# High-Performance Hardware Implementation of Lattice-Based Digital Signatures

Luke Beckwith, Duc Tri Nguyen, Kris Gaj
*George Mason University*, USA
{lbeckwit, dnguye69, kgaj}@gmu.edu

***Abstract*—Many currently deployed public-key cryptosystems are based on the difficulty of the discrete logarithm and integer factorization problems. However, given an adequately sized quantum computer, these problems can be solved in polynomial time as a function of the key size. Due to the future threat of quantum computing to current cryptographic standards, alternative algorithms that remain secure under quantum computing are being evaluated for future use. As a part of this evaluation, high-performance implementations of these candidate algorithms must be investigated. This work presents a high-performance implementation of all operations of CRYSTALS-Dilithium and one operation of FALCON (signature verification) targeting FPGAs. In particular, we present a Dilithium design that achieves the best latency for an FPGA implementation to date and, to the best of our knowledge, the first FALCON hardware implementation to date. We compare our results with the hardware implementations of all viable NIST Round 3 post-quantum digital signature candidates.**

***Index Terms*—Post-Quantum Cryptography, Digital Signature, Number Theoretic Transform, FPGA**

## I. Introduction

Current public key cryptographic standards, such as RSA and ECC, rely on the difficulty of the integer factorization and the discrete logarithm problem. However, with a sufficiently large quantum computer, Shor's algorithm [1] can be applied to solve these problems in superpolynomial run time [2].

While there is no known quantum computer capable of running Shor's algorithm to break current public-key standards, the process of selecting, standardizing, and deploying new public-key cryptosystems may take a substantial amount of time. In 2016, NIST announced the Post-Quantum Cryptography (PQC) standardization process aimed at developing new public-key standards resistant against quantum computers. In July 2020, NIST selected seven third-round finalists, including two lattice-based digital signature schemes: CRYSTALS-Dilithium and FALCON. All remaining candidates are currently being evaluated in terms of their security, key and ciphertext/signature size, performance in both software and hardware, and several other criteria. While software implementations are relatively easy to develop and benchmark, hardware implementations require a substantial amount time and design effort to determine their efficiency in terms of speed, area, power, and energy. FPGA implementations of PQC submissions are necessary to provide insight into the cost and performance of these algorithms in hardware.

**Contributions.** In this work, we present a high-performance FPGA implementation of CRYSTALS-Dilithium designed at the Register Transfer Level (RTL) using Verilog. We also present a high-performance implementation of FALCON verification designed in VHDL. In particular, we make the following contributions:

- We present a high-performance implementation of Dilithium supporting all operations and security levels and achieving the lowest reported latency for all operations.
- We also present a hardware implementation of FALCON verification, which is, to the best of our knowledge, the first reported hardware implementation of FALCON.

The source code for Dilithium in this paper is publicly available at: https://github.com/GMUCERG/Dilithium. The source code for FALCON verification will be made public upon publication.

## II. Previous Work

The existing hardware and hardware/software (HW/SW) implementations of digital signatures schemes qualified to Round 3 of the NIST PQC standardization process are summarized in Table I. Hardware design may focus either on maximizing performance or minimizing the area and power the design consumes. These approaches lead to substantial differences in hardware architecture, and thus both types of implementations are important to measure an algorithm's performance in hardware. Therefore, the implementations in Table I are split into High-Speed or Lightweight. However, the dividing line is not always apparent as designs may seek to make area/performance trade-offs. These labels are to help differentiate the results of various implementations but should not be considered absolute.

Currently, the lattice and symmetric-based algorithms are the most viable candidates as multivariate schemes have received concerning success in cryptanalysis [3]. The lattice-based algorithms have better performance and smaller signatures size, making them strong candidates. However, the symmetric-based signatures may provide more confidence in their security, which is based on preexisting symmetric primitives.

Prior to our previously reported work [14], Dilithium had received two implementations. In [4], a high-performance design for Dilithium version 2 [15] is described. The authors report area for individual modules instantiated for level 3 and performance for security levels 1-4. To improve polynomial arithmetic performance, they used a $2 \times 2$ butterfly for their

TABLE I: Status of RTL implementations for Round 3 digital signature schemes

| Algorithms | High-Speed | Lightweight | HW/SW |
|---|---|---|---|
| **Lattice-based** | | | |
| CRYSTALS-Dilithium | [4], **TW** | [5], [6]*** | [7], [8]*, [9], [10]*** |
| FALCON | **TW** | – | – |
| **Multivariate** | | | |
| GeMSS | – | – | – |
| Rainbow | [11]** | – | – |
| **Symmetric-based** | | | |
| Picnic | [12] | – | – |
| SPHINCS+ | [13] | – | – |

*extended version of [7]
**only for Round 1 and Round 2 parameter sets
*** Supports Dilithium and Saber

Number Theoretic Transform (NTT) and duplicated the module multiple times in certain operations. As an example, their sign operation is split into many parallel operations, including 12 instances of their NTT unit. This allows this design to achieve high performance, but at the cost of a large number of DSPs and BRAM. They also used multiple Keccak cores to enable more parallelization in polynomial sampling. Another implementation, [5], describes a mid-range implementation for version 3.1 of Dilithium and focuses on achieving the best performance possible with reasonable resource utilization. This design consists of three top-level modules, each capable of performing all operations at a single security level and an individual module that only performs a single operation at a single security level. This design does utilize some parallelization, for example, the use of two butterfly cores in the NTT. However, it limits duplication to maintain lower resource utilization. Since releasing our work on Dilithium, a HW/SW implementation [9] and two RTL implementations supporting both Saber and Dilithium [6], [10] have been made public. In particular, [10] notes that the Dilithium NTT parameters can be applied to Saber for polynomial multiplication with almost no modification if a slight failure rate is accepted. A similar multiplier approach is used by [6], which implements the entire Saber and Dilithium algorithms. The HW/SW implementation [9] investigates the performance area trade-offs of moving the Keccak core and polynomial multiplier into FPGA accelerators on several platforms and compares them against C and NEON optimized implementations.

FALCON has not received any form of hardware implementation to date. This is likely due to the complexity of the algorithm. In particular, the key generation and signature generation functions are both complex and not naturally hardware friendly. For example, these functions require floating-point FFT operations with 53 bits of precision, which are costly to implement on FPGA. Further, the authors of the algorithms note that a limitation of FALCON is the complexity of key generation and signature generation, which are delicate to implement [16]. The tree sampling of signature generation requires recursive FFT operations, which are challenging to implement in hardware and require a large amount of memory. Given the difficulty posed in fully implementing all operations of FALCON and the short timeline remaining for the conclusion of NIST round 3, we chose to focus on implementing verification. Verification is a simpler operation requiring only integer polynomial operations. It is also the most common and most likely operation to be performed on resource-constrained Internet of Things (IoT) devices, which rarely generate signatures but frequently verify signatures to confirm the authenticity of connections. This makes verification performance an important metric for evaluation.

## III. Background

### A. Dilithium

Dilithium is a member of the Cryptographic Suite for Algebraic Lattices (CRYSTALS) along with the Key Encapsulation Mechanism (KEM) Kyber. The core operations of Dilithium are the arithmetic of polynomial matrices and vectors. As described in [17], Dilithium is a Fiat-Shamir with Aborts [18], [19] style signature scheme and bases its security upon the Module Learning with Errors (M-LWE) and Module Short Integer Solution (M-SIS) problems. The M-LWE problem can be briefly described as follows: Let $A \in R_q^{k\times l}$ be uniformly chosen, $s_1 \in R_q^l$, and $s_2 \in R_q^k$. Then, the standard M-LWE problem is to distinguish $(A, A\cdot s_1 + s_2)$ from $(A, u)$, where $u$ is a uniformly chosen vector. The M-SIS problem can be briefly described as follows: Let $A$ be a uniformly chosen $R_q^{k\times l}$ matrix, then find a non-zero vector $x \in R_q^l$ such that the norm of $x$ is less than $\beta$ for some $\beta$ and $A\cdot x = 0$. Dilithium is built upon polynomial arithmetic using these hard problems. The security of the algorithm is primarily adjusted by changing the $k, l$ parameters, which determine the size of matrices and arrays.

The full description of the key generation of CRYSTALS-Dilithium is shown in Algorithm 1. The only input is a random 32-byte seed $\zeta$. The public $A$ matrix is sampled from a 32-byte seed $\rho$ using the SHAKE128 XOF, and the secret $s$ polynomials are sampled from the 32-byte seed $\sigma$ using the SHAKE256 XOF. An incrementing nonce is appended to seed when sampling each polynomial in the vectors and matrices to generate a unique output. As an example, $s_2$ is sampled using the same seed as $s_1$, $s_1$ is sampled using the nonce values from $[0, l-1]$ and $s_2$ from $[l, 2l-1]$. The core polynomial operation is the matrix multiplication and vector addition:

$$\vec{t} = \begin{bmatrix} a_{0,0} & \cdots & a_{l-1,0} \\ \vdots & \ddots & \vdots \\ a_{0,k-1} & \cdots & a_{l-1,k-1} \end{bmatrix} \begin{bmatrix} s_1(0) \\ \vdots \\ s_1(l-1) \end{bmatrix} + \begin{bmatrix} s_2(0) \\ \vdots \\ s_2(k-1) \end{bmatrix}$$

The resulting polynomial vector $t$ can be released publicly as the M-LWE problem ensures that the secret values $s_1, s_2$ cannot be recovered only knowing $A, t$. To reduce the size of the public key, only the upper bits of $t$ are transmitted. $Power2Round$ accomplished this by splitting $t$ into two vectors, one containing the $23 - d$ most significant bits of the coefficient and the other containing the $d$ least significant bits, where $d$ is a parameter value shown in Table II. This greatly reduced the transmission size of the public key but required additional information in verification to recover from the missing bits. This is accomplished through the use of a hint, which is generated using the lower bits of $t$ during sign.

```
Algorithm 1: Dilithium Key Generation
Input: $\zeta \in \{0,1\}^{256}$
Output: $pk = (\rho, t_1)$, $sk = (\rho, K, tr, s_1, s_2, t_0)$
1 $(\rho, \sigma, K) \leftarrow H(\zeta)$
2 $A \leftarrow$ExpandA$(\rho)$    $s_1, s_2 \leftarrow$ExpandS$(\sigma)$
3 $t \leftarrow A \cdot s_1 + s_2$
4 $(t_0, t_1) \leftarrow$ Power2Round$(t, d)$
5 $tr \leftarrow H(\rho || t_1)$
```

```
Algorithm 2: Dilithium Signature Generation
Input: $sk = (\rho, K, tr, s_1, s_2, t_0), M \in \{0,1\}^*$
Output: $\sigma = (\hat{c}, z, h)$
1 $\mathbf{A} \leftarrow$ExpandA$(\rho)$    $\mu \leftarrow H(tr || M)$    $\rho' \leftarrow H(K || \mu)$
2 $k \leftarrow 0$    $abort \leftarrow 1$
3 while $abort$ do
4     $abort \leftarrow 0$
5     $y \leftarrow$ExpandMask$(\rho', k)$
6     $w \leftarrow \mathbf{A} \cdot y$
7     $w_1 \leftarrow$HighBits$(w, 2\gamma_2)$
8     $\hat{c} \leftarrow H(\mu || w_1)$
9     $c \leftarrow$SampleInBall$(\hat{c})$
10    $z \leftarrow y + c \cdot s_1$
11    $w_0 \leftarrow$LowBits$(w, 2\gamma_2)$
12    if $||z||_\infty \geq \gamma_1 - \beta$ or $||w_0 - c \cdot s_2||_\infty \geq \gamma_2 - \beta$ then
13        $abort \leftarrow 1$
14    else
15        $h \leftarrow$MakeHint$(w_1, w_0 - c \cdot s_2 + c \cdot t_0, 2\gamma_2)$
16        if $||c \cdot t_0||_\infty \geq \gamma_2$ or $\sum h_i > \omega$ then
17            $abort \leftarrow 1$
18    $k = k + l$
```

```
Algorithm 3: Dilithium Signature Verification
Input: $pk = (\rho, t_1)$, $M \in \{0,1\}^*, \sigma = (\hat{c}, z, h)$
Output: Valid or Invalid
1 $\mathbf{A} \leftarrow$ExpandA$(\rho)$
2 $\mu \leftarrow H(H(\rho || t_1) || M)$
3 $c \leftarrow$SampleInBall$(\hat{c})$
4 $(w_1, w_0) \leftarrow$UseHint$(h, A \cdot z - c \cdot t_1 \cdot 2^d)$
5 if $||z||_\infty < \gamma_1 - \beta$ & $\hat{c} = H(\mu || w_1)$ & $\sum h_i \leq \omega$ then
6     return Valid
7 return Invalid
```

Signature generation is described in Algorithm 2. The first step is to sample a vector $y$ based on the secret key and the hash of the message. This vector is then multiplied by the public matrix $A$, and the result is designated $w$.

