Improved ML-DSA Hardware Implementation With First Order Masking Countermeasure

Kamal Raj, Prasanna Ravi, Tee Kiah Chia, Anupam Chattopadhyay

***Abstract*—We present the protected hardware implementation of the Module-Lattice-Based Digital Signature Standard (ML-DSA). ML-DSA is an extension of Dilithium 3.1, which is the winner of the Post Quantum Cryptography (PQC) competition in the digital signature category. The proposed design is based on the existing high-performance Dilithium 3.1 design. We implemented existing Dilithium masking gadgets in hardware, which were only implemented in software. The masking gadgets are integrated with the unprotected ML-DSA design and functional verification of the complete design is verified with the Known Answer Tests(KATs) generated from an updated ML-DSA software implementation. We also present the practical power side-channel attack experimental results by implementing masking gadgets on the standard side-channel evaluation FPGA board and collecting power traces up-to 1 million traces. The proposed protected design has the overhead of 1.127× LUT, 1.2× Flip-Flop, and 378× execution time compared to unprotected design. The experimental results show that it resists side-channel attacks.**

***Index Terms*—Post Quantum Cryptography, ML-DSA, Hardware Implementation, Masking, Side-Channel Attack, Security**

I. Introduction

A huge amount of data is exchanged between various devices through wired or wireless communication channels. Most of the data is exchanged over the public channel, which is vulnerable to various threats and also has privacy concerns. The cryptographic ciphers protect sensitive data from the attacker and solve the problem of security and privacy concerns [1]. The underlying method to protect data over public channels is a mathematical hard problem that can't be solved by the modern computer unless the adversary has the information about the secret key, which encrypts or decrypts the secret messages.

There are two types of cryptosystems, symmetric key and asymmetric key cryptosystems both have their benefits. For example, symmetric cryptosystems have the benefit of less computational cost while secret key sharing is the biggest issue while asymmetric cryptosystems have the benefit of no secret key sharing but has more computational cost. In practice, both symmetric and asymmetric cryptosystems are used to balance the computational cost by using asymmetric cryptosystems for authentication and key encapsulation over public channels and using symmetric cryptosystems for encrypting or decryption the secret messages [2].

However, asymmetric cryptosystems pose a major security threat to quantum computers because of Shor's algorithms that can solve the current public key cryptosystem's hard problem in polynomial time with the quantum computer [3]. The National Institute of Standard and Technology (NIST) started a competition called Post Quantum Cryptography (PQC) which aims to develop cryptosystems that are quantum-safe, which means the algorithms that are hard for classical as well as quantum computers and also provides long-term security. These PQC cryptosystems are implemented in classical computers but at the same time, they also provide security from quantum computer threats. Although PQC algorithms are quantum-safe, it has some issues compared to non-quantum-safe cryptosystems like larger key size, and most importantly, they are not studied rigorously for side-channel attack threat [4].

There are both software and hardware implementation PQC algorithms are available in the literature but we choose hardware implementation to study side-channel attack threats. A cryptosystem implemented in hardware can give high-speed computation of the cryptographic algorithms. For example, authors in [5] have implemented PQC candidate Dilithium on FPGA SoC which achieved about 40× better performance compared to software-only implementation. Authors in [6] implemented AES on $22nm$ Tri-gate CMOS which gives a throughput of 289 Gbps which is significantly large compared to its software-only implementation. Also, the computational extensive part of the cryptosystem can be implemented in hardware and added to the instruction set architecture which increases the performance of the cryptosystem in software. For example, Intel added six instructions in the processor to improve Advanced Encryption Standard (AES) performance in software [7].

Although hardware implementation of cryptosystesms provide better performance, it has the threat of extracting secret information through side-channel attack techniques like power side-channel attack [8], timing side-channel attacks [9] etc. These attacks can recover secret information from the system running with the cryptographic algorithm. Hence, it is important that it is protected from side-channel attack countermeasure.

