A Hardware-Software Co-Design for the Discrete Gaussian Sampling of FALCON Digital Signature

Emre Karabulut
North Carolina State University
NC, USA
ekarabu@ncsu.edu

Aydin Aysu
North Carolina State University
NC, USA
aaysu@ncsu.edu

2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) | 979-8-3503-7394-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/HOST55342.2024.10545399

***Abstract*—Sampling random values from a discrete Gaussian distribution with high precision is a major and computationally-intensive operation of emerging and existing cryptographic standards. FALCON is one such algorithm that the National Institute of Standards and Technology chose to standardize as a next-generation, quantum-secure digital signature algorithm. The discrete Gaussian sampling of FALCON has *both flexibility and efficiency* needs—it constitutes 72% of total signature generation in reference software and requires sampling from a variable mean and standard deviation. Unfortunately, there are no prior works on accelerating this complete sampling procedure.**

In this paper, we propose a hardware-software co-design for accelerating FALCON's discrete Gaussian sampling subroutine. The proposed solution handles the flexible computations for setting the variable parameters in software and executes core operations with low latency, parameterized, and custom hardware. The hardware parameterization allows trading off area vs. performance. On a Xilinx SoC FPGA Architecture, the results show that compared to the reference software, our solution can accelerate the sampling up to 56.05× and the full signature scheme by 1.67×. Moreover, we quantified that our optimized multiplier circuits can improve the throughput over a straightforward implementation by 2.87×.

***Index Terms*—discrete Gaussian sampling, hardware-software co-design, post-quantum cryptography, digital signatures, FPGA.**

I. Introduction

The security of the current large-scale encryption infrastructure is based on the difficulty of solving mathematical problems such as integer factorization [1] and discrete logarithms [2]. Although these problems are conjectured to be hard for classical computers, quantum algorithms are proven to solve them exponentially faster [3], [4]. This poses a serious risk at the core of existing security systems and creates a critical need to design, implement, and deploy new cryptographic solutions that can survive the quantum threat. Such systems base their security on other *classical* problems like the ones in lattice-based cryptography [5]. The National Institute of Standards and Technology (NIST) has been leading this "post-quantum cryptography" effort by evaluating and standardizing promising algorithms. Recently, NIST has announced algorithms to be standardized for the new, quantum-safe, public-key encryption/establishment and digital signature applications [6].

Following the NIST standardization, a major push has just started for transitioning to post-quantum cryptography led by NIST, the Department of Homeland Security (DHS), Microsoft, Cisco, and Amazon Web Services, among others. A major roadblock to transitioning to practice is efficient implementation, which is especially important for real-time, embedded/edge, and battery-operated devices.

FALCON is one of the algorithms that NIST chose for the post-quantum standards. FALCON uses lattice-based cryptography—a (relatively) new family of cryptographic systems that are based on the short integer solution (SIS) over NTRU lattices [7]. Lattice cryptography includes new types of computations that are absent in earlier cryptographic standards such as RSA and ECDSA. One of these building blocks is the discrete Gaussian sampling needed to create the construction of the trapdoors. Having such high-precision Gaussian distributions reduces the signature size and, indeed, FALCON was explicitly chosen for its small signature sizes.

Unfortunately, the signature size savings of discrete Gaussian sampling comes at the expense of computing overhead. This is especially true for FALCON: for the reference software [7] provided in the NIST submission package, sampling can account for 72% of the total signing execution time[1]. Moreover, the sampling in FALCON is different from other lattice-based cryptographic schemes: it requires sampling from variable means and variances. Therefore, a practical implementation of FALCON's sampling procedure requires *both efficiency and flexibility*. Although there are plentiful works on FALCON's cryptanalysis, its efficient implementations, and especially custom hardware acceleration, are unknown.

In this work, we propose the first accelerated implementation of FALCON's discrete Gaussian sampling. Specifically, we designed and implemented a hardware-software co-design that can address both the flexibility and efficiency needs. We partition the sampling in such a way that the hardware executes the core operations of sampling (*i.e.,* sampling over the cumulative distribution table, exponent calculation, and rejection sampling) in a configurable manner, while the software performs floating-point divisions. The custom hardware consists of fully-pipelined, high-throughput, and low-latency datapath to carry out the floating-point-based sampling operations. Moreover, the custom hardware includes design-time

[1]This obviously changes with respect to the architecture. For implementation result details, please check Section V.

parameters that change the multiplication pipeline stage, allowing a trade-off between area vs. performance. The software runs on an ARM Cortex-A9 within the Xilinx Zynq SoC FPGA architecture and communicates with the hardware over the AXI bus. The results show that, compared to the NIST reference software [7] compiled on the same platform, our solution can accelerate the sampling up to 11.09×.

The rest of the paper is organized as follows. Section II provides the background on the discrete Gaussian sampling and its special use in FALCON digital signature algorithm. Section III describes the target system's architecture and the rationale behind our hardware-software partitioning. Section IV discusses our custom hardware design on the FPGA. Section V presents the implementation results. Section VI explains related aspects and future extensions, and Section VII concludes the paper.

Algorithm 1 FALCON Key Generation Algorithm [7]

```
Input: A monic polynomial $\phi \in \mathbb{Z}[x]$, a modulus $q$
Output: A secret key $sk$ and a public key $h$
 1: $f, g \leftarrow$ Gaussian_CDT_Sampling()
 2: $F, G \leftarrow$ NTRUSolve($f, g, \phi, q$)
 3: $B \leftarrow \begin{bmatrix} g & -f \\ G & F \end{bmatrix}$
 4: $\hat{B} \leftarrow FFT(B)$
 5: $G \leftarrow \hat{B} \times \hat{B}^*$        ▷ × represents matrix multiplication
 6: $T \leftarrow$ ffLDL*($G$)
 7: for each leaf of T do
 8:     $leaf.value \leftarrow \sigma/\sqrt{leaf.value}$
 9: $sk \leftarrow (\hat{B}, T)$
10: $h \leftarrow gf^{-1} mod(q)$
11: return $sk, h$
```

II. Background

This section provides the background information about the core concepts and methods for our work. We explain the specifics of discrete Gaussian Sampling and FALCON post-quantum digital signature scheme, then discuss the sampling procedures within FALCON.

A. Discrete Gaussian Sampling

Gaussian distribution ($D_{\sigma,\mu}$) is determined with a standard deviation σ and a mean (center) μ parameters. The statistical distance from the ideal distribution is determined by a precision parameter λ in the discrete Gaussian distribution. FALCON signature scheme samples its secret coefficients over a discrete Gaussian distribution [7]. Coefficient values range between $-\tau\sigma$ and $\tau\sigma$ where τ is $\sqrt{\lambda \times 2 \times \ln 2}$. The probability of coefficient value x is defined over the distribution and this probability is calculated with $\frac{1}{\sigma\sqrt{2\pi}} e^{(x-\mu)^2/2\sigma^2}$.

There are several algorithmic options to implement the discrete Gaussian sampling. We outline the major approaches as follows.