$$\vec{w} = \begin{bmatrix} a_{0,0} & \cdots & a_{l-1,0} \\ \vdots & \ddots & \vdots \\ a_{0,k-1} & \cdots & a_{l-1,k-1} \end{bmatrix} \begin{bmatrix} y_1(0) \\ \vdots \\ y_1(l-1) \end{bmatrix}$$

This vector is decomposed into two polynomial vectors $w_1$, $w_0$ by a constant $\alpha$ such that $w_1(i) \cdot \alpha + w_0(i) = w(i)$ for all $w(i) \in w$. This allows the effect of small noise to be removed from $w$ before hashing. The quotient coefficients $w_1(i)$ are referred to as the high-order bits, and the remainder $w_0(i)$ is referred to as the low-order bits. $HighBits$ is the function that returns a polynomial vector composed of the higher-order bits of the input vector $w$. We denote this vector as $w_1$. $LowBits$ gives a vector composed of the lower-order bits of $w$. We denote this vector as $w_0$. The coefficients of $w_1$ are concatenated and hashed with $\mu$ to generate a byte vector $\hat{c}$. A short polynomial $c$ is sampled from $\hat{c}$ using a variant of the Fisher-Yates shuffle [17]. The function to perform this operation is called $SampleInBall$. The polynomial $c$ has $\tau$ coefficients equal to $\pm 1$. The remaining coefficients are 0. This polynomial is then used to calculate $z$ as follows.

$$\vec{z} = \begin{bmatrix} y(0) \\ \vdots \\ y(l-1) \end{bmatrix} + c \cdot \begin{bmatrix} s_1(0) \\ \vdots \\ s_1(l-1) \end{bmatrix}$$

As mentioned in key generation, a hint is also needed so that the verifier can recover from the missing bits in $t_1$. This hint contains the coefficient indices that require a carry bit in the verification operation. The hint is generated by the $MakeHint$ operation using $c, w_0, s_2$ and $t_0$. Once this operation is complete, the $(\bar{c}, z, h)$ set is a potential signature candidate. However, it can be observed above that $z$ and $s_1$ are closely related, with the secret $s_1$ only being masked by multiplication with the public polynomial $c$ and addition with the $y$ vector. In some circumstances, $z$ may leak information about the long-term secret $s_1$. As such, the max-norm of several vectors must be checked to ensure security. The max norm of a polynomial $x \in R^n$ is defined as $||x||_\infty = max(x_0, ..., x_n)$. If it exceeds certain predefined bounds, the signature is rejected, and a new attempt is generated by sampling $y$ with a new nonce. Further, the hint has a maximum size $\omega$. If too many hint bits are needed, a new attempt will be needed as well. Dilithium requires $3 - 5$ attempts on average to generate a secure signature depending on the security level.

In Algorithm 3, verification is described. It attempts to recreate the $\hat{c}$ seed using the message, public key, and signature. This is accomplished using the following operations:

$$\vec{w'} = \begin{bmatrix} a_{0,0} & \cdots & a_{l-1,0} \\ \vdots & \ddots & \vdots \\ a_{0,k-1} & \cdots & a_{l-1,k-1} \end{bmatrix} \begin{bmatrix} z(0) \\ \vdots \\ z(l-1) \end{bmatrix} - c \cdot \begin{bmatrix} s_1(0) \\ \vdots \\ s_1(k-1) \end{bmatrix}$$

Presuming all inputs are valid, $w'_1$ will be equivalent to $w_1$ as

$$w' = A \cdot z - c \cdot t = A \cdot (y + c \cdot s_1) - c \cdot (A \cdot s_1 + s_2)$$
$$\implies w' = A \cdot y - c \cdot s_2$$

TABLE II: Dilithium parameters for version 3.1 at all supported security levels (2, 3, and 5).

| Parameter | Value 2 | Value 3 | Value 5 |
|---|---|---|---|
| $q$ [modulus] | | $2^{23} - 2^{13} + 1$ | |
| $d$ [dropped bit from t] | | 13 | |
| $\tau$ [# of +/- 1's in c] | 39 | 49 | 60 |
| $\omega$ [max # of 1's in hint] | 80 | 55 | 74 |
| $(k, l)$ [Vector Dimensions] | (4,4) | (6,5) | (8,7) |
| $\eta$ [secret coefficient range] | 2 | 4 | 2 |
| $\gamma_1$ [y coefficient range] | $2^{17}$ | $2^{19}$ | $2^{19}$ |
| $\gamma_2$ [low-order rounding range] | $\frac{q-1}{88}$ | $\frac{q-1}{32}$ | $\frac{q-1}{32}$ |

| | Dilithium | FALCON (Verify) |
|---|---|---|
| Security Basis | M-LWE, SIS | NTRU-SIS |
| NTT-Friendly | Yes | Yes |
| Hash Function | SHA-3 | SHA-3 |
| Elements | Vectors/matrices of polynomials $(k, l) \in [(4,4),(6,5),(8,7)]$ | Polynomials |
| Polynomial Degree | 256 | [512,1024] |
| Coefficient Modulus | 8,380,417 (23-bit) | 12,289 (14-bit) |

TABLE III: High level comparision between Dilithium and FALCON verification

As noted previously, since only the upper bits of $t$ are used in this operation, a hint is applied using $UseHint$ to account for the carry bits from the missing lower bits of $t$. Both $s_2$ and $c$ have small coefficients and thus will not contribute substantially to the higher-order bits of $w'$. When it is decomposed, the upper bits will be equivalent to the upper bits of $A \cdot y$, and thus when it is hashed, it will recreate $\hat{c}$.

## B. FALCON

FALCON is a lattice-based cryptosystem for digital signatures over NTRU lattices. While Dilithium is lattice-based as well, there are substantial differences. Unlike Dilithium, FALCON builds its security on the NTRU Short Integer Solution (NTRU-SIS) problem [16]. In FALCON, key generation and signature generation are very complex, involving matrix decomposition, a complex trapdoor sampler, and floating-point Fast Fourier Transform operations. However, verification is much simpler and has very high performance. It also has smaller signatures and public keys than Dilithium, reducing the data that must be transmitted for verification. Table III shows a high level comparison between Dilithium and FALCON verification.

The main elements used in verification are integer polynomials of degree 512 or 1024 with coefficients modulo 12289. The ring of FALCON was chosen such that it supports a full NTT transformation.

The core operation of FALCON verification is the calculation of $s_1 = c - h \times s_2$ as shown in line 4 of the Algorithm 4, where the multiplication between two large polynomials is computed using NTT. The signature is accepted or rejected based the norm of $(s_1, s_2)$, where $||(s_1, s_2)||^2 = \sum_{i=0}^{N-1} s_1(i)^2 + s_2(i)^2$.

## C. Number Theoretic Transform

It is well known that a Discrete Fourier Transform (DFT) may be used to compute polynomial multiplication. However,

**Algorithm 4:** FALCON Signature Verification

```
Input: pk = $\bar{h}$, $M \in \{0,1\}^*, \sigma = (r, \bar{s})$
Output: Valid or Invalid
1 $c \leftarrow$ HashToPoint$(r||M)$
2 $h \leftarrow$ Decode$(\bar{h})$
3 $s_2 \leftarrow$ Decompress$(\bar{s})$
4 $s_1 \leftarrow c$ - NTT$^{-1}$(NTT$(h) \circ$ NTT$(s_2)$)
5 if $||(s_1, s_2)||^2 \leq \lfloor \beta^2 \rfloor$ then
6     return Valid
7 return Invalid
```

a straightforward computation of a DFT requires $O(n^2)$ multiplications. This can be reduced using the Fast Fourier Transform (FFT), which computes the same results but only requires $O(n \log n)$ multiplications. Traditionally this is performed on complex numbers, but it can be applied using integers when acting on a quotient ring. This variant is called the Number Theoretic Transform (NTT). A ring is referred to as NTT-friendly if it is of the form $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, where $n$ is power of 2 and $q \equiv 1 (\text{mod } 2n)$. The first condition allows an $n$-point NTT to be used in place of the standard $2n$-point, and the second guarantees that there are primitive roots of unity with high enough order to perform a full transform. However, it is worth noting that while these conditions are optimal for NTT based multiplication, the NTT can still be applied with looser conditions. As an example, Kyber does not have a $2nth$ root of unity but still uses an incomplete NTT to accelerate polynomial multiplication [20], and through several clever tricks, the NTT can be applied to NTRU, which has a power of 2 coefficient modulus and a prime degree reduction polynomial [21].

Both the Dilithium and FALCON verify operations are performed over NTT-friendly rings. Polynomial multiplication can be efficiently calculated using the NTT as follows: $a \times b = NTT^{-1}(NTT(a) \circ NTT(b))$, where $\circ$ is point-wise multiplication of the polynomial coefficients. This approach reduces the complexity of polynomial multiplication from $O(n^2)$ to $O(n \log n)$.

## IV. Hardware Design

### A. Dilithium

*1) Design Methodology:* We present a combined architecture for Dilithium key generation, signature generation, and signature verification. The primary difference between security levels is the dimension of vectors and matrices, which only causes an increase in BRAM utilization. Therefore we chose to support all security levels in a single architecture. This section will discuss the main subcomponents used to implement the Dilithium algorithms, the high-level implementation, and the scheduling of the design. The high-level block diagram is shown in Fig. 2.

Our design decisions were informed by the previous works on Dilithium as well as the other signature candidates. As seen previously in Table I, the symmetric-based candidates have

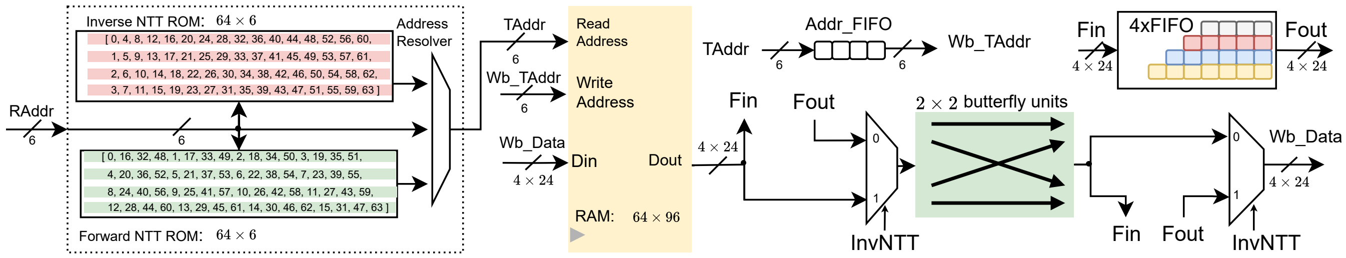


Fig. 1: CRYSTALS-Dilithium $2 \times 2$ NTT architecture with Address Resolver, $4\times$FIFO for data, Addr_FIFO for indices

received only high-performance RTL implementations. Thus, for a fair comparison between families of algorithms, our work also focuses on high-performance RTL implementation. This allows us to compare using latency as a single objective metric as opposed to area comparison's which must compare multiple different resources.

Based on the previous works on Dilithium [4], [22], we chose to use the $2 \times 2$ butterfly approach for our NTT. However, we limit DSP usage to multiplication as simpler operations can be implemented using LUTs without impacting the overall critical path of the design. We also chose to use multiple Keccak cores to parallelize sampling but were careful to use only the minimum number of cores required to prevent sampling from becoming the bottleneck of the design. Our sign operation is split into several stages, but we focus on splitting the operation at steps that require minimal data transfer and hardware duplication.

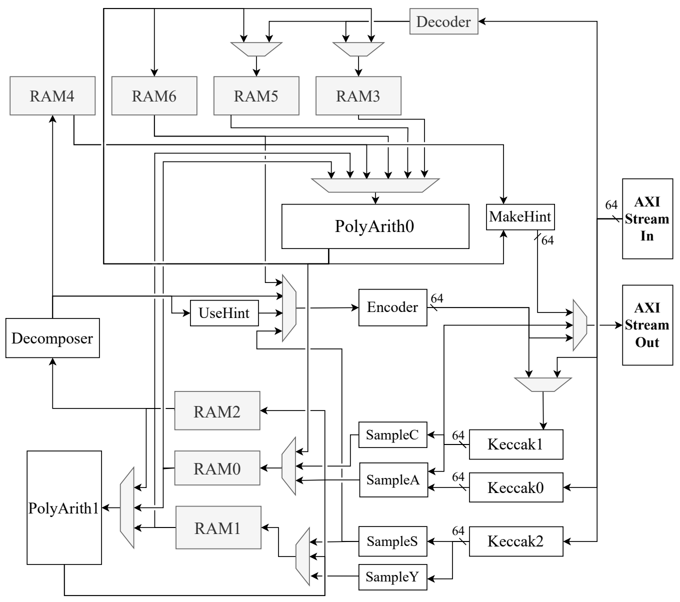


Fig. 2: Block diagram of the combined architecture of CRYSTALS-Dilithium. Bus widths are 96 bits unless shown otherwise.

