has itself as both left child and right child. This means that once the index of a leaf is read from the ROM, all subsequent readings will return this index. Then, to obtain the sampled value, the output of the ROM is used since the leaf index value matches the sampled value represented by the leaf. The unique DDG tree computed using the Knuth-Yao method is well-suited for such an implementation, because its near-optimal entropy ensures a near-minimal number of intermediate nodes.

A total of θ memory readings are performed using random bits to ensure that a sampled value will always be obtained since the last column of the probability matrix, in the chosen distribution for FALCON, has ones which means that the longest path in the tree requires θ random bits. Regarding the number of uniformly random bits available per clock cycle, because the random number generator is not the topic of this work, it will be considered that the used random number generator produces at least 1 bit per cycle. This is not unrealistic, as it is possible to use performant random number generators generating blocks of uniformly random bits every X cycle, with an average of at least 1 bit per cycle, and using a FIFO to store those random bits and retrieve them.

B. Proposed half-Gaussian Knuth-Yao sampler hardware architecture

The resulting hardware architecture, illustrated in Fig. 6, consists of a ROM, to store the DDG tree, a state-machine, to control the dataflow, and two multiplexers, one to make the sampling start from the root of the tree, and the other one to only output the sample on the clock cycle when the signal ready is up. The signals include:

- A `start` input signal to begin the sampling of a value.
- A `random_bit` input signal which corresponds to the random bit to be used in the next memory reading.
- A `ready` output signal to flag the end of the sampling process and the availability of the sample. It only stays up for one clock cycle.
- A `sample` output signal which corresponds to either the sampled value, when it is available, or an idle state code. Its bit-width is 5 since there are 19 sampleable integer values from 0 to 18.

C. Example with a simple distribution

To illustrate the proposed implementation concept, an example with a simple distribution and a precision $\theta = 6$ is given. The chosen distribution respects the property $\sum_i P(X = i) = 1$ and its probability matrix is shown in Fig. 3 alongside the DDG tree generated from it.

After generating the DDG tree, the next step is to index it. In this DDG tree, there are 4 leaves, the root, and 9 intermediate nodes which means that 4 bits are needed to index the nodes. Table IIa summarizes the nodes-indices association.

Once the nodes are indexed, the next step is to build the memory from it. Since the indices are on 4 bits, and the tree traversal is done using one random bit at once, the ROM has a 5-bit address port and a 4-bit data port. Table IIb gives the ROM contents.

As an example, with the sequence of random bits 110010, the DDG tree gives us the sample 2 after the fourth bit. The consecutive memory readings, illustrated in Fig. 7, are:

1) Starting from the root, meaning the index is equal to 4, and receiving the random bit 1, the concatenation gives an address of 20. Hence the index read in the memory is 6.
2) For the second memory reading, the index is equal to 6 and the random bit is equal to 1 which gives the address 22. The index read in the memory is 8.
3) The third memory reading happens with an index of 8 and a random bit equal to 0, so the address is 8. The reading gives the index 9.
4) The fourth reading, for which the index is 9 and the random bit is 0, so an address of 9, returns an index equal to 2. 2 being a leaf, it is already known at this stage that the sampled value is 2.
5) The fifth reading takes place with index=2 and a random bit equal to 1. This means that the ROM address is 18 and the ROM output is 2.
6) For the last memory reading, the index is 2 and the random bit is 0 which gives an address of 2. Finally, the sampled value is 2. Once a leaf is reached, it is indeed the index of that very leaf that is read in the ROM until the end of the sampling.