- *Rejection sampling* post-processes a uniform distribution to match the desired Gaussian distribution, by drawing random numbers from a uniform distribution and then accepting/rejecting the sampled integers, probabilistically, based on their likelihood of occurring on the Gaussian distribution [8]. Calculating the probability requires the support of complex floating point operations.
- *Ziggurat sampling* optimizes the rejection sampling [9] by decreasing the rejection rate. It divides the probability density function into equal regions and pre-defines the acceptance criteria for a subset of those regions. When those conditions are met, the floating point arithmetic is eliminated because the sampled integer will be accepted with 100% probability.
- *Bernoulli sampling* is another optimization on top of the rejection sampling that aims to reduce the rejection rate. This is achieved by approximating the original Gaussian distribution in two steps: (i) constructing an intermediate Gaussian distribution with a standard deviation smaller than the target and (ii) rejecting/accepting based on a carefully chosen Bernoulli distribution. Both Bernoulli and Ziggurat still require support for floating-point arithmetic.
- Knuth-Yao sampling pre-calculates a discrete distribution generating tree from the probability matrix of possible values for each random variable where the leaves represent the sampled value [10]. Then, it performs a random walk on this tree's branches (using uniform random values) to sample from non-uniform distributions. Knuth-Yao, therefore, effectively removes the run-time support requirement of floating-point arithmetic.
- *Cumulative Distribution Table (CDT)-based sampling* (a.k.a., inversion sampling), pre-computes the discrete cumulative distribution value for all possible samples and creates a CDT table that stores them in descending order [11]. The CDT table determines the likelihood of sampling each value and shapes a uniform distribution into a Gaussian distribution. This is achieved by sampling a uniform random value, comparing it against the pre-computed table entries, and returning the table index of the closest and smaller to the sampled value.

The core sampling procedure in FALCON follows a combination of CDT-based and Bernoulli sampling over two distributions due to their implementation efficiency.

B. FALCON Post-Quantum Digital Signature Scheme

FALCON is a post-quantum, lattice-based, hash-and-sign signature scheme [7]. FALCON signature scheme has three main steps: key generation, signature generation, and signature verification. FALCON requires sampling during key generation and signature generation to obtain its secret coefficients; hence, we omit the discussion about the verification procedure in this work. Unlike other NIST PQC finalists [12], [13], FALCON samples its secret coefficients over discrete Gaussian distributions instead of uniform distributions.

Algorithm 1 shows FALCON's key generation that computes the secret key sk and the public key h. The key generation algorithm first samples f and g polynomials over

Algorithm 2 FALCON Signature Generation Algorithm [7]

```
Input: a message $m$, a secret key $sk$, a bound $\beta^2$
Output: a signature $sig$ of $m$
 1: $r \leftarrow \{0,1\}^{320}$ uniformly
 2: $c \leftarrow$ HashToPoint $(r||m)$
 3: $t \leftarrow (\frac{-1}{q} FFT(c) \odot FFT(F), \frac{1}{q} FFT(c) \odot FFT(f))$
 4: do                    ▷ $\odot$ represents FFT multiplication
 5:     do
 6:         $z \leftarrow$ ffSampling $(t, T)$
 7:         s $\leftarrow (t - z) \begin{bmatrix} FFT(g) & -FFT(f) \\ FFT(G) & -FFT(F) \end{bmatrix}$
 8:     while $s^2 > [\beta^2]$
 9:     $(s_1, s_2) \leftarrow invFFT(s)$
10:     $s \leftarrow$ Compress$(s_2, 8 \cdot sbytelen - 328)$
11: while $s = \perp$
12: return $sig = (r, s)$
```

Algorithm 3 SamplerZ [7]

```
Input: Floating point values $\mu, \sigma'$ and $\sigma_{min}$
Input: Constant $\sigma_{inv}$ = $1/(2\sigma_{max}^2)$
Output: a sampled coefficient
 1: $s \leftarrow$ ceil($\mu$)
 2: $r \leftarrow \mu - s$
 3: $ccs \leftarrow \sigma_{min}/\sigma'$
 4: $dss \leftarrow 1/(2 \cdot {\sigma'}^2)$
 5: while (1) do
 6:     $z_0 \leftarrow$ BaseSampler($u$)        ▷ $u$ is uniformly random
    number
 7:     $b \leftarrow$ OneBitUniformRnd()
 8:     $z \leftarrow b + (2 \cdot b - 1)z_0$
 9:     $x \leftarrow ((z - r)^2) \cdot dss - z_0^2 \cdot \sigma_{inv}$
10:     if (BerExp($x, ccs$)) then
11:         return $z + s$
```

Algorithm 4 BaseSampler [7]

```
Input: a 72-bit uniformly sampled random number $u$
Output: a sampled $z_0 \leftarrow D_{\sigma_{base},0}$
 1: $z_0 \leftarrow 0$
 2: for $i$ from 0 to 17 do
 3:     if $u <$ RCDT[$i$] then
 4:         $z_0 \leftarrow z_0 + 1$
 5: return $z_0$ =0
```

a Gaussian distribution and then generates the secret key by using these two polynomials. Since the key generation algorithm works with constant standard deviation σ and the center μ, implementation is relatively straightforward. The prior work [14] that presents a hardware Gaussian sampler implementation for FALCON's key generation procedures can be used to this end.

Algorithm 2 illustrates FALCON's signing procedure, which takes in a message m, the secret key sk, and then returns a signature (r and s). This signing algorithm predominantly relies on three main subroutines: HashToPoint, ffSampling, and Compress. In the HashToPoint subroutine, the algorithm first concatenates the message with a uniformly generated polynomial and then hashes it to the polynomial c. Compress subroutine reduces the signature size with a simple encoding mechanism. ffSampling is referred to as the "heart of the signature generation" in FALCON's specification document [7]. This subroutine is responsible for generating vector z, which is the core polynomial in the signature generation process. The coefficients of z are sampled over a Gaussian distribution. The parameters of this distribution, σ and μ, are variable and based on the coefficients of a polynomial t.

The polynomial t itself is derived from the product of the message and the secret key. As FALCON signature generation accommodates different messages, both the t polynomial and its associated σ, and μ parameters inherently vary. Since ffSampling works with dynamic σ and μ parameters, each coefficient of the signature is likely to be sampled over different Gaussian distributions. Although the software's flexibility allows for the implementation of a discrete Gaussian sampler with varying σ and μ parameters, efficient hardware acceleration of this sampler is significantly challenging—there was no such hardware to date at the time of this paper.

C. Sampling in FALCON

FALCON has two sampling routines, one for the key generation and the other for the signature generation. The key generation requires a sampling operation over Gaussian distribution with constant σ and μ parameters. By contrast, signature generation requires a Gaussian distribution structure that needs varying σ and μ parameters. Therefore, FALCON employs a sampling strategy that comprises two levels of sampling. The first layer samples a value from the base distribution that has constant σ and μ parameters. This base distribution is common to both key and signature generation processes. The second layer rejects or accepts the value sampled from the first layer based on the second distribution that satisfies varying σ and μ. While the key generation employs the first layer for its sampling, the signature generation uses the both layers.