*2) Polynomial Arithmetic:* In our design, the polynomial arithmetic unit, PolyArith, is used to perform the NTT, point-wise multiplication, addition, and subtraction. For all integer multiplications, Barrett reduction is used to perform the modulo operation as it can be implemented efficiently in hardware through the use of shifts and additions in place of the constant multiplications. Our design utilizes four butterfly units, each capable of performing the basic arithmetic operations needed as well as the Cooley-Tukey and Gentlemen-Sande butterfly operations. This design choice was centered around the NTT as it is the most costly and complex polynomial operation. With four butterflies, our design can make use of a $2 \times 2$ butterfly arrangement, which gives the benefits of processing four pairs of coefficients in parallel but with a lower memory throughput requirement than processing four pairs of coefficients in a single layer. This is done by processing two layers of the NTT per memory access. Between the layers, the NTT pipeline must be stalled to wait for the write back to complete. However, using a $2 \times 2$ arrangement reduces the number of write-back conflicts and is combined with coefficient rearrangement, removing the need for stall almost entirely. This is an optimal trade-off, as it provides better performance than processing a single layer at once but does not utilize excessive resources. The next viable arrangement would be to use 16 butterflies in a $4 \times 4$ arrangement. However, this would increase the area by a factor of 4 and provide only minimal improvement to the overall performance. Since the butterfly hardware is reused for all polynomial arithmetic, four pairs of coefficients are processed in parallel for all operations.

Our design for the $2 \times 2$ NTT unit, shown in Fig. 1, builds upon the pipelined NTT design by Nguyen et al. [23], [24], which processed four coefficients per clock cycle and 2 NTT layers at once. The ideal cost of the Forward and Inverse NTT is $\frac{n}{4} \cdot \frac{\log n}{2}$ clock cycles, where $n$ is the degree of the polynomial. The BRAM stores 4 coefficients per row and uses $n/4$ rows. In case of Dilithium, $n = 256$. Thus, the polynomial configuration is $64 \times 4$ coefficients, corresponding to the $64 \times 96$-bit RAM. Similarly, with Falcon $n = 1024$, the configuration is $256 \times 4$ coefficients and corresponding to the $256 \times 56$-bit RAM. Instead of multiplying with $n^{-1}$ in the last stage of Inverse NTT, we incorporate the divide-by-2 technique [25] into the four $2 \times 2$ butterfly units at every level of the Inverse NTT. Each butterfly unit is deeply pipelined and uses DSP units for multiplication to improve the critical path. Please note that the Forward and the Inverse NTT use Decimation-In-Frequency (DIF) and Decimation-In-Time (DIT) variants of NTT, respectively. This prevents the need for the bit-reversal operation. Hence, all butterfly units in our design must support both DIF and DIT so that a single $2 \times 2$ NTT unit is capable of performing both Forward and Inverse NTT.

In our design process, we set the constraint that the memory must be written to in Natural Order from other modules. This constraint simplifies our design since other modules do not have to shuffle coefficients in order to prepare them for the Forward NTT operation, which would delay its commencement. As shown in Fig. 3, the Forward NTT starts with the memory indices in Natural Order. After Forward NTT, the memory indices are shuffled in blocks of four coefficients. To find the True Address, we use a simple mapping (performed by Address Resolver), so any query to the memory can read the correct RAM contents. After Inverse NTT, the memory indices are converted back to Natural Order. By using this approach, our hardware design becomes concise and easy to implement.

One challenge of a $2 \times 2$ butterfly is correctly loading the coefficients. For the Forward NTT with $n = 256$, coefficients are loaded in pairs of distance $2^{7-j}$ at layer $j$. Thus, for layer $j = 0$, $(x_0, x_{128})$ would be the first coefficient pair loaded into a butterfly. Then $(x_0, x_{64})$ would be the first pair for the next layer $j = 1$, and so on. To meet this requirement at full throughput, while using only a single simple dual-port RAM, we introduce components called `4xFIFO` and `Addr_FIFO`, shown in Fig. 1.

The `4xFIFO` is used to transpose a $4 \times 4$ matrix of coefficients to properly order the coefficients. This transpose operation must be executed before data enters the butterfly unit in the Forward NTT (DIF) and after it exits this unit in the Inverse NTT (DIT). The same 4×FIFO unit, operating in two different modes shown in Figs. 4 and 5, is used for both of these operations.

In Fig. 4, we illustrate the operation of this unit at the start of the Forward NTT. `4xFIFO` consists of four shift registers shown in this figure as rows of adjacent squares. Each square represents a register capable of holding one polynomial coefficient. The lengths of these shift registers, considered in the bottom-up order, are 7, 6, 5, and 4 positions, respectively. The four leftmost positions of each shift register can be written to in parallel. The rightmost column of the entire `4xFIFO` component can be read in parallel. Every clock cycle, the contents of each shift register is shifted by one position to the right. Every clock cycle, one and only one of these registers has its four leftmost positions written to in parallel. In Fig. 4, these positions are marked using blue rectangular frames. After the initialization, lasting 3 clock cycles, in each subsequent clock cycle, four coefficients are read from the rightmost column of the entire `4xFIFO` component. These positions are marked using red rectangular frames. The same `4xFIFO` unit can be reused for the Inverse NTT operation, as shown in Fig. 5. During this transformation, the `4xFIFO` is used after the butterfly transform, before the writeback to memory. Rather than write-by-row, read-by-column operation used in Forward NTT, the `4xFIFO` is configured to execute the write-by-column, read-by-row operation.

The Address Resolver unit, shown in Fig. 1, is responsible for converting the Representation Address (`RAddr`) in the NTT algorithm to True Address (`TAddr`) to account for the reordering that occurs in the Forward and Inverse NTT operation. When $n = 256$, it does so by using two $64 \times 6$ ROMs to map the address when required by the operation. Contents of the respective ROMs, Forward NTT ROM and Inverse NTT ROM, are shown Fig. 1. To determine the contents of ROMs, we examined the order of indices before and after the NTT transform, such that the conversion between `RAddr` and `TAddr` guarantees that the RAM words `RAddr` refers to are always correct. The construction of the mapping tables depends only on the parameter $n$ and the writeback pattern of the FIFO buffer, which are fixed at runtime. We decided to use a ROM-based approach for $n \leq 256$ since the entire ROM content is able to fit efficiently into LUTs. For $n > 256$, it should be noted that the ROM content can also be computed on the fly using the Algorithm 5 with bit shifting and masking. With `Address Resolver` unit, we completely eliminate the execution time of a shuffle and re-ordering at the cost of the negligible amount of extra memory. The `Addr_FIFO` unit is responsible for delaying `TAddr` by 4 clock cycles (which is the depth of the pipeline of the $2 \times 2$ butterfly units).

**Algorithm 5:** Address Resolver ROM calculation

```
Input: Representation address (RAddr) $i$, $n = 2^k$
Output: True address (TAddr)
1 $f = n \gg 4 = 2^{k-4}$
2 if mode = Forward NTT then
3     return $(i \bmod 4) \times f + i/4$
4 else if mode = Inverse NTT then
5     return $(i \bmod f) \times 4 + i/f$
6 else
7     return $i$
```

*3) BRAM Configuration:* As discussed in previous sections, the polynomial arithmetic modules process four coefficients per clock cycle. Thus, the bandwidth requirement is $96 = 4 \times 24$ bits. The smallest dual-port BRAM configuration that can accommodate this width in today's Xilinx FPGAs is composed of three 36-kbit BRAMs, each configured as 1024x36 memory [26]. This configuration can efficiently store two vectors of polynomials at the security level 5, with the $4 \cdot 24/3 \cdot 36 = 89\%$ utilization of each memory word. This structure allows us to efficiently utilize BRAM for polynomial storage, leading to lower BRAM utilization than previously reported implementations.

*4) Keccak and Polynomial Samplers:* In Dilithium, the polynomials composing vectors and matrices are independently sampled using a constant seed value and an appended incrementing nonce as the input to either *SHAKE128* or *SHAKE256*. This allows parallel sampling of polynomials if multiple Keccak cores are used. While the Keccak core

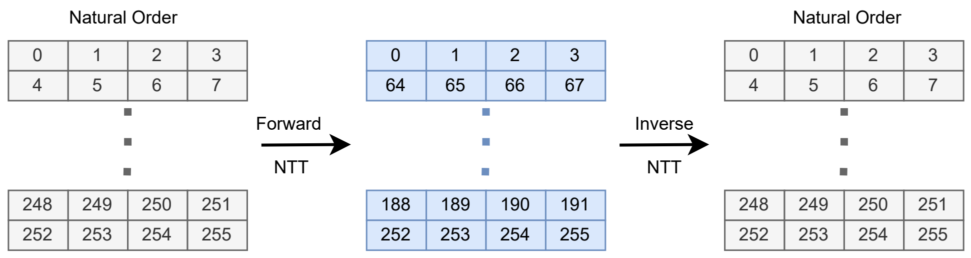


Fig. 3: Example of the memory indices during NTT Transform $n = 256$

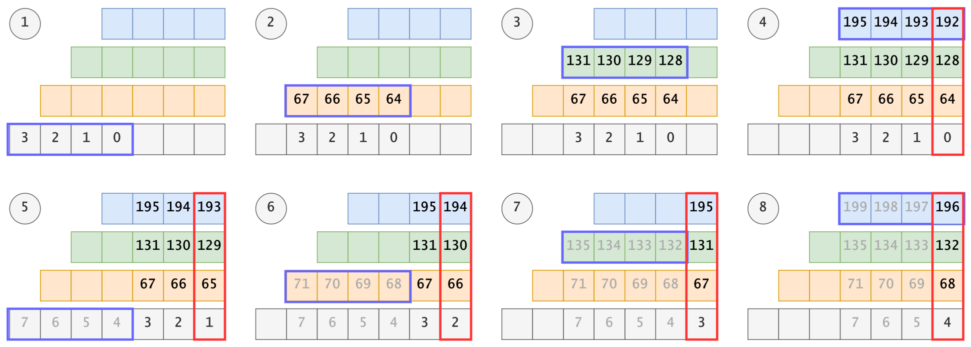


Fig. 4: Example of the operation of the reorder FIFO for Forward NTT

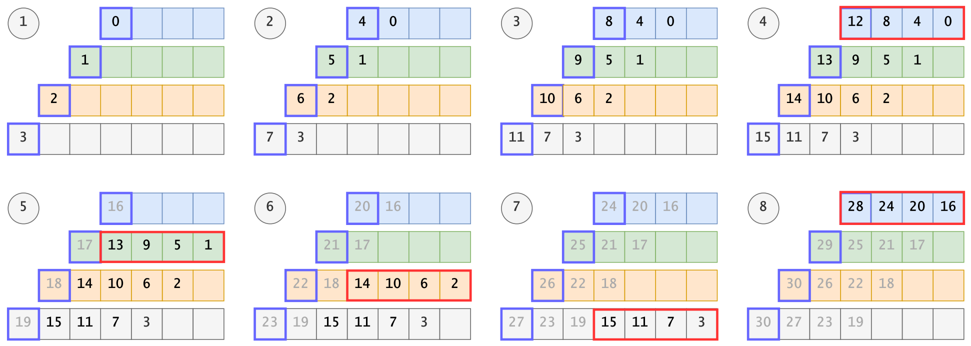


Fig. 5: Example of the operation of the reorder FIFO for Inverse NTT

consumes a substantial amount of resources, a large amount of pseudorandom data is required in Dilithium. As an example, at security level 5, the $\mathbf{A}$ matrix is of dimensions $8 \times 7$, with each sample requiring 24-bits of pseudorandom data. While the rejection rate is low, this still requires a minimum of *8*7*256*3=43KB* of pseudorandom data. Producing that amount of pseudorandomness quickly becomes the performance bottleneck in high-performance designs. Thus, we use three Keccak cores in our design. Two are primarily used for sampling of the $\mathbf{A}$ matrix, and the third is used for the remaining hashing and sampling. To improve the performance of sampling, multiple coefficients are sampled in parallel to fully utilize the Keccak core. The Keccak module utilized in our design is a preexisting implementation [27]. We selected three Keccak cores because it was the minimum number of instances that prevented polynomial sampling from becoming the operational bottleneck of the design.

## B. *FALCON Verification*

As observed earlier, while key and signature generation in FALCON are complex operations, verification is a much simpler operation that lends itself to a very efficient hardware implementation. This work focuses on optimizing FALCON signature verification in hardware for high performance. Parallelization is possible in the decoding of polynomials as well as the hashing of the message and sampling of the $c$ polynomial. The datapath of the architecture is four polynomials wide to take advantage of the same NTT architecture discussed previously. The only modification made to the NTT was to modify the butterfly units for the smaller modulus and to add a conditional bypass of the last layer since FALCON-512 required an odd number of NTT layers.

