Merge-Exchange Sort Based Discrete Gaussian Sampler with Fixed Memory Access Pattern

Shanquan Tian, Wen Wang, Jakub Szefer
Department of Electrical Engineering, Yale University, New Haven, CT, USA
{shanquan.tian, wen.wang.ww349, jakub.szefer}@yale.edu

Abstract*—Discrete Gaussian samplers are used to sample integers from a discrete Gaussian distribution. Since this functionality is used in operations such as key generation, signing, or key encapsulation of lattice-based schemes, it is a fundamental building block of these cryptographic algorithms. One required feature of modern discrete Gaussian samplers when used in cryptographic algorithms is to be constant-time, to ensure security against timing side-channel attacks. Further, it is often desired to minimize potential for power or EM side-channel attacks by limiting how much information an attacker can gain from measuring power traces. To address the need for having a Gaussian sampler with these features in hardware, this paper presents a novel hardware implementation of a constant-time discrete Gaussian sampler with fixed memory access pattern realized on FPGAs. The design uses an approach based on Cumulative Distribution Table (CDT). Further, the new sampler uses a merge-exchange sort algorithm that enables generating the samples in batches. In the hardware, due to the use of the merge-exchange sort algorithm, the memory access pattern is always fixed, regardless of the values of the secret samples. This increases the resistance of the sampler to potential power or EM side-channel attacks as memory usage and accesses are independent of the secret values. The presented sampler can be fully parameterized at compile-time with the following Gaussian parameters: *standard deviation*, *precision*, and *tail cut*, generating a hardware design that matches the exact parameters required by the cryptographic algorithm. In addition, it can be parameterized, at compile-time, with the *batch size* for the number of samples to generate at a time. The design evaluation is based on synthesis data for various Xilinx FPGAs.

***Index Terms*—Gaussian Sampler, FPGA, Lattice-Based Cryptography, Post-Quantum Cryptography**

I. Introduction

Lattice-based cryptography, and in general post-quantum cryptography, has been recently gaining attention due to its ability to withstand attacks using quantum computers; once such computers become available. Post-quantum cryptography especially focuses on digital signature schemes and public-key encryption schemes. Among these, there are a number of lattice-based proposals. Many of the proposals are within a standardization process [1] run by the National Institute for Standards and Technology (NIST). To help NIST's process, the software and hardware performance results of these candidates are needed for understanding the practicality of the different cryptographic schemes, as well as to understand how they can be designed to eliminate the different side-channels.

Key generation, signing, or key encapsulation are core operations in lattice-based schemes, which usually require sampling integers from a Gaussian distribution. Depending on the scheme, different variants of Gaussian distributions are used, but they can be in general parameterized by the Gaussian parameters: standard deviation σ, precision λ, and tail-cut τ.

To gain understanding of the hardware demands of the Gaussian sampler for a variety of parameters (σ, λ, τ), we have developed a new Gaussian sampler in hardware on FPGAs. The sampler is based on Cumulative Distribution Table (CDT) that allows for an efficient implementation without requiring the use of floating-point operations. Further, the design uses a new approach based on merge-exchange sort [2] algorithm. Due to the merge-exchange sort approach, the implementation is constant-time, defending against timing attacks by design. Moreover, the memory access pattern generated during execution of the sampler is always fixed, and this makes our design potentially more resistant against power or EM side-channel attacks on the hardware implementation.

Since typical lattice-based cryptographic algorithms require the generation of hundreds of samples at a time, e.g., [3], [4], [5], our approach is to generate a batch of random samples at a time. This is achieved by generating uniformly random input numbers, then sorting them with the CDT table based on each entry's index (to effectively find the closet index for each random input value) and then extracting the samples from the sorted list. The cost of sorting is amortized among all samples in the batch. As different schemes use different batch size, our design is parameterized, at compile-time, with the batch size n.

Any Gaussian sampler requires generation of random numbers, and our hardware design leverages cSHAKE-256 hash algorithm for pseudo-random number generation (PRNG). The cSHAKE algorithm is a subset of the SHA-3 standard for cryptographic hashes, and is widely used in candidates in the NIST PQC standardization process. We present evaluation data for both cases of PRNG overhead included and excluded; especially as the algorithms are still being tweaked, future lattice-based algorithms may use different hash algorithms so data with PRNG excluded can help evaluate overhead of only the core sampling algorithm.

A. Contributions

The contributions of this paper include:

- A hardware implementation of a fully parameterized merge-exchange sort module, which can sort any number of values, not necessarily a power-of-two.
- A Gaussian sampler based on the CDT approach and using the merge-exchange sort module; the Gaussian sam-

pler supports compile-time Gaussian parameters: σ, λ, τ and the batch size n, which can all be freely chosen by the designers.

- A constant-time design that also has a constant memory access pattern, defending against timing side-channel attacks, and limiting potential for power or EM based side-channel attacks on the implementation.

The source code for the Gaussian sampler module is available under open-source license at https://caslab.csl.yale.edu/code/merge-exchange-gaussian-sampler.

II. Background

In this section, we present background on discrete Gaussian sampling and the different methods presented in literature. An in-depth survey of Gaussian sampling methods can be found, for example, in [6].

A. Discrete Gaussian Sampling

The centered Discrete Gaussian Distribution (DGB) over $\mathbb{Z}$ with standard deviation σ (denoted by $D_{\mathbb{Z},\sigma}$) is defined such that the probability of sampling a value $x \in \mathbb{Z}$ is given by $\rho_\sigma(x)/S_\sigma$, where $\rho_\sigma(x) = \exp(\frac{-x^2}{2\sigma^2})$ and $S_\sigma = \rho_\sigma(\mathbb{Z}) = \sum_{k=-\infty}^{\infty} \rho_\sigma(k) \approx \sqrt{2\pi}\sigma$. The specific value of the standard deviation used is usually defined by the algorithm's designers. For example in qTESLA [7] the standard deviation is mainly chosen such that the corresponding Ring Learning With Errors (R-LWE) problem [8] yields sufficient hardness.