D. Hardware architecture for the FALCON algorithm

With the recommended [12] parameters for FALCON, the computed DDG tree possesses $\theta = 72$ levels with 458 intermediate nodes and 19 leaves, so 478 nodes to index after

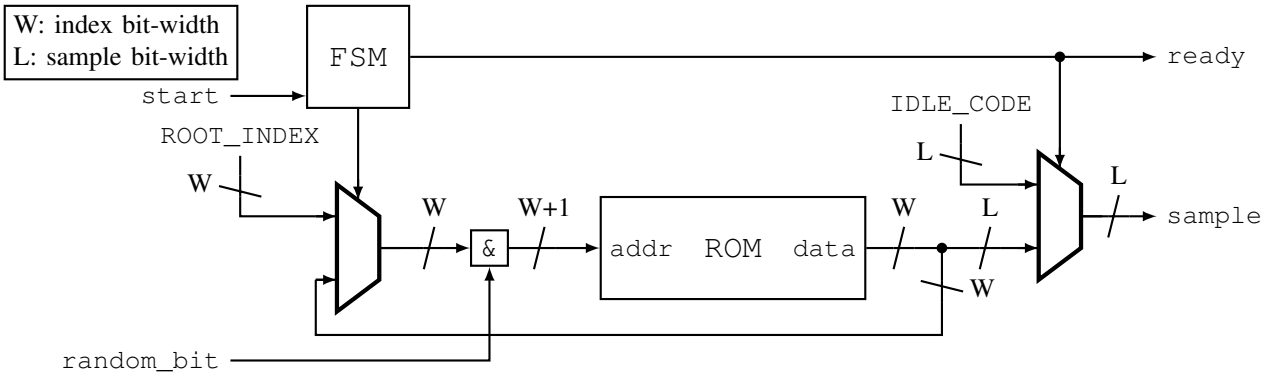

Fig. 6: Hardware architecture of the proposed half-Gaussian Knuth-Yao sampler

TABLE II: Example DDG tree node-index association and resulting ROM construction

(a) DDG tree node-index association

Tree node	Index
Leaf 0 (0)	0000
Leaf 1 (1)	0001
Leaf 2 (2)	0010
Leaf 3 (3)	0011
Root (R)	0100
Node n°1 (I1)	0101
Node n°2 (I2)	0110
Node n°3 (I3)	0111
Node n°4 (I4)	1000
Node n°5 (I5)	1001
Node n°6 (I6)	1010
Node n°7 (I7)	1011
Node n°8 (I8)	1100
Node n°9 (I9)	1101

(b) ROM contents

Address	Data_{out}
00000 (Leaf 0)	0000 (Leaf 0)
00001 (Leaf 1)	0001 (Leaf 1)
00010 (Leaf 2)	0010 (Leaf 2)
00011 (Leaf 3)	0011 (Leaf 3)
00100 (Root)	0101 (I1)
00101 (I1)	0001 (Leaf 1)
00110 (I2)	0111 (I3)
00111 (I3)	0010 (Leaf 2)
01000 (I4)	1001 (I5)
01001 (I5)	0010 (Leaf 2)
01010 (I6)	1011 (I7)
01011 (I7)	0010 (Leaf 2)
01100 (I8)	0000 (Leaf 0)
01101 (I9)	0011 (Leaf 3)
⋮	⋮
10000 (Leaf 0)	0000 (Leaf 0)
10001 (Leaf 1)	0001 (Leaf 1)
10010 (Leaf 2)	0010 (Leaf 2)
10011 (Leaf 3)	0011 (Leaf 3)
10100 (Root)	0110 (I2)
10101 (I1)	0000 (Leaf 0)
10110 (I2)	1000 (I4)
10111 (I3)	0000 (Leaf 0)
11000 (I4)	1010 (I6)
11001 (I5)	0000 (Leaf 0)
11010 (I6)	1100 (I8)
11011 (I7)	0001 (Leaf 1)
11100 (I8)	1101 (I9)
11101 (I9)	0001 (Leaf 1)

