Parameterized Hardware Accelerators for Lattice-Based Cryptography and Their Application to the HW/SW Co-Design of qTESLA

Wen Wang[1], Shanquan Tian[1], Bernhard Jungk[2], Nina Bindel[3], Patrick Longa[4] and Jakub Szefer[1]

[1] Yale University, USA
{wen.wang.ww349,shanquan.tian,jakub.szefer}@yale.edu

[2] MAN Truck & Bus SE, Germany
bernhard.jungk@man.eu

[3] University of Waterloo, Canada
nlbindel@uwaterloo.ca

[4] Microsoft Research, USA
plonga@microsoft.com

Abstract. This paper presents a set of efficient and parameterized hardware accelerators that target post-quantum lattice-based cryptographic schemes, including a versatile cSHAKE core, a binary-search CDT-based Gaussian sampler, and a pipelined NTT-based polynomial multiplier, among others. Unlike much of prior work, the accelerators are fully open-sourced, are designed to be constant-time, and can be parameterized at compile-time to support different parameters without the need for re-writing the hardware implementation. These flexible, publicly-available accelerators are leveraged to demonstrate the first hardware-software co-design using RISC-V of the post-quantum lattice-based signature scheme qTESLA with provably secure parameters. In particular, this work demonstrates that the NIST's Round 2 level 1 and level 3 qTESLA variants achieve over a 40-100x speedup for key generation, about a 10x speedup for signing, and about a 16x speedup for verification, compared to the baseline RISC-V software-only implementation. For instance, this corresponds to execution in 7.7, 34.4, and 7.8 milliseconds for key generation, signing, and verification, respectively, for qTESLA's level 1 parameter set on an Artix-7 FPGA, demonstrating the feasibility of the scheme for embedded applications.

Keywords: Lattice-based cryptography · Post-quantum cryptography · qTESLA · Hardware accelerators · Hardware-software co-design · FPGA · RISC-V

1 Introduction

Most common cryptographic protocols such as RSA and ECC will become insecure once a sufficiently large and fault-tolerant quantum computer is built with the capability to run Shor's algorithm [47] and its variants. To address this security threat, cryptographers have been working on alternative algorithms which are not known to be vulnerable to attacks using quantum computers – the so-called post-quantum or quantum-safe cryptographic algorithms. In order to advance this effort, in 2017 NIST launched a new standardization process with the goal of selecting the next generation of quantum-safe public-key cryptographic algorithms [36]. The initial phase of this process, which ended with the selection of 17 key encapsulation mechanisms (KEMs) and 9 digital signature schemes, mainly focused on aspects related to security and cryptanalysis, and gave much less emphasis to efficiency characteristics. However, as NIST's Round 2 progresses since its start in 2019, performance analysis of the various candidates on different software and hardware platforms is becoming a more crucial aspect of the evaluation.

Among the various post-quantum families, lattice-based cryptography [1,44] represents one of the most promising and popular alternatives. For instance, from the 9 NIST Round 2 digital signature candidates that were selected, 3 belong to this cryptographic family: Dilithium [32], Falcon [43], and qTESLA [9]. For this work, we selected qTESLA, which is a signature scheme based on the hardness of the ring learning with errors (R-LWE) problem that comes with built-in defenses against some implementation attacks such as simple side-channel and fault attacks, and against key substitution (KS) attacks [3]. Since instantiations of qTESLA are *provably-secure* by construction, the signature scheme enjoys an important security guarantee: the security hardness of a given instantiation is *provably-guaranteed* as long as its corresponding R-LWE instance remains secure. This feature, however, comes at a price which is reflected in the larger sizes, especially of public keys, and a slower performance.

In this work, we focus on the development of efficient and flexible lattice-based cryptography accelerators and their application to realize the first hardware-software co-design of provably-secure instances of qTESLA using a RISC-V core[1]. Our work demonstrates that even demanding, provably-secure schemes can be realized efficiently with proper use of hardware-software co-design.

Related Work and Contributions. There are many hardware designs in the literature targeting the computing blocks that are necessary for the implementation of lattice-based systems, such as the Gaussian sampler and the number theoretic transform (NTT) [20, 41]. However, a recurrent issue is that most existing works, especially in the case of the NTT, are not fully scalable or parameterized and are, hence, limited to specific cryptographic schemes [27, 37, 41, 45]. Post-quantum cryptography (PQC), including lattice-based cryptography, is still an active research area and, as a consequence, there is a proliferation of schemes and a rapid evolution in the parameters that are used in practical instantiations, as can be observed in the ongoing NIST PQC standardization process. This issue is markedly problematic and expensive for hardware. Hence, unlike much of prior work, the accelerators developed in this work are designed to be fully parameterized at compile-time to help implement different parameters and support different lattice-based schemes.

Concretely, we present the following accelerators:

- a unified hardware core for both SHAKE-128/256 and cSHAKE-128/256,
- a novel, parameterized binary-search CDT sampler in hardware,
- a novel, hardware pipelined NTT-based polynomial multiplier, and
- a parameterized sparse polynomial multiplier in hardware.

Additionally, we provide a lightweight Hmax-Sum hardware module for qTESLA.

These flexible accelerators are then used to realize the first RISC-V based hardware-software co-design of qTESLA with the provably-secure parameter sets. This successfully

[1]https://github.com/SpinalHDL/VexRiscv/

demonstrates the significant impact of offloading complex functions from software to hardware accelerators. The modules are fully parameterized and, hence, allow us to quickly change parameters and re-synthesize the design. For example, in our design, it is made easy to switch from qTESLA's Round 2 provably-secure parameters to prior heuristic parameters, if desired.

In addition, in contrast to most existing hardware designs, the full hardware code developed in this work is publicly released and available at https://caslab.csl.yale.edu/code/qtesla-hw-sw-platform.

Finally, a relevant feature of our design is the use of a simple and standard 32-bit interconnect to the microcontroller. This design feature aims at providing platform flexibility and showing that hardware accelerators can achieve good performance even with this conservative choice. This is different from many designs proposed in the literature. At an extreme end, some hardware designs for lattice-based cryptography focus on low power and low cycle counts, at the cost of custom interfaces and very low clock frequencies. In particular, [6] proposes a design that achieves a very low cycle count for lattice-based schemes, but this is in part due to the use of a single-cycle bus interface, and not due to the accelerator hardware design itself. Despite the low cycle count, the actual execution time in [6] is higher than comparable operations in our design due to their longer critical path delay. Other designs use standard interfaces and a hardware-software co-design approach, but are not flexible in the choice of parameters [16]. We remark that since our hardware-software co-design is open-sourced, users can also improve the performance further by using a specialized bus or interconnect.

Among recent existing work with similar goals to ours, Banerjee et al. [6] proposed Sapphire, a configurable lattice crypto-processor coupled with a RISC-V processor that has been tested on an ASIC using several NIST candidates. Sapphire supports qTESLA, but the results correspond to outdated parameters that are no longer part of the NIST PQC process. Also, our implementation of qTESLA's key generation performs much better because Banerjee et al.'s Gaussian sampler is based on a merge-sort CDT algorithm that is intended for software-use only. Furthermore, our design takes advantage of the faster sparse polynomial multiplications.

Farahmand et al. [16] proposed a hardware-software co-design architecture to benchmark various lattice-based KEMs. To speed up the design process they use the popular Zynq UltraScale+ SoC which contains a hard processor core coupled to the FPGA fabric. Thus, they benefit from the high working clock frequencies of the Arm processor built into the FPGA. However, their work only supports designs with modulus q being a power-of-two or NTRU-based KEMs. Hence, their arithmetic blocks do not support any of the Round 2 lattice-based digital signature candidate proposals. Furthermore, they only include a simple schoolbook multiplier.

Paper Outline. We first review the lattice-based signature scheme qTESLA and explain the selection of our functions to be implemented in hardware in Section 2. Afterwards, we present the five proposed hardware accelerators in Section 3. Section 4 presents an in-depth experimental evaluation of each hardware module and the final qTESLA design, as well as a comparison to the state-of-the-art. We conclude this paper in Section 5.

2 Preliminaries

This section gives a brief introduction to the lattice-based signature scheme qTESLA. It also provides background on software profiling used to identify most suitable functions for hardware acceleration, and gives details about these functions.

2.1 qTESLA

qTESLA is a provably-secure post-quantum signature scheme, based on the hardness of the decisional R-LWE problem [9]. The scheme is based on the "Fiat-Shamir with Aborts" framework by Lyubashevsky [31] and is an efficient variant of the Bai-Galbraith signature scheme [5] adapted to the setting of ideal lattices. A distinctive feature of qTESLA is that its parameters are *provably secure*, i.e., they are generated according to the security reduction from R-LWE.

Notation. We define the rings $\mathcal{R} = \mathbb{Z}[x]/\langle x^n + 1\rangle$ and $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, where n is the dimension and $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ for a prime modulus $q \equiv 1 \bmod 2n$. We further define the sets $\mathbb{H}_{n,h} = \{\sum_{i=0}^{n-1} f_i x^i \in \mathcal{R} \mid f_i \in \{-1, 0, 1\}, \sum_{i=0}^{n-1} |f_i| = h\}$ and $\mathcal{R}_{q,[B]} = \{\sum_{i=0}^{n-1} f_i x^i \in \mathcal{R} \mid f_i \in [-B, B]\}$ for fixed system parameters h and B. For some even (odd) modulus $m \in \mathbb{Z}_{\geq 0}$ and an element $c \in \mathbb{Z}$, $c' = c \bmod^{\pm} m$ denotes the unique element $-m/2 < c' \leq m/2$ (resp., $-\lfloor m/2 \rfloor \leq c' \leq \lfloor m/2 \rfloor$) with $c' = c \bmod m$. We also define the rounding functions $[\cdot]_L : \mathbb{Z} \to \mathbb{Z},\ c \mapsto (c \bmod^{\pm} q) \bmod^{\pm} 2^d$, and $[\cdot]_M : \mathbb{Z} \to \mathbb{Z},\ c \mapsto (c \bmod^{\pm} q - [c]_L)/2^d$ for a fixed system parameter d. These definitions are extended to polynomials by applying the operators to each polynomial coefficient, i.e., $[f]_L = \sum_{i=0}^{n-1} [f_i]_L\, x^i$ and $[f]_M = \sum_{i=0}^{n-1} [f_i]_M\, x^i$ for a given $f = \sum_{i=0}^{n-1} f_i x^i \in \mathcal{R}$. Given $f \in \mathcal{R}$, we define the function $\max_i(f)$ which returns the i-th largest absolute coefficient of f. For an element $c \in \mathbb{Z}$, we have that $\|c\|_\infty = |c \bmod^{\pm} q|$, and define the infinity norm for a polynomial $f \in \mathcal{R}$ as $\|f\|_\infty = \max_i \|f_i\|_\infty$. To denote sampling each coefficient of a polynomial f with centered discrete Gaussian distribution $\mathcal{D}_\sigma$ with standard deviation σ, we write $f \leftarrow_\sigma \mathcal{R}$.

Besides the number of polynomial coefficients n and the modulus q, the R-LWE setup also involves defining the number of R-LWE samples that are used by the scheme instantiation, which we denote by k. The values E and S define the coefficient bounds for the error and secret polynomials, B determines the interval from which the random coefficients of the polynomial y are chosen during signing, and $b_{\mathsf{GenA}} \in \mathbb{Z}_{>0}$ represents the number of blocks requested in the first cSHAKE call during generation of the so-called public polynomials $a_1, \ldots, a_k$ [9]. Finally, we define two additional system parameters: λ, which denotes the targeted bit-security of a given instantiation, and κ, which denotes the input and output bit length of the hash and pseudo-random functions (PRFs).

The signature scheme qTESLA. qTESLA is parameterized by λ, κ, n, k, q, σ, E, S, B, d, h, and b_{GenA}, discussed above. The pseudo-code of qTESLA's key generation, sign and verify algorithms are presented in Algorithms 1, 2 and 3, respectively. A brief description of the algorithms, highlighting the most important operations of the scheme, follows. For complete information and details about the different qTESLA functions, readers are referred to [9].

From a computational perspective, as in the case of other lattice-based schemes, qTESLA's main operations are polynomial operations defined over the rings $\mathcal{R}$ and $\mathcal{R}_q$. Consequently,

Algorithm 1 qTESLA's key generation [9]

```
Require: -
Ensure: $sk = (s, e_1, \ldots, e_k, \mathsf{s}_a, \mathsf{s}_y, g)$ and
    $pk = (t_1, \ldots, t_k, \mathsf{s}_a)$
1: counter $\leftarrow 1$
2: seed $\leftarrow_\$ \{0,1\}^\kappa$
3: $\mathsf{s}_s, \mathsf{s}_{e_1}, \ldots, \mathsf{s}_{e_k}, \mathsf{s}_a, \mathsf{s}_y \leftarrow \mathsf{PRF}_1(\mathsf{seed})$
4: $a_1, \ldots, a_k \leftarrow \mathsf{GenA}(\mathsf{s}_a)$
5: do
6:     $s \leftarrow \mathsf{GaussSampler}(\mathsf{s}_s, \text{counter})$
7:     counter $\leftarrow$ counter $+ 1$
8: while $\mathsf{checkS}(s) \neq 0$
9: for $i = 1, \ldots, k$ do
10:     do
11:         $e_i \leftarrow \mathsf{GaussSampler}(\mathsf{s}_{e_i}, \text{counter})$
12:         counter $\leftarrow$ counter $+ 1$
13:     while $\mathsf{checkE}(e_i) \neq 0$
14:     $t_i \leftarrow a_i s + e_i \mod q$
15: $g \leftarrow \mathsf{G}(t_1, \ldots, t_k)$
16: $sk \leftarrow (s, e_1, \ldots, e_k, \mathsf{s}_a, \mathsf{s}_y, g)$
17: $pk \leftarrow (t_1, \ldots, t_k, \mathsf{s}_a)$
18: return $sk$, $pk$
```

Algorithm 2 qTESLA's sign [9]

```
Require: $m$, $sk = (s, e_1, \ldots, e_k, \mathsf{s}_a, \mathsf{s}_y, g)$
Ensure: $(z, c')$
1: counter $\leftarrow 1$
2: $r \leftarrow_\$ \{0,1\}^\kappa$
3: rand $\leftarrow \mathsf{PRF}_2(\mathsf{s}_y, r, \mathsf{G}(m))$
4: $y \leftarrow \mathsf{ySampler}(\mathsf{rand}, \text{counter})$
5: $a_1, \ldots, a_k \leftarrow \mathsf{GenA}(\mathsf{s}_a)$
6: for $i = 1, \ldots, k$ do
7:     $v_i = a_i y \bmod^{\pm} q$
8: $c' \leftarrow \mathsf{H}(v_1, \ldots, v_k, \mathsf{G}(m), g)$
9: $c \triangleq \{pos\_list, sign\_list\} \leftarrow \mathsf{Enc}(c')$
10: $z \leftarrow y + sc$
11: if $z \notin \mathcal{R}_{q,[B-S]}$ then
12:     counter $\leftarrow$ counter $+ 1$
13:     Restart at step 4
14: for $i = 1, \ldots, k$ do
15:     $w_i \leftarrow v_i - e_i c \bmod^{\pm} q$
16:     if $\|[w_i]_L\|_\infty \geq 2^{d-1} - E \vee \|w_i\|_\infty \geq \lfloor q/2 \rfloor - E$ then
17:         counter $\leftarrow$ counter $+ 1$
18:         Restart at step 4
19: return $(z, c')$
```

a main focus of this work is the efficient implementation of the NTT, which allows an efficient realization of the expensive polynomial multiplications, because $q \equiv 1 \bmod 2n$ for qTESLA scheme.