Module-Lattice-Based Digital Signature Standard or ML-DSA [10] is the extension of Dilithium 3.1 [11] recently standardized by NIST. However, it also has a threat from side-channel attacks. For example, authors in [12] presented the power side-channel attack on Dilithium targeting the

Kamal Raj and Tee Kiah Chia are with Temasek Laboratories @ Nanyang Technological University, Singapore 639798 (email: kamalraj@ieee.org, tkchia@ntu.edu.sg)

Prasanna Ravi and Anupam Chattopadhyay are with College of Computing and Data Science, Nanyang Technological University, Singapore 639798. (email: prasanna.ravi@ntu.edu.sg, anupam@ntu.edu.sg)

randomness leakage in the signature generation algorithm and recovered secret information from Dilithium level 3 using 10,000 power traces within half an hour.

There are various hardware designs of Dilithium are available in the literature. However, these designs are not protected with the side-channel attack countermeasure which can lead to the extraction of secret information through leakage concerning security and privacy issues. At present, there is a lack of protected hardware implementation of Dilithium or ML-DSA which motivates us to implement side-channel attack countermeasures to protect ML-DSA for long-term security.

In this paper, the implementation of masking countermeasures for ML-DSA hardware design, the challenges of implementing it in hardware, and its effect on side-channel attack resistance, area, and performance are presented. The main contributions are as follows:

1) Existing hardware implementation, Dilithium 3.1 implemented by the authors in [13], is updated to the final ML-DSA standard, and its functionality is verified using Known Answer Tests (KATs) generated from an updated ML-DSA software implementation.
2) Masking gadgets proposed by the authors in [14] to mask entire signature generation algorithm are implemented in hardware which is then integrated with ML-DSA design. The masked implementation is also verified with the same Known Answer Tests (KATs).
3) The proposed protected ML-DSA design has an overhead of 1.127× LUT, 1.2× Flip-Flop, and 378× execution time compared to the high-performance unprotected ML-DSA hardware design.
4) Side-Channel attack experiments are performed to evaluate proposed protected ML-DSA hardware. We collected up to 1 million traces for each module and t-test result shows that it doesn't leak side-channel information.

This paper is organized as follows. Section II present the background on existing hardware implementation, Side-Channel attack and its countermeasures of Dilithium algorithms. Section III briefly present the internal algorithms of ML-DSA and masking gadgets for ML-DSA. Section IV present the proposed hardware implementation of masked ML-DSA signature generation. Section V present the experimental results and Section VI concludes the paper.

II. BACKGROUND

Modular Lattice-Based Digital Signature Standard (ML-DSA) [10] is a lattice-based post-quantum cryptography (PQC) digital signature algorithm, recently standardized by the National Institute of Standard and Technology (NIST) as FIPS 204. It is an extension of Dilithium Version 3.1 [11] which is the winner of the NIST PQC competition in the digital signature scheme. The initial public draft of ML-DSA was finalized to the final ML-DSA standard on 13th August 2024 with changes to the initial draft. NIST also provided a set of test vectors for the finalized standard (under https://github.com/usnistgov/ACVP-Server/). The authors of Dilithium 3.1 also published a reference implementation of Dilithium – without the changes in the final standard – which is available in public.

A. Dilithium Hardware Implementation

Although there is no official Dilithium or ML-DSA hardware implementation available from authors of Dilithium, authors in [13] present high-performance hardware implementation of Dilithium 3.1 which is available in the public repository. It is a high-performance design that uses multiple cores of different hardware modules to increase the performance of the design. More details about this implementation are given in Section IV. There are also other hardware designs of Dilithium 3.1 available in the literature which are given below.

Authors in [15] have presented a lightweight hardware implementation of Dilithium 3.1. It uses various methods to optimize the design which includes efficient implementation of modular reduction module, implementation of pipeline architecture, and hardware resource sharing. They also use this hardware design as an accelerator for the Zynq processing system by integrating the core design of Dilithium with the Zynq processing system using the AXI interface. Authors in [5] have presented the Dilithium hardware implemented on the FPGA SoC platform. The authors use the hardware-software co-design method to implement Dilithium on the SoC platform. Those computations that are expensive for software are implemented in hardware like polynomial multiplication, expandmask, SampleInBall. The high-level design is then integrated with the processer using AXI interface.

Authors in [16] have presented the hardware implementation of a crypto-processor that is targeted for accelerating operations in Dilithium algorithms. The authors implemented low latency NTT butterfly using the Karatsuba algorithm which is embedded in the NTT module. The crypto-processor is designed in such a way that one object (256-degree polynomial) is processed in each instruction enabling computation of parallel data resulting in speeding up the execution time. Authors in [17] presented a unified crypto-processor for lattice-based post-quantum cryptography Kyber and Dilithium. The authors targeted polynomial multiplication design to optimize the design. The sampling unit that samples pseudo-random numbers from the KECCAK module is customized for Dilithium and Kyber.