When implementing a Discrete Gaussian Sampler (DGS), the resulting implemented distribution (denoted by $D'_{\mathbb{Z},\sigma}$) differs from the theoretical distribution due to the infinite representation of $\rho_\sigma(x)/S_\sigma$ and the infinite tail of the Gaussian distribution, while only finite values are possible in hardware (or software). Hence, the precision (λ) and the tail-cut (τ) are defined to realize and analyze a DGS. Both λ and τ are related to the security parameter κ of the cryptographic scheme that uses the DGS. The precision λ should be larger than $|p_x - p'_x|$ where p_x (resp., p'_x) is the probability of sampling x according to $D_{\mathbb{Z},\sigma}$ (resp., $D'_{\mathbb{Z},\sigma}$). In other words, the probability p'_x is represented by λ. Typically $\lambda = \kappa$ is recommended to fix the lower bound of the statistical distance of $D_{\mathbb{Z},\sigma}$ and $D'_{\mathbb{Z},\sigma}$ at about $2^{-\kappa}$. Saarinen [9], however, argues that a precision of $\lambda = \kappa/2$ is sufficient for a security level of κ bits in most cases, since there is no known algorithm able to distinguish a sampler with statistical distance $2^{-\kappa/2}$ from one with $2^{-\kappa}$. For example, in qTESLA [7], the precision λ is chosen larger than $\kappa/2$ plus a few extra bits to be more conservative, e.g., for $\kappa = 95$ a precision of $\lambda = 64$ is chosen. The tail-cut τ determines the bounded support that can be implemented, i.e., instead of sampling values over $\mathbb{Z}$, they are sampled over $\mathbb{Z} \cap [-\sigma\tau, \sigma\tau]$.

B. Sampling Techniques

There are several techniques described in the literature for implementing Gaussian samplers for lattice-based schemes. We briefly summarize the most popular options and highlight their advantages and limitations in the following paragraphs.

Rejection sampling [10] works by first sampling x in $[-\sigma\tau, \sigma\tau] \cap \mathbb{Z}$ and y in $[0, 1)$ uniformly at random. If $y \leq \rho_\sigma(x)$, x is accepted as a valid output. Otherwise, x is rejected and the process is repeated again with a new pair (x, y). The repeated need for generating the values $\rho_\sigma(x)$ and a potentially high rejection rate in this method are disadvantageous: Computing $\rho_\sigma(x)$ during run-time can be very expensive in hardware; using a table of pre-computed values $\rho_\sigma(x)\ \forall x \in [-\sigma\tau, \sigma\tau] \cap \mathbb{Z}$, on the other hand, may result in increase of required memory.

An optimization of the basic rejection sampling technique is *Bernoulli sampling* [11]. Essentially, the idea of Bernoulli sampling approach is to combine uniform sampling over $\mathrm{U}(\{0, \ldots, \xi - 1\})$ with the distribution $\xi \cdot D_{\mathbb{Z}^+,\sigma_2}$ for a width parameter ξ. The desired distribution $D_{\mathbb{Z},\sigma}$ with $\sigma = \xi\sigma_2$ is then obtained by rejection sampling guided by the so-called Bernoulli distributions $\mathcal{B}_c$ [11]. This method reduces drastically the rejection rate but is hard to be implemented in constant-time in both hardware and software.

Another popular sampling method is the *Knuth-Yao* algorithm [12] which is based on a binary tree called the discrete distribution generating (DDG) tree. The DDG tree is constructed from the binary representation of p'_z. Sampling a value with DGD is achieved by a sequence of uniformly sampling over $\{0, 1\}$ to decide on the path through the tree until a leaf is reached. Due to the heights of different leaves, sampling using this approach is highly non-constant time, and hence, it requires countermeasures against timing attacks, e.g., the Fisher-Yates shuffle [13] is used in [14], [15] to randomly swap all the values in every block of the generated samples.

Yet another efficient method to realize Gaussian sampling is *cumulative distribution sampling*, first proposed for cryptographic applications by Peikert [16]. This method consists of pre-computing a so-called Cumulative Distribution Table (CDT) containing values of the Cumulative Distribution Function (CDF) scaled to the range $[0, 2^\lambda)$ for a given precision λ. A sample of the DGD is derived as follows: First a value x is chosen uniformly at random in the range $[0, 2^\lambda)$. Then the index z in the CDT is found such that $\text{CDT}[z] \leq x < \text{CDT}[z + 1]$. The CDT-based method is one of the most efficient and flexible methods in the literature. It is portable and can be combined with other techniques for improved performance. CDT based samplers are for example used in qTESLA [7] and FrodoKEM [4] – two candidates of NIST's PQC standardization process [1]. In addition, the applicability of the CDT-based approach to other lattice-based schemes was recently shown by [17]. The efficiency of this approach can be limited for larger standard deviations σ due to the size of the CDT. However, if large σ is required, Peikert's convolution lemma [18] can be applied which transforms the problem to equivalently building two samplers with much smaller σ_1 and σ_2. Therefore, multiple CDT-based samplers with smaller standard deviations can be used as building blocks to construct samplers requiring a big standard deviation [15].

The CDT based samplers are greatly affected by the size of the CDT. Typically, the width of the CDT is determined by the sampling precision λ, which determines how many bits are needed in total to represent one sample. Meanwhile, the depth of the CDT is determined by the standard deviation σ and the tail-cut τ, since $\sigma\tau$ determines the range of the samples. Note, since the Gaussian distribution used in Gaussian sampling is symmetric in the x axis, only values in range $[0, \sigma\tau]$ are need to be sampled, and a random sign $\{-1, 1\}$ can be generated to get samples from the full range $[-\sigma\tau, \sigma\tau]$.

C. Problems in Existing CDT Samplers

Commonly used CDT samplers usually work as follows: A random number is first generated, then it is compared against the CDT in order to extract one sample after the comparison process; for generating multiple such samples, this processed is repeatedly carried out until enough samples are collected. There are two main existing approaches for scanning the CDT. A *binary search* on the CDT can be performed efficiently. However, this method is susceptible to timing side-channel attacks in software since the branching depends on the private uniform samples. To mitigate such timing-based side-channel attacks, a more conservative software method can be used: A *full scan* on the CDT is always carried out, regardless of the input random number value. However, this approach is very inefficient especially when the size of the CDT table is relatively large since a full CDT scan is always needed each time a random sample is needed.