Algorithm 3 presents the $SamplerZ$ method that includes two layers of samplings and provides a discrete distribution for the varying σ and μ. This algorithm first calls $BaseSampler$ at step 6 to sample a non-uniform value over the described Gaussian distribution that has constant σ and μ parameters. Then, it rejects or accepts the sampled value based on the second sampling that is performed by the $BerExp$ algorithm at step 10. Note that the $SamplerZ$ function is called many times during FALCON's signature generation with varying inputs (μ and σ'). Therefore, the variables s, r, ccs, and dss are calculated at run-time. These calculations require floating point arithmetic.

Algorithm 4 shows the first layer of the sampling algorithm used in FALCON, where the input is a 72-bit uniform random number (u) and the output (z_0) is a 5-bit integer from the distribution $D_{\sigma_{base},0}$. This algorithm requires a table that stores pre-computed reverse cumulative distribution values (stored

Algorithm 5 BerExp [7]

```
Input: Floating point values $x$, $ccs \leq 0$
Output: 1-bit sampling result
 1: $s \leftarrow$ x $\cdot\ ln(2)^{-1}$
 2: $r \leftarrow x - s \cdot ln(2)$
 3: if $s > 63$ then
 4:     $s \leftarrow 63$
 5: $z \leftarrow$(2·ApproxExp($r, ccs$) - 1) $>> s$
 6: $i \leftarrow 64$
 7: do
 8:     $i \leftarrow (i{-}8)$
 9:     $urnd \leftarrow$ OneByteUniformRnd()
10:     $w \leftarrow urnd - ((z >> i)\&0\text{xFF})$
11: while $(w == 0\ \&\&\ i > 0)$
12: if $w < 0$ then
13:     return True
14: else
15:     return False
```

in the RCDT array, line 3). The algorithm first compares a uniform random number against the table and then returns the index of the first entry that is larger. Since all distribution values are pre-calculated and then stored in a table, this algorithm does not need to perform the $exp()$ calculation.

Algorithm 5 shows the $BerExp$ algorithm which is the second layer of the sampling procedure. Since the algorithm input x is a varying value, this algorithm needs to work with dynamic floating-point operations, and it also requires the $exp()$ calculation. FALCON reduces the computation demand by introducing $ApproxExp$ implementation (see step 5 in Algorithm 5). Within this context, $ApproxExp$ leverages a precomputed table to approximate the $exp()$ function. After processing the arithmetic and logical operations shown in the algorithm and using one byte of uniform randomness (via OneByteUniformRnd), the $BerExp$ algorithm returns a one-bit sampling result. This bit serves as a decision bit-—determining whether to accept or reject the output from $BaseSampler$. If $BaseSampler$'s output is rejected, $SamplerZ$ repeats the operations from steps 6 through 10 until a sampled value is accepted by $BerExp$.

III. System Architecture and Hardware-Software Partitioning

FALCON's sampling consists of floating-point addition, multiplication, and division. These operations are expensive to implement in hardware because of the large operands. Thus, FALCON's C reference implementation (software) approximates these floating-point operations with large integer values. Listing 1 shows FALCON's SamplerZ reference implementation and our partitioning into hardware and software in our system. The sampling requires floating point-based division operations (lines 12 and 13). The reference implementation approximates the operations over large integers r, dss, ccs, and x using **fpr** which is defined with a 64-bit int data type. The algorithm has also varying input that increases hardware

Listing 1: FALCON SamplerZ reference implementation [7]. This also shows the partitioning of the reference implementation for hardware(HW) and software (SW).

```
 1  int Zf(sampler)
 2      (void *ctx, fpr mu, fpr isigma)
 3  {
 4
 5  //==================SW=====================
 6     sampler_context *spc;
 7     int s;
 8     fpr r, dss, ccs;
 9     spc = ctx;
10     s = (int)fpr_floor(mu);
11     r = fpr_sub(mu, fpr_of(s));
12     dss = fpr_half(fpr_sqr(isigma));
13     ccs = fpr_mul(isigma, spc->sigma_min);
14  //=========================================
15
16  //==================HW=====================
17     for (;;) {
18       int z0, z, b;
19       fpr x;
20       z0 = Zf(gaussian0_sampler)(&spc->p);
21       b = (int)prng_get_u8(&spc->p) & 1;
22       z = b + ((b << 1) - 1) * z0;
23       x = fpr_mul(
24            fpr_sqr(fpr_sub(fpr_of(z), r)),
25            dss);
26
27       x = fpr_sub(x,
28            fpr_mul(fpr_of(z0 * z0),
29            fpr_inv_2sqrsigma0));
30
31       if (BerExp(&spc->p, x, ccs))
32           return s + z;
33     }
34  //=========================================
35
36  }
```

implementation complexity and that requires flexibility in the implementation. In addition to the division, the sampling operation needs floating point multiplications in $samplerZ$ (lines 23 and 28) as well as in **BerExp**. There are also memory read operations in **gaussian0_sampler**. Unlike divisions, they are relatively easier to implement on the hardware but these are time-consuming operations in software since the operands are large numbers.

To enhance throughput without compromising flexibility, we strategically partition the sampling operations between hardware and software workloads. Our partitioning aims to increase parallelism in the sampling operation. The key decider in our hardware-software partitioning is the separation of floating point divisions from multiplications and table read operations. The crux of our partitioning approach lies in offloading floating-point divisions to software, benefiting from a highly efficient and flexible ALU datapath in the processor. Other computational units are accelerated in custom-designed hardware. The software part consists of floating point-based division and parameter initializations and covers steps between 10 and 13 in Listing 1. The hardware partitioning covers the remaining steps (17 to 32).

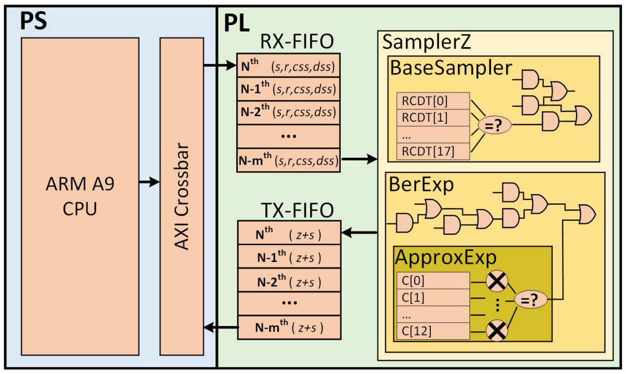


Fig. 1: The proposed architecture for FALCON's sampling operation. The architecture has two partitions that provide a flexible environment where the software and hardware can collaborate and run parallel.

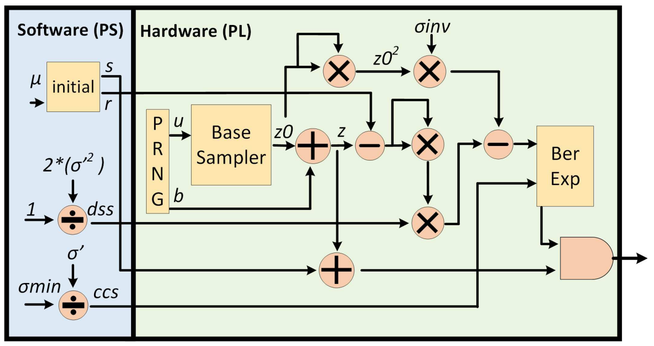


Fig. 2: Software and hardware building blocks of $SamplerZ$ subroutine. The software (blue) executes floating-point divisions, while the hardware part (green) performs $BerExp$ and $BaseSampler$.