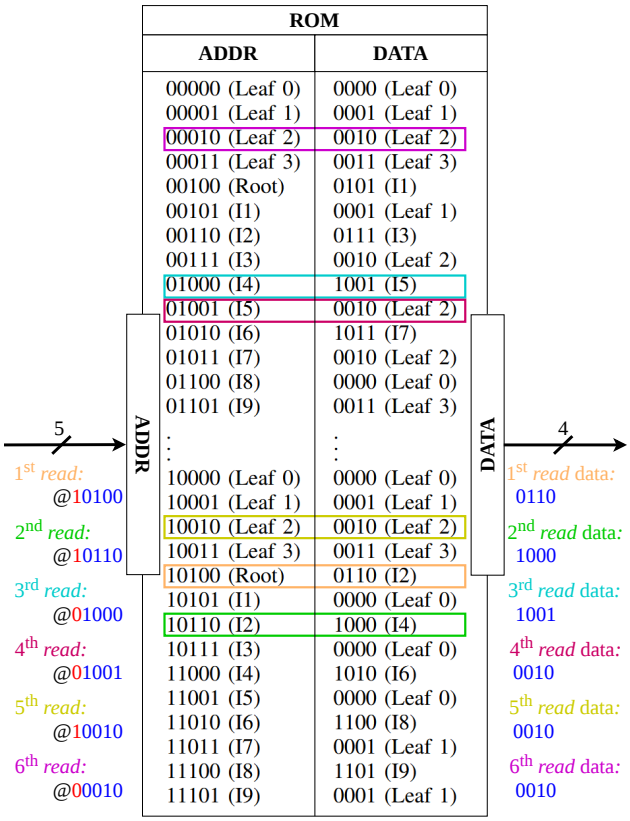

Fig. 7: Example of consecutive memory readings

counting the root as well. Therefore, the counter in the state-machine counts 72 cycles, 1 cycle per memory reading to perform. This also means that 9 bits are needed to index the tree, so the ROM has a 10-bit address bus and a 9-bit data bus. The hardware architecture, for this discrete half-Gaussian sampler, is the same as shown in Fig. 6 with $W = 9$, $L = 5$, and $\theta = 72$.

This approach yields better resource usage for FPGA implementations than the classical probability matrix approach. To store the probability matrix, storing 19 words of 72 bits is necessary, which represents a requirement of 1,368 bits of memory capacity. The proposed solution requires storing 2×478 words of 9 bits, which represents a requirement of 8,604 bits of memory capacity. However, such a reasoning does not take into account the structure of the physical memory blocks on FPGAs:

1) AMD modern FPGA families, such as 7-series, Ultrascale, Ultrascale+, and Versal ACAP, all use the same BRAM block architecture. The BRAM blocks are 36-kbit dual-port memories that can be configured as either two independent 18-kbit RAMs or a single 36-kbit RAM. A 18-kbit BRAM block can be configured

as a 2048×9-bit memory, meaning an 11-bit wide address bus and a 9-bit wide data bus, which would be enough to fit the ROM of the proposed architecture. In classical implementations, the probability matrix is typically stored in columns, corresponding to 72 words of 19 bits. An 18-kbit BRAM blocks would also be enough to store it. In the end, both solutions use the same number of BRAM block on AMD modern FPGAs.

2) Intel modern FPGA families, such as Arria V, Cyclone V, and Stratix V have two kinds of memory block that can be configured as ROM. M10K memory blocks for Arria V and Cyclone V FPGAs, and M20K memory blocks for Stratix V FPGAs. A M10K memory block, which is smaller than a M20K memory block, can be configured as a 1024×10-bit ROM which would be enough to store the DDG tree representation of the proposed architecture. It is also enough to store the probability matrix for classical implementations. Older Intel FPGAs use M9K memory blocks which can be configured as a 1024×9-bit ROM. A single one of those M9K blocks would also be enough for each of the discussed solutions. In the end, both solutions use the same number of memory blocks in Intel FPGA architectures.

We can conclude that on modern FPGAs from AMD and Intel, both solutions consume the same number of memory resources. However, the proposed architecture also benefits from using fewer logic resources, compared with classical implementations using the probability matrix, since it is purely based on consecutive memory readings without need for any logic operation. The critical path of the proposed solution is very close to that of its ROM, allowing the design to operate at high clock frequencies.