When implemented in hardware, both *binary search* and *full scan* CDT sampling methods can be designed to be constant-time. Unfortunately, the *binary search* based hardware design of CDT sampler can be potentially vulnerable to power or EM based side-channel attacks since the memory access pattern during the CDT *binary search* is uniquely determined by the secret random samples. For example, in [19] a Single Trace Analysis (STA) based power attack was proposed and demonstrated on the implementation of a CDT-based gaussian sampler by observing the Hamming weight differences between values used in computation or memory addresses.

This vulnerability is mitigated by use of the *full scan* method, however, a limitation of a naive *full scan* based hardware design of CDT sampler is that it leads to poor performance when a large number of random samples are needed (which is the case for most of the modern lattice-based schemes): For each sample the full CDT needs to be scanned to hide the true value that was searched for. Therefore, a better solution is needed for designing a CDT sampler in hardware which mitigates secret-dependent memory accesses and at the same time maintains a good performance for generating a relatively large batch of random samples.

The solution we propose in our design to solve these issues is discussed in detail in the next section (Section III). Details of the hardware implementation of our CDT sampler are presented in Section IV. A thorough evaluation of the CDT sampler with different parameters are provided in Section V. Comparisons with the state-of-the-art software implementations and hardware implementations of the CDT sampler is given in Section VI.

Algorithm 1 CDT-based Gaussian sampling using Batcher's odd-even mergesort algorithm (adapted from [20] and modified to reflect our hardware implementation details).

```
INPUT: integer $n$ of requested random numbers, a CDT (of $t$ positive
    values of precision $\lambda$, where $t = \lceil \sigma\tau \rceil$)
OUTPUT: a list $z$ of $n$ Gaussian samples
▷ The depth of the list (memory) samp is $n + t$, where the first
  $n$ entries store random numbers (initially empty), the rest store
  CDT values. Each entry has 3 parts $s$, $k$, $g$.
▷ At compile-time, initialize part of samp with the CDT, and
  append an index to each entry to keep track of their original
  Gaussian indices:
for $0 \le i < t$ do
    samp$[i+n].s \leftarrow \infty$   // search sentinel
    samp$[i+n].k \leftarrow$ cdt$[i]$
    samp$[i+n].g \leftarrow i$
end for
▷ At run-time, Generate $n$ random values (size of each value is the
  precision $\lambda$) and fill in the list (memory) samp and keep track
  of their original sampling order:
for $0 \le i < n$ do
    samp$[i].s \leftarrow i$
    samp$[i].k \leftarrow_{\$} \{0, ..., 2^{\lambda} - 1\}$
    samp$[i].g \leftarrow 0$   // placeholder
end for
▷ Sort samp list (memory) in constant-time according to $k$ field
  using merge-exchange sort approach:
BatcherMergeExchange(samp, key:$k$, data:$s, g$)
Set each entry's Gaussian index $g$: the values of $g$ of each random
  value entry get updated with the $g$ value of the closest CDT entry.
▷ Sort samp list (memory) again in constant-time, now according
  to $s$ field. Because $s$ values for CDT are all infinity, random
  numbers will be at the top of samp, and in their original order:
BatcherMergeExchange(samp, key:$s$, data:$g$)
▷ Discard the trailing entries of samp and output the rest as
  samples with $k[0]$ determining sign bits:
for $0 \leqslant i < n$ do
    if samp$[i].k[0] > 0$ then
        $z[i] \leftarrow 1 \cdot$ samp$[i].g$
    else
        $z[i] \leftarrow -1 \cdot$ samp$[i].g$
    end if
end for
return $z$
```

III. CDT Method with Batcher's Odd-Even Mergesort Algorithm

Since typical lattice-based applications require the generation of hundreds of samples at a time, one possible way to increase the throughput during Gaussian sampling is to use the CDT method combined with a suitable sorting algorithm, as proposed by Alkim *et al.* [20]. The basic idea is to sort a batch of uniformly random samples together with the CDT, and then identify the unique CDT entry immediately below each sample in the list. The cost of sorting is thus amortized among all the samples in the batch.

Since the sorting needs to be done efficiently and in constant-time, Batcher's odd-even mergesort algorithm [21], also called merge-exchange sort [2], is used in [20] in the software implementation of the sampler in qTESLA [7]. Compared to most of the other sorting algorithms, one important feature in the merge-exchange sort algorithm is that the data comparisons during the sorting process have a fixed pattern, regardless of the values to be sorted. This feature is ideal for our hardware design since one of our main targets of the Gaussian sampler is to eliminate the potential leakages due to secret-dependent memory accesses, which could be abused in power or EM based side-channel attacks. Therefore, we follow the approach as used in the qTESLA software implementation [20] and present the first implementation of CDT sampler based on merge-exchange sort in hardware.

Let `BatcherMergeExchange`($\langle$*sequence*$\rangle$, `key`:$\langle$*key*$\rangle$, `data`:$\langle$*data*$\rangle$) be a constant-time implementation of Batcher's sorting algorithm for $\langle$*sequence*$\rangle$, using the field $\langle$*key*$\rangle$ of each of its entries for ordering, and carrying the corresponding field(s) $\langle$*data*$\rangle$ as associated data. Algorithm 1 generates n Gaussian samples using the algoritm from Alkim *et al.* [20]. In contrast to [20], the generation of the sign bit in the hardware design does not require another sampling process. Instead, when outputting samples, $\texttt{samp}[i].k[0]$ is used to determine the sign of its corresponding $\texttt{samp}[i].g$: If $\texttt{samp}[i].k[0]$ is 1, our Gaussian sampler will output $1 \cdot \texttt{samp}[i].g$, otherwise it will output $-1 \cdot \texttt{samp}[i].g$.[1]

Algorithm 1 gets as input a number n of requested random values and a CDT with t entries of width λ bits (recall the precision is λ), the entries represent the absolute values of the CDF. λ is usually $\geq \kappa/2$, as discussed in Section II-A. The algorithm outputs n values of the Gaussian samples.