Intermediate dependencies might act as bottlenecks in unoptimized systems, slowing down performance as one component waits for another to complete. It is noteworthy that while there are input and output data dependencies between our software and hardware partitions, no intermediate dependencies are present in our partitioning. Our approach avoids this potential bottleneck by isolating the software and hardware in terms of intermediate dependencies. To illustrate, consider the scenario wherein the hardware works on the first call's while loop, the software can execute s, r, css, and dss for the next call. This design choice enables the simultaneous execution of software and hardware tasks. This structure brings about a marked improvement in the efficiency of the sampling processes.

We use a Xilinx SoC FPGA that enables running both software and hardware implementations in a single SoC. Figure 1 presents the SoC FPGA architecture and the proposed design's block diagram. Our design utilizes the Processing System (PS) part for performing the operations that require a flexible architecture such as the floating point-based division, whereas the Programmable Logic (PL) part has dedicated hardware designs to accelerate the operations. Our design has two independent FIFOs that synchronize the software and hardware executions and also enable running in parallel. The software implementation generates the inputs of the hardware accelerator and sends them via RX FIFO. The hardware design sends the output of the $SamplerZ$ algorithm to the software with TX FIFO.

IV. Hardware Design

The proposed hardware design has one accelerator core for $SamplerZ$ and FIFO data paths. $SamplerZ$'s accelerator has three sub-modules: $BaseSampler$, $BerExp$, and $ApproxExp$. Figure 2 illustrates the $SamplerZ$ algorithm's major building blocks and their hardware-software partitions. The software performs initial floating point-based divisions for ccs and dss and parameter initializations for r and s. The hardware accelerator executes $BerExp$, $BaseSampler$, and $ApproxExp$ that require heavy multiplications and memory read operations. For example, the software would sequentially read table entries in the reference $BaseSampler$, and $ApproxExp$ algorithms, while the hardware can read the entire table and perform all comparisons in one clock cycle. The figure also shows $SamplerZ$'s four data paths where the hardware parallelizes the operations[2].

A. A Half Gaussian Sampling with BaseSampler

We implement the $BaseSampler$ algorithm with a precomputed reverse cumulative distribution (RCDT) table in hardware, which is carried out in software in FALCON's reference implementation. The $BaseSampler$ algorithm samples a value over a Gaussian distribution centered at 0. Figure 1 outlines the $BaseSampler$ implementation in $samplerZ$ module. It has one register file to store RCDT and one comparison circuit. The input of the design is a 72-bit uniform random number and the output is an unsigned integer ranging from 0 to 17.

The design first receives the 72-bit number and then performs a parallel comparison between the input and each entry of RCDT. The last step of the design is to count and return the number of entries that are larger than the given input. The software implementation may require multiple cycles to execute a comparison between the input and one entry of RCDT. By contrast, our hardware implementation can complete the entire table comparison in one cycle.

B. Rejection Sampling with BerExp

The $BerExp$ algorithm returns a single bit and the probability of returning 1 is $css \cdot exp(-x)$. The css value is a floating-point number and defines the scaling factor for each σ' value. The variable x is also a floating-point input that is calculated with the $BaseSampler$'s sampled value.

[2]The figure also depicts a PRNG (pseudo-random number generator). In FALCON's reference implementation, an external randomness source is assumed, hence an additional C function for PRNG is not implemented. For testing purposes in our system's functionality, we introduced a PRNG that deterministically produces uniformly random numbers. This ensures that both the reference implementation and our hardware utilize the same source of randomness, facilitating direct comparison of their final outputs.

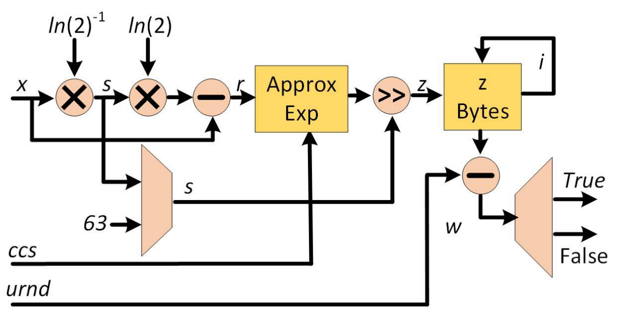


Fig. 3: The block diagram of $BerExp$ sub-routine. It is fully implemented on the hardware.

Although the $BerExp$ does not have a floating-point division, it requires floating-point addition and multiplication. FALCON documentation provides the minimum and maximum ranges of the $SamplerZ$ inputs. Therefore, we can simply extend the floating-point numbers with 2^{72} and can work over their integer values with 72-bit precision.

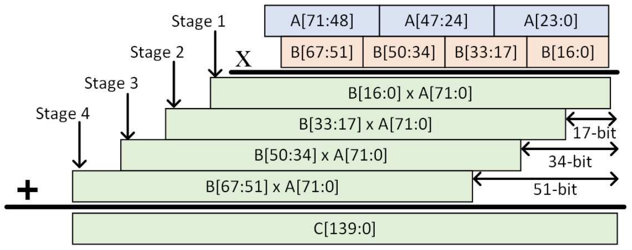
Figure 3 illustrates the $BerExp$ algorithm's computations. Algorithm 5 shows that the input x first goes through a floating-point division at the first step with $ln(2)$ to obtain s. Since $ln(2)$ is a constant value, we calculate its inverse ($ln(2)^{-1}$) and then extend it with 2^{72} at the compile time. After this pre-calculation, the first floating-point division operation simply becomes a 72-bit unsigned multiplication. The first multiplication result is 144 bits. However, we do not reduce this multiplication result to 72 bits before the next multiplication to preserve the precision.

$BerExp$ sampling design also has the $ApproxExp$ module to compute an approximation of $2^{63} \cdot css \cdot exp(-x)$. FALCON's NIST submission package already provides a C implementation of the $ApproxExp$ module. This module computes the approximation over a pre-calculated table (C), like $BaseSampler$. The implementation iteratively performs 64-bit multiplications between table entries and the output of the previous multiplication. The first multiplication happens between the input x and the first table entry. Table C has 13 entries and therefore $ApproxExp$ module initially requires 13 multiplications and then obtains $2^{63} \cdot exp(-x)$. The final step of $ApproxExp$ is a multiplication between ccs and $2^{63} \cdot exp(-x)$.

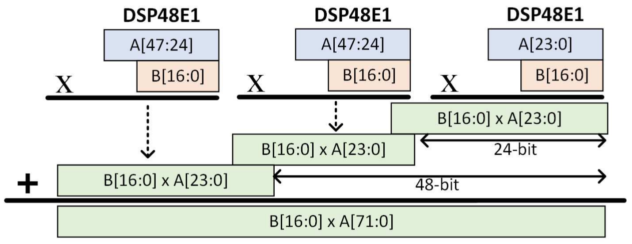
$BerExp$ algorithm, at its last step, compares the output of $ApproxExp$ module with a uniformly sampled number ($urnd$). This comparison is performed byte by byte and continues until the $BerExp$ algorithm accepts the comparison output. Since hardware, unlike most software platforms, allows performing the comparison in parallel for each byte, our design completes the rejection operation in one cycle.