B. Side-Channel Attack and Countermeasure

Since ML-DSA is a digital signature algorithm, some of its variables are public and can be skipped for masking and some variables are sensitive and need to be protected because if they are leaked, they can help attackers to compute secret parts. The first variable in the signing algorithm is y, which needs to be protected since attackers can compute coefficients of s_1 from $z = y + cs_1$, because z and c are public. The variable y is generated pseudo-randomly from the seed (ρ) and a nonce.

Authors in [12] have presented a practical Side-Channel attack on Dilithium signature generation by targeting y variable and they recommend that the entire Dilithium signature generation must be protected including pseudo-random number. To protect the Dilithium from side-channel attack, masked Dilithium is proposed by the authors in [14] and [18]. Authors in [14] have not provided any software and hardware

implementation but presented the practical result of a Side-Channel attack on their proposed algorithm. Authors in [18] have not provided any practical results but they have provided C program implementation of their proposed algorithm. We use the masking gadget proposed by the authors in [14] to implement protected ML-DSA hardware design.

III. Algorithm Description

All the notations used in this section are as per the notations used in the FIPS-204 document [10]. ML-DSA has three internal algorithms that are key generation, signature generation, and signature verification. It is divided into three categories, category 2 (ML-DSA-44), category 3 (ML-DSA-65), and category 5 (ML-DSA-87). The parameter set of different categories is given in TABLE I.

TABLE I
ML-DSA Parameter Set [10]

Parameter	ML-DSA-44	ML-DSA-65	ML-DSA-87
Public key size	2560	1312	2420
Private key size	4032	1952	3309
Signature size	4896	2592	4627
γ_1	2^{17}	2^{19}	2^{19}
γ_2	(q-1)/88	(q1-/32)	(q-1)/32
(k, l)	$(4, 4)$	$(6, 5)$	$(8, 7)$
η	2	4	2
β	78	196	120

A. Key Generation

The key generation algorithm (Algorithm 1) generates a public key and a secret key pair using a random 32-bit seed ξ. This seed is expanded to to generate other seeds $\rho, \rho'$and K. ρ is expanded using the XOF function to pseudo-randomly sampling matrix A which is then stored in NTT representation. ρ' is expanded to sample s_1 and s_2 in the range $[-\eta, \eta]$. After expanding A, s_1 and s_2, $t = As_1 + s_2$ which is a public value. K is stored as it is and it is used in the signature generation algorithm. The public key is then packed which is byte encoding of t_1 and ρ. Then hash of the public key is computed and it is packed with other secret key variables.

Algorithm 1 Key Generation

Input: Seed $\xi \in \mathbb{B}^{32}$
Output: Public Key(pk) and Private Key(sk)
1: $(\rho, \rho', K) \in \mathbb{B}^{32} \times \mathbb{B}^{64} \times \mathbb{B}^{32} \leftarrow \text{H}(\xi||\kappa||l)$
2: $\hat{A} \leftarrow \text{ExpandA}(\rho)$
3: $(s_1, s_2) \leftarrow \text{ExpandS}(\rho')$
4: $t \leftarrow NTT^{-1}(\hat{A} \circ NTT(s_1)) + s_2$
5: $(t_1, t_2) \leftarrow \text{Power2Round}(t)$
6: $pk \leftarrow \text{pkEncode}(\rho, t_1)$
7: $tr \leftarrow H(pk, 64)$
8: $sk \leftarrow \text{skEncode}(\rho, K, tr, s_1, s_2, t_0)$

B. Signature Generation

Signature generation algorithm takes message M' which is message M padded with the contextual message, and secret key (sk) as input and generate the signature. The signature generation algorithm is shown in Algorithm 2. First, the secret key is decoded to get the seed and other variables. Then s_1, s_2, and t_0 are converted to NTT domain. ρ is expanded to sample matrix A which is then converted to NTT domain. Other seed ρ'' is computed which is expanded to sample y. From y, it computes $w = Ay$. w_1 nd μ and hashed to gen $\tilde{c}$. The signature z is computed as $z = y + cs_1$.