Another important operation is Gaussian sampling, which for qTESLA is used only for key generation. Gaussian sampling is used to generate the secret and error polynomials in $\mathcal{R}$ with centered discrete Gaussian distribution $\mathcal{D}_\sigma$. The polynomials produced by the Gaussian sampler (denoted by GaussSampler) have to pass two security checks, namely, checkE and checkS, which make sure that $\sum_{i=1}^{h} \max_i(f)$ (called Hmax-Sum in the remainder) is less than or equal to the fixed bounds E and S, respectively. For the generation of the public keys, we need to derive the public polynomials $a_1, \ldots, a_k \in \mathcal{R}_q$. This operation is denoted by the function $\mathsf{GenA} : \{0,1\}^\kappa \to \mathcal{R}_q^k$. The random seed s_a that is used to generate the public polynomials is transmitted to the signing and verification algorithms through the secret and public keys, respectively. We highlight that the fresh generation of $a_1, \ldots, a_k$ using a random seed saves bandwidth, makes the introduction of backdoors more difficult and minimizes the impact of all-for-the-price-of-one attacks [9]. We also point out that the secret key includes a value denoted by g, which is the hash of the polynomials $t_1, \ldots, t_k$ (which are part of the public key), computed via the function $\mathsf{G} : \{0,1\}^* \to \{0,1\}^{320}$ [3]. This is then used during the hashing operation to derive the challenge value c' at signing. This design feature protects against key substitution attacks [10], by guaranteeing that any attempt by an attacker of modifying the public key will be detected during verification when checking c'.

During signing, the sampling function ySampler samples a polynomial $y \in \mathcal{R}_{q,[B]}$. To produce the randomness rand used to generate y, one uses a secret-key value s_y and some fresh randomness r. The use of s_y makes qTESLA resilient to fixed-randomness attacks such as the one demonstrated against Sony's Playstation 3 [12], and the random value r guarantees the use of a fresh y at each signing operation, which makes qTESLA's

Algorithm 3 qTESLA's verify [9]

```
Require: $m$, $(z, c')$, $pk = (t_1, \ldots, t_k, \mathsf{s}_a)$
Ensure: $\{0, -1\}$ ▷ accept, reject signature
1: $c \triangleq \{pos\_list, sign\_list\} \leftarrow \mathsf{Enc}(c')$
2: $a_1, \ldots, a_k \leftarrow \mathsf{GenA}(\mathsf{s}_a)$
3: for $i = 1, \ldots, k$ do
4:     $w_i \leftarrow a_i z - t_i c \bmod^{\pm} q$
5: if $z \notin \mathcal{R}_{q,[B-S]} \vee c' \neq \mathsf{H}(w_1, \ldots, w_k, \mathsf{G}(m), \mathsf{G}(t_1, \ldots, t_k))$ then
6:     return $-1$
7: return 0
```

signatures *probabilistic* and, hence, more difficult to attack through side-channel analysis. In addition, the fresh y protects against some powerful fault attacks against deterministic signature schemes [11, 39]. Signing and verification also require the generation of the challenge c' by using the hash-based function H, which computes $[v_1]_M, \ldots, [v_k]_M$ for some polynomials v_i (or w_i during verification) and hashes these together with the digests $\mathsf{G}(m)$ and $\mathsf{G}(t_1, ..., t_k)$. This value is then mapped deterministically (using the function Enc) to a pseudo-randomly generated polynomial $c \in \mathbb{H}_{n,h}$ which is encoded as the two arrays $pos\_list \in \{0, \ldots, n-1\}^h$ and $sign\_list \in \{-1, 1\}^h$ representing the positions and signs of the nonzero coefficients of c, respectively. At signing, in order for the *potential* signature $(z \leftarrow sc + y, c')$ to be returned by the signing algorithm, it needs to pass a *security* check, which verifies that $z \notin \mathcal{R}_{q,[B-S]}$, and a *correctness* check, which verifies that $\| [w_i]_L \|_\infty < 2^{d-1} - E$ and $\|w_i\|_\infty < \lfloor q/2 \rceil - E$. At verification, if for a given signature (z, c') it holds that $z \in \mathcal{R}_{q,[B-S]}$ and c' matches the value computed using the function H as described above, the signature is accepted; otherwise, it is rejected.

Hashing and pseudo-random generation are required by several computations in the scheme. This functionality is provided by the extendable output functions SHAKE [15], in the realization of the functions G and H, and cSHAKE [26], in the realization of the functions PRF_1, PRF_2, $\mathsf{ySampler}$, $\mathsf{GaussSampler}$, GenA and Enc. Although implementers are free to pick a cryptographic PRF of their choice to implement PRF_1, PRF_2, $\mathsf{ySampler}$, and $\mathsf{GaussSampler}$, we chose to reuse the same (c)SHAKE core to also support these functions in order to save area. According to the specifications [9], the use of cSHAKE-128 is fixed for GenA and Enc. For the remaining functions, level 1 and level 3 parameter sets use (c)SHAKE-128 and (c)SHAKE-256, respectively.

	λ	κ	n	k	q	σ	h	$E = S$	B	d	b_{GenA}
qTESLA-p-I	95	256	1024	4	$343,576,577$	8.5	25	554	$2^{19}-1$	22	108
qTESLA-p-III	160	256	2048	5	$856,145,921$	8.5	40	901	$2^{21}-1$	24	180

Table 1: Parameters of the two qTESLA parameter sets (Round 2).

Parameter Sets. qTESLA's NIST PQC submission for Round 2 includes two parameter sets: qTESLA-p-I and qTESLA-p-III, which target NIST security levels 1 and 3, respectively, and are assumed to provide post-quantum security equivalent to AES-128 and AES-192, respectively. We recall the instantiations with their relevant parameters in Table 1. The parameters for qTESLA-p-I lead to a signature, public key and secret key of $2,592$ bytes, $14,880$ bytes and $5,224$ bytes, respectively. The corresponding figures for qTESLA-p-III are $5,664$ bytes, $38,432$ bytes and $12,392$ bytes, respectively.

2.2 Basis Software Implementation

In our design, we used qTESLA's most recent portable C reference implementation that was submitted to the NIST PQC standardization process (Round 2) as the basis software implementation[2]. It is a state-of-the-art 32/64-bit software implementation of qTESLA, targeting a low clock cycle count. This is the fastest reference software implementation of qTESLA we are aware of. We chose the definitions of the targeted architecture and basic data types to ensure that the code runs correctly on 32-bit architectures (i.e., on our RISC-V target) and we used the available compiler flags to enable the highest optimization levels of the GCC compiler. We remark that further optimizations using low-level, hand-written assembly code will probably lead to faster code. As a reference point, assembly optimizations using vector instructions on an x64 Intel processor achieve a $1.5\times$ speedup in comparison to the reference implementation of qTESLA [3]. However, we caution readers that this improvement is obtained on a platform completely different to the RISC-V platform targeted in this work. Moreover, speedups obtained with assembly optimizations are also highly dependent, among other aspects, on the particular cryptographic scheme. For example, based on the benchmarking results on an ARM Cortex-M4 platform [25], when using low-level, hand-written assembly optimizations, Kyber-768 [46] achieves around $1.5\times$ speedup while the speedup for ntruhrss701 [52] is over $11\times$ (compared to their respective portable C reference implementations). For this work, developing such assembly optimizations for RISC-V is considered out of scope.

2.3 Software Profiling

To determine potential functions for promising speedups using hardware acceleration, we profiled qTESLA's reference software implementation. We profiled the code with `gprof` on a 3.4GHz Intel Core i7-6700 (Skylake) CPU with TurboBoost disabled. As a result, we found that the two most expensive operations are (c)SHAKE and the NTT-based polynomial multiplication: about 39.4% of the computing time is spent by the Keccak function performing cSHAKE and SHAKE computations, and about 27.9% of the time is spent by the polynomial multiplier performing NTT computations. Other costly operations include the sparse polynomial multiplications (6.3% of the total cost) and the Gaussian sampler (4.5% of the total cost). Accordingly, these four functions were selected for hardware acceleration. Interestingly, after acceleration, we discovered that the Hmax-Sum function became a new bottleneck, and it was accelerated as well. This highlights the importance of repeated profiling in order to reassess the performance of functions that are originally considered inexpensive.

2.4 Functions Selected for Acceleration

Based on the profiling results in Section 2.3, we designed accelerators for (c)SHAKE, NTT-based polynomial multiplier, Gaussian sampler, sparse multiplication and Hmax-Sum. The first 3 of these functions are also targeted because they are commonly found in lattice-based cryptography and can be used to accelerate other cryptographic schemes.

(c)SHAKE. SHAKE [15] and cSHAKE [26] are extendable output functions (XOF) based on the Keccak algorithm [7, 8], which is also the basis of NIST's SHA-3 standard [15]. XOFs are similar to hash functions, but while hash functions only produce a fixed length output, XOFs produce a variable amount of output bits.

[2]The software is available at `https://github.com/qtesla/qTesla`, `commit-id d8fd7a5`.

Keccak is a parameterizable sponge function, where b denotes the state size, r the rate, and c the capacity and $b = r + c$. For current NIST algorithms based on Keccak, the state is set to $b = 25 \times 2^6 = 1600$, while c (and r) vary. Therefore, NIST's algorithms are usually described in the form $\text{Keccak}[c](\text{message}, \text{outputlength})$. For SHAKE128 and cSHAKE128, $c = 128$ and for the other two variants $c = 256$. Based on Keccak, they can be defined as:

$$\begin{aligned}\text{SHAKE-c}(M, d) =& \text{Keccak}[c](M||1111, d)\\ \text{cSHAKE-c}(M, N, S, d) =& \text{Keccak}[c](\text{bytepad}(\text{encode\_string}(N)||\\ & \text{encode\_string}(S), c/8)||M||00, d).\end{aligned}$$

N is a so-called function-name string, defined by NIST, and S a customization bit string. It is further defined that, if N and S are empty strings, cSHAKE = SHAKE.

Sponge functions such as Keccak have an absorption and a squeezing phase. In the absorption phase r bits are combined with the internal state using XOR, followed by a computation of the internal Keccak permutation. Hence, if n bits have to be absorbed, $\lceil \frac{n}{r} \rceil$ absorptions have to be performed. Similarly, in the squeezing phase r bits of output are produced, followed by one or more executions of the Keccak permutation, if more than r bits are requested. Due to this general design, it is possible to use the same implementation of the internal permutation for all needed SHAKE and cSHAKE implementations, if the inputs are properly prepared and padded.

Polynomial Multiplication. Setting $q \equiv 1 \bmod 2n$ enables the use of the efficient NTT for polynomial multiplication, which we define next.

Let ω and ϕ be primitive n-th and $2n$-th roots of unity in $\mathbb{Z}_q$, respectively, where $\phi^2 = \omega$. Then, for a polynomial $c = \sum_{i=0}^{n-1} c_i x^i$ the forward NTT transform is defined as

$$\mathsf{NTT} : \mathcal{R} = \mathbb{Z}_q[x]/\langle x^n + 1\rangle \to \mathcal{R}_q, \quad c \mapsto \tilde{c} = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} c_j \phi^j \omega^{ij} \right) x^i.$$

Likewise, the inverse NTT transform is defined as

$$\mathsf{NTT}^{-1} : \mathcal{R}_q \to \mathcal{R} = \mathbb{Z}_q[x]/\langle x^n + 1\rangle, \quad \tilde{c} \mapsto c = \sum_{i=0}^{n-1} \left(n^{-1}\phi^{-i} \sum_{j=0}^{n-1} \tilde{c}_j \omega^{-ij} \right) x^i.$$

In qTESLA, the NTT is used to carry out the polynomial multiplications in line 7 of Algorithm 2 and in line 4 of Algorithm 3. In particular, given that public polynomials $a_1, \ldots, a_k$ are assumed to be generated directly in the NTT domain, multiplications have the form $a_i \cdot b$ in $\mathcal{R}_q$, for some $b \in \mathcal{R}_q$, and can be computed as $\mathsf{NTT}^{-1}(a_i \circ \mathsf{NTT}(b))$, where $\circ$ is the coefficient-wise multiplication.

Sparse Polynomial Multiplication. In addition to standard polynomial multiplications which are dealt with the NTT, qTESLA also performs polynomial multiplications with the sparse polynomial $c \in \mathbb{H}_{n,h}$, in lines 10 and 15 of Algorithm 2 and in line 4 of Algorithm 3. Recall that c is encoded as two lists *pos_list* and *sign_list* $\in \{-1, 0, 1\}^h$ which represent the positions and signs of its nonzero coefficients, respectively. These multiplications can be specialized with an algorithm that exploits the sparseness; see [3, Alg. 11].

Parameter set	CDT parameters
qTESLA-p-I	64 : 63 : 78 : 624
qTESLA-p-III	128 : 125 : 111 : 1776

Table 2: CDT parameters used in qTESLA's Round 2 implementation (targeted precision β : implemented precision in bits : number of rows t : table size in bytes).

Discrete Gaussian Sampler. Discrete Gaussian samplers are parameterized by the precision of the samples (which we denote by β), the standard deviation σ of the Gaussian distribution, and the tail-cut τ, such that the range of the samples is $[-\sigma\tau, \sigma\tau] \cap \mathbb{Z}$. There are several sampling techniques, such as reject, Bernoulli, Ziggurat, CDT, and Knuth-Yao. Among them, the cumulative distribution table (CDT) of the normal distribution [38] is one of the most efficient methods when σ is relatively small, as is the case in, e.g., the R-LWE encryption schemes by Lyubashevsky et al. [33] and by Linder and Peikert [28] and the NIST PQC candidates FrodoKEM [35] and qTESLA [9]. In addition, this method is also easy to implement securely in constant-time and avoids the need for floating point operations, which are especially expensive in hardware.

The method consists of pre-computing a table $\mathsf{CDT}[i] := \lfloor 2^{\beta} \Pr[c \leqslant i \mid c \leftarrow_{\sigma} \mathbb{Z}] \rfloor$ for $i \in [0, \ldots, t-1]$ *offline*, using the smallest t such that $\Pr[|c| \geqslant t \mid c \leftarrow_{\sigma} \mathbb{Z}] < 2^{-\beta}$. Then, during the *online* computation one picks a uniform sample $u \leftarrow_{\$} \mathbb{Z}/2^{\beta}\mathbb{Z}$ generated by a PRNG, scans the table, and finally returns the value s such that $\mathsf{CDT}[s] \leqslant u < \mathsf{CDT}[s+1]$. To cover the full sampling range, a random bit is used to assign the sign to the Gaussian sample s. Table 2 includes the specific CDT parameters used in qTESLA implementations.

Hmax-Sum. In qTESLA, after sampling a secret polynomial e_i or s during key generation, the polynomial has to be checked to see if the sum of its largest h coefficients is smaller than a pre-defined bound E or S. If the sum is smaller than the bound, then the sampled polynomial is accepted as valid. Otherwise, it is rejected and the procedure is repeated again. We denote this procedure as the Hmax-Sum function.