C. Optimizations in Large Multiplications

Gaussian sampling over the integers is one of the unique features of the FALCON digital signature algorithm. Although FALCON works with varying standard deviations



(a) A schoolbook multiplication with four stages.



(b) One stage multiplication with 3 DSP48E multipliers.

Fig. 4: A multiplication representation for 72-bit and 68-bit unsigned operands. The multiplication is constructed with 17-bit and 24-bit multiplication steps due to the input size constraint of the Xilinx DSP48E1 block.

and arbitrary centers that are ideally floating-point numbers, FALCON approximates these numbers with large integers by maintaining the required precision. Therefore, FALCON's Gaussian sampling algorithm requires large integer multiplications. The $ApproxExp$ function, for instance, requires 64-bit integer multiplications (14 times) to approximate the exponent operation in FALCON's reference implementation. In addition, FALCON's Gaussian sampling algorithm might call this function more than once to sample a single coefficient based on the number of rejected samples. $BerExp$ and $SamplerZ$ are other steps that require large integer ($\geq$ 72-bit) multiplications, and they may also call several iterations of these multiplications based on the rejection rate.

If FALCON's Gaussian sampling is designed without optimization, dedicated multipliers are separately allocated for each multiplication[3]. This implementation approach results in 116 DSP48E1 utilization because a single DSP48E1 block can multiply at most signed 18-bit with 25-bit variables (or unsigned 17-bit with 24-bit variables) [15]. Hence, large integer multiplication mandates the cascade of several DSP48E1 blocks. Moreover, this unoptimized design operates with low frequencies due to long critical paths caused by the cascaded DSP48E1 blocks. Our proposed design offers a pipelined multiplier structure and optimizes DSP utilization. Therefore, the proposed design operates with a significantly higher frequency.

We first exemplify large integer multiplication with multistage 24-bit and 17-bit multiplications using DSPs with maximum utilization. Figure 4a illustrates the product operation of 72-bit and 68-bit operands with the schoolbook multiplication method. The first operand (A) is 72-bit and split into 3 parts

[3]Simply, the product of A and B is implemented as $A * B$ in HDL.

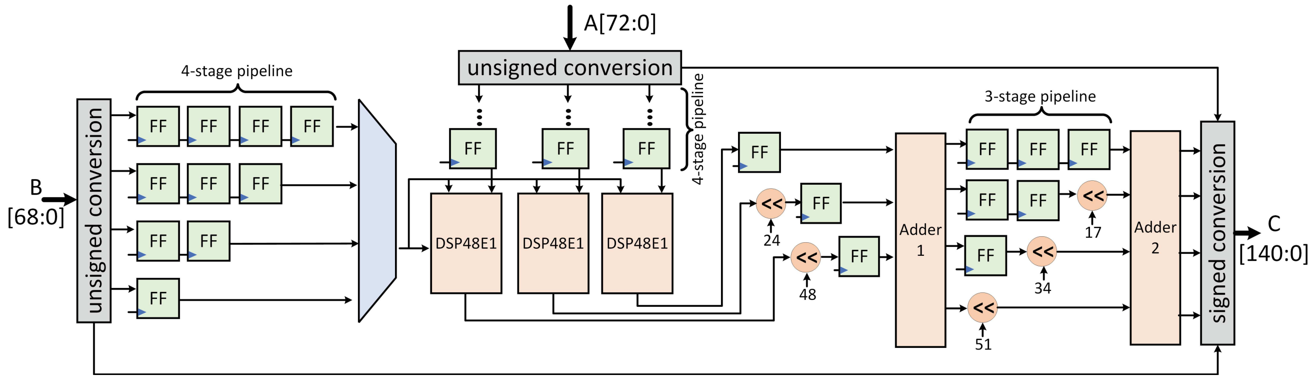


Fig. 5: The proposed multiplier design. This design has multi-stage pipelining that multiplies 73-bit and 69-bit signed operands only with 3 DSP48E1 instead of 16 DSP48E1 blocks. The latency is 8 cycles and throughput is 1 multiplication per 4 cycles.

with an equal length of 24-bit, while the second operand (B) is split into 17-bit vectors. First, the operand A is multiplied by each 17-bit vector of the second operand (B); hence, each stage consists of a 72-bit by 17-bit multiplication. Second, each block multiplication result is shifted left accordingly. The final result is the sum of each stage output.

Figure 4b presents a multiplication block for the first stage of the aforementioned schoolbook multiplication method using three DSP48E1s. Each DSP48E1 shares the operand A with 24-bit fractions, while their other operands are identical, the LSB 17-bit of operand B. Similar to the previous instance, the final result is obtained by first shifting and then summing each multiplication output. The complete hardware, therefore, requires 12 DSP48E1 multiplications, 11 shift operations, and 5 additions to multiply A by B.

Figure 5 shows the proposed pipelined large integer multiplier design. This hardware multiplies two signed operands, one of which can have a maximum size of 73 bits while the other can have a maximum size of 69 bits. The design has a fully-pipelined architecture with 8 stages and thus its latency is 8 cycles but its throughput is one multiplication per 4 cycles. The presented multiplier first converts the two signed operands to unsigned operands before the multiplication operation. Second, it splits the first operand (72-bit) into 3 24-bit groups and the second operand (68-bit) into 4 17-bit groups to fully utilize the DSP48E1 block. Figure 4a shows each stage of the multiplication is executed between the single but different 17-bit vector of B and the second operand A. To sort B's vectors in a cyclic order, the design pipelines each vector with a different number of registers and also selects the corresponding B's vector with a cycle count. For example, the LSB 17-bit of operand B arrives at the DSP48E1 units after the first cycle, while the next 17-bit arrives at the next clock cycle. However, A's vectors are pipelined with the same number of registers since A should remain stable every 4 cycles. Our design utilizes enabled registers and therefore the implementation requires fewer registers for the pipelining. For example, A's vectors are pipelined with one stage of register rather than 4 stages and the register enable signal is activated after every 4 cycles.

The DSP48E1 blocks generate the output of one of the stages in schoolbook multiplication as shown in Figure 4a. Then, the outputs are shifted accordingly before going through the adder circuits. Adder 1 executes the addition operation to execute the model described in Figure 4b, while the second adder is used to perform the addition shown in Figure 4a. The final step of design is the sign extension operation that assigns the corresponding sign to the multiplication product. If these two signed operands' multiplication is implemented with a single asterisk ($*$), the implementation requires 16 DSP48E1 blocks, and that increases DSP48E1 usage by 5.33×.

Our multiplier design also provides design-time flexibility with different area-performance settings. The default configuration uses our performance-optimized design which is described in detail above. If the area-optimization flag is set, the multiplier design works with a single DSP48E1 but, it reduces the throughput and performs one multiplication per 15-cycle. In addition, one multiplication operation in the $samplerZ$ algorithm requires two operands larger than 69-bit but smaller than 73-bit. Therefore, we implemented an additional multiplier that has a 10-stage pipelined design with 3 DSP48E1s.