Let S be a finite set, then we denote sampling an element s uniformly random in S by $s \leftarrow_{\$} S$. As shown in Algorithm 1, λ-bit precision CDT of depth t is pre-calculated and later combined with a sequence of n uniformly random values of λ-bit precision, forming a new list `samp` of $n+t$ elements. The first n entries store random numbers, and the rest store CDT values. Each element in `samp` is further expanded as a triple of form (s, k, g) where s holds a record of the entry type (index $0, 1, ..., n-1$ for random numbers and infinity for CDT), k denotes the actual value and g keeps track of the CDT index (zero for random numbers). Accessing the i-th entry in `samp` is denoted by $\texttt{samp}[i].s$, $\texttt{samp}[i].k$, and $\texttt{samp}[i].g$. Two merge-exchange sorting steps are needed: The first sort blends the n random values with the CDT in a sorted order; then the values of g of the random value entries get updated with its closest CDT entry's Gaussian index; once all the entries of `samp` have an updated Gaussian index, the second sort starts by use of s, which records the entry type (random value or CDT), as the key. Finally, the random values entries are lifted to the upper part of `samp` and the Gaussian indices carried by these entries are returned as the output.

IV. Hardware Implementation

This section presents details of the hardware implementation of our Gaussian sampler. Figure 1 shows the top-level view of the sampler. The top module contains the sampler module and the pseudo-random number generator (PRNG) module, which is used for generating random numbers.

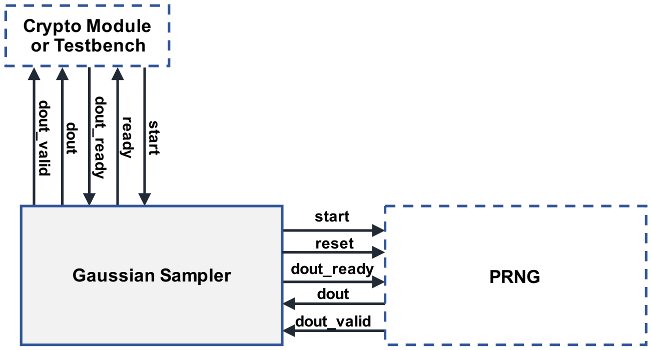

Fig. 1. High-level block diagram of the CDT-based Gaussian sampler, two key parts are the sampler itself and a module in this case cSHAKE, that is used for pseudo-random number generation (PRNG) needed by the sampler.

The top module communicates with the outside world through an AXI-like interface. There are five signals: `start` to start the sampling process, `ready` to indicate that sampled data can be read out, and `dout`, `dout_valid`, and `dout_ready` for reading samples from the module. The interface can be easily extended to, for example, AXI4, for integration with other cryptographic modules, or some of the proposed APIs for PQC algorithms. A similar interface is used internally within the module for communication between the sampler and the PRNG module.

A. CDT Sampler Design Parameters

Depending on the user needs, the proposed CDT sampler can be tuned by defining the following parameters at synthesis time: Batch size n, which determines how many samples are generated after one complete process of CDT sampling; CDT table width W_{cdt}, with $W_{cdt} = \lambda$; and CDT table depth D_{cdt}, where $D_{cdt} = \lceil \sigma\tau \rceil$.

In the following evaluation sections, we will show the effect of these parameters on the performance of the CDT sampler, while here we give a brief overview. The batch size n most affects the runtime of the PRNG, as bigger batches require pre-generation of more random numbers. Further, bigger batch size requires more time for sorting the numbers and generating the output. CDT table width W_{cdt} does not affect the runtime of the sampler as it only changes the widths of the memories, however, W_{cdt} has a major impact on the logic usage of the CDT sampler as it determines the size of the basic computation units in the design (e.g., register size, comparator size). CDT table depth D_{cdt} has similar effects as n on the performance of the CDT sampler, as the size of the list to be sorted depends on $n+t$.

B. CDT Sampler Module

The hardware design of the CDT sampler follows the Algorithm 1 as proposed in Section III. Details of the CDT sampler are shown in Figure 2. A dual-ported memory mem_{samp} is

[1] Re-use of $k[0]$ bit for determining the sign is done to reduce the required number of random numbers needed, use of the bit for both data and sign may introduce some correlation, and security of this design decision needs to be further evaluated.

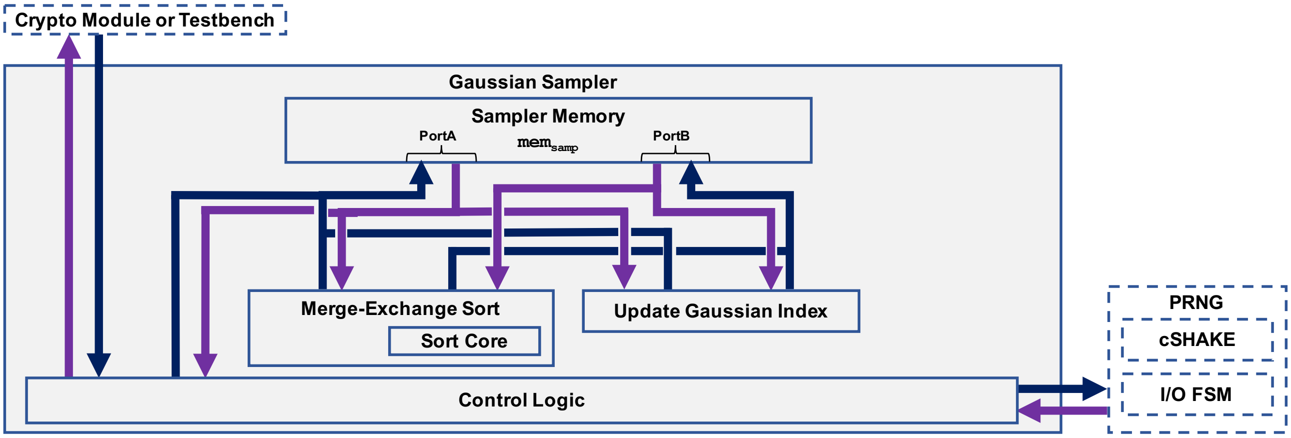

Fig. 2. Diagram showing main components of the sampler module: one memory for storing CDT table and random numbers, of which each entry is of form (s, k, g); logic for the merge-exchange sort, output generation, and control logic. Smaller elements such as MUXes or details of the signals are not shown.

needed in the CDT sampler module to store CDT table and random numbers. It has $n + t$ entries, each stores a triple (s, k, g), of which the bit widths are $(log(n) + 1)$, (λ), and $(log(t))$, respectively. The highest bit in s is reserved to distinguish infinity from regular indices. k is used to store CDT values or random numbers generated by a PRNG module, thus needs λ bits. The CDT values are pre-initialized at compile time with the CDT table defined by the lattice-based scheme.