3 Hardware Acceleration

In this section, we describe the details of the proposed hardware modules.

3.1 SHAKE

Our SHAKE core is based on the scalable slice-oriented SHA-3 architecture introduced in [22, 23]. In our design, we extended the basic architecture to include the padding function and support for both cSHAKE and SHAKE with variable rate. As shown in [23], the architecture scales very well, depending on the number of slices processed per cycle. The slice-orientation allows several possibilities of folding the permutation by a factor of 2^l with $0 \leq l \leq 6$. With this strategy, the area is reduced, while an acceptable throughput and throughput-area ratio is maintained.

Our main goal in this work is to build a hardware accelerator which is directly connected to a processor core with a 32-bit interconnect, using its available standard interfaces. Therefore, we chose to explore the mid-range implementations since the extreme ends have several drawbacks in our use case. For the smallest cores, the main drawback is that they are quite slow (e.g., [17] reports execution in more than 18,000 cycles and [24] in

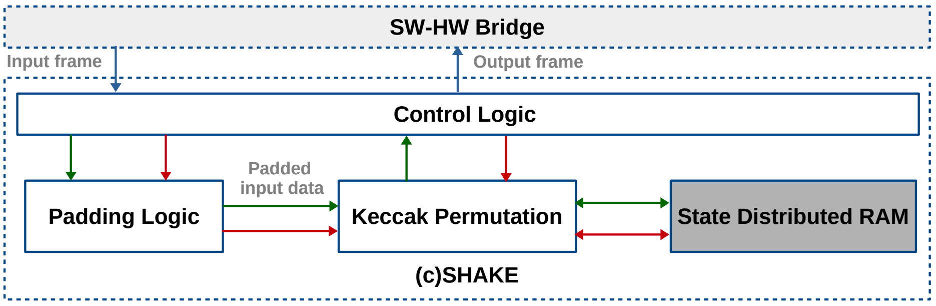

Figure 1: Dataflow diagram of the SHAKE hardware module. Red arrows represent control signals, green arrows represent data signals, and blue arrows represent the external I/O.

more than 2,600 cycles). For high-speed cores, a high amount of parallelism, unrolling or pipelining are used [19], which would waste lots of resources in our scenario given that the interconnect would be a bottleneck. For example, if a faster design such as the one from [49] is used to implement SHAKE-128, it would take at least $\frac{1344}{32} = 42$ cycles to load the data over a 32-bit wide interface, but only between 2 and 24 cycles for the processing itself. Consequently, for our design we chose the low-end to mid-range with $0 \leq l \leq 5$ (skipping $l = 6$, as in this case loading a new message block would take more time than the actual computation).

Our architecture is summarized in the dataflow diagram in Figure 1. In comparison to the original SHA-3 architecture [22, 23], the following major changes have been made:

- Support for cSHAKE and SHAKE, instead of SHA-3.
- Support for both 128-bit and 256-bit parameter sets.
- Direct integration of the padding functionality into the core.

Protocol. Our core communicates with the processor using a new protocol with several different 32-bit frames for data transmission:

- A *command frame* to distinguish between the four different operation modes cSHAKE-128, cSHAKE-256, SHAKE-128, and SHAKE-256. This command frame also specifies the output length generated by the SHAKE core.
- A *customization frame* to transfer the cSHAKE customization string to the core. Our implementation follows the cSHAKE-simple strategy and supports a 16-bit customization string [2].
- A *length frame*, which specifies the length of the incoming data block. This length information has to be either equal to the rate of the selected function, or less. If the block to be transferred is the last message block to be absorbed, an additional end flag in this length frame is set.
- A *message frame* that contains the message block to be absorbed. For a message block of length $m \leq r$, $\lceil \frac{m}{32} \rceil$ frames have to be transmitted.

The interface uses a handshake mechanism borrowed from AXI4-Lite [29] to implement the data transfer.

Control Logic. The control logic uses the incoming frames to control the padding logic, the permutation, and indirectly the distributed RAM used as state memory. If the core is idle and a command frame is received, the control logic switches to the appropriate internal state and expects as the next frame either the customization frame, if cSHAKE is requested, or the length frame. The rate r for the relevant variant and the requested output length d are stored internally. The rate r and the information, if SHAKE or cSHAKE has to be performed, is later used to calculate the number of bits to absorb per message block and the number of bits to squeeze. The information also controls the different encodings of the customization string and the padding (since SHAKE and cSHAKE use slightly different padding schemes).

If cSHAKE is requested, a customization frame is processed next. The necessary absorption phase for the customization string is quicker than absorbing a full message block. According to the cSHAKE encoding rules, the total length to be absorbed is only 64 bits for a 16-bit customization string. Therefore, it is possible to absorb the customization string in only $\frac{64}{2^l}$ cycles, independently of the actual rate, plus the time to execute the Keccak permutation once. After absorbing the string, the length frame is expected. A length frame describes how many message frames have to be transmitted to the SHAKE core and also if it is the last message block. Each message frame is directly absorbed, needing $\frac{r}{2^l}$ cycles per block, depending on the configuration of the core. If the last message frame is received, the SHAKE or cSHAKE padding, as well as Keccak's padding, is applied. Afterwards, the core automatically starts to squeeze out the requested amount of output data and sends it back to the processor. Each step in the squeezing phase consists of transferring r bits over the communication link, followed by one computation of the Keccak permutation, if more bits need to be squeezed.

Sending data back to the processor is much simpler, as it is only necessary to transfer the data in 32-bit chunks over the interface without any additional protocol overhead.

Padding Logic. The padding needs to fill up a message block to a multiple of the rate r. Since our core supports bit-wise input lengths, this leads to 25×2^l multiplexers, depending on the number of slices processed in parallel. These multiplexers switch between the input data, '0' and '1', depending on the length of the message to be absorbed. Beside the length of the message block, the output of the multiplexer depends on the selected operation mode of the SHAKE core, because the padding differs between SHAKE and cSHAKE. Additionally, if the padding does not fit into the message block, an extra message block needs to be absorbed.

Permutation. The implementation of the permutation follows the original slice-oriented design from [23]. In summary, the implementation uses the following ideas. Firstly, if 2^l slices are processed in parallel in each cycle, only a smaller part of the total Keccak permutation – namely, $\frac{2^l}{64}$ of the combinational logic – must be implemented, but then reiterated for $\frac{64}{2^l}$ cycles for a complete round. The required combinational logic is implemented in the permutation module, while the required bit-shuffling is implemented using an addressing scheme in the state RAM module.

One important complicating factor for the implementation is data dependencies between consecutive slices. These dependencies require that the permutation keep some internal state between consecutive clock cycles, and also between two consecutive rounds, which adds some overhead to the otherwise straightforward implementation of the combinational logic part.

Evaluation and Related Work. Table 3 shows the evaluation results for our SHAKE core and some state-of-the-art results from the literature. The approximate number of clock

Design	Features (Func./Pad./Standard IO)	Platform	Slices/LUTs/FFs	TP/Area (MBit/s/slice)	Fmax (MHz)
p=1	(c)SHAKE-128/256, ✓, ✓	Artix-7	249/811/490	0.29	178
p=2	(c)SHAKE-128/256, ✓, ✓	Artix-7	273/908/450	0.48	163
p=4	(c)SHAKE-128/256, ✓, ✓	Artix-7	312/1069/361	0.81	158
p=8	(c)SHAKE-128/256, ✓, ✓	Artix-7	404/1466/270	1.31	164
p=16	(c)SHAKE-128/256, ✓, ✓	Artix-7	657/2401/226	1.62	165
p=32	(c)SHAKE-128/256, ✓, ✓	Artix-7	1149/4436/180	1.79	161
p=1 [21]	SHA-3-256, —, ✓	Artix-7	172/—/—	0.42	179
p=2 [21]	SHA-3-256, —, ✓	Artix-7	207/—/—	0.62	159
p=4 [21]	SHA-3-256, —, ✓	Artix-7	247/—/—	1.19	179
p=8 [21]	SHA-3-256, —, ✓	Artix-7	293/—/—	1.61	145
p=8 [21]	SHA-3-256, ✓, ✓	Artix-7	482/—/—	2.03	180
p=16 [21]	SHA-3-256, —, ✓	Artix-7	463/—/—	2.1	150
p=32 [21]	SHA-3-256, —, ✓	Artix-7	900/—/—	1.99	138
[4]	SHA-3-256, —, —	Virtex-6	49/193/41	0.22	198
[4]	SHA-3-256, —, —	Virtex-6	60/174/71	0.42	426
[49]	SHA-3-256, —, —	Virtex-6	1432/—/—	10.33	327

Table 3: Performance of the proposed SHAKE hardware module and comparison with state-of-the-art related work.

cycles for SHAKE and cSHAKE is calculated as follows:

$$\begin{aligned}\text{cycles}_{\text{SHAKE}} &= 1 + \lceil\frac{m_1}{r}\rceil(1 + \frac{r+1600}{2^l}) + \lceil\frac{m_2}{r}\rceil(\frac{r+1600}{2^l}) - \frac{1600}{2^l}\\ \text{cycles}_{\text{cSHAKE}} &= \frac{64+1600}{2^l} + \text{cycles}_{\text{SHAKE}}\end{aligned}$$

where m_1 is the length of the message to be absorbed, r is the rate, $p = 2^l$ is the number of slices processed in parallel, and m_2 is the output length. Both m_1 and m_2 are given in bits. The number is only approximated, since it assumes that no extra message block for the padding is needed.

For the purpose of comparing the throughput with previous works on SHA-3, we assume that long messages are processed and only a short output with $m_2 < r$ is requested. Also, Table 3 only includes results corresponding to SHA-3-256's rate. As expected, the area consumption of our core goes up compared to the implementation reported in [21]. However, the general trend is very similar, with an offset between 70 and 249 slices, which is due to the increased feature set, i.e., the original core does not implement any padding functionality, and only includes one fixed hash function, namely SHA-3-256.

As expected, our design cannot compete with the smallest design from [4], nor with the high-throughput core from [49] in their respective benchmark categories. However, as mentioned earlier, both design targets would lead to a sub-optimal performance in our use case with a standard 32-bit interface, because either the processing time of a low-end core would not provide sufficient speed or a high-speed core would waste resources since it would spend most of the time waiting for new input. Overall, we can see that an extended feature set of a Keccak core can be implemented with a reasonable overhead.

Applicability to Other Cryptographic Schemes. SHAKE and cSHAKE are versatile crypto primitives with broad applications in cryptographic protocols. Importantly, similar to our qTESLA's profiling results in Section 2.3, SHAKE and cSHAKE have been found to be responsible for significant portions of the computing cost of several of the post-quantum

Algorithm 4 Binary-search CDT-based Gaussian sampler

```
Require: a random number $x$ of precision $\beta$ generated by a PRNG.
Ensure: a signed Gaussian sample $s$ of bit length $\lceil \log_2(t) \rceil + 1$.
  ▷ Fix pre-computed CDT table with $t$ entries of precision $\beta$.
  ▷ Split CDT into two power-of-two parts s.t. the first sub-table's last entry index "end_1" and
    the second sub-table's first entry index "first_2" are:
 1: $\text{end\_1} \leftarrow 2^{\lceil \log_2(t) \rceil - 1} - 1$
 2: $\text{first\_2} \leftarrow t - 2^{\lceil \log_2(t) \rceil - 1}$
 3: $\text{sign} \leftarrow \text{MSB}(x)$, $\text{MSB}(x) \leftarrow 0$
 4: if $x < \mathsf{CDT}[\text{end\_1} + 1]$ then
 5:     $\min \leftarrow 0,\ \max \leftarrow \text{end\_1} + 1$          // To search sub-table $[0, \text{end\_1}]$
 6: else
 7:     $\min \leftarrow \text{first\_2},\ \max \leftarrow t$          // To search sub-table $[\text{first\_2}, t - 1]$
 8: while $\min + 1 \neq \max$ do
 9:     if $x < \mathsf{CDT}[(\min + \max)/2]$ then
10:         $\max \leftarrow (\min + \max)/2$
11:     else
12:         $\min \leftarrow (\min + \max)/2$
13: return $s \leftarrow \text{sign}\ ?\ (-\min)\ :\ (\min)$
```

schemes in the NIST PQC process, including FrodoKEM [35], Saber [13], NewHope [40], Kyber [46], and others. Thus, our SHAKE core offers a flexible and efficient architecture with different area and performance trade-offs that can be easily used to accelerate the hash and XOF computations of (post-quantum) schemes for different applications.

3.2 Gaussian Sampler

As discussed in Section 2.4, we chose a CDT-based Gaussian sampler for our design due to its simplicity and efficiency in hardware when the standard deviation σ is relatively small. This sampler can be implemented with different search algorithms, such as full-scan search, binary search, and others. Since binary search does not run in constant-time on general-purpose computers due to the presence of cache memory, qTESLA's software implementation [3] employs full-scan search to prevent timing and cache attacks. However, for hardware implementations, by exploiting the fact that the memory access time is fixed and constant, we can speed up the CDT-based Gaussian sampler by use of binary search.

We present our novel *time-invariant* CDT-based Gaussian sampling algorithm using binary search in Algorithm 4. In the algorithm, CDT is a pre-computed table intended to be saved into a memory block in hardware, the input to the Gaussian sampler is a random number x of precision β generated by a PRNG, and the output is a signed Gaussian sample s of width $\lceil \log_2(t) \rceil + 1$, where t is the depth of CDT; see Table 2. The sign is determined by the most significant bit of x. The basic idea of the algorithm is to use the CDT table to fix two overlapping "power-of-two" sub-tables with the same size $2^{\lceil \log_2(t) \rceil - 1}$, and then run a binary search in which the first, lower-address sub-table is given priority. For example, for $t = 624$ the CDT table is split into the sub-tables of ranges $[0, 511]$ and $[112, 623]$. The former table is given priority and, hence, inputs falling in the overlapping range execute binary search on it. Since memory access time is constant in our setting and the sampler runs the same number of iterations for all possible inputs, the algorithm is protected against timing attacks.

Figure 2 depicts the hardware architecture of our discrete Gaussian sampler GaussSampler,

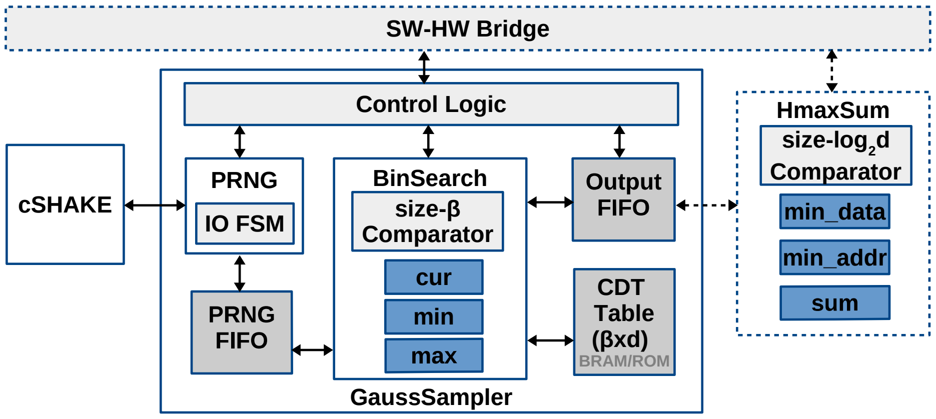

Figure 2: Dataflow diagram of the GaussSampler and HmaxSum hardware modules. The HmaxSum module can be conditionally added in the design to accelerate the qTESLA computations.

which fetches uniform random numbers from the cSHAKE-based PRNG and outputs samples to the outside modules. One PRNG FIFO is added in the design to buffer the input random numbers. Similarly, one Output FIFO is added and used to buffer the output samples. All the interfaces between the sub-modules are all implemented in AXI-like format. This ensures that these sub-modules can easily communicate and coordinate the computations with each other by following the same handshaking protocol. Our GaussSampler module is implemented in a fully parameterized fashion: users can freely tune the design parameters β, σ and τ depending on their scheme. Details of the sub-modules in Figure 2 follow next.