It is pertinent to note that the critical path in our design is not primarily determined by the DSPs' multiplication process. Instead, it's the operations associated with the second adder (Adder 2). Since Adder 2 has 140-bit operands along with shifting operations, it creates the longest path. Although pipelining Adder 2 with additional registers increases the operating frequency, it negatively impacts the area×delay product, a crucial efficiency metric in hardware design. Given this trade-off, the multiplier design does not utilize additional registers for the Adder 2, aligning with our design objectives and setting our latency goal.

V. Implementation Results

We used Xilinx Zedboard XC7Z020 from the Zynq-7000 SoC family as our platform. This board has an SoC FPGA, which includes an ARM Cortex-A9 processor for software

TABLE I: Hardware area utilization of the proposed solution. A straightforward implementation of the large multipliers would require **116** DSPs.

Task	Module	LUT	FF	DSP	BRAM
Base Design	SamplerZ	3,456	732	**116**	0
Perf. Opt. Design		2,523	2,028	**9**	0
Area. Opt. Design		2,328	1,030	**3**	0
Sync + Communication	AXI_FIFO	592	632	0	2
Communication	AXI_Periph	405	587	0	0
Reseting	rst_sys	16	23	0	0

execution and a programmable logic part for hardware acceleration. We first designed and functionally verified the proposed $SamplerZ$ and created an AXI-Stream wrapper on it. Second, we hooked the wrapped $SamplerZ$ to FIFOs and Zynq processor as described in Section III. Then, we tested the proposed implementation and measured its performance with FALCON's implementation's test vectors [16].

The summary of our results is as follows. Compared to FALCON's reference implementation running on the same platform, our solution accelerates the sampling by up to $11.09\times$. The bottleneck becomes the slow AXI communication, and our improvement can theoretically increase to $90.36\times$ with better AXI. Our optimized multipliers improve area efficiency by 12.88–38.66×. The proposed hardware core is even faster than a high-end Intel processor by $38.34\times$. Accelerating the sampling process with our solution leads to a $1.67\times$ improvement on the entire FALCON signature performance. Compared to the earlier hardware designs that uses fixed parameters, our hardware with varying σ and μ is obviously more costly as expected—this is the first accelerated implementation of FALCON's discrete Gaussian sampling that can be used during signature generation.

A. Comparison of Hardware Design Choices

Table I shows a breakdown of area utilizations for each IP block as well as their tasks in our design. We present a baseline design, along with two optimization settings: the Perf. Opt. design (performance-optimized) and the Area Opt. design (area-optimized). The baseline design is with the straightforward implementation of the multiplier without our proposed optimizations. Both the Perf. Opt. and Area Opt. designs require three multiplier circuits, but their multiplier structures differ. The Perf. Opt. design employs three DSP-based multipliers, using a total of 9 DSP48E1s, while the Area Opt. design uses a single DSP-based multiplier, consuming 3 DSP48E1s. As shown in Table I, the baseline design, devoid of multiplier-specific optimizations, unsurprisingly demands a significantly higher number of DSP48E1 units. Since Perf. Opt. design has a larger data path due to its parallel multipliers, it requires more registers than the design with area-optimized configuration.

B. Performance Evaluation and Throughput Improvement of $SamplerZ$

Table II presents a performance comparison between FALCON's reference implementation [7] and the proposed designs

TABLE II: Performance comparison with AXI communication overhead. Our hardware-software co-design solution improves the throughput by up to **11.09**×.

Design	Frequency	Latency (#Cycle)	Throughput (sample/sec)	Throughput Improvement
Reference SW [7]	666 MHz	96,747	6,890	-
Base Design	45 MHz	1,691	26,611	**3.86**×
Perf. Opt.	132 MHz	1,726	76,447	**11.09**×
Area Opt.	134 MHz	1,807	74,156	**10.76**×

TABLE III: Peak performance comparison of cores. Our hardware-software co-design solution can theoretically outperform software implementation by up to **90.36**× with maximum AXI performance.

Design	Frequency	Latency (#Cycle)	Throughput (sample/sec)	Throughput Improvement
Reference SW [7]	666 MHz	96,747	6,890	-
Base Design	45 MHz	124	362,903	**52.67**×
Perf. Opt.	132 MHz	212	622,641	**90.36**×
Area Opt.	134 MHz	406	330,049	**47.90**×

with different optimization settings. We calculated the improvement result based on the designs' throughputs. The base design has a lower cycle count than other designs as expected. However, its frequency is the lowest (45 MHz) among other designs since it does not have a pipelined architecture. In contrast, both the Perf. Opt. and Area Opt. designs operate at much higher frequencies, 132 MHz and 134 MHz respectively, resulting in a substantial throughput improvement. Specifically, Perf. Opt. design improves throughput by $11.09\times$, whereas the Area Opt. design improves it by $10.76\times$. Despite their different multiplier units, the performance difference between the Area Opt. and Perf. Opt. designs is marginal. This is caused by the AXI components and we later explain the AXI components' impact on the performance with Table III. Also, the achieved frequencies of 132 MHz and 134 MHz are especially noteworthy since our platform has a relatively low-end FPGA. Even with a mid-level FPGA, the achievable frequency exceeds 200 MHz, as detailed in Section V-F.

Our hardware design consists of two main parts: $SamplerZ$ accelerator core and AXI components. Table II presents a performance result for the entire architecture including the AXI interconnect and AXI FIFO overheads. Accelerating the AXI communication bandwidth is beyond the scope of this study. Therefore, we also present Table III that illustrates only $SamplerZ$ accelerator theoretical performance results with maximum AXI performance The results show that a peak throughput improvement of $90.36\times$ is theoretically possible. Since the base design does not have pipelined multipliers, its latency is the best one compared to the other two. However, the Perf. Opt. design's theoretical throughput is $1.71\times$ and $1.88\times$ more than base designs and Area Opt. designs, respectively. Although the base design's throughput is $1.09\times$ better than the Area Opt. design, its DSP requirement overhead is $38.66\times$ more than the Area Opt. design. The key takeaway from Table III is that further acceleration may be possible by improving the I/O communication overhead or by porting the

TABLE IV: Improvement from hardware acceleration. The tasks allocated to hardware take 35 cycles with our design, but it takes several hundred to thousands of cycles on ARM and Intel CPU.

Platform	Optimization	Latency (#Cycle)	Latency Improvement
ARM Cortex M7 [17]	FPU	664	**18.97** ×
	EMU	3820	**109.14**×
ARM Cortex M4 [17]	FPU	5418	**154.14**×
	EMU	31,744	**906.97**×
Intel® Core® i7-8700 [18]	Reference SW	1,342	**38.34** ×
Proposed Hardware	Perf Opt	35	-

TABLE V: Performance comparison on the full FALCON-512 signature generation and its sub-routines. Due to our accelerated sampling, the full signature generation is accelerated by **2.7**×.

Operation	Reference SW [7] #Cycle	Our Design #Cycle	#Cycle Improvement
SamplerZ	96,747	1,726	**56.05**×
Gaussian Sampling	99,068,928	1,767,424	**56.05**×
ffSampling	172,559,792	75,258,288	**2.29**×
FALCON Signature Generation	245,706,666	148,405,162	**1.67**×

TABLE VI: Comparison with prior works. We propose the first sampler that can support FALCON signature generation scheme. As expected, the sampler requires more resources compared to fixed μ and σ samplers.