Apart from the memory block, four main submodules are needed to build a full CDT sampler: `Control Logic` functioning as the bridge between different submodules and the memory block; a `PRNG` module is used for generating random numbers; `Merge-Exchange Sort` module for sorting random numbers following Batcher's odd-even mergesort algorithm; and `Update Gaussian Index` module for updating g values for the final output samples.

1) Control Logic: When the CDT sampler starts, the `Control Logic` is responsible for controlling generation of a batch of samples. The control logic accepts as input the size of batch n, which determines how many outputs the PRNG generates to initialize mem_{samp}.

Once mem_{samp} is fully initialized with the randomly-generated values, the control logic triggers the `Merge-Exchange Sort` module for the first time, to sort the whole memory according to k. After sorting is done, the `Update Gaussian Index` module updates g values of random numbers with their closest CDT entry's Gaussian indices. Then the `Merge-Exchange Sort` module is triggered again to sort the whole memory according to s. After that, the samples stored in the first entries of mem_{samp} can be read out through control logic once instructed by the cryptographic module (or the testbench).

2) PRNG Module: Different PRNG modules can be used in our design, including ones base on XORs or using constant values for testing. For full evaluation, we use a `PRNG` module constructed based on an area-efficient cSHAKE module proposed by Jungk and Apfelbeck [22]. cSHAKE is a subset of the SHA-3 standard and is used widely in modern cryptographic algorithms, such as the lattice-based signature scheme qTELSA [20]. When a request is made to generate a batch of samples, the first module triggered is the cSHAKE which starts generating the pseudo-random numbers. The seed and domain separator of cSHAKE module are initialized at compile time. The top module controls how many samples need to be generated, i.e. n. Since the output of cSHAKE is written in a unit of 32 bits, in total $\lambda/32 \cdot n$ outputs are generated. As showed in line 6-10 in Algorithm 1, cSHAKE is called at the beginning of Gaussian sampler operation to initialize the first n entries of the mem_{samp} memory.

3) Merge-Exchange Sort Module: The design of the `Merge-Exchange Sort` module closely follows the Batcher's odd-even mergesort algorithm [21]. This module is triggered twice, with k or s as the sorting key (line 11 and 13 in Algorithm 1). This module is built upon a core submodule `Sort Core`, which is called for a fixed number of rounds where within each round data pairs are chosen and sorted sequentially following a fixed pattern. When `Merge-Exchange Sort` module starts the computation, a local parameter set (p, q, r, d), which determines the location of the comparison pairs, is initialized and sent to `Sort Core`. The values (p, q, r, d) are initialized with: $p \leftarrow 2^{a-1}$ where a is the least integer such that $2^a \geq N$ (N is the list size, and in our work $N = n + t$), $q \leftarrow 2^{a-1}$, $r \leftarrow 0$, and $d \leftarrow p$[2]. `Sort Core` then sequentially chooses comparison pairs with indices $(i, i + d)$ from the sorting list, where $0 \leq i < N - d$ and $iANDp = r$. Once the memory contents mapping to indices $(i, i+d)$ from the list are read out, the two values are compared. Depending on the comparison result, the values are either written back to the memory in the original order, or in a reversed order. These main steps in `Sort Core` (memory reads, data comparison, memory writes) are implemented in a fully pipelined fashion. While `Sort Core` is running, the parameter set (p, q, r, d) is updated, waiting to be fed to the next round once the current computation in `Sort Core` finishes. (p, q, r, d) is updated as follows: If $q \neq p$, set $q \leftarrow q/2$, $r \leftarrow p$, $d \leftarrow q - p$; Else, set $p \leftarrow \lfloor p/2 \rfloor$, $q \leftarrow 2^{a-1}$.

n	Cyc. Sampler	Cyc. Total	LUT Sampler	FF Sampler	LUT Total	FF Total	BRAM	Time×Area per Sample	Time (μs) per Sample	Fmax (MHz)
1	4048	4331	630	681	3056	1053	3	13.2×10^6	38.1	114
2	4012	4299	624	685	3048	1057	3	6.55×10^6	19.0	113
4	4230	4525	635	695	3070	1067	3	3.47×10^6	9.69	117
8	4490	4801	647	705	3072	1077	3	1.84×10^6	5.06	119
16	4994	5337	721	715	3144	1087	3	1.05×10^6	3.01	111
32	6130	6637	655	725	3081	1097	3	639×10^3	1.86	111
64	8799	9634	702	751	3131	1123	3	471×10^3	1.39	108
128	14,315	15,806	791	760	3246	1132	3	401×10^3	1.01	123
256	27,287	30,090	866	787	3319	1159	3	390×10^3	0.991	119
512	56,929	62,256	907	812	3336	1184	3	406×10^3	1.06	115
1024	125,451	135,826	820	837	3261	1209	5	433×10^3	1.12	119
80	10,155	11,154	786	758	3187	1130	3	444×10^3	1.25	112
100	11,947	13,126	793	760	3197	1132	3	420×10^3	1.11	118
160	17,081	18,900	798	768	3206	1140	3	379×10^3	1.01	116
320	34,039	37,398	866	795	3267	1167	3	382×10^3	0.977	120
640	73,301	79,840	836	820	3260	1192	3	407×10^3	1.02	123
1000	121,643	131,722	826	837	3226	1209	5	425×10^3	1.15	115
1280	164,607	177,506	819	844	3267	1216	5	453×10^3	1.20	116
2560	376,973	402,492	833	871	3224	1243	10	507×10^3	1.38	114
5120	868,001	918,860	869	896	3280	1268	22	589×10^3	1.57	114

TABLE I

EVALUATION OF THE PERFORMANCE FOR DIFFERENT VALUES OF n. THE DATA IS FOR $(W_{cdt}, D_{cdt}) = (64, 78)$, ON ARTY A7-100T DEVELOPMENT BOARD, USING ARTIX-7 FPGA CHIP. SPECIFIC FPGA DEVICE IS XC7A100TCSG324-1. IN TIME×AREA PER SAMPLER, TIME IS IN TOTAL CYCLES, AREA IS IN TOTAL LUTS.

$r = 0$, $d = p$[2]. This process is repeated until p and q are both equal to 1.