Control Logic. When a valid request is received by the Control Logic, it immediately triggers the PRNG module to generate new random numbers. When these random numbers are generated, they are fed into the PRNG FIFO. Once there are values in the FIFO, the Control Logic starts the binary search step by raising the start input signal in BinSearch. After a valid sample gets generated by BinSearch, it is further sent to the Output FIFO. The samples in the Output FIFO are further read by the outside modules. By introducing the input and output FIFOs in the design, we can make sure that PRNG can keep generating new pseudo-random numbers while BinSearch is working on the binary search computations based on the previously generated random numbers. The computations of different sub-modules are easily and well coordinated by handshaking with each other through their AXI-like interfaces.

cSHAKE PRNG. The PRNG module uses the SHAKE module described in Section 3.1 to generate secure pseudo-random numbers. This module accepts a string seed as input data, which is further fed to the SHAKE module together with a customization bit string for cSHAKE computations. In order to generate random numbers of width β, $\frac{\beta}{32}$-bit outputs from SHAKE are buffered and further sent out as a valid random number.

Binary Search. As shown in Figure 2, the BinSearch module stores the pre-computed values of the CDT table in a BRAM block, which is configured as single-ported with width β and depth t. The design of the binary search module closely follows Algorithm 4. Three registers are defined in the design: cur stores the current address of the CDT memory entry that is being read; min and max store the range of the memory section that need to be searched for. Apart from these registers, a size-β Comparator is also needed for the comparison between the input random number from PRNG and the actual CDT value stored at memory address cur. Depending on the comparison result, the cur value is updated accordingly. In order to eliminate the idle cycles in the computation unit and at the same maintain a relatively short logic path in the design, we pre-computed all the possible values of cur and stored them in two separate registers pred1 and pred2. One of the values in these registers are then used to update the value of cur once the comparison finishes. This design choice guarantees that the total runtime of one full binary search reaches the theoretical computational complexity $\lceil \log_2(t) \rceil$. More importantly, we achieve a good maximum frequency in the final design, as shown in Table 4.

Input/Output FIFOs. The PRNG FIFO and Output FIFO are deployed in our design in order to flexibly adjust the overall performance of the GaussSampler module when integrated with different outside modules. A larger PRNG FIFO allows the buffering of more pseudo-random numbers from the PRNG while a larger Output FIFO makes sure that the BinSearch module can generate more outputs even if the outside module is not fetching the output on time. Depending on the input and output data rates, these two FIFOs can adjust their sizes independently to make sure that the overall performance is optimal. A series of experiments was carried out in our work in order to determine the best sizes for these two FIFOs. We found that, given the hardware-software interface overhead, large input/output FIFOs do not contribute to a better performance, and thus we pick 8 and 2 as the sizes for PRNG FIFO and Output FIFO, respectively. These two sizes are also used for all the sampler-dependent evaluations in this work.

Evaluation and Related Work. Table 4 shows the performance and synthesis results of our GaussSampler module when synthesized with the qTESLA-p-I and qTESLA-p-III parameters. The exact cycle count of our GaussSampler design for generating n samples depends on the actual interface, and in our case, we provide cycles in an ideal setting, i.e., the outside modules are always holding valid inputs and are ready to read out outputs. Given the fixed interface delay, our Gaussian sampler runs in constant-time. For lattice-based schemes, usually a relatively large number of random samples are needed. For qTESLA-p-I and qTESLA-p-III, $n = 1024$ and $n = 2048$ samples are needed in one Gaussian sampling function call. To get these samples, GaussSampler can generate samples in batches of size b. For the cycle reports, we show both the *total* cycle count, i.e., cycle counts for the whole Gaussian sampling operation, as well as the cycle counts for running the standalone PRNG module in order to generate n pseudo-random numbers.

As shown in Table 4, the best cycle count is achieved when $b = n$, as each new Gaussian sampling function call requires to absorb a new customization bit string during the cSHAKE computation. Further, we can see that the *total* cycle count of the sampler is very close to the PRNG cycle count. This shows that the computations of PRNG and BinSearch are perfectly interleaved by use of the input and output FIFOs.

In Howe et al. [20], constant-time hardware designs of Gaussian samplers based on different methods are presented, including a binary-search CDT sampler. While Howe et al. demonstrate that the runtime for generating one Gaussian sample by use of their CDT-based Gaussian sampler can reach the theoretical bound $\lceil \log_2(t) \rceil$, it is hard to apply their design to real-life applications since, in their case the PRNG and the binary search steps are carried out in sequence and there is no architectural support for the data and control signal synchronizations between different modules. Also, we note that they use a significantly faster, but arguably less cryptographically secure [17], PRNG based on

Design	$\sigma/\beta/\tau$	Batch size(b)	Device	Total cycles (n samp.)	PRNG cycles	Slices/LUTs /FFs/BRAMs	Fmax (MHz)
Ours,	8.5/64/9.17	512	Artix-7	19,046	18,948	113/278/295/2.5	131
qTESLA-p-I	8.5/64/9.17	1024	Artix-7	18,451	18,370	118/279/296/2.5	134
Ours,	8.5/128/13.0	512	Artix-7	83,040	82,952	217/485/487/4.5	128
qTESLA-p-	8.5/128/13.0	1024	Artix-7	81,904	81,860	191/450/487/4.5	123
III	8.5/128/13.0	2048	Artix-7	81,335	81,314	191/470/490/4.5	123
Ours	3.33/64/9.5	512	Artix-7*	9,506	9,474	114/268/283/2.5	101
[20]	3.33/64/9.5	512	Virtex-6	2,560 (w/o PRNG)		15/53/17/1	193
[50]	3.33/64/9.5	512	Artix-7*	50,700 (w/o PRNG)		-/893/796/3	113

Table 4: Performance of the GaussSampler module and comparison with state-of-the-art related work. The synthesis results for our and related work exclude the PRNG overhead. The "total cycles" in [20, 50] excludes the PRNG, whereas our work does include it. Results for Artix-7 with * correspond to the device model XC7A100TCSG324, otherwise they correspond to XC7A200TFBG676.

Trivium. In contrast, our GaussSampler module uses the NIST-approved, cryptographically strong cSHAKE primitive as the underlying PRNG. Moreover, our design is fully pipelined and highly modular, and users can easily replace the SHAKE core with their own PRNG design, if desired.

The authors in [50] presented a merge-sort based Gaussian sampler following an older version of the qTESLA software implementation. Their design provides a fixed memory access pattern which eliminates some potential timing and power side-channel attacks. However, the merge-sort based sampling method is much more expensive compared to our binary search based approach in terms of both cycle counts and area usage, as shown in Table 4.

Applicability to Other Lattice-Based Schemes. Our Gaussian sampler hardware module is very flexible and can be directly used in many lattice-based constructions with relatively small σ, as is the case of, for example, the NIST PQC key encapsulation candidate FrodoKEM [35] and the binary variant of the lattice-based signature scheme Falcon [43].

3.3 Polynomial Multiplier

Figure 3 shows the dataflow of the hardware module PolyMul, including four main sub-modules: Controller, NTT, ModMul, and PointwiseMul. The Controller module contains all of the controlling logic while the other sub-modules serve different computation purposes: NTT is used for forward or inverse NTT transformation, ModMul is the modular Montgomery multiplier, and PointwiseMul is used for the coefficient-wise polynomial multiplications.

The core of PolyMul is the hardware implementation of the NTT algorithm. Hence, in this section we first discuss and describe our memory efficient NTT algorithm that is suitable for hardware implementations. Afterwards, we describe the other three sub-modules. At the end of the section we evaluate the performance, explain the applicability of the accelerator to other schemes, and discuss related work.

Memory Efficient and Unified NTT Algorithm. Most hardware implementations of the NTT-based polynomial multipliers are based on a unified NTT algorithm [14, 27, 45] in

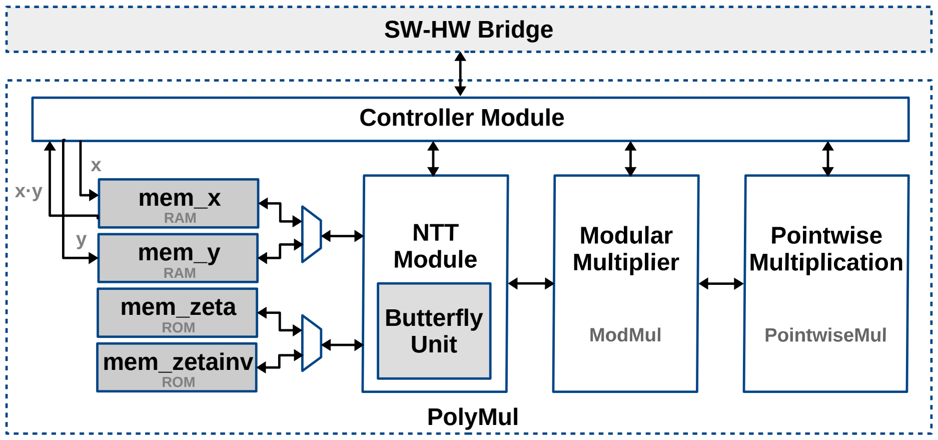

Figure 3: Dataflow diagram of the PolyMul hardware module.

which both the forward and inverse NTT transformations are performed using the Cooley-Tukey (CT) butterfly algorithm (denoted as CT-NTT algorithm in what follows). Using the same algorithm, however, requires a pre-scaling operation followed by a bit-reversal step on the input polynomials in NTT and NTT^{-1}, and one additional polynomial post-scaling operation after NTT^{-1}. In recent years, the CT-NTT algorithm has been greatly optimized, e.g., in [14,45]. Unfortunately, these optimizations increase the complexity of the hardware implementation.

In this work, we took a different direction: we unified the algorithms proposed by Pöppelmann, Oder and Güneysu in [42] for lattice-based schemes. In their software implementation, [42] adopted an NTT algorithm which relies on a CT butterfly for NTT and a Gentlemen-Sande (GS) butterfly for NTT^{-1}. By using the two butterfly algorithms, the bit-reversal steps are naturally eliminated. Moreover, polynomial pre-scaling and post-scaling operations can be merged into the twiddle factors by taking advantage of the different structures within the CT and GS butterflies.

A direct implementation of the CT and GS algorithms to support polynomial multiplication is inexpensive in software. However, when mapping them to hardware, the two separate algorithms would require two different hardware modules, leading to twice as much hardware logic when compared to a CT-NTT based hardware implementation. In our work, we unify the CT and GS butterflies based on the observation that both algorithms require the same number of rounds and within each round, a fixed number of iterations are applied. This leads to a unified module that performs both NTT and NTT^{-1} computations with reduced hardware resources while keeping the performance advantage of using the two butterflies. Our unified algorithm, called CT-GS-NTT in the remainder, is depicted in Algorithm 5. Depending on the operation type (NTT or NTT^{-1}), the control indices m_0, m_1 and the coefficients $a[j], a[j+m]$ are conditionally updated based on the CT or GS butterfly.

Roy et al. [45] presented a new memory access scheme by carefully storing polynomial coefficients in pairs. Inspired by their idea, we incorporate a variant of their memory access scheme in our unified CT-GS-NTT algorithm to reduce the required memory; see lines 5–16 of Algorithm 5.

Apart from the logic units, four memory blocks are needed in our NTT design: mem_x stores the input polynomial a, which is already represented in the NTT domain, mem_y stores the input polynomial b, which later needs to be transformed by the NTT module,

Algorithm 5 Memory-efficient and unified CT-GS-NTT Algorithm

Require: $a = \sum_{i=0}^{n-1} a_i x^i \in \mathcal{R}_q$, with $a_i \in \mathbb{Z}_q$; pre-computed twiddle factors W
Ensure: $\mathsf{NTT}(a)$ or $\mathsf{NTT}^{-1}(a) \in \mathcal{R}_q$

```
▷ Depending on NTT or NTT^{-1}, $n/2$ or 1 is assigned to $m_0$; similarly in the lines below
1: $m_0 \leftarrow n/2$ or 1; $m_1 \leftarrow 1/2$ or 2; $n_0 \leftarrow 1$ or 0; $n_1 \leftarrow n$ or $n/2$
2: $k \leftarrow 0$, $j \leftarrow 0$
3: for $m = m_0; n_0 < m < n_1; m = m \cdot m_1$ do          // First $(\log_2(n) - 1)$ NTT rounds
4:     for $i = 0; i < \frac{n}{2}; i = j + \frac{m}{2}$ do
5:         $w \leftarrow W[k]$
6:         for $j = i; j < i + \frac{m}{2}; j = j + 1$ do
7:             $(t_1, u_1) \leftarrow (a[j], a[j + m])$          // From mem[$j$]
8:             $(t_2, u_2) \leftarrow (a[j + m \cdot m_1], a[j + m + m \cdot m_1])$ // From mem[$j + m \cdot m_1$]
9:             $r_1 \leftarrow w \cdot u_1$ or $w \cdot (t_1 - u_1)$
10:            $r_2 \leftarrow w \cdot u_2$ or $w \cdot (t_2 - u_2)$
11:            $a[j] \leftarrow t_1 + r_1$ or $t_1 + u_1$
12:            $a[j + m] \leftarrow t_1 - r_1$ or $r_1$
13:            $a[j + m \cdot m_1] \leftarrow t_2 + r_2$ or $t_2 + u_2$
14:            $a[j + m + m \cdot m_1] \leftarrow t_2 - r_2$ or $r_2$
15:            mem[$j$] $\leftarrow (a[j], a[j + m \cdot m_1])$
16:            mem[$j + m \cdot m_1$] $\leftarrow (a[j + m], a[j + m + m \cdot m_1])$
17:        $k \leftarrow k + 1$
18: for $i = 0; i < \frac{n}{2}; i = i + 1$ do          // Last NTT round
19:     $w \leftarrow W[k]$
20:     $(t_1, u_1) \leftarrow (a[i], a[i + m])$          // From mem[$i$]
21:     $r_1 \leftarrow w \cdot u_1$ or $w \cdot (t_1 - u_1)$
22:     $a[j] \leftarrow t_1 + r_1$ or $t_1 + u_1$
23:     $a[j + 1] \leftarrow t_1 - r_1$ or $r_1$
24:     mem[$i$] $\leftarrow (a[i], a[i + m])$
25:     $k \leftarrow k + 1$ or $k \leftarrow k$
26: return $a$
```

mem_zeta and mem_zetainv store the pre-computed twiddle factors needed in the NTT and NTT^{-1} transformations, respectively. mem_x and mem_y are both configured as dual-port RAMs with width $2 \cdot (\lceil \log_2(q) \rceil + 1)$ and depth $n/2$, while mem_zeta and mem_zetainv are configured as single-port ROMs with width $(\lceil \log_2(q) \rceil + 1)$ and depth n. Details of the sub-modules are expanded next.