Decoding polynomials is simple deserialization that can be performed using a bus width converter. Decompression is somewhat more complicated, but it can be performed simply in hardware using a priority encoder and a conditional subtraction, as shown in Fig. 6. The least significant 7 bits are unchanged for each coefficient, and the upper bits are encoded using a unary encoding. There is also a sign bit prefix for each coefficient. Each coefficient is mapped to 9-25 bits, Fig. 7 demonstrates the compression format. The decoded and decompressed polynomials are multiplied together after being converted to the NTT domain. The decoder is optimized to handle four coefficients per cycle so that the first NTT operation can begin as soon as possible. However, decompression is performed in parallel with an NTT optimization. Thus, it only decompresses one coefficient per cycle to minimize area. The norm is calculated by the modules shown in Fig. 8 and can handle four coefficient inputs in parallel. The top-level architecture can be seen in Figure 9.

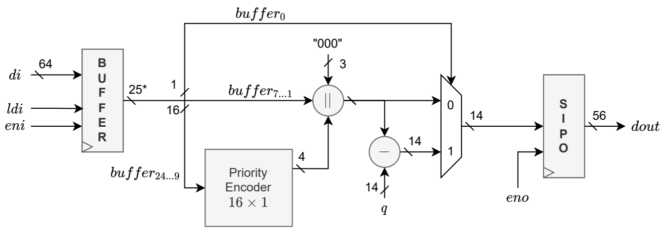


Fig. 6: FALCON Decompression Architecture

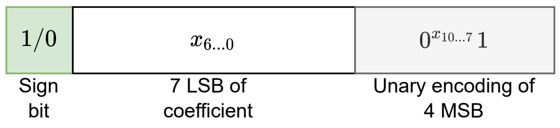


Fig. 7: FALCON Decompression Format

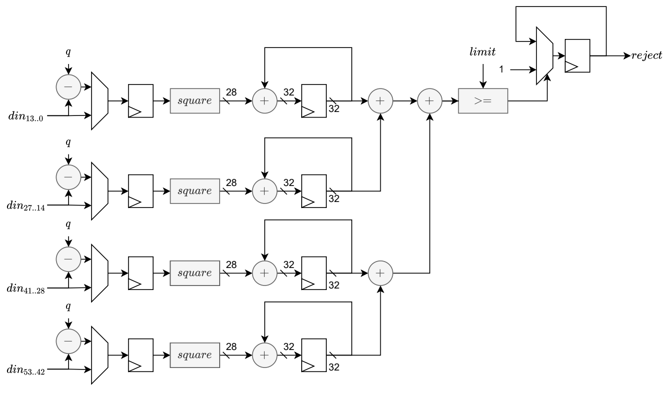


Fig. 8: FALCON Norm Architecture

## V. Operation Scheduling

Thoroughly designed and optimized scheduling of operations is crucial for efficient hardware implementations. Creating a high-performance design with an acceptable design area is only possible with high utilization of components and constant progress on the core operation. As such, we have highly optimized our operation scheduling to maximize utilization of the polynomial arithmetic units, which are the core of our design and are responsible for the majority of the operations in Dilithium.

### *A. Dilithium Key Generation*

The scheduling diagrams of key generation is shown in Fig. 10. The core operations are matrix multiplication and vector addition. The first polynomial vector $s_1$ is sampled immediately in parallel with the sampling of the $A$ matrix. The vector $s_1$ is encoded as it is sampled. The NTT is applied to $s_1$ and as soon as NTT completes, the matrix multiplication with the fully sampled $A$ matrix is performed. The inverse NTT operation is then performed on the result. The result of the addition with $s_2$ is immediately split into the MSB and LSB components, and the MSB components are encoded and hashed in parallel with the addition. When the hash computations are finished, the LSB components are encoded, and the operation is then complete.

### *B. Dilithium Signature Generation*

Signature generation is the most complex operation and is split between Figs. 11 and 12. As mentioned previously, to improve performance, we split the design into multiple sections that run in parallel. In particular, we have two different groups of modules which are used to complete three stages: *Precomputation*, *Stage0*, and *Stage1*. *Precomputation* is performed only once at the start of signature generation and consists of sampling and decoding the secret key. Initially, this is performed by the first group of modules while another group performs *Stage0*. *Stage0* consists of the operations required to generate the $w$ vector needed for a signature attempt. This includes the sampling of $y$ and the matrix multiplication.

Once both of the stages are completed for the first time, the group of modules initially used for *Precomputation* use the results of *Stage0* and attempt to create a valid signature. This is what *Stage1* accomplishes. While *Stage1* is being performed, the other group of modules runs again so that if *Stage1* fails, there is another $w$ vector immediately ready for a new attempt. Essentially, this system is a high-level two-stage pipeline after the initial precomputation is performed. We split the signature operation after the $w$ operations because it requires the fewest dependencies between stages. The polynomial vectors are large, and transfer between stages will require BRAM buffers. Splitting at this operation requires only the $w$ and $y$ vectors to be transferred from *Stage0* to *Stage1*.

### *C. Dilithium Verify*

The scheduling diagrams of verification are shown in Fig. 13. As with key generation, the decoding of the first vector and the sampling of $A$ begin immediately. The $c$ polynomial is also sampled at the start of the operation, after which the message and $\rho$ seed are hashed. The main polynomial operation is performed sequentially, with all polynomials being transformed to the NTT domain, the multiplications being performed, and the difference of the values taken before

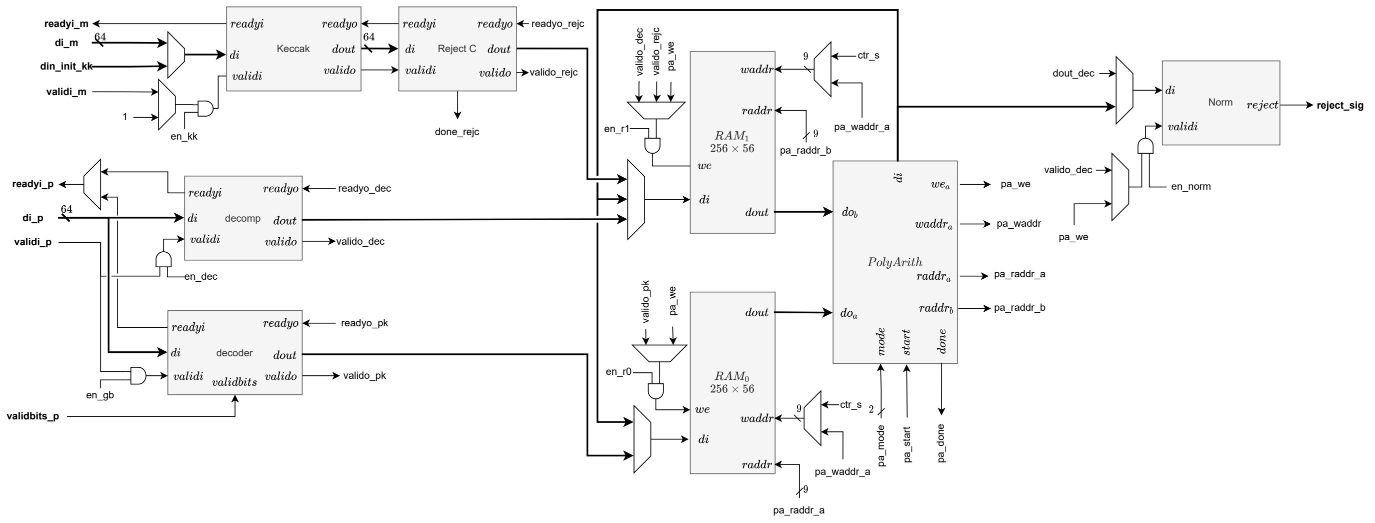


Fig. 9: Top level verification architecture of FALCON. Default bus width is 54-bit

converting back to the time domain. The hint is applied to the result after decomposing it. This is then encoded and hashed and then compared with $\bar{c}$.

### D. FALCON Verification

The scheduling diagrams of FALCON verification are shown in Fig. 14. The $h$ polynomial is decoded immediately at the start of the operation and then transformed to the NTT domain in parallel with decompression of $s_2$. The second NTT operation, multiplication, and inverse NTT operation are then performed. In parallel with the main polynomial operation, hashing of the message and the rejection sampling of $c$ are executed. The message hash only becomes the bottleneck for a large message. In particular, if hashing the message is not complete by the time the inverse NTT operation begins, the polynomial module will stall. So for level I, hashing the message can take up to 1,536 cycles and at level V up to 3,072 cycles without slowing down the operation. Since the Keccak module can ingest and hash 136 bytes in 41 clock cycles, this corresponds to a message of length 5KB and 10KB for level I and level V, respectively.

## VI. Results

All results were generated using Xilinx Vivado 2020.2. Maximum clock frequencies were determined using the Minerva hardware optimization tool [28]. The critical path of the design is within the interconnect for the shared Keccak modules. Since our design targets high performance, we primarily report our results for Virtex UltraScale+. However, we also include selected results for the Artix-7 and Kintex-7 FPGAs to perform a fair comparison with previous work.

The detailed resource utilization of our implementations is summarized in Tables IV and V. Table IV reports the area breakdown of the submodules used in our design and the percentage of the total LUTs they consume in the combined architecture. The entry "Other" represents the entire control logic of the top-level module and minor components of the datapath not listed explicitly in the table. Table V shows the results for both our combined architecture and the individual modules that only perform one major operation.

TABLE IV: Resource utilization of Dilithium submodules in the combined architecture

| Submodule | Resource Utilization | | | | % of Total |
|---|---|---|---|---|---|
| | LUT | FF | DSP | BRAM | (LUT) |
| $96 \times 1024$ RAM | 0 | 0 | 0 | $3 \times 6$ | 0 |
| $96 \times 4096$ RAM | 0 | 0 | 0 | 11 | 0 |
| MakeHint | 2,389 | 740 | 0 | 0 | 4.5 |
| UseHint | 6,453 | 2,808 | 0 | 0 | 12.1 |
| Encoder | 1,626 | 461 | 0 | 0 | 3.1 |
| Decoder | 2,189 | 239 | 0 | 0 | 4.1 |
| Decomposer | 1,437 | 680 | 0 | 0 | 2.7 |
| NTT/PolyArith | $4,509 \times 2$ | $3,146 \times 2$ | 16 | 0 | 16.9 |
| SampleA | 1,793 | 619 | 0 | 0 | 3.4 |
| SampleS | 1,755 | 396 | 0 | 0 | 3.2 |
| SampleY | 2,220 | 630 | 0 | 0 | 4.2 |
| SampleC | 1,856 | 868 | 0 | 0 | 3.5 |
| Keccak | $5,483 \times 3$ | $4,451 \times 3$ | 0 | 0 | 30.1 |
| Other | 6,002 | 1,231 | 0 | 0 | 11.3 |
| Combined Architecture | 53,187 | 28,318 | 16 | 29 | 100.0 |

There is a substantial amount of resource sharing possible between the implementations of three major operations, with the combined architecture only consuming 48% of the sum of the LUTs of the individual modules. The limited area increase over the most complex signature generation module is due to specific units only being required for certain operations, such as the secret sampler, SampleS, which is only utilized by key generation. However, many modules such as the Keccak cores, polynomial arithmetic modules, and RAMs can be fully shared between the different operations.