Given the size N of the list to be sorted, in total $1/2 \times log(N) \times (log(N)+1)$ rounds are needed to finish the merge-exchange sort process. Due to the pipelined design of the Merge-Exchange Sort module, the cycles needed for one complete sorting approaches the theoretical limit: For example, with $N = 512$, in total 45 rounds, 9,727 comparisons are required, and for each comparison one cycle is needed for memory read and write respectively, this leads to a theoretical limit of 19,454 cycles; by use of our hardware design, in total 22,382 cycles are needed to sort a list of $N = 512$ elements, which is very close to the theoretical limit. As we can see from the above analysis, the memory access pattern in the Merge-Exchange Sort module are fully dependent on the parameter set (p, q, r, d) which is further decided only by the size of the list N, therefore, the memory access pattern is fixed and will not leak any secret-dependent information.

4) Update Gaussian Index Module: Between two merge-exchange sorts, the g values of random number entries are updated (line 12 in Algorithm 1). The mem_{samp} is scanned and updated from the beginning to the end, while each update operates differently based on the entry type (CDT or random values): If the entry stores a CDT value, its Gaussian index is saved in an internal register reg_g and the memory entry gets written with the same content; on the other hand, if the entry stores a random number, the module will write the same memory content with the g part updated with the value stored in reg_g to the same memory entry. This way the Update Gaussian Index module also runs in constant time with a fixed memory access pattern.

V. EVALUATION

As described in Section IV, the CDT sampler module can be flexibly configured by tuning three parameters: Batch size n, CDT table width W_{cdt}, and CDT table depth D_{cdt}. In this section, a detailed analysis on the effects of these parameters on the performance of the CDT sampler is given. The logic usage data provided in the following tables exclude the cSHAKE overhead unless pointed out specifically.

A. Sensitivity to Batch Size (n)

In modern lattice-based schemes, a large number of random gaussian-distributed samples are usually needed. For example, for qTESLA, in total 5120 random samples are needed in variant *qTESLA-p-I* and for variant *qTESLA-p-III*, 12288 are needed [7]. To generate such random samples, a simple solution would be to use our CDT sampler with batch size equals to the number of expected samples. However, this will lead to a big memory overhead for storing a large number of random numbers. A more time-area efficient solution would be to repeatedly use a smaller-sized CDT sampler until all the needed samples are collected.

Table I shows the sensitivity of the design to the batch size n. Note that n can be chosen randomly without being constrainted as a power-of-two. Further, when $n = 1$, this is equivalent to the design of *full scan* CDT approach (as discussed in Section II-C). The evaluation is based on generating n samples for *qTESLA-p-I* with security parameters $(\sigma, \lambda, \tau) = (8.5.64, 9.06)$ leading to a derived CDT of size $(W_{cdt}, D_{cdt}) = (64, 78)$. As we can see from Table I, the batch size n has a big impact on the design efficiency: As n increases starting from $n = 1$, “Time×Area per Sample”

W_{cdt}	Cyc. PRNG	Cyc. Total	LUT	FF	BRAM	Fmax (MHz)
32	999	18,100	590	543	2	113.4
64	1820	18,900	798	768	3	116.5
96	2640	19,700	988	994	4	112.7
128	3360	20,400	988	1215	4	114.6
160	4180	21,300	1102	1443	5	117.0
192	5000	22,100	1224	1663	6	117.8
224	5720	22,800	1361	1893	7	113.6
256	6540	23,600	1762	2118	8	114.5

TABLE II

EVALUATION OF THE PERFORMANCE FOR DIFFERENT VALUES OF W_{cdt} (EXCLUDING PRNG). $(n, D_{cdt}) = (160, 78)$. THE DATA IS FOR ARTY A7-100T DEVELOPMENT BOARD, USING ARTIX-7 FPGA CHIP. SPECIFIC FPGA DEVICE IS XC7A100TCSG324-1.

and “Time per Sample” both decrease; however, when n is too large, e.g., $n = 1280$, the efficiency of the design starts dropping with an increasing memory overhead. In terms of time-area product per sample, the design is most efficient in when $n = 160$, therefore, $n = 160$ is chosen in the discussions in the following subsections.

B. Sensitivity to CDT Width

Table II shows the sensitivity of the CDT sampler to the precision. Recall that implemented precision is slightly larger than $\kappa/2$ to ensure a security margin. Further, with the current approach, 32-bit numbers are generated by the cSHAKE-based PRNG. Consequently, as expected, the number of required cycles for generating random numbers increases linearly with the increase of the precision. However, for the merge-exchange sort, the cycles are not affected by the precision since the size of the memory is fixed given n. In terms of logic utilization, the precision has a big impact since it determines the size of the basic computation units, e.g., register size, comparator size, etc.

C. Sensitivity to CDT Depth

Table III shows the sensitivity of the design to different CDT table depth. Recall that CDT table depth is equal to $\lceil \sigma\tau \rceil$. Gaussian sampling parameters are chosen as $(\sigma, \lambda, \tau) = (\sigma, 64, 9.00)$ with σ being a variant. This leads to a CDT of size $D_{cdt} = \lceil 9.00 \cdot \sigma \rceil$ and $W_{cdt} = 64$. For a fixed batch size $n = 160$, as expected, the performance is impacted similarly as being impacted by n shown in Table I, since merge-exchange sort module sorts the whole memory of depth $n + \lceil \sigma\tau \rceil$. The area increases slightly due to the change in the size (depth) of the CDT memory. For comparison with existing work, [4], [7], [3], the Table III also lists the evaluation of CDT table depths of: 28, 78, 110, and 1718.

VI. COMPARISON AND RESULTS

This section compares the presented work with the software CDT-based implementations of Gaussian samplers proposed in qTESLA [20], and the state-of-the-art CDT-based hardware implementation based on the *binary search* method as discussed in Section II-C.