Controller Module. The controller module in PolyMul is responsible for coordinating the different sub-modules. For the execution of a polynomial multiplication of the form $x \cdot y = \mathsf{NTT}^{-1}(x \circ \mathsf{NTT}(y))$, the polynomial y is first received and written to mem_y. Then, the forward NTT transformation on y begins by use of the NTT module. The computation result $\mathsf{NTT}(y)$ is written back to mem_y. While the forward NTT transformation is ongoing, the polynomial x can be sent and stored in mem_x. Once mem_x gets updated with polynomial x and mem_y gets updated with the result $\mathsf{NTT}(y)$, the PointwiseMul module is triggered. The PointwiseMul module writes back its result to mem_x, which is later used in computing NTT^{-1}. The final result of NTT^{-1} is kept in mem_x, from which it can be sent in 32-bit chunks over the interconnect bus.

NTT Module. The NTT module is designed according to our unified CT-GS-NTT algorithm in Algorithm 5. It uses a Butterfly unit as a building block and interacts with two memories: one stores the polynomial, and the other one stores the pre-computed

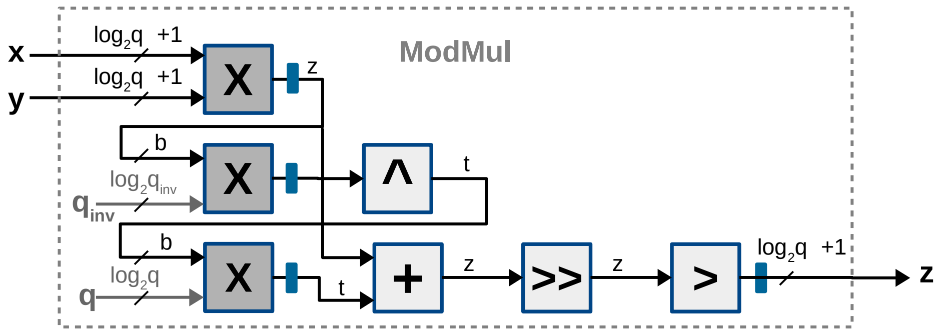

Figure 4: Dataflow diagram of the ModMul module. Register buffers are shown as small blue boxes in the diagram.

twiddle factors. The polynomial memory is organized in a way that each memory content contains a coefficient pair, as defined in Algorithm 5. The organization of the polynomial memory ensures that two concurrent memory reads prepare two pairs of the coefficients needed for two butterfly operations. In this way, we can fully utilize the Butterfly unit. The architecture of NTT is fully pipelined. By use of our NTT module, one forward NTT or inverse NTT operation takes around $(\frac{n}{2} \cdot \log_2(n))$ cycles (plus a small fixed overhead for filling the pipelines).

Modular Multiplier. Typically, integer multiplication is followed by modular reduction in $\mathbb{Z}_q$ in lattice-based implementations operating over the ring $\mathcal{R}_q$ (this, for example, is the case of qTESLA's software implementation). Hence, we designed a ModMul module that combines both operations. Since our design does not exploit any special property of the modulus q, our modular multiplier supports a configurable modulus. Figure 4 shows the dataflow of the ModMul module.

For the reduction operation we use Montgomery reduction [34], as shown in Algorithm 6. The input operands are two signed integers $x, y \in \mathbb{Z}_q$, and the modular multiplication result is $z = x \cdot y \bmod q$ with output range $(-q, q]$. One modular multiplication involves three integer multiplication operations, one bit-wise AND operation, one addition operation, and one right shift operation. One final correction operation is also needed to make sure that the result is in the range $(-q, q]$.

To be able to do one modular multiplication within each clock cycle, while maintaining a short logic path, we implemented a pipelined modular multiplier module in hardware. As shown in Figure 4, three integer multipliers are instantiated in the ModMul module: one multiplier accepting two input operands of bit length $\lceil \log_2(q) \rceil + 1$, one multiplier with an operand fixed to the constant q_{inv}, and one multiplier with inputs q and an operand of some width b (typically, b is the multiple of the computer word-size immediately larger than $\lceil \log_2(q) \rceil$). The multiplication results of these multipliers are all buffered before being used in the next step to make sure that the longest critical path stays within the multiplier. The final result is also buffered. Therefore, one modular multiplication takes four cycles to complete. However, since the ModMul module is fully pipelined, right after the inputs are fed into the design, new inputs can always be sent in the very next clock cycle. This ensures that within each clock cycle one modular multiplication operation can be finished on average.

Pointwise Multiplication. The PointwiseMul module simply multiplies two polynomials in an entry-wise fashion. Once the forward NTT transformation on input polynomial y

Algorithm 6 Signed modular multiplication with Montgomery reduction

```
Require: $x, y \in (-q, q]$ and $q_{inv} = -q^{-1} \bmod 2^b$ for a suitable value $b$
Ensure: $z = x \cdot y \bmod q$ with $z \in (-q, q]$
1: $z = x \cdot y$
2: $t = (z \cdot q_{inv}) \wedge (2^b - 1)$
3: $t = t \cdot q$
4: $z = z + t$
5: $z = z \gg b$
6: if $(z > q)$ then
7:     $z = z - q$
8: return $z$
```

finishes, the memory contents in mem_y are updated with $\mathsf{NTT}(y)$. Then the PointwiseMul module is triggered: memory contents from mem_y and mem_x are read out, multiplied, reduced, and finally get written back to mem_x. This process is carried out repeatedly until all the memory contents are processed. For both NTT and PointwiseMul modules, the modular multiplications are realized by interacting with the same ModMul.

Evaluation. Table 5 provides the performance and synthesis results of our modular multiplier as well as the polynomial multiplier. As we can see, when synthesized with the parameters (n, q) required by qTESLA-p-I and qTESLA-p-III, the cycles achieved by the PolyMul module are close to the theoretically estimated $n \cdot \log_2(n) + \frac{n}{2}$ cycles. The area utilization for the qTESLA-p-III design only increases slightly when compared to that of qTESLA-p-I, and both have similar maximum frequency.

Related Work. Most of the existing designs of NTT-based polynomial multipliers are implemented for fixed parameters. While this might lead to efficient hardware implementation, the implementations are not easily reusable by other than the targeted schemes or as soon as new parameters arise. To be able to discuss the differences of these works and our fully parameterizable design, we first compare with a compact, state-of-the-art NTT-based polynomial multiplier [14]. This design shares one butterfly operator for NTT and NTT^{-1} computations and therefore is better suited for embedded systems, which fits to our design target.

The design [14] adopts a CT-NTT based approach, and exploits some optimizations, such as the improved memory scheme [45]. However, their design is based on a fixed modulus q, where q is the biggest Fermat prime $q = 2^{16} + 1 = 65537$. The shape of q supports very cheap reduction essentially using additions and shifts and, therefore, can be finished within one clock cycle. In this case, the pipelines within the polynomial multiplier in [14] are quite straightforward to design as the most expensive modular reduction operation gets its result within the same clock cycle. This explains for the most part the synthesis results gap observed in Table 5 between [14] and our design.

Another line of optimizations is to use multiple butterfly units to parallelize the NTT, such as in [27] where four butterfly units are used to support the parameters $(n, q) = (1024, 12289)$. We synthesized our PolyMul module with the same parameters for comparison. As shown in Table 5, the use of multiple butterfly units working in parallel improves the performance in terms of cycles, but increases significantly the area overhead.

Fair comparisons with these works [14, 27] are hard to achieve as none of them support flexible parameters (n, q). Our design does not pose any constraints on the polynomial size n or the modulus q, given its fully pipelined architecture.

Parameters (n, q)	Tunable (n, q)	Platform	Cycles	Slices/LUTs/FFs /DSPs/BRAMs	Fmax (MHz)
		ModMul			
(—, 343576577)	✓,✓	Artix-7	1	102/212/313/0/11	151
(—, 856145921)	✓,✓	Artix-7	1	96/219/243/0/11	147
		PolyMul, w/o ModMul			
(1024, 343576577)	✓,✓	Artix-7	11,455	502/1735/758/6/0	126
(2048, 856145921)	✓,✓	Artix-7	24,785	506/1736/783/8/0	124
		PolyMul, w/ ModMul			
(1024, 343576577)	✓,✓	Artix-7	11,455	582/1977/991/6/11	124
(2048, 856145921)	✓,✓	Artix-7	24,785	555/1981/1021/8/11	124
		PolyMul, w/ ModMul, comparison with related work			
(1024, 65537), Ours	✓,✓	Spartan-6	11,455	545/1576/361/4/5	90
(1024, 65537) [14]	✓,—	Spartan-6	11,826	251/—/—/4.5/1	241
(2048, 65537), Ours	✓,✓	Spartan-6	24,785	543/1601/368/8/5	90
(2048, 65537) [14]	✓,—	Spartan-6	25,654	269—/—/9/1	207
(1024, 12289), Ours	✓,✓	Artix-7	11,455	271/944/467/3/3	141
(1024, 12289) [27]	—,—	Artix-7	5494	—/2832/1381/8/10	150

Table 5: Performance of the hardware modules ModMul and PolyMul (with and without ModMul included) and comparison with related state-of-the-art work.

Applicability to Other Lattice-Based Schemes. Our NTT module is flexible in the sense that it can support any NTT implementation with $q \equiv 1 \bmod 2n$ over the ring $\mathcal{R}_q$ with n being a power-of-two. Hence, it can be used to accelerate the NTT computations of, e.g., the lattice-based signature scheme Dilithium [32] and the KEM scheme NewHope [40].

3.4 Sparse Polynomial Multiplier

In qTESLA, the sparse polynomial multiplication involves a dense polynomial $a = \sum_{i=1}^{n-1} a_i x^i \in \mathcal{R}_q$ and a sparse polynomial $c = \sum_{i=1}^{n-1} c_i x^i$, where $c_i \in \{-1, 0, 1\}$ with exactly h coefficients being non-zero. Two arrays *pos_list* and *sign_list* are used to store the information of the indices and signs of the non-zero coefficients of c, respectively. In the software implementation of qTESLA, Algorithm 7 is used for sparse polynomial multiplications to improve the efficiency by exploiting the sparseness of c.

Polynomial multiplication in $\mathcal{R}_q$ can be seen as the following matrix-vector product:

$$a \cdot c = \underbrace{\begin{bmatrix} a_0 & -a_{n-1} & \cdots & -a_2 & -a_1 \\ a_1 & -a_0 & \cdots & -a_3 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-2} & a_{n-3} & \cdots & a_1 & -a_{n-1} \\ a_{n-1} & a_{n-2} & \cdots & a_0 & a_0 \end{bmatrix}}_{=:A} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-2} \\ c_{n-1} \end{pmatrix}.$$

Since the polynomial c is very sparse, the sparse polynomial multiplication can be implemented in a column-wise fashion. First, a non-zero coefficient c_i is identified. Its index i determines which column of the matrix A will be needed for the computation while the value of $c_i \in \{-1, 1\}$ determines whether it is a column-wise subtraction or addition.

Algorithm 7 Sparse Polynomial Multiplication

Require: $a = \sum_{i=0}^{n-1} a_i x^i \in \mathcal{R}_q$ with $a_i \in \mathbb{Z}_q$, and list arrays $pos\_list \in \{0, ..., n-1\}^h$ and $sign\_list \in \{-1, 1\}^h$ containing the positions and signs, resp., of the nonzero elements of a polynomial $c \in \mathbb{H}_{n,h}$

Ensure: $f = a \cdot c \in \mathcal{R}_q$

```
1: Set all coefficients of $f$ to 0
2: for $i = 0, ..., h-1$ do
3:     $\text{pos} \leftarrow \text{pos\_list}[i]$
4:     for $j = 0, ..., \text{pos}-1$ do
5:         $f_j \leftarrow f_j - \text{sign\_list}[i] \cdot a_{j+n-\text{pos}}$
6:     for $j = \text{pos}, ..., n-1$ do
7:         $f_j \leftarrow f_j + \text{sign\_list}[i] \cdot a_{j-\text{pos}}$
8: return $f$
```

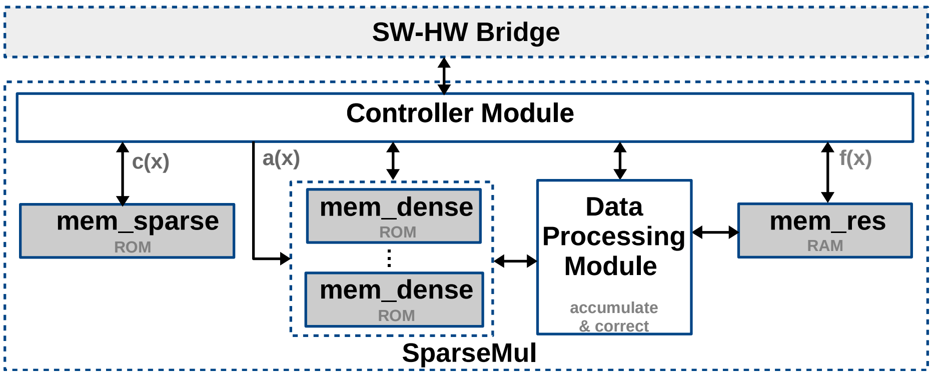

Figure 5: Dataflow diagram of the SparseMul hardware module.

Once c_i is chosen, the i_{th} column of A needs to be constructed based on the non-sparse polynomial a and the index i. While constructing the i_{th} column of A, the column-wise computation between the intermediate result and the newly constructed column A_i can happen in parallel. Computations above are repeated until the columns of A mapping to the h non-zero entries in c are all reconstructed and processed.

In the software implementation of qTESLA, two sparse polynomial functions are defined: SparseMul8 and SparseMul32, depending on the size of the coefficients of a. For our hardware implementation it is advantageous to implement one unified module where all coefficients are assumed to be in $[-q, q)$.

Hardware Module. For the implementation of our hardware module SparseMul, we followed the idea above but added more flexibility in the design. Moreover, our sparse polynomial multiplier is pipelined and fully parameterized. In particular, users can choose the following two parameters: the size of the polynomial n and the number of non-zero coefficients h in the sparse polynomial c. In addition, the performance parameter p can be used to achieve a trade-off between performance and area where $p \in \{2, 4, \ldots, \frac{h}{2}\}$. Essentially, p determines the number of columns of the matrix A that are to be processed and computed in parallel.