IV. Design space exploration

The proposed architecture works with one uniformly random bit per cycle. It is possible to adapt the proposed design to tackle more than one level of the DDG tree at once with multiple random bits. In this Section, solutions dealing with $\{2, 3, 4\}$ uniformly random bits per cycle are explored.

In practice, it is possible to have more than one random number generator, or to have a random number generator generating more than 1 random bit per cycle on average. In such cases, solutions where the tree traversal is performed with more than a single random bit per clock cycles can provide a higher throughput. If θ is the bit-precision used for the distribution, then the number of random bits that can be used at every clock cycle must divide θ. In the case of FALCON, this means that only solutions using $\{1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 36, 72\}$ random bits per cycle are compatible with the distribution χ. To use X bits simultaneously to go through the DDG tree, the tree is compacted by a factor X. Instead of representing every level of the DDG tree, only every X-th level from the root is kept, while intermediate levels remain implicit. Leaves that would appear in those implicit levels are represented at the next kept level. This compaction method guarantees the bitwise equivalence between the compacted DDG tree and the original DDG tree meaning that for a given set of random bits, both will give the same sample. When compacting the DDG tree, the number of intermediate node is reduced leading to lower bit-width for the index, but at the same time it increases the number of random bits to concatenate to the index. Fig. 8 shows the evolution of the memory cost of the proposed architecture depending on the number of random bits used to go through the DDG tree. From this figure, it appears that, in the case of this specific distribution, the lowest memory cost is achieved when traversing the tree with two random bits per cycle. After that point, the memory cost increase, caused by the concatenation of the random bits with the indices to form the memory addresses, is far greater than the memory cost decrease caused by the reduction of the number of nodes in the tree.

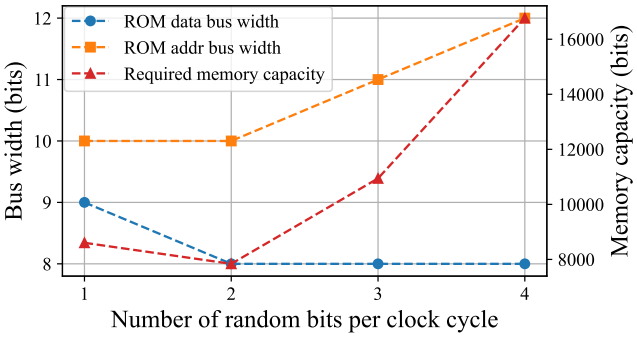

Fig. 8: Comparison of the memory cost evolution depending on the number of random bits used per clock cycle

Regarding the other changes in the hardware architecture, when increasing the number of random bits used from 1 to X, the counter counts $X\times$ less values, and the multiplexer sizes are reduced according to the bit-width of the indices. Ultimately, the combinational part of the design stays negligible.

V. Implementation results

The proposed hardware implementation is synthesized using AMD Vivado 2023.2 with the default settings. The results reported, for the proposed hardware designs, target the AMD ZCU104+ (xczu7ev-ffvc1156-2-e) FPGA from the Zynq Ultrascale+ FPGA family. The proposed architecture was described using VHDL, and verified using a custom Python testbench made with the cocotb library[2].

A. Resources and performances

In Table III, the proposed architectures' LUT and FF counts, as well as the clock cycles required per sample, decrease as the number of random bit used per cycle grows. This is due to the reduction in counter size and multiplexer sizes when more random bits per cycle are used, as explained in Section IV.