D_{cdt}	Cyc. Total	LUT	FF	BRAM	Fmax (MHz)
20	14,300	706	752	3	111.9
32	15,200	699	752	3	116.8
40	16,000	786	760	3	124.8
64	18,000	784	760	3	114.2
80	19,500	787	768	3	115.7
128	24,700	859	785	3	118.6
160	28,000	864	793	3	119.6
256	37,900	868	793	3	116.3
320	44,400	784	801	3	108.8
512	68,800	821	816	3	119.1
640	84,900	913	824	3	113.4
1280	177,000	905	847	5	115.0
2560	391,000	925	871	10	120.2
28 [4]	14,900	706	752	3	114.1
78 [7]	19,400	798	768	3	116.5
110 [7]	22,800	863	785	3	116.0
1718 [3]	244,000	907	847	5	115.0

TABLE III

EVALUATION OF THE PERFORMANCE FOR DIFFERENT VALUES OF σ AND FIXED $\tau = 9.00$, THUS DIFFERENT D_{cdt}. n=160 (BEST TIME×AREA PRODUCT) EXCLUDING PRNG. THE DATA IS FOR ARTY A7-100T DEVELOPMENT BOARD, USING ARTIX-7 FPGA CHIPS SUGGESTED BY NIST. SPECIFIC FPGA DEVICE IS XC7A100TCSG324-1. NOTE, BOTTOM PART OF THE TABLE SHOWS DATA FOR SPECIFIC PARAMETERS MATCHING THE CORRESPONDING CRYPTOGRAPHIC ALGORITHM DESIGNS CITED IN THE BRACKETS.

A. Comparison with Software-Based CDT Samplers

Table IV shows comparison to software based CDT sampler used in qTESLA [20]. For the sampler itself, the data for Intel Core i7-6700 was provided by qTESLA developers, while data for ARM Cortex-M4 is estimated by us.[2] The entry “Total On-Chip Power” in the Vivado synthesis report, which is generated by Power Analysis Tool in Xilinx Vivado IDE, is adopted as measure of energy consumption of the sampler on FPGAs.

Comparing high-end CPUs and FPGAs, the software implementation is faster than the hardware design when synthesized for a high-end UltraScale+ FPGA in terms of the run-time given the much higher CPU frequency. However, based on average 65W power usage by the high-end CPU, the high-end FPGA has about 3× slower runtime, but is about 20× better in power. Therefore, the FPGA design of the CDT sampler is much more efficient in terms of power usage compare to a CPU-based implementation.

Meanwhile, for comparing low-resource CPUs and FPGAs, on ARM Cortex-M4, the software implementation does have lower power usage compared to low-resource Artix-7 FPGA, but in terms of running time, the low-resource FPGA is over 100× faster than the low-end ARM CPU. Therefore, our hardware-based CDT sampler is more efficient compared to the software implementations when taking both run-time and power consumption into account.

[2] While we do not have data for the software sampler for ARM Cortex-M4, based on comparison of whole qTESLA software on ARM, data from https://eprint.iacr.org/2019/844.pdf, and on Intel, data from https://eprint.iacr.org/2019/085.pdf, the M4 is about 6× slower in cycles and has 150× slower clock. ARM Cortex-M4 power estimates are from https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4, assuming 151 uW/MHz.

Design	Implementation	Method	Device	Parameters (σ, λ, τ)	Fmax	Cycles Total	Cycles PRNG	Time	Power (W)
p-I	SW, [20]	CDT	Intel i7	(8.5, 64, 9.06)	3400 MHz	228,704	36,345	67.3 us	65.000 W
p-III	SW, [20]	CDT	Intel i7	(8.5, 125, 12.94)	3400 MHz	344,894	77,894	101.4 us	65.000 W
p-I	SW, [20], Our Est.	CDT	ARM M4	(8.5, 64, 9.06)	24 MHz	1,372,224	218,070	60,570.0 us	0.004 W
p-III	SW, [20], Our Est.	CDT	ARM M4	(8.5, 125, 12.94)	24 MHz	2,069,364	467,364	91,260.0 us	0.004 W
p-I	HW, Our	CDT	Artix-7	(8.5, 64, 9.06)	115 MHz	62,256	5,327	541.2 us	0.299 W
p-III	HW, Our	CDT	Artix-7	(8.5, 128, 12.94)	118 MHz	71,328	10,375	603.0 us	0.332 W
p-I	HW, Our	CDT	UltraScale+	(8.5, 64, 9.06)	236 MHz	62,256	5,327	264.2 us	3.254 W
p-III	HW, Our	CDT	UltraScale+	(8.5, 128, 12.94)	259 MHz	71,328	10,375	275.2 us	3.284 W

TABLE IV

Performance comparison of our hardware based CDT sampler with the software sampler implementation in qTESLA [20], which is most closely related to our hardware design. SW indicates software implementation, while HW indicates hardware implementations, both generate 512 samples. All implementations are constant time and have fixed memory access patterns. Artix-7 is the XC7A100TCSG324-1 and UltraScale+ is the XCVU13P-FHGA2104-3-E.

Design	Method	Const. Time	Const. Acc. Pattern	Device	LUT	FF	BRAM	Fmax	Cycles
				PRNG Overhead Excluded					
[15]	CDT, binary search	✓	✗	Virtex-6	53	17	1	193 Mhz	2560
				PRNG Overhead Excluded					
Our	CDT, merge-exhange sort	✓	✓	Virtex-6	946	813	3	126 Mhz	50,700
Our	CDT, merge-exhange sort	✓	✓	Artix-7	893	796	3	113 Mhz	50,700
Our	CDT, merge-exhange sort	✓	✓	UltraScale+	839	796	3	251 Mhz	50,700
				PRNG (cSHAKE) Overhead Included					
Our	CDT, merge-exhange sort	✓	✓	Virtex-6	3949	1222	4	126 Mhz	56,000
Our	CDT, merge-exhange sort	✓	✓	Artix-7	3334	1168	3	113 Mhz	56,000
Our	CDT, merge-exhange sort	✓	✓	UltraScale+	3074	1168	3	251 Mhz	56,000

TABLE V

Performance of the CDT sampler and comparison with related work [15], all with $(\sigma, \lambda, \tau) = (3.33, 64, 9.42)$. Cycles provided in this table are all for generating 512 samplers. For data from [15], we show only their designs which store CDT in BRAMs, as we do. Virtex-6 is 6VLX75T-2FF484, Artix-7 is XC7A100TCSG324-1, and UltraScale+ is XCVU13P-FHGA2104-3-E.

B. Comparison with Hardware-Based CDT Samplers

Howe *et al.* [15] provided a comprehensive evaluation of discrete Gaussian samplers in hardware and, among others, presented one implementation of binary-search based CDT sampler, which works in constant time. The binary search and sort-based CDT samplers are both constant-time in hardware (although binary search is not constant-time in software and sort-based CDT sampling is constant-time in software). When implemented in hardware, the binary-search based CDT sampler is efficient both in area and performance, however, as discussed in Section II-C, the memory access pattern in the design is uniquely determined by the random uniform sample, therefore their design *et al.* [15] is possibly vulnerable to power or EM side-channel attacks.