To enable such parallelism, $\frac{p}{2}$ dual-port memory blocks (denoted by mem_dense in Figure 5) each keeping a copy of a's coefficients are needed. Note that since mem_dense are of dual ports, two memory reads can be issued in parallel and thus two columns of A can be

Design	Cycles	Slices	LUTs	FFs	Fmax (MHz)	BRAMs (RAMB36)	Time×Area
qTESLA-p-I, $n = 1024$, $h = 25$							
p=2	$13,404$	127	393	240	134	2	$1,702,308$
p=4	7225	212	661	375	138	3	$1,531,700$
p=8	4133	336	1063	610	135	5	$1,388,688$
p=16	2069	573	1819	1050	139	9	$1,185,537$
qTESLA-p-III, $n = 2048$, $h = 40$							
p=2	$41,101$	143	431	252	174	4	$5,877,444$
p=4	$20,561$	222	702	391	175	6	$4,564,542$
p=8	$10,286$	356	1138	643	173	10	$3,661,815$
p=16	6175	618	1926	1109	138	18	$3,816,150$

Table 6: Performance of the hardware module SparseMul.

constructed in parallel. mem_dense are instantiated with ROMs configured with width $(\lceil \log_2 q \rceil + 1)$ and depth n. To store the information of the sparse polynomial c, given its sparsity, we allocated a much smaller memory chunk mem_sparse which is of width $p \cdot (\log_2 n + 1)$ and depth $\lceil \frac{h}{p} \rceil$. Each entry of mem_sparse contains p {index, sign} tuples mapping to p non-zero coefficients in c. To be able to read and update the intermediate results in parallel during computation, mem_res is allocated for storing the intermediate results and it has the same configuration as mem_dense.

Apart from the memory blocks, one controller module and one data processing module are needed. The controller module issues read and write requests to all the memory modules and passes data through the rest of the modules. Once the SparseMul module starts, the controller module issues a read request to mem_sparse. The output of mem_sparse contains p tuples of {index, sign}. Based on these index values, the controller module starts issuing separate reads continuously to each mem_dense. In parallel, the controller issues continuous read requests to mem_res (initialized with zeroes) starting from memory address 0. The data processing unit keeps taking p memory outputs from the mem_dense memories as input. These values first get conditionally negated based on the construction of matrix A and later get further accumulated based on the sign values. The accumulation result later gets corrected to range $[-q, q)$ through $\log_2(p)$ comparisons. The corrected result then gets added to the intermediate result (the output of the mem_res memory), corrected to range $[-q, q)$ and finally written back to mem_res in order. Once all the memory contents of mem_res get updated, a new memory read request is issued to mem_sparse whose output then specifies the next p columns of A to be processed. This process repeats for $\lceil \frac{h}{p} \rceil$ times. When SparseMul finishes, the resulting polynomial $f = a \cdot c$ is stored in mem_res memory.

Evaluation. In total, it takes around $n \cdot \lceil \frac{h}{p} \rceil$ cycles to finish the sparse polynomial multiplication by use of the SparseMul hardware module. As shown in Table 6, the achieved cycle counts are close to the theoretical bound. As the performance parameter p doubles, the cycle count halves, approximately. However, the area overhead of the design also increases as the parallelism of the SparseMul design increases, especially for $p \geq 8$. Depending on the user's requirements, the design parameter p can be freely tuned to achieve a certain area-performance trade-off.

Applicability to Other Lattice-Based Schemes. To our knowledge, this is the first hardware module of a fully parameterized sparse polynomial multiplier targeting the multiplication between a dense polynomial a and a sparse polynomial c with h non-zero coefficients from $\{-1, 1\}$. Since SparseMul is fully parameterized, it could be adapted to other schemes performing a similar computation. Examples of modern schemes using some variant of these sparse multiplications are, for example, the signature scheme Dilithium [32] and the KEM scheme LAC [30].

3.5 Hmax-Sum

Algorithm 8 Hmax-Sum Problem

Require: $a = \sum_{i=0}^{n-1} a_i x^i$.
Ensure: sum of the h largest coefficients of $a \in \mathcal{R}_q$.

```
1:  sum ← 0
2:  for $i = 0; i < h; i = i + 1$ do                      // Initialize the size-$h$ array.
3:      hmax_array[$i$] ← 0
4:  for $i = 0; i < n; i = i + 1$ do
5:      min_data ← hmax_array[0]
6:      for $j = 0; j < h; j = j + 1$ do                  // Find the least value in the array.
7:          comp ← (hmax_array[$j$] < min_data)
8:          if (comp = true) then
9:              min_data ← hmax_array[$j$]
10:             min_index ← $j$
11:     update ← ($a[i]$ > min_data)
12:     if (update = true) then                           // Update the array.
13:         hmax_array[min_index] ← $a[i]$
14:         sum ← (sum − min_data + $a[i]$)
15: return sum
```

To solve the Hmax-Sum problem, a natural solution is to first find out the largest h coefficients of the polynomial and then to compute the sum of them. The software implementation of qTESLA adopts this method: bubble sort is repeatedly used for h rounds. All the coefficients are first written in a list. For the first round, the elements in the list are scanned, compared and conditionally swapped until the biggest element sinks to the end of the list. This element then gets removed from the list and added to the sum. The above steps are repeated h times.

A naive implementation of the above method can be easily migrated to hardware, but it would require allocating memory of size $O(n)$ since all polynomial coefficients have to be stored. In our work, we observed that such a large memory requirement can be reduced to $O(h)$ as described next and shown in Algorithm 8. First a size-h array hmax is initialized. When a coefficient is fed to the algorithm, a full scan of the hmax array is carried out with the target of finding out the value min_data and the index min_index of the smallest element in hmax. Afterwards, the input coefficient is compared with min_data: if the input coefficient is bigger, min_data stored at index min_index in the array is updated with the coefficient. In parallel, the sum is updated according to line 14 of Algorithm 8. This algorithm ensures that the hmax array always stores the biggest coefficients that have been scanned.

Implementation. Based on Algorithm 8, we designed the following hardware module HmaxSum (see Figure 2): when a reset signal is received, the memory of depth h (mem_h)

Module	Cycles (PRNG Incld.)	Slices	LUTs	FFs	Memory (RAMB36)	Fmax (MHz)
qTESLA-p-I						
GaussSampler	$18,451$	118	279	296	2.5	134
HmaxSum	$28,686$	27	83	66	0	356
GaussSampler + HmaxSum	$29,293$	144	362	360	2.5	130
qTESLA-p-III						
GaussSampler	$81,335$	191	470	490	4.5	123
HmaxSum	$88,093$	40	94	70	0	389
GaussSampler + HmaxSum	$88,676$	236	560	558	4.5	125

Table 7: Performance of the GaussSampler, HmaxSum, and GaussSampler + HmaxSum hardware modules; the last combination of modules gives the best performance due to parallelized execution.

within the module is initialized with zeroes. Afterwards, valid coefficients can be sent to HmaxSum through its AXI-like interface. To find out the smallest memory content, a full-scan is carried out on mem_h and after the scan finishes, the value and the address of the smallest element are stored in two separate registers min_data and min_addr. Afterwards, a comparison between the input coefficient and min_data is carried out, and the memory content stored at memory address min_addr is conditionally updated: if the input coefficient is larger, the memory content stored at address min_addr is overwritten with the coefficient value. In parallel, the sum register is conditionally updated. After all the input coefficients of a polynomial are processed by HmaxSum, the value of sum is returned as the result.

Evaluation. Apart from low memory requirements, another advantage of adopting Algorithm 8 is that the HmaxSum module can run in parallel with the GaussSampler module. Once a valid sample is generated by GaussSampler, HmaxSum can immediately start processing it. As shown in Table 7, when running the HmaxSum module alone, it is quite expensive in terms of cycles as the complexity of Algorithm 8 is $O(n \cdot h)$. However, parallelizing the execution of GaussSampler and HmaxSum leads to almost the same cycle count as running HmaxSum alone. In terms of area utilization, the HmaxSum module is quite lightweight and, hence, introduces a very small overhead.

4 Performance Evaluation and Comparison

Based on our flexible, to-be open-sourced modules we implemented a hardware-software co-design of the qTESLA algorithm with provably secure parameter sets. To demonstrate the effectiveness of the hardware accelerators, while targeting system-on-chip type designs with standard 32-bit interfaces, we prototyped the hardware-software co-design on a RISC-V based Murax SoC. Figure 6 shows the block diagram of the SoC, with the hardware accelerators highlighted in gray. The whole design was further implemented on an Artix-7 FPGA board from Xilinx.

The SoC uses the VexRiscv[3] 32-bit RISC-V CPU implementation written in SpinalHDL[4]. It supports the RV32IM instruction set and implements a 5-stage in-order pipeline. The open-source RISC-V implementation was selected so the whole design can be freely distributed.

[3] https://github.com/SpinalHDL/VexRiscv/
[4] https://spinalhdl.github.io/SpinalDoc/

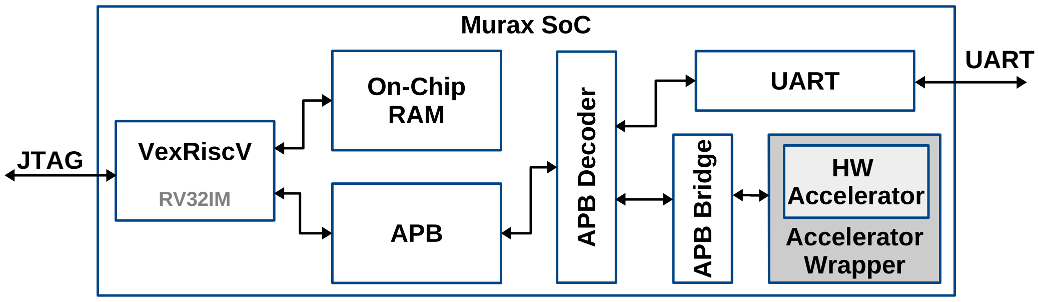

Figure 6: Schematic diagram of the Murax SoC. Hardware accelerators are connected to the APB. Details of the hardware accelerators are shown in Figure 8.

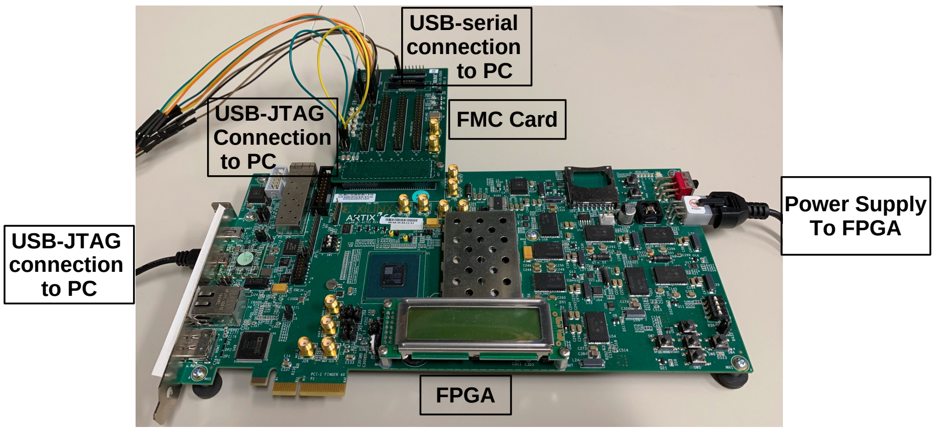
USB-serial connection to PC
USB-JTAG Connection to PC
FMC Card
Power Supply To FPGA
USB-JTAG connection to PC
FPGA

Figure 7: Evaluation setup with an Artix-7 AC701 FPGA and an FMC XM105 Debug Card.

Other processors such as Arm on Zynq board, however, can easily be used with our work (requiring only minor modifications to the interfaces). The closed-source Cortex-M series may be most similar to the open-source RISC-V used in this work, in terms of performance, and we compare to designs based on Cortex-M where possible.

4.1 FPGA Evaluation Platform

We evaluated our design using an Artix-7 AC701 FPGA as test-platform which is a platform recommended by NIST for PQC hardware evaluations. This board has a Xilinx XC7A200T-2FBG676C device. We used Vivado Software Version 2018.3 for synthesis. Figure 7 shows the evaluation setup for our experiments. Since the AC701 board has very limited number of GPIOs pins, we connected an FMC XM105 Debug Card to the FMC connector on the FPGA. This allows for sufficient GPIO pins to connect JTAG and UART to the SoC instantiated in the FPGA (in addition to the usual JTAG used to program the FPGA itself). We tested our implementations on the AC701 board at its default clock of 90 MHz. However, to achieve a fair comparison, our speedup reports presented in the following sections are based on the maximum frequency reported by the synthesis tools.

We also successfully tested our implementations on a DE1-SoC evaluation board from Terasic which has an Intel (formerly Altera) Cyclone V SoC 5CSEMA5F31C6 device. The same cycle counts and similar synthesis results are achieved on this platform as our implementation is neither platform-specific nor dependent on a specific FPGA vendor.

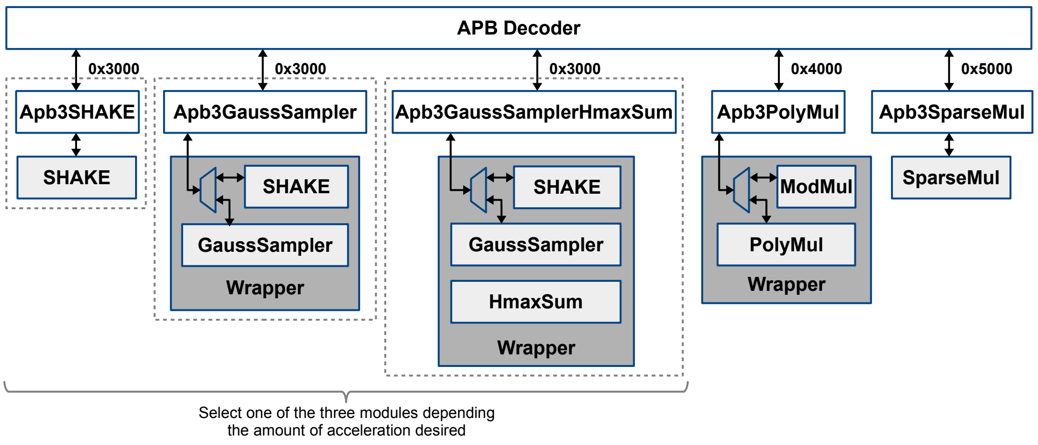

Figure 8: Detailed diagram of the connections between the APB Decoder, APB bridge modules and hardware accelerators. Dotted squares all contain a SHAKE module and thus one peripheral from these three can be chosen depending on user's requirements when a SHAKE accelerator is needed in the design.

This shows the open-source design can be easily ported to different development platforms and does not depend on a specific hardware setup.

4.2 Hardware-Software Interface used in Evaluation

To accelerate the compute-intensive operations in qTESLA, the dedicated hardware accelerators described in Section 3 are added to the SoC as peripherals. The SoC uses an 32-bit APB bus for connecting its peripherals to the main processor core. Our hardware modules are connected to this APB bus, as shown in Figure 8.

Different peripherals on the APB bus can be accessed by the software through control and data registers that are memory mapped to different addresses. On the software side, programs use read and write instructions to pre-defined addresses to access the peripherals. On the hardware side, the 32-bit interface bus includes a decoder for controlling which peripheral a read or write should go to. Further, for each accelerator, an `Apb3Bridge` module is developed to translate the APB signals to/from the control signals of the accelerator. When porting the design to another standard bus, only the `Apb3Bridge` modules would require modification.

Software Modifications. We modified the corresponding software functions in qTESLA's software reference implementation to replace them with function calls to our hardware accelerators. When the hardware accelerators are added in the Murax SoC, the adapted software functions simply communicate to/from the accelerators. The new functions maintain the same parameters as their original counterparts. Thus, only the function definitions are changed, and software is re-compiled to use the hardware.