Algorithm 2 Signature Generation

Input: (sk, M', rnd)
Output: Signature σ
1: $(\rho, K, tr, s_1, s_2, t_0) \leftarrow \text{skDecode}(sk)$
2: $\hat{s_1} \leftarrow \text{NTT}(s_1)$
3: $\hat{s_2} \leftarrow \text{NTT}(s_2)$
4: $\hat{t_0} \leftarrow \text{NTT}(t_0)$
5: $\hat{A} \leftarrow \text{ExpandA}(\rho)$
6: $\mu \leftarrow H(BytesToBits(tr)||M', 64)$
7: $\rho'' \leftarrow H(K||rnd||\mu, 64)$
8: $\kappa \leftarrow 0$
9: $(z, h) \leftarrow \perp$
10: **while** $(z, h) = \perp$ **do**
11: $y \in R_q^l \leftarrow ExpandMask(\rho'', \kappa)$
12: $w \leftarrow NTT^{(-1)}(\hat{A} \circ NTT(y))$
13: $\tilde{c} \leftarrow H(\mu||w1Encode(w_1), \lambda/4)$
14: $c \in R_q \leftarrow SampleInBall(\tilde{c})$
15: $\tilde{c} \leftarrow NTT(c)$
16: $\langle\langle cs_1 \rangle\rangle \leftarrow NTT^{-1}(\hat{c} \circ \hat{s_1})$
17: $\langle\langle cs_2 \rangle\rangle \leftarrow NTT^{-1}(\hat{c} \circ \hat{s_2})$
18: $z \leftarrow y + \langle\langle cs_1 \rangle\rangle$
19: $r_0 \leftarrow LowBits(w - \langle\langle cs_2 \rangle\rangle)$
20: **if** $||z||_\infty \geq \gamma_1 - \beta$**or**$||r_o||_\infty \geq \gamma_2 - \beta$**then**$(z, h) \leftarrow \perp$ **then**
21: **else**
22: $\langle\langle ct_o \rangle\rangle \leftarrow NTT^{-1}(\hat{c} \circ \hat{t_0})$
23: $h \leftarrow MakeHint(-\langle\langle ct_0 \rangle\rangle, w - \langle\langle cs_2 \rangle\rangle + \langle\langle ct_0 \rangle\rangle)$
24: **end if**
25: **end while**
26: $\sigma \leftarrow sigEncode(\tilde{c}, z, h)$

After signature generation, validity check is performed and if validity check is failed, the signature generation continue the rejection sampling. If the validity is passed, hint is computed and it is encoded with final signature.

C. Signature Verification

The signature verification takes message M' and public key and generates a boolean output. If the output is true then the signature is valid else signature is invalid. The signature verification algorithm is given in Algorithm 3. First, the signature is decoded and if lengths are not as per defined value, it generates an invalid output. If decoding returns valid, signature verification continues by first expanding the seed ρ to sample matrix A which is then stored in NTT representation.

Then it samples challenge and computes $W'_{approx} = Az - ct_1$. It uses signer's hint to obtain w'_1 form W'_{approx}. Finally, it checks whether response z and hint are valid or not. If all succeeds, it returns true otherwise it returns false.

Algorithm 3 Signature Verification

Input: (pk, M', σ)
Output: Boolean
1: $(\rho, t_1) \leftarrow$ pkDecode(pk)
2: $(\tilde{c}, z, h) \leftarrow$ sigDecode(σ)
3: **if h** $= \perp$ **then return false**
4: **end if**
5: $\hat{A} \leftarrow$ **ExpandA**(ρ)
6: $tr \leftarrow H(pk, 64)$
7: $\mu \leftarrow (H(BytesToBits(tr)||M', 64))$
8: $c \leftarrow R_q \leftarrow SampleInBall(\tilde{c})$
9: $w'_{Approx} \leftarrow NTT^{-1}(A \circ NTT(z) - NTT(c) \circ NTT(t_1 \circ 2_d))$
10: $w'_1 \leftarrow UseHint(h, w'_{Approx})$
11: $\tilde{c}' \leftarrow H(\mu||w1Encode(w'_1), \lambda/4)$
12: **return**$[[||z||_\infty < \gamma_1 - \beta]]$**and**$[[\tilde{c} = \tilde{c}']]$

D. Masking Gadgets

The higher-order masking gadget algorithms form authors in [14] is used in the proposed protected ML-DSA design. Although they used these gadgets to protect Dilithium from side-channel attack but it can also be used for masking ML-DSA since the internal algorithms of ML-DSA is similar to Dilithium algorithms [10]. The masking is basically splitting the sensitive variables into shares such that combining the shares give the original value. The masking applies for boolean values such that $(b_1 \oplus b_2 \oplus \ldots \oplus b_n = b)$ where b is the original value. It applies for arithmetic value such that $(a_1 + a_2 + \ldots + a_n)$ mod q = a, where a is the original value.