### A. Comparison with Previous Dilithium Works

The performance results and comparison with existing implementations are detailed in Table VII. For this work (TW) and the paper by Land et al. [5], we list the best and average execution times for signing. The grouping by security

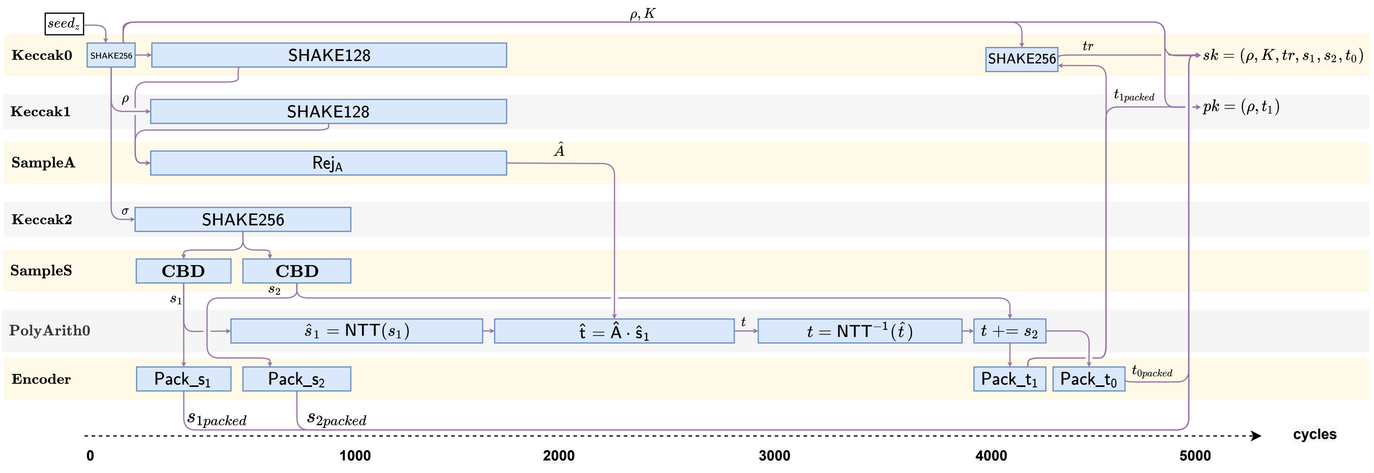


Fig. 10: Schedule of Dilithium key generation for security level 2

TABLE V: Resource utilization of top level modules on Virtex UltraScale+ FPGA

| Algorithm | Module | LUT | FF | DSP | BRAM |
|---|---|---|---|---|---|
| **Dilithium** | **Keygen** | 29,021 | 18,952 | 8 | 18.5 |
| | **Sign** | 42,440 | 24,419 | 16 | 29 |
| | **Verify** | 39,341 | 22,743 | 8 | 20 |
| | **Combined** | 53,187 | 28,318 | 16 | 29 |
| **FALCON-512** | **Verify** | 14,500 | 7,287 | 4 | 2 |
| **FALCON-1024** | **Verify** | 13,956 | 6,737 | 4 | 2 |

level is based on the NIST-defined PQC security levels. In Dilithium, the clock cycle cost of each operation generally increases by 50% as the security level increases due to the larger vector dimensions. The one exception to this rule is the average case for signature generation, which largely depends on the average number of attempts needed to generate a valid signature. According to the specification [17], on average, level 2 requires 4.25 attempts, level 3 requires 5.1 attempts, and level 5 requires 3.85 attempts. In our pipelined design, each additional attempt requires 5.8K cycles for security level 2, 8.1K cycles for level 3, and 10.8K cycles for level 5.

Ricci et al. [4] report the number of clock cycles for all security levels but the maximum clock frequency and area only for security level 2. Their results for sign report best-case results. The area is reported individually for each operation, so we will compare it against the area results for our individual modules. Compared to this high-performance implementation, our implementation achieves performance improvements with a lower utilization in all metrics except for BRAM in key generation and verification. In terms of latency, we achieve 1.5-3.7× better performance in terms of latency in microseconds. Our designs utilize 38%-46% fewer LUTs, 25%-72% fewer FFs, and 96%-98% fewer DSP. Our implementation of signature generation also uses 80% fewer BRAMs.

Our area and performance improvements are enabled by our efficient NTT design and optimized operation scheduling. The NTT design reported in [4] requires 48 DSP units for the forward NTT and 84 for the inverse NTT, while our design utilizes only 8. Our splitting of the rejection loop in signing also leads to a much lower area. Ricci et al. [4] duplicate modules so that there are 18 components running in parallel, including 12 NTT instances. This requires a large amount of BRAM to buffer data between modules and leads to a much larger implementation of signature generation.

Compared to the mid-range implementation by Land et al. [5], our design achieves substantially better performance at the cost of moderate increases in LUTs and FFs. Since this design includes results reported for modules that perform all operations at a single security level, we will compare our combined module implemented for Artix-7 with their module for security level 5.

The design by Land et al. [5] employs some parallelization, such as the use of multiple butterfly units in their NTT, but focuses on keeping a low area. They also utilize different operation modes of FPGA DSPs in their design, such as pre-addition and Single Instruction Multiple Data (SIMD) addition. This allows their NTT to achieve an impressive maximum frequency of 311 MHz, but also results in very high DSP usage. However, this high-frequency NTT core is not able to improve overall performance since it resides in the same clock domain as the rest of the design, forcing it to run at a lower clock rate. Our polynomial arithmetic unit is optimized so that it is not the critical path of the design but also seeks to minimize DSP usage and cycle latency. This allows our design to achieve higher NTT performance in the application of Dilithium with a much lower DSP count. In particular, their NTT requires 533/536 clock cycles for the forward and inverse operation, while our design is able to complete the same operations in 300/294 cycles. The additional cycles on top of the ideal of 256 cycles are caused by the pipeline depth. We achieve between $2.2 - 3\times$ lower latency in microseconds. These improvements come at the cost of 19% more LUTs and 100.5% more FFs, but our design uses 64.6% fewer DSP units

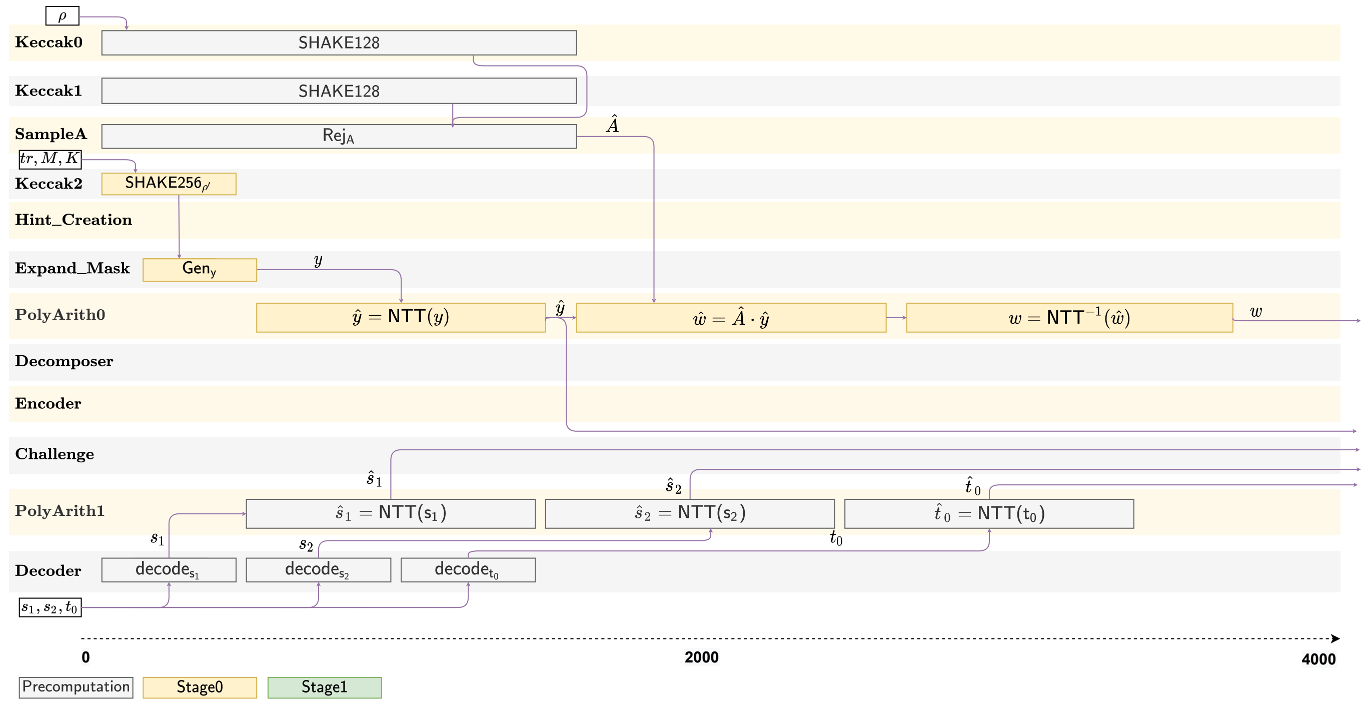


Fig. 11: Schedule of Dilithium precomputation stage of signature generation at security level 2

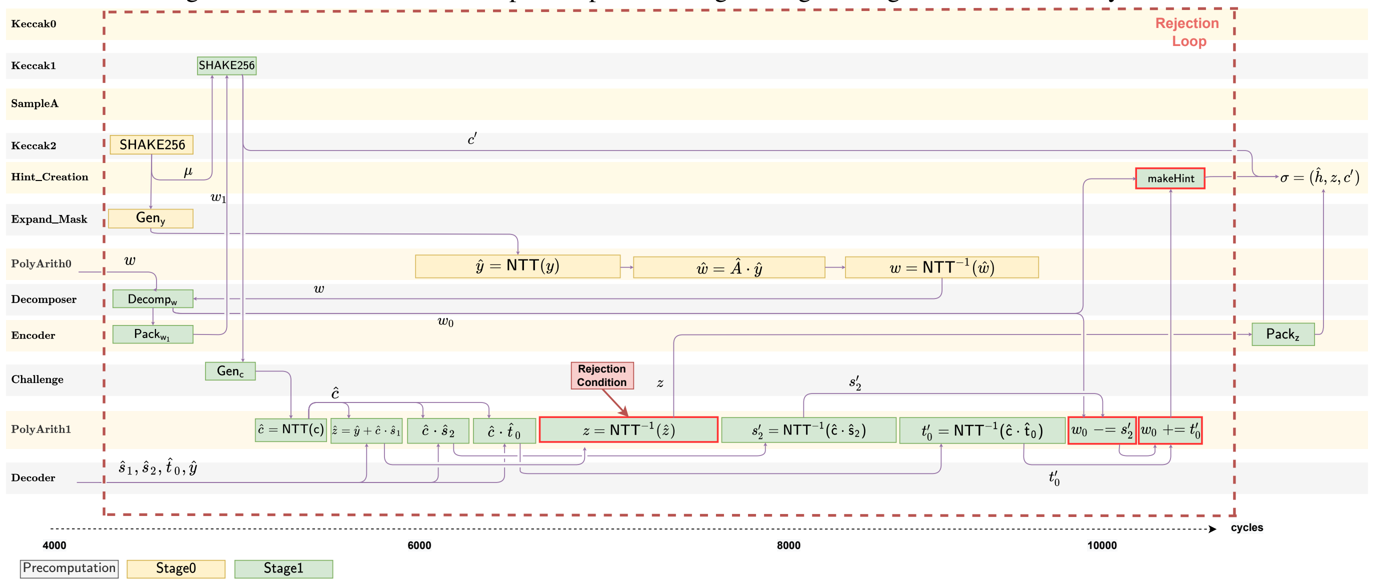

Rejection Loop
Keccak0
Keccak1
SHAKE256
SampleA
Keccak2
SHAKE256
$c'$
Hint_Creation
$\mu$
makeHint
$\sigma = (\hat{h}, z, c')$
$w_1$
Expand_Mask
Gen$_y$
PolyArith0
$w$
$\hat{y} = \mathsf{NTT}(y)$
$\hat{w} = \hat{A} \cdot \hat{y}$
$w = \mathsf{NTT}^{-1}(\hat{w})$
Decomposer
Decomp$_w$
$w$
$w_0$
Encoder
Pack$_{w_1}$
Pack$_z$
Challenge
Gen$_c$
Rejection Condition
$z$
$s_2'$
$\hat{c}$
PolyArith1
$\hat{c} = \mathsf{NTT}(c)$
$\hat{z} = \hat{y} + \hat{c} \cdot \hat{s}_1$
$\hat{c} \cdot \hat{s}_2$
$\hat{c} \cdot \hat{t}_0$
$z = \mathsf{NTT}^{-1}(\hat{z})$
$s_2' = \mathsf{NTT}^{-1}(\hat{c} \cdot \hat{s}_2)$
$t_0' = \mathsf{NTT}^{-1}(\hat{c} \cdot \hat{t}_0)$
$w_0 \; -= s_2'$
$w_0 \; += t_0'$
Decoder
$\hat{s}_1, \hat{s}_2, \hat{t}_0, \hat{y}$
$t_0'$
cycles
4000
6000
8000
10000
Precomputation
Stage0
Stage1

Fig. 12: Schedule of Dilithium signature rejection loop for signature generation at security level 2

and 6.5% fewer BRAMs.

The work reported in [6] details a unified Dilithium and Saber architecture fully supporting both algorithms. The two algorithms are able to fully share the Keccak core and the polynomial multiplier but otherwise require specialized modules. Their design focuses on low area, but supporting two algorithms naturally leads to a larger area than an implementation supporting only Dilithium. We achieve between $2.9-4\times$ lower latency in microseconds. These improvements come at the cost of $2.8\times$ more LUTs, $3\times$ more FFs, $4\times$ more DSP, and $1.2\times$ more BRAM.