Regarding the BRAM block cost, as explained in Section III, a single half-BRAM block of 18 kbit can be configured as a

[2]https://doi.org/10.5281/zenodo.20596529

TABLE III: Hardware implementation results of the proposed architectures (AMD ZCU104+ FPGA)

RNG	Resources			Performances	
bit(s)/cycle	LUT	FF	BRAM	Cycles/sample	Freq (MHz)
1	24	9	0.5	72	568.8
2	20	8	0.5	36	568.8
3	19	7	0.5	24	570.1
4	18	7	1	18	553.7

2048×9-bit memory, which is enough to store the DDG tree when using $\{1, 2, 3\}$ random bits as shown in Fig. 8. However, Fig. 8 also shows that when using 4 random bits, a 2048×9-bit memory is no longer enough, and that configuring the 18-kbit half-BRAM block as a 4096×4 is also not enough. Hence, using a 36-kbit BRAM block configured as a 4096×9-bit memory becomes necessary. This explains why the solutions using $\{1, 2, 3\}$ random bits per cycle consume half a BRAM block, which means that they consume a 18-kbit half-BRAM block, whereas the solution using 4 random bits consumes a full BRAM block, which means that it consumes a 36-kbit BRAM block. The target FPGA, ZCU104+, is a Zynq Ultrascale+ FPGA with a speed grade of -2 which means that the BRAM block in ROM configuration has a maximum frequency of 637 MHz. All of the proposed implementations have frequencies between 550 MHz and 575 MHz, on default implementation settings, which is at 86.3 % of the BRAM block maximum frequency, and corroborates the fact that the critical paths in the proposed implementations are close to the critical path of a BRAM block.

Among the four proposed architectures, the design with 3 random bits per cycle stands out the most. It uses less resources and achieves better speed than the design with 1 and 2 random bits per cycle. And though it uses 25% more cycles than the design with 4 random bits per cycles, it consumes only one 18-kbit half-BRAM block instead of one 36-kbit BRAM block.

B. Comparison with previous works

Table IV compares the proposed 3 random bits implementation with existing works. When comparing the different reported implementation results, it is important to note that the Gaussian distribution parameters may vary from an implementation to another. Furthermore, the target platforms often differ from one implementation to another, meaning that fair comparisons are difficult to do due to the differences in technology. Regarding area, Virtex-5 FPGAs slices contain four 6-input LUTs and four flip flops, and all other FPGAs slices contain four 6-input LUTs and eight flip flops.

When comparing CDT implementations [26]–[28] with classical Knuth-Yao implementations [14], [17], Knuth-Yao designs tends to have higher LUT and FF usage, more clock cycles per sample, but also better maximum frequency, whereas CDT designs tend to rely on BRAM blocks and lower resource usage. The proposed architecture achieves the lowest resource usage compared with the literature, reducing it by at least 60 % compared to the smallest architecture in the literature. Unlike previous architectures, the proposed design is sequential, which explains the lower throughput in cycles per sample shown in the table. The proposed architecture can be easily pipelined by fragmenting the memory according to the tree levels. Furthermore, it can be parallelized at low cost thanks to its very low resource usage.

TABLE IV: Comparison of the proposed sampler with existing works

Architecture	Target	LUT / FF / Slice	BRAM	Cycles /sample	Freq (MHz)
Knuth-Yao [14]	Virtex-5 (5VLX30)	140 / 66 / 47	0	17	333
CDT [26]	Virtex-6 (6VLX75T-2)	863 / 6 / 231	0	1	61
Inverse Transform [26]	Virtex-6 (6VLX75T-2)	136 / 5 / 42	0	1	115
CDT [27]	Virtex-5 (5VLX30)	43 / 33 / 17 85 / 65 / 39	1 1	2.28 1.14	259 256
CDT [28]	Virtex-6 (6VLX75T-2)	112 / 19 / 43 53 / 17 / 15	0 1	5 5	297 193
Knuth-Yao [17]	Virtex-5 (5VLX30-3)	99 / 21 / 35	0	10	310
Knuth-Yao [29]	Virtex-6 (6VLX75T-2)	195 / 284 / 113	0	1	745
Knuth-Yao [18]	Kintex-7 (XC7K325T)	98 / 16 / 38	0	1	150
Knuth-Yao This work	**ZCU104+**	**19 / 7 / 5**	**0.5**	**24**	**570.1**

VI. Conclusion

In this paper, a novel approach to implement the Knuth-Yao algorithm on FPGAs is proposed. This approach is applied to the discrete half-Gaussian sampler of the FALCON post-quantum signature algorithm. The design is parameterized and reusable with other distributions. Optimizations, based on the number of uniformly random bits used for each traversal of the DDG tree, have been proposed. The resulting implementations achieve both lower area than previous reported works and high-frequency.