The ML-DSA algorithms use both arithmetic and boolean values. Hence conversion from boolean to arithmetic and arithmetic to boolean is required to mask ML-DSA. The authors have proposed an efficient gadgets ShiftMod which is the base component in masking Dilithium [14]. They also provided boolean to arithmetic conversion algorithms which is used in Dilithium expandMask and decompose. There are three functions where masking is used which are expandMask, decompose and rejection sampling. In ML-DSA, these functions are the same as Dilithium hence these gadgets are also applicable to masking ML-DSA.

IV. Proposed Hardware Implementation

The proposed design is based on existing Dilithium 3.1 design from the authors in [13]. The source code of this design which includes the implementation of key-pair generation, signature generation, signature verification and KATs are available online in a GitHub repository https://github.com/GMUCERG/Dilithium.

The design exchanges data using 64-bit input-output ports. The authors have provided the Verilog testbench with the hardware implementation, which demonstrates how the data is

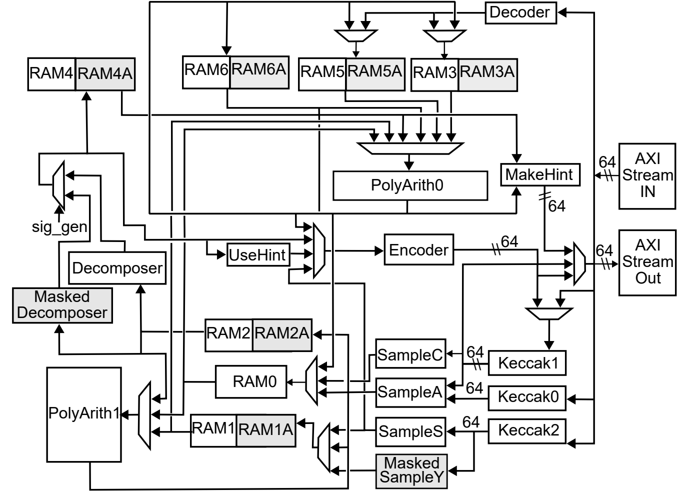

Fig. 1. Masked ML-DSA Hardware Implementation Architecture. Redrawn Dilithium 3.1 Architecture From [13].

exchanged. Other than the 64-bit port, there are more input and output ports like sec_level input port (2,3 and 5 are the valid inputs) for setting Dilithium security level, mode (0,1,2 are valid inputs) for setting which algorithm to run like key-pair generation (mode 0), sign (mode 2), or verify (mode 1) and handshaking signals valid_i, ready_o, ready_i, and valid_o.

The design uses two cores of polynomial arithmetic unit (PolyArith), which performs NTT, polynomial addition, and subtraction operations. The PolyArith core uses the four butterfly unit to perform arithmetic operations. The BRAM used to store the coefficient has 96-bit in each address which is capable of storing 4 coefficients per address. It also uses three KECCAK cores which perform computation independently to each other for parallel operation.

A. Masked ML-DSA Hardware Implementation Flow

First, we updated the existing Dilithium 3.1 design to the final ML-DSA standard. There are a few changes in the Dilithium 3.1 to ML-DSA which can be implemented easily in hardware. These changes are listed below:

1) Domain separation added in the key generation algorithm.
2) The length of tr is increased from 256-bit to 516-bit.
3) The size of the $\tilde{c}$ is increased from 256-bit to 384-bit and 512-bit.
4) Addition of 256-bit random number in signature generation to generate ρ'.
5) Addition of contextual message padded with the original message.