In [9], Zhou et al. report their results for security level 2 of the round 2 Dilithium parameter set. They provide results for both the Xilinx Zynq-7000 SoC as well as an Altera Cyclone-IV FPGA using the Nios-II softcore processor. Area results for the number LUTs, DSPs, and BRAMs are included. Only the average performance for sign is reported. The authors make performance comparisons between a pure software implementation, a HW/SW co-design with Keccak and polynomial-arithmetic hardware accelerators, and a HW/SW

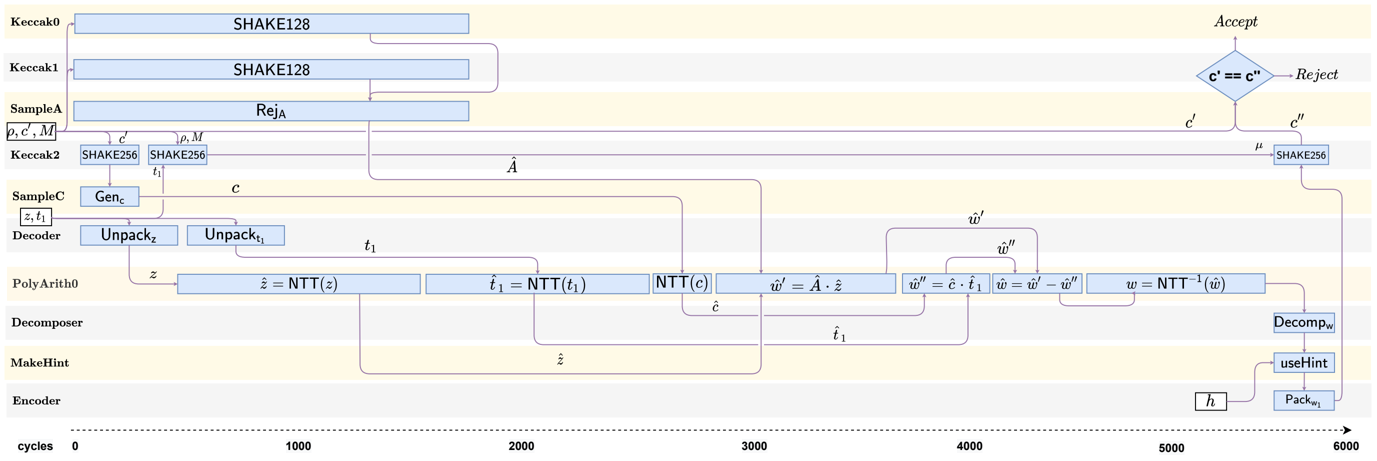


Fig. 13: Schedule of Dilithium signature verification for security level 2

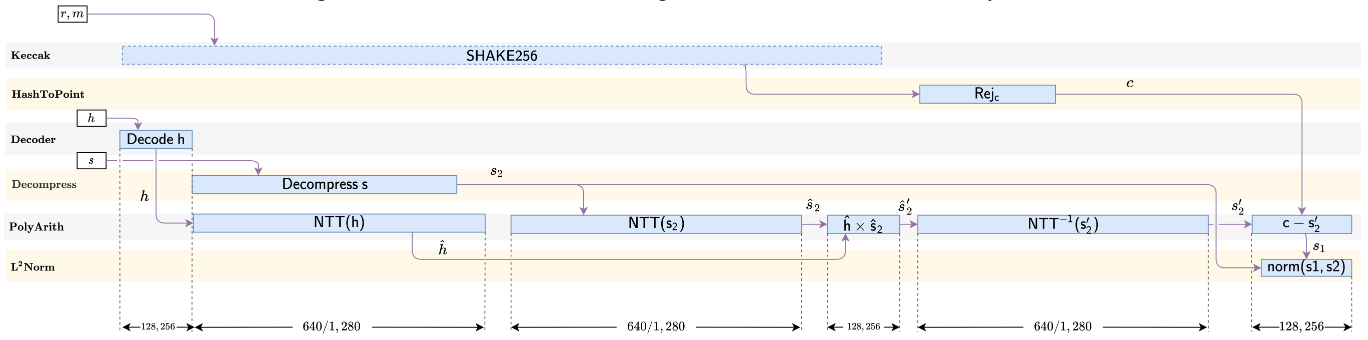


Fig. 14: FALCON verify schedule. Cycle count of operations shown for FALCON-512/FALCON-1024

| Author | Algorithm | Design Approach | Multiple operations | Multiple security levels | Keygen Included | Reported Results | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | A7 | K7 | US+ |
| This Work | FALCON | High Performance | | | | ✓ | ✓ | ✓ |
| | Dilithium | High Performance | ✓ | ✓ | ✓ | | | |
| [4] | Dilithium (Round 2) | High Performance | | | ✓ | | | ✓ |
| [5] | Dilithium | Mid-Range | ✓ | | ✓ | ✓ | | |
| [6] | Dilithium | Unified/Lightweight | ✓ | ✓ | ✓ | | | ✓ |
| [12] | Picnic | High Performance | ✓ | | | | ✓ | |
| [13] | SPHINCS+ | High Performance | ✓ | | | ✓ | | |

TABLE VI: Design decisions for investigated algorithms in this and previous work.

co-design using NEON instructions for Keccak and a hardware accelerator for polynomial arithmetic. The last approach using NEON instructions for Keccak and a hardware accelerator for polynomial arithmetic gave the best results with an area of 2.6K LUTs and a speed-up of 1.44-2×. Compared to this approach, our design is 49-174.4× faster but uses 20.3× more LUTs, 1.3× more DSP, and 4× more BRAM.

Software implementations of Dilithium for embedded devices are substantially slower than hardware. For example, a recent optimized implementation on Cortex-M4 takes 1.35M/3M/1.26M clock cycles for key Generation, Signing, and Verifying at security level 2 [29]. Their implementation was tested on an STM32F407 with a 24 MHz clock, so our hardware implementation has 2961×/1071×/2042× lower latency.

In comparison with high-end CPUs, the highly optimized NEON implementation on ARMv8 Apple M1 *Firestorm* core at 3.2 GHz, reported by Becker et al. [30] for Dilithium-III takes $51/36\mu s$ for Key generation and Verifying. Our hardware implementation is 1.60×/0.92× faster in the two mentioned above operations. Due to rejection in the signing operation, it takes $149\mu s$ for the average case in Apple M1, while our hardware is 0.77× and 2.36× faster in the average and best case, respectively.

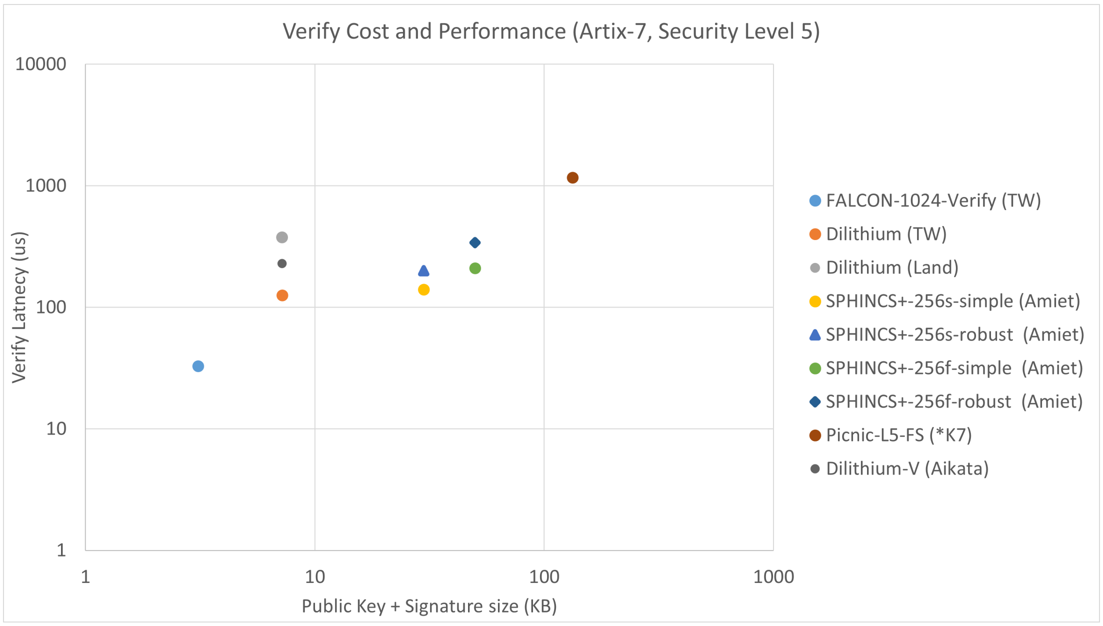
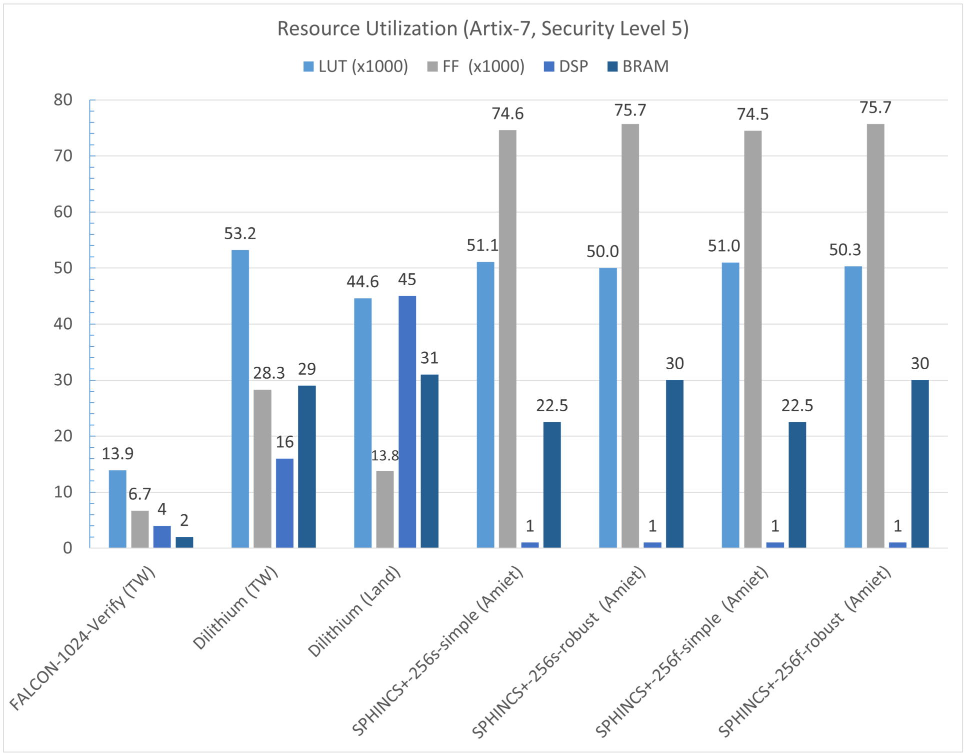


Fig. 15: Verification performances and transmission overhead

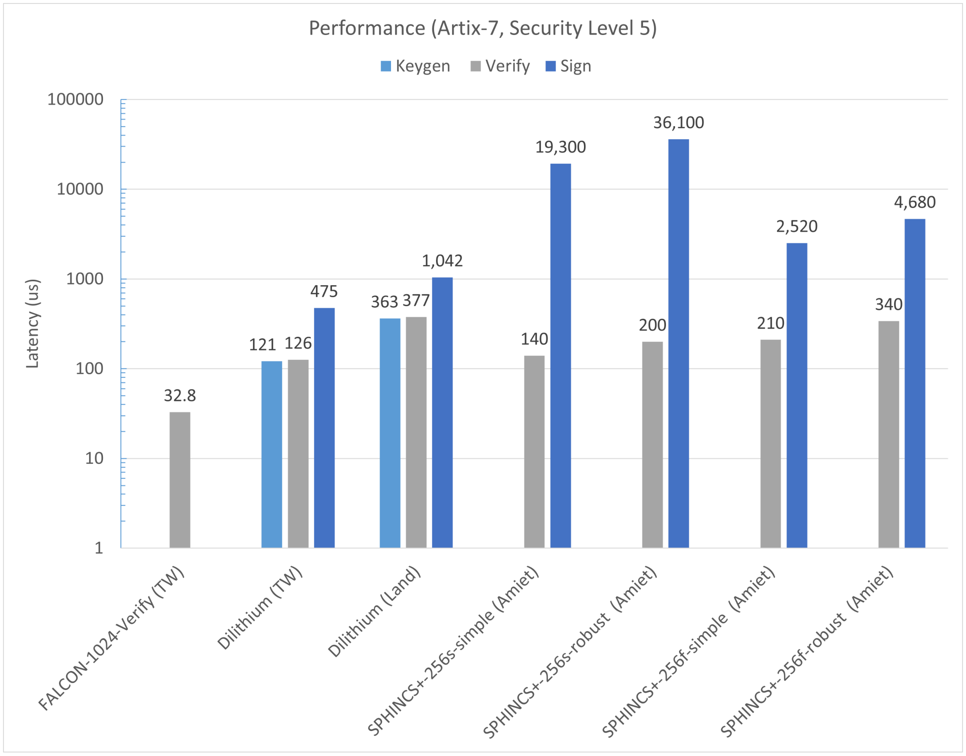
TABLE VII: Full hardware implementations of digital signature schemes qualified as finalists to Round 3 of the NIST PQC standardization process. Results are grouped by FPGA family, sorted by verification latency. **TW** denotes This Work. Notation for FPGA families - A7: Artix-7, K7: Kintex-7, VUS+: Virtex UltraScale+, ZUS+: Zynq UltraScale+