Work	FALCON Support	σ/λ	Platform	LUT/FF/ BRAM	F_{Max} (MHz)	#Cycle
This Work Area Opt.	**Full Support**	varying	XC7Z020	3341/2272/2	134	1807
[14]	**KeyGen only**	2/53	Virtex-7	151/7/0	322	1
		$\sqrt{5}/200$		455/8/0	317	1
[19]	**No**	8.5/64	Artix-7	907/812/3	115	111
[20]		8.5/64	Artix-7	511/343/0	353	1
[21]	**No**	3.33/64	Virtex-6	112/19/0	297	5
[22]		3.33/90	Virtex-5	43/33/1	259	3
[23]		3.33/80	Virtex-6	863/6/0	61	1
[24]	**No**	215/128	Spartan-6	928/1121/0	129	8
[25]	**No**	4.41/112	Spartan-6	426/123/1	102	8
[26]		4.41/112	Spartan-6	463/45/0	80	30

design to a more capable FPGA that can execute at a higher operating frequency.

C. Comparison with Software on Other Platforms

Next, we compare the impact of our hardware acceleration by comparing it with its corresponding software running on other platforms. This corresponding software of workload is shown in Listing 1, between lines 17 and 32. Table IV presents this comparison. This table categorizes the software versions into two primary optimized implementations: one that utilizes emulated floating-point (EMU) operations and the other that harnesses dedicated floating-point instructions (FPU)—both are included in the FALCON NIST submission package. Table IV also presents Intel® Core® i7-8700 performance results for FALCON's reference implementation. Since FPU employs ARM's dedicated floating-point instructions, Intel® Core® i7-8700 does not have performance results for FPU. To provide a fairer comparison, we compared implementations' performance using cycle count rather than throughput and frequency, taking into account that they are different platforms. The cycle count we provide for our performance-optimized design specifically omits considerations for software partitioning and I/O overheads.

Table IV depicts that our novel and efficient hardware implementation significantly improves the performance (up to 906.97×) in terms of cycle count. FALCON's sampling procedure requires operations with large operands so ARM Cortex M4 performs the poorest due to its basic structure. Although ARM Cortex M7 is able to benefit from 64-bit floating-point instructions, our hardware implementation executes the same operation 18.97× and 109.14× faster. Our hardware even outperforms the high-end CPU, Intel® Core® i7-8700, by showing 38.34× better performance. We also note that FALCON's Gaussian sampler execution is not constant for its software implementations. Therefore, the cycle count numbers of the software execution might slightly vary for different runs.

D. Impact of Gaussian Sampling Optimization on Signature Generation Performance

We next quantify the impact of the improvement on the whole digital signature scheme, not just on the discrete Gaussian distribution. Table V compares the performance between FALCON's reference implementation vs. our performance-optimized design for discrete Gaussian sampling. We first profiled the $SamplerZ$ function by running reference implementation and then ran the same function by enabling our accelerator. The result shows that our accelerator improves the $SamplerZ$ execution time by 56.05×. Since the Gaussian sampler is called $2n$ times during the ffSampling, our design decreased the Gaussian sampler's execution cycle count from 99M to 1.7M cycles. FALCON's signature generation heavily depends on the performance of the Gaussian sampler. Therefore, our design improves the entire signature generation by 1.67×, from 245M cycles down to 148M.

E. Comparison to Previous Works on Gaussian Sampling

Table VI presents a comparison between the proposed design and prior Gaussian samplers. There are earlier sampler designs for BLISS [21], [27], LP [21]–[23], FrodoKEM [28] and qTESLA [19]. These hardware implementations are fixed for a single σ, μ parameter setting; hence, they **cannot** support FALCON. An earlier FALCON implementation [14] proposed a 'design-time' flexible hardware for σ, μ parameters—this hardware cannot support run-time flexibility and thus is limited to FALCON key generation (i.e., cannot support FALCON signature generation). Other FALCON implementations include FALCON's verification steps [29], which excludes Gaussian sampling, and SIMD acceleration on software [30] [4]. Likewise, another prior work passes FALCON software implementation

[4]Unfortunately, the discrete Gaussian sampling sub-routine latency is not provided in this work; hence, a comparison is infeasible

through an HLS tool but does not provide a performance or area profiling for the Gaussian sampler sub-routine [31]. These results show that, despite our hardware optimizations, FALCON's sampling needs still incur more time and area overheads compared to other lattice-based cryptosystems. FALCON has also a GPU implementation [32]. However, this work does not provide a performance result for FALCON's Gaussian sampler. As a result, we cannot provide a discussion about comparing our work with GPU platforms.

F. Platform Choices and Comparative Performance.

Our design inherently requires a close integration of software and hardware components. This necessitates a platform that can adeptly accommodate both. The Xilinx Zedboard XC7Z020, an SoC FPGA, emerged as our primary choice for its ability to support our hardware-software co-design. With the PS side of the Zedboard managing the software partition and the PL side tailored for the hardware task. Furthermore, this platform enabled testing its functionality and measuring its performance in real-time.

Building upon our experiments, we identified two main objectives for further exploration. First, evaluating the performance of our multiplier architecture across different FPGAs might provide a fresh perspective on our overall contributions. Second, it is essential to synthesize our design on a mid-level FPGA board, Virtex 7 FPGA VC709, for a more aligned comparison given that a majority of prior research gravitates towards the use of Virtex series FPGAs, we considered. As a result, our performance-optimized design attained a frequency of 213 MHz on the Virtex 7 FPGA VC709 board, which is significantly higher than the Zedboard's 132 MHz. This variation emphasizes the crucial significance of the chosen FPGA board and its inherent properties on the timing performance of a design.

VI. Discussions

This section explains related considerations and perspectives surrounding our solution. We clarify our standpoint on implementation security, explore potential design adaptations for lower-end FPGA devices, and highlight the significance of FPGA choice in enhancing Gaussian sampling efficiency.

A. Implementation Security

We do not cover implementation attacks and associated defenses in this work. There are various attack vectors including fault injection attacks, power/EM side-channel attacks, microarchitectural attacks, acoustic/photonic side-channel attacks, and cold-boot attacks, among others [33]. These attacks have to be evaluated and related defenses could be added on top of our solution. Note that we propose the first-ever hardware acceleration of the FALCON sampling procedure. The natural steps in this line of work are to first develop hardware/software solutions for algorithms and then to consider such attacks in follow-on studies. This is exemplified in many previous HOST papers, including the hardware design of discrete Gaussian samplers without implementation security [34], hardware design of lightweight cryptography algorithms without implementation security [35], and software design for fully homomorphic encryption without implementation security [36], among others.

B. Performance Comparison on Lower-End FPGA Devices

If the design is implemented on even lower-end devices that contain an embedded microcontroller without floating-point hardware support, the software side can take longer. In such cases, the floating point arithmetic (i.e., the entire sampling process) can be moved to FPGA for acceleration, provided that the FPGA contains sufficient space. This work demonstrates a novel hardware-software co-design method for accelerating FALCON's discrete Gaussian sampling sub-routine. This work does not aim to optimize and accelerate the AXI components because we argue that this optimization effort does not contribute to the novelty of the proposed method.