A future research perspective following up on this work is to implement the proposed architecture on an ASIC platform and evaluate its performances and area cost. Future works could also evaluate the security, of the proposed architecture, in regards to side-channel attacks and propose countermeasures. To do so, the proposed design should be integrated into a full FALCON sampler architecture with a suitable random number generator. The countermeasures put forward should be evaluated in terms of both resource and performance costs and the protection level provided.

Acknowledgment: This work received funding from the France 2030 program, managed by the French National Research Agency under grant agreement No. ANR-22-PETQ-0008 PQ-TLS

REFERENCES

[1] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, November 20-22, 1994*, pp. 124–134, IEEE Computer Society, 1994.

[2] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, G. Seiler, D. Stehlé, and J. Ding, "CRYSTALS-Kyber," tech. rep., National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/selected-algorithms.

[3] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehlé, and S. Bai, "CRYSTALS-Dilithium," tech. rep., National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/selected-algorithms.

[4] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "FALCON: Fast-Fourier Lattice-based Compact Signatures over NTRU," tech. rep., National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/selected-algorithms.

[5] A. Hülsing, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, P. Kampanakis, S. Kölbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, J.-P. Aumasson, B. Westerbaan, and W. Beullens, "SPHINCS$^+$," tech. rep., National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/selected-algorithms.

[6] P. Gaborit, C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, E. Persichetti, G. Zémor, J. Bos, A. Dion, J. Lacan, J.-M. Robert, P. Veron, P. Barreto, S. Gueron, T. Güneysu, R. Misoczki, N. Sendrier, J.-P. Tillich, V. Vasseur, S. Ghosh, and J. Richter-Brockmann, "Hamming Quasi-Cyclic (HQC)," tech. rep., National Institute of Standards and Technology, 2025. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/selected-algorithms.

[7] X. Yu, Y. Sun, Y. Zhao, H. Kuang, and J. Han, "RVCE-FAL: A RISC-V scalar-vector custom extension for faster FALCON digital signature," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2024, Valencia, Spain, March 25-27, 2024*, pp. 1–6, IEEE, 2024.

[8] E. Karabulut and A. Aysu, "A hardware-software co-design for the discrete gaussian sampling of FALCON digital signature," in *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2024, Tysons Corner, VA, USA, May 6-9, 2024*, pp. 90–100, IEEE, 2024.

[9] Y. Lee, J. M. Youn, K. Nam, H. H. Jung, M. Cho, J. Na, J. Park, S. Jeon, B. G. Kang, H. Oh, and Y. Paek, "An efficient hardware/software co-design for FALCON on low-end embedded systems," *IEEE Access*, vol. 12, pp. 57947–57958, 2024.

[10] Y. Ouyang, Y. Zhu, W. Zhu, B. Yang, Z. Zhang, H. Wang, Q. Tao, M. Zhu, S. Wei, and L. Liu, "Falconsign: An efficient and high-throughput hardware architecture for falcon signature generation," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 1, pp. 203–226, 2025.

[11] H.-C. Chen and Y. Asau, "On generating random variates from an empirical distribution," *AIIE Transactions*, vol. 6, no. 2, pp. 163–166, 1974.

[12] T. Prest, T. Ricosset, and M. Rossi, "Simple, fast and constant-time gaussian sampling over the integers for falcon," *Post-Quantum Cryptography Project of NIST*, 2019.