4.3 Evaluation of qTESLA

The hardware accelerators described in Section 3 can be added to the Murax SoC by use of the APB bridge modules as described in Section 4.2 to accelerate the operations in the qTESLA scheme. Due to the modularity in the design of the SoC, the hardware accelerators

Function	SW Cycles	HW Cycles	HW-SW Cycles	IO Overhead(%)	Speedup SW/HW	Speedup SW/HW-SW
qTESLA-p-I						
SHAKE128	44,683	505	1586	214.1	88.5	28.2
GaussSampler	3,540,807	18,451	26,286	42.5	191.9	134.7
GaussSampler + HmaxSum	4,009,628	29,293	29,397	0.4	136.9	136.4
PolyMul	558,365	11,455	31,473	174.8	48.7	17.7
SparseMul8	365,207	7225	28,181	290.1	50.5	13.0
SparseMul32	571,165	7225	28,180	290.0	79.1	20.3
qTESLA-p-III						
SHAKE256	45,581	473	1562	230.2	96.4	29.2
GaussSampler	16,707,765	81,335	81,505	0.2	205.4	205.0
GaussSampler + HmaxSum	18,195,064	88,676	88,748	0.1	205.2	205.0
PolyMul	1,179,949	24,785	63,743	157.2	47.6	18.5
SparseMul8	1,160,213	20,561	62,216	202.6	56.4	18.6
SparseMul32	1,780,940	20,561	62,226	202.6	86.6	28.6

Table 8: Performance of different functions on software, hardware and HW-SW co-design. The "Speedup" columns are expressed in terms of cycle counts.

can be easily added to and removed from the SoC before synthesis. Depending on the users' requirements, any of the hardware accelerators (e.g., SHAKE, GaussSampler, GaussSampler + HmaxSum, PolyMul and SparseMul) can be added to the design for accelerating part of the compute-intensive operations in qTESLA. Different hardware accelerators can also be combined and added to accelerate different computations. Below we evaluate the three operations: key generation, signature generation, and signature verification with different combinations of the accelerators.

4.3.1 Speedup over Software Functions

Table 8 shows the performance of calling the SHAKE-128 and SHAKE-256 functions from the pure software, pure hardware, and hardware-software co-design. The input length is fixed to 32 bytes and the output length is fixed to 128 bytes, as a testing example. As we can see from the table, the SHAKE hardware accelerator achieves very good speedups in terms of clock cycles compared to running the corresponding functions on the pure software. Smaller speedups are achieved when the SHAKE module is added to the Murax SoC as an accelerator due to the IO overhead for sending the inputs and returning the outputs between the software and the hardware. With the IO overhead taken into account, function calls to the SHAKE function in the "Murax + SHAKE" design still leads to an over 28× speedup over the pure software implementation.

Table 8 also shows the performance of calling the Gaussian Sampler function from the pure software, pure hardware, and hardware-software co-design. The input seed is fixed to 32 bytes and the output length is fixed to 1024 and 2048 for qTESLA-p-I and qTESLA-p-III, respectively, as is the case in the qTESLA software reference implementation. As we can see, when the Gaussian Sampler function is called in the design "Murax + GaussSampler", over 134× and 205× speedups are achieved compared to calling the functions on the pure software for qTESLA-p-I and qTESLA-p-III, respectively. The reason for achieving such high speedups is threefold: from the algorithm level, we adopted a binary-search based CDT sampling algorithm in our design while the qTESLA software reference implementation uses a more conservative full-scan based CDT sampling algorithm. In terms of implementation, our fully pipelined hardware design brings a very good hardware acceleration over a pure software-based implementation. Moreover, when the GaussSampler module is added as an accelerator to the Murax SoC, the valid outputs from the hardware accelerator are returned to the software in parallel with the hardware computation phase. In this case, the IO overhead is very well hidden and the speedups brought by the hardware accelerator can be well exploited.

Table 8 then shows the performance of calling the Gaussian Sampler and Hmax-Sum functions from the pure software, pure hardware, and hardware-software co-design. As we can see, when these two functions are called in the design "Murax + GaussSampler + HmaxSum", over 136× and 205× speedups are achieved compared to calling the functions on the pure Murax SoC for qTESLA-p-I and qTESLA-p-III, respectively. We find it interesting to note that by introducing a lightweight HmaxSum accelerator to the "Murax + GaussSampler" design, the IO overhead for calling the Gaussian sampling function is almost negligible as the output returning phase is perfectly overlapped with the computations of the HmaxSum module.

Next, Table 8 shows the performance of calling the polynomial multiplication function from the pure software, pure hardware, and hardware-software co-design. As shown in the table, running one polynomial multiplication operation by use of the PolyMul accelerator takes more than 47× less cycles compared to the pure software implementation. However, when the function is called from the "Murax + PolyMul" design, two polynomials with large coefficients have to be sent to the hardware and one polynomial has to be returned to the software, leading to a rather big IO overhead. Therefore, only 17× and 18× speedups are achieved for qTESLA-p-I and qTESLA-p-III, respectively.

Table 8 finally shows the performance of calling the sparse polynomial multiplication functions SparseMul8 and SparseMul32 from the pure software, pure hardware, and hardware-software co-design. As we can see, running one SparseMul8 operation by use of the hardware accelerator takes the same number of cycles as running one SparseMul32 operation since the same SparseMul module is used. When calling the sparse polynomial functions in the "Murax + SparseMul" design, one polynomial with large coefficients and two small arrays have to be sent to the hardware and one big polynomial has to be returned to the software, yielding a big IO overhead. With these IO overhead taken into account, when the SparseMul8 and SparseMul32 functions are called in the "Murax + SparseMul" design, over 20× and 28× speedups are achieved compared to running the same function in pure software for qTESLA-p-I and qTESLA-p-III, respectively.

4.3.2 Key Generation Evaluation

Table 9 shows the performance and maximum frequency of running qTESLA's key generation on different designs. The cycles are reported as the average cycle counts for 100 executions. The column "speedup" reports the speedup of the time when adding the hardware module(s) of the corresponding row compared to running on the pure Murax SoC (first row). As we can see, adding a SHAKE accelerator gives over 2.4× and 2.2× speedups compared to running the key generation operation on the pure Murax SoC for qTESLA-p-I and qTESLA-p-III, respectively. Larger speedups are achieved when the GaussSampler accelerator is added to the design as Gaussian sampling is the most compute-intensive operation in the key generation step. By adding an extra lightweight HmaxSum accelerator into the "Murax + GaussSampler" design, around 16× and 37× speedups are achieved which is a larger improvement compared to adding a standalone GaussSampler accelerator in the design. This is due to the fact that when the GaussSampler accelerator is added, the most expensive Gaussian Sampler function gets greatly sped up and this in turn leaves the less expensive Hmax-Sum function costly in the "Murax + GaussSampler" design. Interestingly, while adding the PolyMul accelerator improves the cycle counts, the speedup compared to running on the pure Murax SoC is 0.83, i.e., adding (only) PolyMul slows down the runtime. This is due to the fact that the maximum frequency of the design drops when a hardware accelerator is integrated. Adding a SparseMul accelerator to the Murax SoC does not bring any speedup in terms of cycles as there is no sparse polynomial multiplication during key generation. The best speedups are achieved when all the available hardware accelerators are added ("Murax + All"): an around 40× speedup is achieved for qTESLA-p-I and an around 100× speedup is achieved for qTESLA-p-III. The best time-area product for key generation is also achieved in the "Murax + All" design.

4.3.3 Signature Generation and Signature Verification Evaluation

Table 10 and Table 11 show the performance and maximum frequency of running the qTESLA sign and verify operations on different designs. We report the average cycle counts for 100 executions. The column "speedup" reports the speedup of the time when adding the hardware module(s) of the corresponding row compared to running on the pure Murax SoC (first row). As the signing and verification steps in qTESLA do not involve Gaussian sampling, adding a GaussSampler accelerator to the design is equivalent to adding a SHAKE accelerator. The small difference in the cycle counts comes from the wrapper function that embeds SHAKE in the GaussSampler accelerator. Thus, the clock cycles achieved on a "Murax + GaussSampler" design for signing and verification are similar to those achieved on a "Murax + SHAKE" design. Apart from SHAKE computations, NTT-based polynomial multiplication and sparse polynomial multiplication are two of the most compute-intensive computations in signature generation and verification. As we can see from the tables, adding a PolyMul accelerator to the design brings a good reduction in clock cycles (and a speedup of 1.12) for signature generation compared to the pure software, while adding a SparseMul accelerator improves the cycle counts for verification (leading to a speedup of 1.25). The best speedups are achieved when all available hardware accelerators are added to the design ("Murax + All"): for qTESLA-p-III, speedups of 10.59× and 16.21× are achieved for signing and verification operations, respectively. The best time-area product for the signature generation and verification is also achieved in the "Murax + All" design.

4.4 Comparison with Related Work

Table 12 provides a detailed comparison of our design with other designs targeting modern lattice-based digital signature schemes running on embedded systems. A thorough benchmarking of NIST PQC schemes on the ARM Cortex-M4 platform was presented in 2019 [25], and it reports the testing results of different variants of qTESLA, Dilithium and Falcon. However, the *provably-secure* variants of qTESLA, namely qTESLA-p-I and qTESLA-p-III, are excluded from their report due to the memory constraint of the Cortex-M4 device. Unlike closed-source processors like Cortex-M4, the open-source Murax SoC can be easily integrated and adapted into specific processor setups as needed, e.g., users can set the size of on-chip RAM or enable optional plugins depending on their requirements.

As explained earlier in the paper, the performance of qTESLA-p-I and qTESLA-p-III is slower compared to the reference software implementations of Dilithium and Falcon in exchange for the *provably secure* feature. As shown in [3], the performance of qTESLA-p-I when running on an Intel Core-i7 CPU is about 3× slower compared to the Dilithium-II scheme. Similarly, when compared with the reference software implementation of Falcon-

Design	Cycles	Fmax (MHz)	Time (ms)	Time×Area (slice×ms)	Speedup
qTESLA-p-I					
Murax	48,529,602	156	310.9	328,299	1.00
+ SHAKE	18,214,784	145	125.5	164,472	2.48
+ GaussSampler (incl. SHAKE)	6,653,608	137	48.7	73,436	6.38
+ GaussSampler + HmaxSum	2,525,853	126	20.1	30,792	15.47
+ PolyMul (incl. ModMul)	46,933,182	126	373.8	602,596	0.83
+ SparseMul	48,529,602	134	361.8	424,795	0.86
+ All	925,431	121	7.7	18,651	40.64
qTESLA-p-III					
Murax	297,103,198	156	1903.3	2,009,841	1.00
+ SHAKE	120,731,265	145	831.5	1,090,134	2.29
+ GaussSampler (incl. SHAKE)	28,394,350	126	224.8	350,687	8.47
+GaussSampler + HmaxSum	6,494,464	126	51.7	83,606	36.79
+ PolyMul (incl. ModMul)	292,924,220	125	2340.8	3,829,482	0.81
+ SparseMul	297,103,153	161	1849.8	2,199,459	1.03
+ All	2,305,220	121	19.0	47,001	100.14

Table 9: Performance of qTESLA key generation on software and different HW-SW co-designs. All = GaussSampler + HmaxSum + PolyMul + SparseMul. The "Speedup" column is provided in terms of time.

Design	Cycles	Fmax (MHz)	Time (ms)	Time×Area (slice×ms)	Speedup
qTESLA-p-I					
Murax	47,180,534	156	302.2	319,171	1.00
+ SHAKE	22,914,215	145	157.8	206,905	1.91
+ GaussSampler (incl. SHAKE)	23,348,731	137	171.0	257,697	1.77
+ GaussSampler + HmaxSum	24,580,234	126	195.6	299,647	1.55
+ PolyMul (incl. ModMul)	34,013,026	126	270.9	436,707	1.12
+ SparseMul	41,356,497	134	308.4	362,007	0.98
+ All	4,165,160	121	34.4	83,944	8.78
qTESLA-p-III					
Murax	105,525,187	156	676.0	713,865	1.00
+ SHAKE	54,013,152	145	372.0	487,710	1.82
+ GaussSampler (incl. SHAKE)	55,308,030	126	437.9	683,084	1.54
+ GaussSampler + HmaxSum	53,024,762	126	422.4	682,611	1.60
+ PolyMul (incl. ModMul)	78,377,348	125	626.3	1,024,655	1.08
+ SparseMul	86,540,888	161	538.8	640,664	1.25
+ All	7,745,088	121	63.9	157,916	10.59

Table 10: Performance of qTESLA signature generation on software and different HW-SW co-designs. All = GaussSampler + HmaxSum + PolyMul + SparseMul. The "Speedup" column is provided in terms of time.

512 on an Intel Core-i7 CPU, the performance of qTESLA-p-III is around 5× slower for signing and 20× slower for verification.

By integrating our dedicated hardware accelerators to the Murax SoC, the performance of qTESLA-p-I on the "Murax +All" platform achieves a big improvement compared to the pure software implementation, as shown in Table 12. As there is no existing work

Design	Cycles	Fmax (MHz)	Time (ms)	Time×Area (slice×ms)	Speedup
qTESLA-p-I					
Murax	17,871,157	156	114.5	120,895	1.00
+ SHAKE	4,625,094	145	31.9	41,763	3.59
+ GaussSampler (incl. SHAKE)	4,625,505	137	33.9	51,052	3.38
+ GaussSampler + HmaxSum	4,623,861	126	36.8	56,368	3.11
+ PolyMul (incl. ModMul)	16,274,763	126	129.6	208,960	0.88
+ SparseMul	15,793,283	134	117.8	138,243	0.97
+ All	946,520	121	7.8	19,076	14.63
qTESLA-p-III					
Murax	48,309,625	156	309.5	326,810	1.00
+ SHAKE	14,899,621	145	102.6	134,535	3.02
+ GaussSampler (incl. SHAKE)	14,892,149	126	117.9	183,927	2.63
+ GaussSampler + HmaxSum	14,889,776	126	118.6	191,684	2.61
+ PolyMul (incl. ModMul)	44,130,687	125	352.6	576,934	0.88
+ SparseMul	39,915,065	161	248.5	295,490	1.25
+ All	2,315,950	121	19.1	47,220	16.21

Table 11: Performance of qTESLA signature verification on software and different HW-SW co-designs. All = GaussSampler + HmaxSum + PolyMul + SparseMul. The "Speedup" column is provided in terms of time.

on hardware designs of Dilithium, an apples-to-apples comparison between qTESLA on hardware and Dilithium on hardware is currently not possible. However, if we regard the performance of Dilithium-II running on the ARM Cortex-M4 device as being efficient, then we can conclude that, with proper use of hardware accelerators, provably-secure schemes like qTESLA can also be considered practical and that these schemes can be competitive in terms of efficiency when running on embedded systems. In particular, running qTESLA-p-III on "Murax +All" achieves a comparable efficiency to the Cortex-M4 benchmarking result for Dilithium-III. When compared to the Falcon-512 scheme, qTESLA-p-III running on our "Murax +All" platform is around 62× and 3.5× faster in terms of key generation and signing time, respectively. Again, a fair comparison between qTESLA on hardware and Falcon on hardware is currently not possible, as there is no publicly-available hardware implementation of Falcon. However, we emphasize again that the proposed hardware accelerators are not restricted to use in qTESLA and, hence, can benefit other schemes such as Dilithium and Falcon. In summary, by taking advantage of the hardware acceleration, the practical feasibility of running the *provably-secure* qTESLA variants qTESLA-p-I and qTESLA-p-III on resource-constrained embedded systems is successfully demonstrated in the present paper.