C. Significance of FPGA Acceleration for FALCON

Our work demonstrates that FPGA significantly enhances FALCON's performance by accelerating its Gaussian sampling sub-routine. Although the application of discrete Gaussian sampling with varying means and variances may seem narrow, it is a fundamental and computationally intensive sub-routine of FALCON. Therefore, a specialized FPGA acceleration is justified because it allows significant performance savings of this important algorithm chosen by NIST, as we quantified in this work. Such crypto-unique subroutine acceleration via FPGA is a common practice in cryptographic engineering as well as in real-world applications with many relevant publications including those at previous HOST conferences [34], [37].

VII. Conclusions

A massive effort has recently started to transition NIST's next-generation post-quantum encryption standards into practice. One important aspect of deploying these standards is designing efficient implementations. FALCON is one of the algorithms that NIST chose yet its implementation has been omitted in prior work. Implementing FALCON efficiently requires accelerating its discrete Gaussian sampling algorithm, which is non-trivial because it includes different components compared to other Gaussian samplers used in lattice cryptography. This paper demonstrates that a hardware-software co-design method is suitable for addressing both the efficiency and flexibility needs used in FALCON. Our solution accelerates the reference sampling software by 56.05$\times$, which corresponds to a total improvement of 1.67$\times$ for the signature generation. Further acceleration may be possible by improving the I/O communication overhead or by porting the design to a more capable FPGA that can execute at a higher operating frequency.

VIII. Acknowledgements

This research is supported in part by the National Science Foundation under Grant No. 2146881.

References

[1] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[3] P. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Review*, vol. 41, no. 2, pp. 303–332, 1999.

[4] J. Proos et al., "Shor's discrete logarithm quantum algorithm for elliptic curves," *Quantum Info. Comput.*, vol. 3, no. 4, pp. 317–344, Jul. 2003.

[5] J. Krämer and P. Struck, "Encryption schemes using random oracles: from classical to post-quantum security," in *International Conference on Post-Quantum Cryptography*. Springer, 2020, pp. 539–558.

[6] The National Institute of Standards and Technology, "Post-quantum cryptography pqc," https://csrc.nist.gov/projects/post-quantum-cryptography.

[7] P. A. Fouque et al., "Falcon: Fast-fourier lattice-based compact signatures over NTRU."

[8] J. Von Neumann, "13. various techniques used in connection with random digits," *Appl. Math Ser*, vol. 12, no. 36-38, p. 5, 1951.

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

[10] D. E. Knuth, K. DE, and Y. AC, "The complexity of nonuniform random number generation." 1976.

[11] C. Peikert, "An efficient and parallel gaussian sampler for lattices," in *Annual Cryptology Conference*. Springer, 2010, pp. 80–97.

[12] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehlé, and S. Bai, "Crystals-dilithium," *Submission to the NIST Post-Quantum Cryptography Standardization [NIS]*, 2017.

[13] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The sphincs+ signature framework," Cryptology ePrint Archive, Paper 2019/1086, 2019, https://eprint.iacr.org/2019/1086. [Online]. Available: https://eprint.iacr.org/2019/1086

[14] E. Karabulut, E. Alkim, and A. Aysu, "Efficient, flexible, and constant-time gaussian sampling hardware for lattice cryptography," *IEEE Transactions on Computers*, 2021.

[15] *7 Series DSP48E1 Slice*, Xilinx Inc, 3 2018, v1.10.

[16] T. Prest, "falcon.py," 12 2022. [Online]. Available: https://github.com/tprest/falcon.py.git

[17] J. Howe and B. Westerbaan, "Benchmarking and analysing nist pqc lattice-based signature scheme standards on the arm cortex m7."

[18] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, Z. Zhang *et al.*, "Fast-fourier lattice-based compact signatures over ntru," https://falcon-sign.info/.

[19] S. Tian, W. Wang, and J. Szefer, "Merge-exchange sort based discrete gaussian sampler with fixed memory access pattern," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 126–134.

[20] L. Kong, R. Liu *et al.*, "High-performance constant-time discrete gaussian sampling," *IEEE Transactions on Computers*, 2020.

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

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

[23] T. Pöppelmann and T. Güneysu, "Towards practical lattice-based public-key encryption on reconfigurable hardware," in *International Conference on Selected Areas in Cryptography*. Springer, 2013, pp. 68–85.

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

[25] C. Zhang, Z. Liu, Y. Chen, J. Lu, and D. Liu, "A flexible and generic gaussian sampler with power side-channel countermeasures for quantum-secure internet of things," *IEEE Internet of Things Journal*, 2020.

[26] D. Liu, C. Zhang, H. Lin, Y. Chen, and M. Zhang, "A resource-efficient and side-channel secure hardware implementation of ring-lwe cryptographic processor," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 4, pp. 1474–1483, 2018.

[27] T. Pöppelmann and T. Güneysu, "Area optimization of lightweight lattice-based encryption on reconfigurable hardware," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014, pp. 2796–2799.

[28] J. Howe, T. Oder, M. Krausz, and T. Güneysu, "Standard lattice-based key encapsulation on embedded devices," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 372–393, 2018.

[29] P. Karl, J. Schupp, T. Fritzmann, and G. Sigl, "Post-quantum signatures on risc-v with hardware acceleration," Cryptology ePrint Archive, Paper 2022/538, 2022, https://eprint.iacr.org/2022/538. [Online]. Available: https://eprint.iacr.org/2022/538

[30] K. Kiningham, P. Levis, M. Anderson, D. Boneh, M. Horowitz, and M. Shih, "Falcon — a flexible architecture for accelerating cryptography," in *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, 2019, pp. 136–144.

[31] D. Soni, K. Basu, M. Nabeel, N. Aaraj, M. Manzano, and R. Karri, *FALCON*. Cham: Springer International Publishing, 2021, pp. 31–41. [Online]. Available: https://doi.org/10.1007/978-3-030-57682-0_3

[32] W.-K. Lee, R. K. Zhao, R. Steinfeld, A. Sakzad, and S. O. Hwang, "High throughput lattice-based signatures on gpus: Comparing falcon and mitaka," *Cryptology ePrint Archive*, 2023.

[33] M. Devi and A. Majumder, "Side-channel attack in internet of things: A survey," in *Applications of Internet of Things: Proceedings of ICCCIOT 2020*. Springer, 2021, pp. 213–222.

[34] R. Agrawal, L. Bu, and M. A. Kinsy, "A post-quantum secure discrete gaussian noise sampler," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 295–304.

[35] A. A. Ayoub and M. D. Aagaard, "Application-specific instruction set architecture for an ultralight hardware security module," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 69–79.

[36] T. Morshed, M. M. A. Aziz, and N. Mohammed, "Cpu and gpu accelerated fully homomorphic encryption," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 142–153.

[37] T. White, C. Gouert, C. Yang, and N. G. Tsoutsos, "Fhe-booster: Accelerating fully homomorphic execution with fine-tuned bootstrapping scheduling," in *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2023, pp. 293–303.