[13] D. E. Knuth and A. C.-C. Yao, "The complexity of nonuniform random number generation," *Algorithm and Complexity: New Directions and Recent Results*, 1976.

[14] S. S. Roy, F. Vercauteren, and I. Verbauwhede, "High precision discrete gaussian sampling on fpgas," in *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers* (T. Lange, K. E. Lauter, and P. Lisonek, eds.), vol. 8282 of *Lecture Notes in Computer Science*, pp. 383–401, Springer, 2013.

[15] N. C. Dwarakanath and S. D. Galbraith, "Sampling from discrete gaussians for lattice-based cryptography on a constrained device," *Applicable Algebra in Engineering, Communication and Computing*, vol. 25, no. 3, pp. 159–180, 2014.

[16] S. S. Roy, O. Reparaz, F. Vercauteren, and I. Verbauwhede, "Compact and side channel secure discrete gaussian sampling," *IACR Cryptology ePrint Archive*, p. 591, 2014.

[17] 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, 2018.

[18] P. Baidya, R. Paul, S. Mandal, and S. K. Debnath, "Efficient implementation of knuth yao sampler on reconfigurable hardware," *IEEE Computer Architecture Letters*, vol. 23, no. 2, pp. 195–198, 2024.

[19] M. X. Lyons and K. Gaj, "Sampling from discrete distributions in combinational hardware with application to post-quantum cryptography," in *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pp. 610–613, IEEE, 2020.

[20] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions," in *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008* (C. Dwork, ed.), pp. 197–206, ACM, 2008.

[21] D. Boneh, Ö. Dagdelen, M. Fischlin, A. Lehmann, C. Schaffner, and M. Zhandry, "Random oracles in a quantum world," in *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings* (D. H. Lee and X. Wang, eds.), vol. 7073 of *Lecture Notes in Computer Science*, pp. 41–69, Springer, 2011.

[22] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings* (J. Buhler, ed.), vol. 1423 of *Lecture Notes in Computer Science*, pp. 267–288, Springer, 1998.

[23] L. Ducas and T. Prest, "Fast fourier orthogonalization," in *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2016, Waterloo, ON, Canada, July 19-22, 2016* (S. A. Abramov, E. V. Zima, and X. Gao, eds.), pp. 191–198, ACM, 2016.

[24] A. Karmakar, S. S. Roy, F. Vercauteren, and I. Verbauwhede, "Pushing the speed limit of constant-time discrete gaussian sampling. A case study on the falcon signature scheme," in *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, p. 88, ACM, 2019.

[25] A. Karmakar, S. S. Roy, O. Reparaz, F. Vercauteren, and I. Verbauwhede, "Constant-time discrete gaussian sampling," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1561–1571, 2018.

[26] T. Pöppelmann and T. Güneysu, "Towards practical lattice-based public-key encryption on reconfigurable hardware," in *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers* (T. Lange, K. E. Lauter, and P. Lisonek, eds.), vol. 8282 of *Lecture Notes in Computer Science*, pp. 68–85, Springer, 2013.

[27] C. Du and G. Bai, "Towards efficient discrete gaussian sampling for lattice-based cryptography," in *25th International Conference on Field Programmable Logic and Applications, FPL 2015, London, United Kingdom, September 2-4, 2015*, pp. 1–6, IEEE, 2015.

[28] A. Khalid, J. Howe, C. Rafferty, and M. O'Neill, "Time-independent discrete gaussian sampling for post-quantum cryptography," in *2016 International Conference on Field-Programmable Technology, FPT 2016, Xi'an, China, December 7-9, 2016* (Y. Song, S. Wang, B. Nelson, J. Li, and Y. Peng, eds.), pp. 241–244, IEEE, 2016.

[29] L. Kong, S. Li, and R. Liu, "High-performance constant-time discrete gaussian sampling," *IEEE Transactions on Computers*, vol. 70, no. 7, pp. 1019–1033, 2021.