| **Design** | **Algorithm** | **pk+sig. (KB)** | **Max Freq. (MHz)** | **LUT** | **FF** | **DSP** | **BRAM** | **Keygen** | | **Verify** | | **Sign** | | **Family** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | **cycles** | **$\mu$s** | **cycles** | **$\mu$s** | **cycles** | **$\mu$s** | |
| | | | | | **Security Level 1** | | | | | | | | | |
| **TW** | FALCON-512 | 1.5 | 142 | 14,500 | 7,287 | 4 | 2 | - | - | 2,399 | 16.8 | - | - | A7 |
| [13] | SPHINCS$^+$-128s-simple | 7.9 | 250 & 500[3] | 48,231 | 72,514 | 0 | 11.5 | - | - | - | 70 | - | 12,400 | A7 |
| [13] | SPHINCS$^+$-128s-robust | 7.9 | 250 & 500[3] | 49,146 | 73,069 | 0 | 15.5 | - | - | - | 110 | - | 21,100 | A7 |
| [13] | SPHINCS$^+$-128f-simple | 17.1 | 250 & 500[3] | 47,991 | 72,505 | 1 | 11.5 | - | - | - | 160 | - | 1,010 | A7 |
| [13] | SPHINCS$^+$-128f-robust | 17.1 | 250 & 500[3] | 48,930 | 72,505 | 1 | 15.5 | - | - | - | 230 | - | 1,640 | A7 |
| [12] | Picnic-L1-FS | 34 | 125 | 90,535 | 23,516 | 0 | 52.5 | - | - | 29,600 | 237 | 31,300 | 250 | K7 |
| **TW** | FALCON-512 | 1.5 | 314 | 14,327 | 7,314 | 4 | 2 | - | - | 2,399 | 7.6 | - | - | VUS+ |
| [4] | Dilithium-II[1,4] | 3.2 | - | - | - | - | - | 12,600 | - | 10,546 | - | 18,338/- | - | VUS+ |
| | | | | | **Security Level 2** | | | | | | | | | |
| **TW** | Dilithium-II[4] | 3.7 | 116 | 53,187 | 28,318 | 16 | 29 | 4,875 | 42 | 6,582 | 57 | 10,945/29,876 | 94/257 | A7 |
| [5] | Dilithium-II[4] | 3.7 | 163 | 27,433 | 10,681 | 45 | 15 | 18,761 | 115 | 19,687 | 121 | 29,057/76,613 | 178/470 | A7 |
| **TW** | Dilithium-II[4] | 3.7 | 256 | 53,907 | 28,435 | 16 | 29 | 4,875 | 19 | 6,582 | 26 | 10,945/29,876 | 43/117 | VUS+ |
| [4] | Dilithium-III[1,2,4] | 4.2 | 350/333/158 | 54,183/68,461/61,738 | 25,236/86,295/34,963 | 182/965/316 | 15/145/18 | 18,193 | 52 | 15,032 | 97 | 21,033/- | 63/- | VUS+ |
| [6] | Dilithium-II[5] | 3.7 | 200 | 19,100 | 9,300 | 4 | 24 | 14,183 | 71 | 15,044 | 75 | 30,358/- | 152/- | ZUS+ |
| | | | | | **Security Level 3** | | | | | | | | | |
| **TW** | Dilithium-III[4] | 5.2 | 116 | 53,187 | 28,318 | 16 | 29 | 8,291 | 71 | 9,724 | 84 | 16,178/49,437 | 139/426 | A7 |
| [13] | SPHINCS$^+$-192s-simple | 16.3 | 250 & 500[3] | 48,725 | 72,514 | 0 | 17 | - | - | - | 100 | - | 21,400 | A7 |
| [13] | SPHINCS$^+$-192s-robust | 16.3 | 250 & 500[3] | 50,064 | 74,462 | 0 | 22.5 | - | - | - | 150 | - | 38,300 | A7 |
| [13] | SPHINCS$^+$-192f-simple | 35.7 | 250 & 500[3] | 48,398 | 73,476 | 1 | 17 | - | - | - | 190 | - | 1,170 | A7 |
| [5] | Dilithium-III[4] | 5.2 | 145 | 30,900 | 11,372 | 45 | 21 | 33,102 | 228 | 32,050 | 221 | 45,068/123,218 | 310/850 | A7 |
| [13] | SPHINCS$^+$-192f-robust | 35.7 | 250 & 500[3] | 47,277 | 74,279 | 1 | 22.5 | - | - | - | 310 | - | 2,120 | A7 |
| **TW** | Dilithium-III[4] | 5.2 | 256 | 53,907 | 28,435 | 16 | 29 | 8,291 | 32 | 9,724 | 39 | 16,178/49,437 | 63/193 | VUS+ |
| [6] | Dilithium-III[5] | 5.2 | 200 | 19,100 | 9,300 | 4 | 24 | 22,957 | 115 | 25,535 | 128 | 47,418/- | 237/- | ZUS+ |
| [4] | Dilithium-IV[1,4] | 5.1 | - | - | - | - | - | 22,981 | - | 20,221 | - | 22,362/- | - | VUS+ |
| | | | | | **Security Level 5** | | | | | | | | | |
| **TW** | FALCON-1024 | 3.1 | 142 | 13,956 | 6,737 | 4 | 2 | - | - | 4,687 | 32.8 | - | - | A7 |
| **TW** | Dilithium-V[4] | 7.2 | 116 | 53,187 | 28,318 | 16 | 29 | 14,037 | 121 | 14,642 | 126 | 24,358/55,070 | 210/475 | A7 |
| [13] | SPHINCS$^+$-256s-simple | 29.8 | 250 & 500[3] | 51,130 | 74,576 | 1 | 22.5 | - | - | - | 140 | - | 19,300 | A7 |
| [13] | SPHINCS$^+$-256s-robust | 29.8 | 250 & 500[3] | 50,00 | 75,738 | 1 | 30 | - | - | - | 200 | - | 36,100 | A7 |
| [13] | SPHINCS$^+$-256f-simple | 49.9 | 250 & 500[3] | 51,009 | 74,539 | 1 | 22.5 | | - | - | 210 | - | 2,520 | A7 |
| [13] | SPHINCS$^+$-256f-robust | 49.9 | 250 & 500[3] | 50,341 | 75,664 | 1 | 30 | - | - | - | 340 | - | 4,680 | A7 |
| [5] | Dilithium-V[4] | 7.2 | 140 | 44,653 | 13,814 | 45 | 31 | 50,982 | 363 | 52,712 | 377 | 70,376/145,912 | 503/1,042 | A7 |
| **TW** | Dilithium-V[4] | 7.2 | 173 | 54,468 | 28,639 | 16 | 29 | 14,037 | 81 | 14,642 | 85 | 24,358/55,070 | 141/318 | K7 |
| [12] | Picnic-L5-FS | 133 | 125 | 167,530 | 33,164 | 0 | 99 | - | - | 146,600 | 1,173 | 154,500 | 1,236 | K7 |
| **TW** | FALCON-1024 | 3.1 | 314 | 13,729 | 6,771 | 4 | 2 | - | - | 4,687 | 14.9 | - | - | VUS+ |
| **TW** | Dilithium-V[4] | 7.2 | 256 | 53,907 | 28,435 | 16 | 29 | 14,037 | 55 | 14,642 | 57 | 24,358/55,070 | 95/215 | VUS+ |
| [6] | Dilithium-V[5] | 7.2 | 200 | 19,100 | 9,300 | 4 | 24 | 38,841 | 194 | 45,789 | 229 | 68,500/- | 342/- | ZUS+ |

[1]Uses Round 2 parameter set [2]Area reported separately for Key Generation, Sign, and Verify [3]Split frequency domain: Keccak at 500 MHz, other units at 250 MHz [4]Best/average execution times reported for the Sign operation of Dilithium [5]Supports both Dilithium and Saber