We synthesized our CDT sampler with the same parameters as in [15], that is $(\sigma, \lambda, \tau) = (3.33, 64, 9.42)$. Table V shows the results for various FPGA boards. Based on the data, as expected, our design is less efficient compared to the binary-search based CDT sampler [15]. However, our design fully eliminates the memory-trace dependency on secret samples and therefore is much more resistant to potential power or EM side-channel attacks.

VII. Conclusion

This paper presented a novel hardware implementation of a constant-time discrete Gaussian sampler with fixed memory access pattern realized on FPGAs. The design used an approach based on Cumulative Distribution Table and a merge-exchange sort algorithm that enables generating samples in batches. Due to the use of the merge-exchange sort algorithm, the memory access pattern is always fixed, regardless of the values of the secret samples. The presented sampler can be fully parameterized at compile-time based on the needed *standard deviation*, *precision*, *tail cut*, and the *batch size*. It is the first CDT based design in hardware that is constant-time and has fixed memory access pattern.

Acknowledgment

This work was funding in part by National Science Foundation grants 1716541 and 1901901. We would like to thank Patrick Longa and Nina Bindel for their help and background information on Gaussian samplers. We would also like to thank Bernhard Jungk for help with cSHAKE code, and Xiayuan Wen for help with editing Verilog HDL code.

References

[1] National Institute of Standards and Technology (NIST), "Post-Quantum Cryptography Standardization." https://csrc.nist.gov/projects/post-quantum-cryptography, 2017. Accessed: 2018-07-23.

[2] D. E. Knuth, *The Art of Computer Programming*, vol. 3: Sorting and Searching. Addison-Wesley, 2nd ed., 1998.

[3] T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "FALCON – Submission to the NISTs post-quantum cryptography standardization process (round 2)," 2019. Available at https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/Falcon-Round2.zip.

[4] M. Naehrig, E. Alkim, J. Bos, L. Ducas, K. Easterbrook, B. LaMacchia, P. Longa, I. Mironov, V. Nikolaenko, C. Peikert, A. Raghunathan, and D. Stebila, "FrodoKEM – Submission to the NISTs post-quantum cryptography standardization process (round 2)," 2019. Available at https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/FrodoKEM-Round2.zip.

[5] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Dilithium – Submission to the NISTs post-quantum cryptography standardization process (round 2)," 2019. Available at: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/CRYSTALS-Dilithium-Round2.zip.

[6] H. Nejatollahi, N. Dutt, S. Ray, F. Regazzoni, I. Banerjee, and R. Cammarota, "Post-quantum lattice-based cryptography implementations: A survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, p. 129, 2019.

[7] N. Bindel, S. Akleylek, E. Alkim, P. S. L. M. Barreto, J. Buchmann, E. Eaton, G. Gutoski, J. Kramer, P. Longa, H. Polat, J. E. Ricardini, and G. Zanon, "qTESLA – Submission to the NISTs post-quantum cryptography standardization process (round 2)," 2019. Available at https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/qTESLA-Round2.zip.

[8] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, pp. 43:1–43:35, 2013.

[9] M.-J. O. Saarinen, "Gaussian sampling precision in lattice cryptography." Cryptology ePrint Archive, Report 2015/953, 2015. https://eprint.iacr.org/2015/953.

[10] J. Von Neumann, "Various techniques used in connection with random digits," *Appl. Math Series*, vol. 12, no. 36–38, p. Art. no. 1, 1951.

[11] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky, "Lattice signatures and bimodal gaussians," in *Advances in Cryptology - CRYPTO 2013 - Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I* (R. Canetti and J. A. Garay, eds.), vol. 8042 of *Lecture Notes in Computer Science*, pp. 40–56, Springer, 2013.

[12] D. E. Knuth and A. Yao, "The complexity of nonuniform random number generation," *Algorithms and Complexity: New Directions and Recent Results, San Diego, CA, USA: Academic*, pp. 357–428, 1976.

[13] R. Fisher and F. Yates, "Statistical tables for biological, agricultural and medical research," *Statistical Tables for Biological, Agricultural and Medical Research, 3rd ed. Edinburgh, Scotland: Oliver and Boyd*, pp. 25–27, 1948.

[14] S. S. Roy, O. Reparaz, F. Vercauteren, and I. Verbauwhede, "Compact and side channel secure discrete gaussian sampling." Cryptology ePrint Archive, Report 2014/591, 2014. https://eprint.iacr.org/2014/591.

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

[16] C. Peikert, "An efficient and parallel gaussian sampler for lattices," in *Advances in Cryptology - CRYPTO 2010* (T. Rabin, ed.), vol. 6223 of *Lecture Notes in Computer Science*, pp. 80–97, Springer, 2010.

[17] J. Howe, A. Khalid, C. Rafferty, F. Regazzoni, and M. O'Neill, "On practical discrete gaussian samplers for lattice-based cryptography," *IEEE Trans. Computers*, vol. 67, no. 3, pp. 322–334, 2018.

[18] D. Micciancio and C. Peikert, "Hardness of sis and lwe with small parameters," in *Advances in Cryptology - CRYPTO 2013* (R. Canetti and J. A. Garay, eds.), pp. 21–39, Springer, 2013.

[19] S. Kim and S. Hong, "Single trace analysis on constant time cdt sampler and its countermeasure," *Applied Sciences*, vol. 8, p. 1809, 2018.

[20] E. Alkim, P. S. L. M. Barreto, N. Bindel, P. Longa, and J. E. Ricardini, "The lattice-based digital signature scheme qtesla." Cryptology ePrint Archive, Report 2019/085, 2019. https://eprint.iacr.org/2019/085.

[21] K. E. Batcher, "Sorting networks and their application," in *AFIPS Spring Joint Computer Conference*, vol. 32 of *AFIPS Conference Proceedings*, pp. 307–314, Thomson Book Company, 1968.

[22] B. Jungk and J. Apfelbeck, "Area-efficient fpga implementations of the sha-3 finalists," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, pp. 235–241, IEEE, 2011.