In 2019, a RISC-V based hardware-software co-design [6] focused on lattice-based schemes was proposed and demonstrated the performance of some qTESLA variants with prior heuristic parameters. As we can see in Table 12, as the design in [6] focuses on low-power and low-cycles ASIC applications, their work presents very small clock cycles for qTESLA-I signing and verification operations by packing more computations into one clock cycle. However, such a design choice leads to a very low frequency; e.g., their HW-SW co-design [6] can only run at 10MHz on an Artix-7 FPGA. Moreover, [6] only partly accelerated qTESLA's key generation since they followed the merge-sort based CDT algorithm for Gaussian sampling as used in the reference software implementation. To better compare our results with this design, we synthesized the "Murax +All" design for

Design	Platform	Freq. (MHz)	KeyGen./Sign/Verify $\times 10^3$ Cycles	KeyGen./Sign/Verify Time (ms)
NIST Security Level = 1				
qTESLA-p-I, our	Murax+HW^p	121	925/4165/947	7.7/34.4/7.8
qTESLA-p-I, —	Cortex-M4	—	—	—
Dilithium-II [25]	Cortex-M4	168	1400/6157/1461	8.3/36.6/8.7
NIST Security Level = 3				
qTESLA-p-III, our	Murax+HW^p	121	2305/7745/2316	19.0/63.9/19.1
qTESLA-p-III, —	Cortex-M4	—	—	—
Dilithium-III [25]	Cortex-M4	168	2282/9289/2229	13.6/55.3/13.3
Falcon-512 [25]	Cortex-M4	168	197,794/38,090/474	1177.3/226.7/2.8
Designs not matching latest NIST Security Levels				
qTESLA-I^o, our	Murax+HW^p	125	181/781/137	1.4/6.2/1.1
qTESLA-I^o [6]	RISC-V+HW^p	10	4847/168/39	484.7/16.8/3.9
qTESLA-I^o [25]	Cortex-M4	168	6748/5831/788	40.2/34.7/4.7

Table 12: Comparison with related work on lattice-based digital signature schemes for embedded systems. All the tests running on platform "Murax+HW" are based on the "Murax + All" design, see Section 4.3.3. ^{o} denotes the use of an old qTESLA reference implementation with outdated instantiations. Platforms noted with ^{p} are all synthesized on an Artix-7 AC701 FPGA. The "—" indicates the Cortex-M4 platform is not able to support qTESLA-p-I and qTESLA-p-III due to memory limits.

qTESLA-I [5], modified the software reference implementation of qTESLA-I, and successfully demonstrated its performance by running it on an Artix-7 FPGA. Given the much higher frequency achieved in our design, as shown in Table 12, running qTESLA-I on our design is 346×, 2.7×, and 3.5× faster for key generation, signature generation, and verification, respectively, when compared to the results achieved in [6].

Hardware evaluations for other qTESLA instantiations using a High-Level Synthesis (HLS)-based hardware design methodology have been also explored [48]. However, the hardware designs generated by the HLS tool are too inefficient for embedded systems, e.g., for the smallest qTESLA-I parameters, it takes over 16× more LUTs compared to our "Murax +All" design when synthesized on the same Artix-7 FPGA.

4.4.1 Comparison to Digital Signature Schemes Beyond NIST's Candidates

When comparing with hardware acceleration for schemes not submitted to NIST's PQC standardization effort, arguably the most important work is the RISC-V based HW-SW co-design of XMSS [51] — a stateful hash-based scheme that was published as Request for Comments (RFC) 8931 in 2018. Several hardware accelerators based on the SHA256 hash function were provided in their work for accelerating the computations in XMSS. Comparing performance of [51] with our qTESLA design paints about the same picture as for the original software implementations: while qTESLA's key generation is much faster, qTESLA's sign and verification algorithms are slower compared to the corresponding XMSS algorithms. Interestingly the speedup from SW to HW-SW is larger for qTESLA than the speedup achieved for XMSS due to the efficient design of our GaussSampler accelerator.

[5]We would like to emphasize that this result should only be used for comparison purposes since qTESLA-I is outdated and withdrawn from the NIST submission.

A few publications [18,41] also focused on the pure FPGA based implementation targeting a specific lattice-based digital signature scheme. Their implementation only focuses on the signing and verification operations. More importantly, their design only supports fixed parameter set of (n, q) and this renders their hardware based designs not usable nowadays as the parameters and the construction of the schemes evolve.

5 Conclusion and Future Work

This paper presented a set of efficient and constant-time hardware accelerators for lattice-based operations. The key accelerator modules implemented are: a versatile cSHAKE core, a novel and elegant binary-search CDT-based Gaussian sampler, and a novel pipelined NTT-based polynomial multiplier with unified Cooley-Tukey and Gentlemen-Sande butterflies. In addition, sparse polynomial multiplier and Hmax-Sum modules were implemented. All of the modules can be fully parameterized at compile-time to help implement different sets of parameters without the need to re-write the hardware code. The accelerators were interfaced with the processor using a standard 32-bit interconnect bus, demonstrating ability to gain significant performance improvements without using unusual or custom interfaces. These flexible accelerators were then used to implement the first hardware-software co-design of the provably-secure lattice-based signature scheme qTESLA, namely qTESLA-p-I and qTESLA-p-III. We achieved over a 40-100x speedup for key generation, about a 10x speedup for signing, and about a 16x speedup for verification, compared to the baseline RISC-V software-only implementation. The hardware-software co-design demonstrated that with the hardware acceleration, the computationally intensive qTESLA-p-I and qTESLA-p-III can run as fast or faster than other lattice-based signature schemes (with smaller parameters or without provable parameters). In contrast to most hardware designs, this is also a fully open-source work and all of the hardware code is publicly available at `https://caslab.csl.yale.edu/code/qtesla-hw-sw-platform`.

The hardware accelerators provided in this work are also of interest for other lattice-based schemes, e.g., Dilithium and Falcon. Given that no research efforts in hardware have been made for these schemes, it would be interesting to see how they can benefit from the hardware accelerators proposed in this work. Another interesting line of research would be to analyze the side-channel resistance of the hardware accelerators and of the hardware-software co-design platform proposed in this work. Our design is fully constant-time and thus is resistant to timing side-channel attacks. However, other side channel attacks such as those exploiting statistical information of power and electromagnetic emanations are not covered in this work. Therefore, to protect against these more advanced attacks additional countermeasures might be necessary. In this context, since qTESLA signatures are probabilistic, it would also be interesting to analyze the level of protection that this feature provides in practice.

Acknowledgments

We would like to thank the reviewers and the shepherd for their useful feedback. Nina Bindel was supported by the NSERC Discovery Accelerator Supplement grant RGPIN-2016-05146. This work was supported in part by the Institute for Quantum Computing (IQC); IQC is supported in part by the Government of Canada and the Province of Ontario. This work was also supported in part by United States National Science Foundation grant number 1716541. We would like to also acknowledge Xilinx for donation of FPGA boards used in this work.

References

[1] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *28th ACM STOC*, pages 99–108. ACM Press, May 1996.

[2] Sedat Akleylek, Erdem Alkim, Paulo S. L. M. Barreto, Patrick Longa, Harun Polat, Jefferson E. Ricardini, and Gustavo Zanon. qTESLA's reference implementation. Available at https://github.com/qtesla/qTesla, commit-id d8fd7a5.

[3] Erdem Alkim, Paulo S.L.M. Barreto, Nina Bindel, Juliane Krämer, Patrick Longa, and Jefferson E. Ricardini. The lattice-based digital signature scheme qTESLA. In *18th ACNS (to appear)*, 2020. Available at https://eprint.iacr.org/2019/085.

[4] Victor Arribas. Beyond the limits: SHA-3 in just 49 slices. In *IEEE FPL 2019*, pages 239–245. IEEE, 2019.

[5] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In Josh Benaloh, editor, *CT-RSA 2014*, volume 8366 of *LNCS*, pages 28–47. Springer, Heidelberg, February 2014.

[6] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR TCHES*, 2019(4):17–61, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8344.

[7] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 313–314. Springer, Heidelberg, May 2013.

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak Reference, 2011. Available at http://keccak.noekeon.org/Keccak-reference-3.0.pdf.

[9] Nina Bindel, Sedat Akleylek, Erdem Alkim, Paulo S. L. M. Barreto, Johannes Buchmann, Edward Eaton, Gus Gutoski, Juliane Krämer, Patrick Longa, Harun Polat, Jefferson E. Ricardini, and Gustavo Zanon. qTESLA. Technical report, National Institute of Standards and Technology, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[10] Simon Blake-Wilson and Alfred Menezes. Unknown key-share attacks on the station-to-station (STS) protocol. In Hideki Imai and Yuliang Zheng, editors, *PKC'99*, volume 1560 of *LNCS*, pages 154–170. Springer, Heidelberg, March 1999.

[11] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR TCHES*, 2018(3):21–43, 2018. Available at https://tches.iacr.org/index.php/TCHES/article/view/7267.

[12] H. M. Cantero, S. Peter, Bushing, and Segher. Console hacking 2010 – PS3 epic fail. 27th Chaos Communication Congress, 2010. https://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf.

[13] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Technical report, National Institute of Standards and Technology, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[14] Chaohui Du and Guoqiang Bai. Efficient polynomial multiplier architecture for ring-LWE based public key cryptosystems. In *IEEE ISCAS*, pages 1162–1165. IEEE, 2016.

[15] Morris J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. *Federal Inf. Process. Stds. (NIST FIPS) – 202*, 2015. Available at https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[16] Farnoud Farahmand, Viet Ba Dang, Michal Andrzejczak, and Kris Gaj. Implementing and benchmarking seven round 2 lattice-based key encapsulation mechanisms using a software/hardware codesign approach. *Second PQC Standardization Conference*, 2019. Available at https://csrc.nist.gov/Events/2019/second-pqc-standardization-conference.

[17] Pierre-Alain Fouque and Thomas Vannet. Improving key recovery to 784 and 799 rounds of Trivium using optimized cube attacks. In Shiho Moriai, editor, *FSE 2013*, volume 8424 of *LNCS*, pages 502–517. Springer, Heidelberg, March 2014.

[18] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Lattice-based signatures: optimization and implementation on reconfigurable hardware. *IEEE Transactions on Computers*, 64(7):1954–1967, 2014.

[19] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. Throughput vs. area tradeoffs in high-speed architectures of five round 3 SHA-3 candidates implemented using Xilinx and Altera FPGAs. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 491–506. Springer, Heidelberg, September / October 2011.

[20] James Howe, Ayesha Khalid, Ciara Rafferty, Francesco Regazzoni, and Máire O'Neill. On practical discrete Gaussian samplers for lattice-based cryptography. *IEEE Transactions on Computers*, 67(3):322–334, 2018.

[21] Bernhard Jungk. *FPGA-based evaluation of cryptographic algorithms*. PhD thesis, Johann Wolfgang Goethe-Universität Frankfurt am Main, 2016.

[22] Bernhard Jungk and Jurgen Apfelbeck. Area-efficient FPGA implementations of the SHA-3 finalists. In *IEEE ReConFig*, pages 235–241. IEEE, 2011.

[23] Bernhard Jungk and Marc Stöttinger. Among slow dwarfs and fast giants: A systematic design space exploration of Keccak. In *IEEE ReCoSoC*, pages 1–8. IEEE, 2013.

[24] Bernhard Jungk and Marc Stöttinger. Serialized lightweight SHA-3 FPGA implementations. *Microprocessors and Microsystems*, 71:102857, 2019.

[25] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: testing and benchmarking NIST PQC on ARM Cortex-M4. *Second NIST PQC Standardization Conference*, 2019. Available at https://csrc.nist.gov/Events/2019/second-pqc-standardization-conference.

[26] John Kelsey, Shu-jen Chang, and Ray Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. Technical report, National Institute of Standards and Technology, 2016. Available at https://csrc.nist.gov/publications/detail/sp/800-185/final.

[27] Po-Chun Kuo, Wen-Ding Li, Yu-Wei Chen, Yuan-Che Hsu, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. High performance post-quantum key exchange on FPGAs. Cryptology ePrint Archive, Report 2017/690, 2017. Available at https://eprint.iacr.org/2017/690.

[28] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 319–339. Springer, Heidelberg, February 2011.

[29] ARM Ltd. AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite, 2019. Available at https://developer.arm.com/docs/ihi0022/d.

[30] Xianhui Lu, Yamin Liu, Dingding Jia, Haiyang Xue, Jingnan He, Zhenfei Zhang, Zhe Liu, Hao Yang, Bao Li, and Kunpeng Wang. LAC. Technical report, National Institute of Standards and Technology, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[31] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 598–616. Springer, Springer, Heidelberg, December 2009.

[32] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[33] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May / June 2010.

[34] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[35] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Technical report, National Institute of Standards and Technology, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[36] National Institute of Standards and Technology (NIST). Post-Quantum Cryptography Standardization. https://csrc.nist.gov/projects/post-quantum-cryptography, 2017. Accessed: 2020-04-08.

[37] Tobias Oder and Tim Güneysu. Implementing the NewHope-simple key exchange on low-cost FPGAs. In Tanja Lange and Orr Dunkelman, editors, *LATINCRYPT 2017*, volume 11368 of *LNCS*, pages 128–142. Springer, Heidelberg, September 2017.

[38] Chris Peikert. An efficient and parallel Gaussian sampler for lattices. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 80–97. Springer, Heidelberg, August 2010.

[39] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. Cryptology ePrint Archive, Report 2017/1014, 2017. Available at http://eprint.iacr.org/2017/1014.

[40] Thomas Poppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope. Technical report, National Institute of Standards and Technology, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[41] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 353–370. Springer, Heidelberg, September 2014.

[42] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *LATINCRYPT 2015*, volume 9230 of *LNCS*, pages 346–365. Springer, Heidelberg, August 2015.

[43] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2019. Available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`.

[44] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

[45] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-LWE cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 371–391. Springer, Heidelberg, September 2014.

[46] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2019. Available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`.

[47] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[48] Deepraj Soni, Kanad Basu, Mohammed Nabeel, and Ramesh Karri. A hardware evaluation study of NIST post-quantum cryptographic signature schemes. *Second NIST PQC Standardization Conference*, 2019. Available at `https://csrc.nist.gov/Events/2019/second-pqc-standardization-conference`.

[49] Abderrahim Tragha Soufiane El Moumni, Mohamed Fettach. High throughput implementation of SHA3 hash algorithm on field programmable gate array (FPGA). *Microprocessors and Microsystems*, 93:104615, 2019.

[50] Shanquan Tian, Wen Wang, and Jakub Szefer. Merge-exchange sort based discrete Gaussian sampler with fixed memory access pattern. In *IEEE FPT*, pages 126–134. IEEE, 2019.

[51] Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. XMSS and embedded systems. In *SAC*, volume 11959 of *LNCS*, pages 523–550. Springer Berlin Heidelberg, 2019.

[52] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, William Whyte, John M. Schanck, Andreas Hulsing, Joost Rijneveld, Peter Schwabe, and Oussama Danba. NTRU. Technical report, National Institute of Standards and Technology, 2019. Available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`.