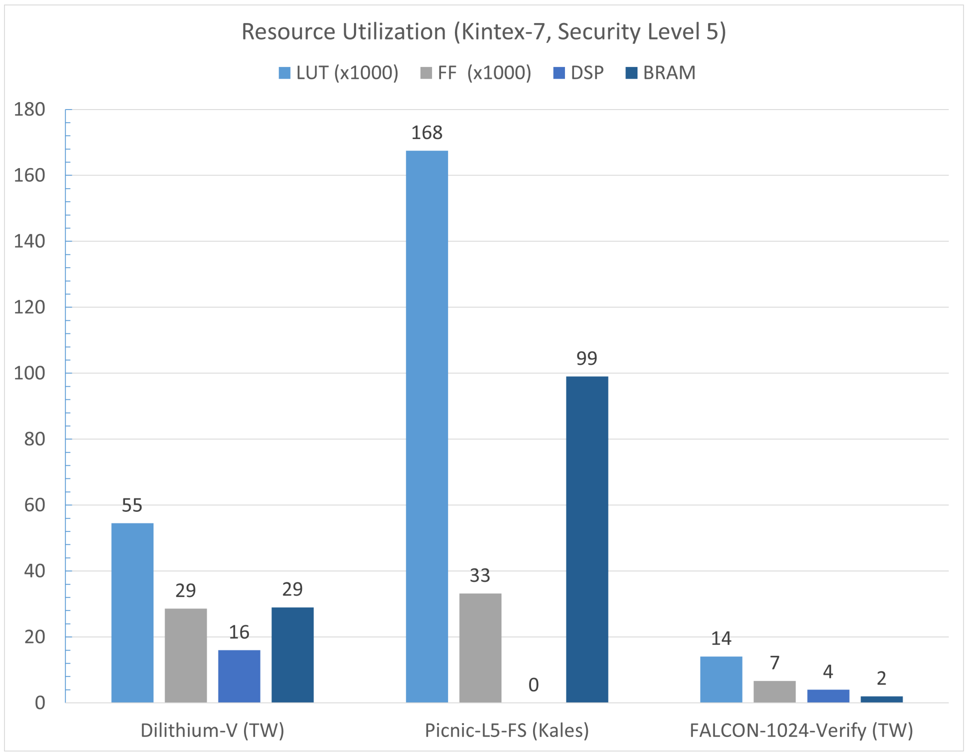


((a)) Artix-7 Area

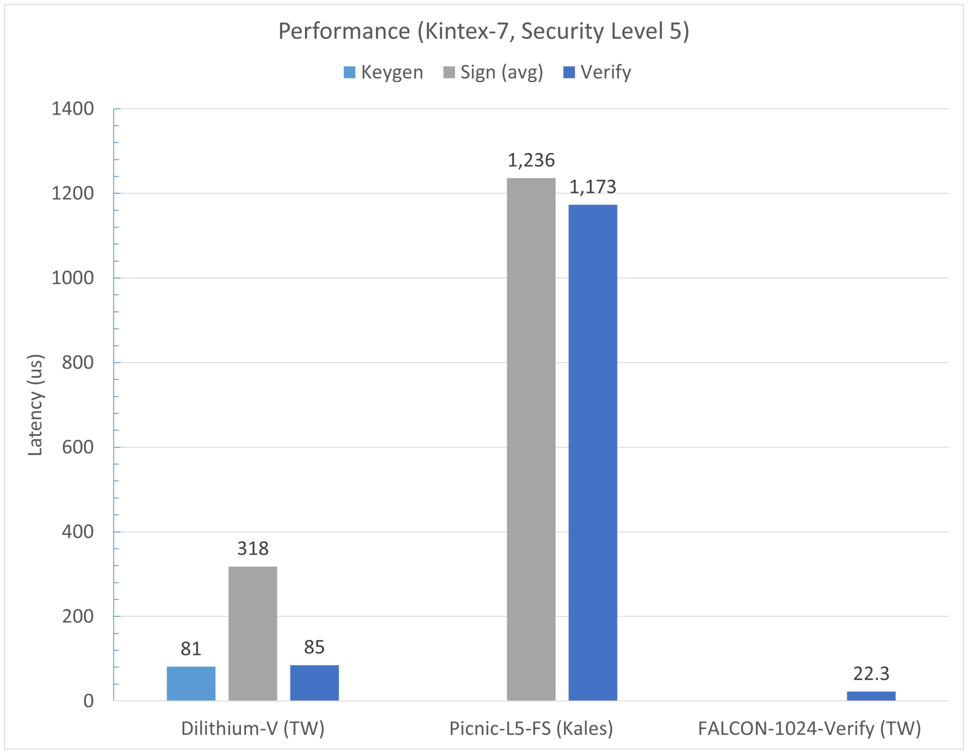


((b)) Artix-7 Performance

Fig. 16: Comparison of digital signatures on Artix-7

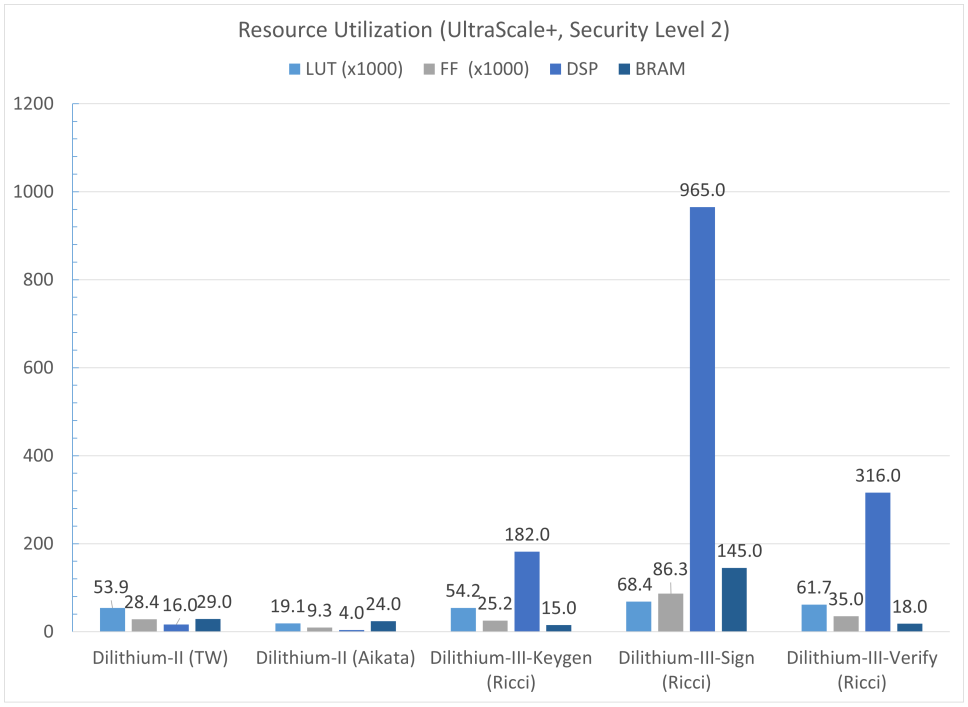


((a)) Kintex-7 Area

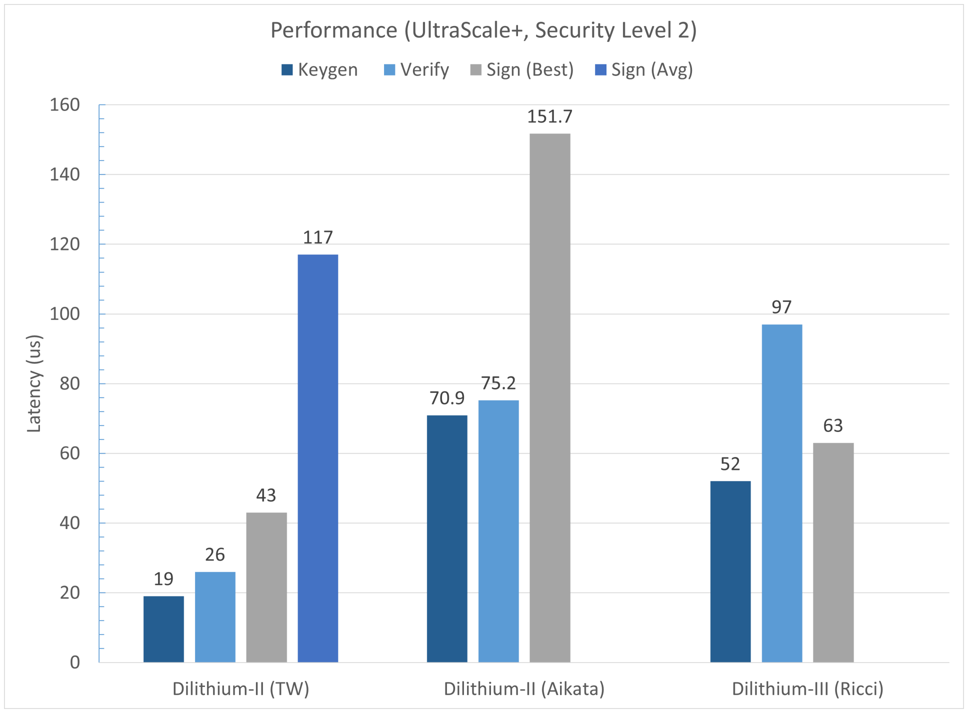


((b)) Kintex-7 Performance

Fig. 17: Comparison of digital signatures on Kintex-7



((a)) Virtex UltraScale+ Area



((b)) Virtex UltraScale+ Performance

Fig. 18: Comparison of digital signatures on UltraScale+

### *B. Comparison of Algorithm Families*

Previous works on SPHINCS$^+$ and Picnic are also reported. SPHINCS$^+$ has twelve parameter sets that determine the security level and trade-offs between sign performance, resource consumption, and signature size. Picnic also has several parameter sets, but only a subset have hardware implementations available for comparison.

Table VII provides some basic insights into the comparison between the remaining viable digital signature schemes in terms of performance in hardware. These comparisons are also illustrated by Figs. 16, 17, 18. The figures show results for each FPGA family that has results reported in Table VII. A single security level was chosen for each set of figures based on whichever security level had the most-complete results for that family of FPGA. Further, Fig. 15 shows a comparison of verification latency and the approximate size of certificates (estimated as the sum of a public key of a user and digital signature of a certification authority) for all algorithms. As verification is the most common operation, this is an interesting metric to consider for digital signature comparison. It should be noted that Picnic is included in this figure even though its results are reported for Kintex-7 and not Artix-7. In the context of this comparison, the only advantage it gains is lower latency from the higher performance of the Kintex-7 family of FPGAs. Since it is the longest latency operation, even with this advantage, we can still conclude that the lattice-based algorithms give better verification performance and smaller certificate sizes.

Based on verification performance and on the approximate size of certificates (which may need to be transmitted to enable signature verification), FALCON is a strong candidate with the fastest verification and smallest combined public key and signature size, as shown in Fig. 15. FALCON also has a lower area footprint than the equivalent operation for Dilithium, as shown in Table V. These qualities would reduce the cost of message transmission and power consumption on client devices, which are more likely to verify signatures than generate them. However, if the performance of all operations is equally weighted, then the complexity of FALCON key generation and signing may make Dilithium a stronger and much easier to implement candidate. As shown in Figs. 16(a), 16(b), 17(a), 17(b), the code-based algorithms both suffer from slow signature generation and large resource consumption while also have the largest signatures. Even with SPHINCS$^+$'s verification latency being competitive with Dilithium, it has a much larger signature size. Thus their primary benefit is the potentially stronger security claims if there is doubt in the security of the lattice-based candidates.

## VII. Conclusions

This paper presents a high-performance implementation of CRYSTALS-Dilithium, which achieves the best-known latency and a smaller area than the best previously reported high-performance design. The implementation includes both a combined module capable of performing three major operations at all security levels and individual modules supporting one operation each. Additionally, the first hardware work on FALCON is presented in the form of a full hardware implementation of FALCON verification. These results are used to provide a more complete comparison between all remaining viable digital signature candidates in the NIST PQC competition.

## References

[1] P. Shor, “Algorithms for quantum computation: Discrete logarithms and factoring,” in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, Santa Fe, NM, USA: IEEE Comput. Soc. Press, 1994, pp. 124–134.

[2] L. Chen, S. Jordan, Y.-K. Liu, *et al.*, “Report on Post-Quantum Cryptography,” National Institute of Standards and Technology, Tech. Rep. NIST IR 8105, Apr. 2016, NIST IR 8105.

[3] W. Beullens, “Improved Cryptanalysis of UOV and Rainbow,” Tech. Rep. 1343, 2020. [Online]. Available: https://eprint.iacr.org/2020/1343 (visited on 01/10/2022).

[4] S. Ricci, L. Malina, P. Jedlicka, *et al.*, “Implementing CRYSTALS-Dilithium Signature Scheme on FPGAs,” in *16th International Conference on Availability, Reliability and Security, ARES 2021*, Vienna Austria: ACM, Aug. 2021, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.1145/3465481.3465756 (visited on 02/06/2022).

[5] G. Land, P. Sasdrich, and T. Guneysu, “A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware,” Cryptology ePrint Archive 2021/355, Mar. 2021.

[6] A. C. Mert, D. Jacquemin, A. Das, D. Matthews, S. Ghosh, and S. S. Roy, “A Unified Cryptoprocessor for Lattice-based Signature and Key-exchange,” en, p. 7,

[7] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, “Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 4, Aug. 2019.

[8] ——, “Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols (Extended Version),” Cryptology ePrint Archive 2019/1140, Sep. 2020.

[9] Z. Zhou, D. He, Z. Liu, M. Luo, and K.-K. R. Choo, “A Software/Hardware Co-Design of Crystals-Dilithium Signature Scheme,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 14, no. 2, 11:1–11:21, Jun. 2021. [Online]. Available: https://doi.org/10.1145/3447812 (visited on 06/10/2021).

[10] A. Basso, F. Aydin, D. Dinu, J. Friel, M. Sastry, and S. Ghosh, “SABER and Dilithium on the Same Polynomial Multiplier,” en, p. 21,

[11] A. Ferozpuri and K. Gaj, “High-speed FPGA Implementation of the NIST Round 1 Rainbow Signature Scheme,” in *2018 International Conference on ReCon-*

*Figurable Computing and FPGAs (ReConFig)*, Cancun, Mexico: IEEE, Dec. 2018, pp. 1–8.

[12] D. Kales, S. Ramacher, C. Rechberger, R. Walch, and M. Werner, "Efficient FPGA Implementations of LowMC and Picnic," in *The Cryptographers' Track at the RSA Conference 2020, CT-RSA 2020*, San Francisco: Springer, Feb. 2020.

[13] D. Amiet, L. Leuenberger, A. Curiger, and P. Zbinden, "FPGA-based SPHINCS+ Implementations: Mind the Glitch," en, in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, Kranj, Slovenia: IEEE, Aug. 2020, pp. 229–237.

[14] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of crystals-dilithium," in *2021 International Conference on Field-Programmable Technology (ICFPT)*, 2021, pp. 1–10.

[15] L. Ducas, E. Kiltz, T. Lepoint, *et al.*, "CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation," NIST Round 2, Mar. 2019.

[16] P.-A. Fouque, J. Hoffstein, P. Kirchner, *et al.*, "Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU," p. 67,

[17] S. Bai, L. Ducas, E. Kiltz, *et al.*, "CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation (Version 3.1)," Tech. Rep., Feb. 2021.

[18] Ö. Dagdelen, M. Fischlin, and T. Gagliardoni, "The Fiat–Shamir Transformation in a Quantum World," in *Advances in Cryptology - ASIACRYPT 2013*, ser. LNCS, vol. 8270, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 62–81.

[19] V. Lyubashevsky, "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures," in *Advances in Cryptology – ASIACRYPT 2009*, vol. 5912, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 598–616.

[20] R. Avanzi, J. Bos, L. Ducas, *et al.*, "Kyber - Algorithm Specifications And Supporting Documentation," en, p. 43,

[21] C.-M. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y. Yang, "NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2," en, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 159–188, Feb. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8791 (visited on 04/18/2021).

[22] G. Land, P. Sasdrich, and T. Guneysu, "A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware," en, p. 16, 2021.

[23] D. T. Nguyen, V. B. Dang, and K. Gaj, "A High-Level Synthesis Approach to the Software/Hardware Codesign of NTT-Based Post-Quantum Cryptography Algorithms," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, Tianjin, China: IEEE, Dec. 2019, pp. 371–374.

[24] ——, "High-Level Synthesis in Implementing and Benchmarking Number Theoretic Transform in Lattice-based Post-Quantum Cryptography using Software/Hardware Codesign," in *16th International Symposium on Applied Reconfigurable Computing, ARC 2020*, Apr. 2020.

[25] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 49–72, Mar. 2020.

[26] "7 Series FPGAs Memory Resources User Guide," 2019.

[27] CERG, *SHAKE*, https://github.com/GMUCERG/SHAKE, 2021.

[28] F. Farahmand, A. Ferozpuri, W. Diehl, and K. Gaj, "Minerva: Automated hardware optimization tool," in *2017 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017*, Cancun: IEEE, Dec. 2017, pp. 1–8.

[29] D. O. C. Greconici, M. J. Kannwischer, and D. Sprenkels, "Compact Dilithium Implementations on Cortex-M3 and Cortex-M4," en, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 1–24, Dec. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8725 (visited on 08/01/2021).

[30] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, "Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1," Cryptology ePrint Archive 2021/986, Jul. 2021.