In hardware, these changes are made by changing the counter read, and counter write value of the KECCAK module since the changes either increase the number of absorb data or squeeze data. After the design update, it is verified using the Known Answer Tests(KATs) generated from an updated ML-DSA software implementation. We further tried to optimise the design by locating the critical path and reducing it. We identified the critical path parallel in serial out buffer and we reduced the buffer size which reduces the critical path.

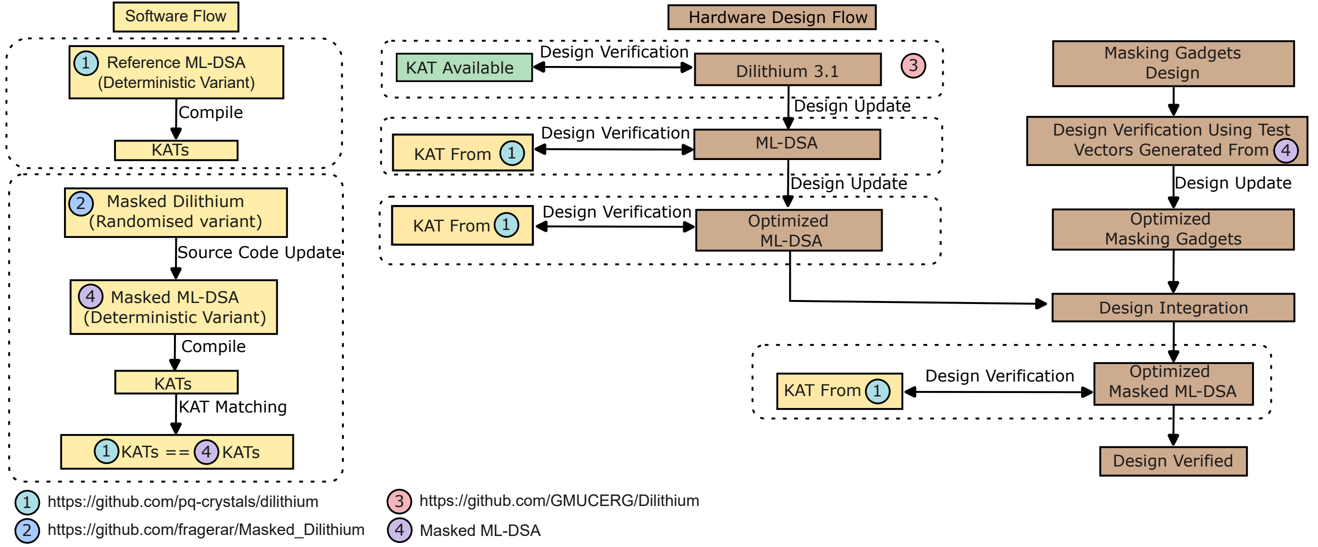

Fig. 2. Design and Verification Flow of Proposed Masked ML-DSA Hardware Implementation.

Next, we designed and integrated masking gadgets with the ML-DSA design. The architecture of the proposed design is shown in Fig. 1.It is similar to the Dilithium 3.1 design but the only difference is the modules that perform computation with sensitive variables are replaced with masked modules, and extra BRAM is added to store the shares of sensitive variables.

The most important part of designing hardware is the design should functionally correct. It can be checked using the Known Answer Tests (KAT). There are different variants of ML-DSA signature generation which is deterministic, hedged, and randomized. The author in [18] have provided the masked implementation for randomised Dilithium. The randomised variant makes verification of the design difficult and also it makes difficult to find errors in the design. To solve this issue, we updated the masked Dilithium C implementation to the masked ML-DSA deterministic variant with minor changes in the C code. Then we used this implementation to verify individual masking gadgets as well as to verify complete masked signature generation. The design and verification flow of the proposed hardware is shown in Fig. 2. The hardware design of various modules used to implement masking countermeasures are given below.

B. Modular Reduction

Modular reduction is the most expensive computation in both ML-DSA and masking algorithms. Optimising this can lead to significant improvement in area and performance. Since we are implementing masking, it is essential that we implement modular reduction in an efficient way. There are different methods for modular reduction which are Barrett reduction and Montgomery reduction.

There is a modular reduction method which is very efficient in hardware and it is presented by the authors in [19] [20]. The authors in [15] used the same method to implement modular reduction for Dilithium. We implemented and used the same design to computed mod q, mod 2q and mod 4q, which is required in designing masking gadgets. The

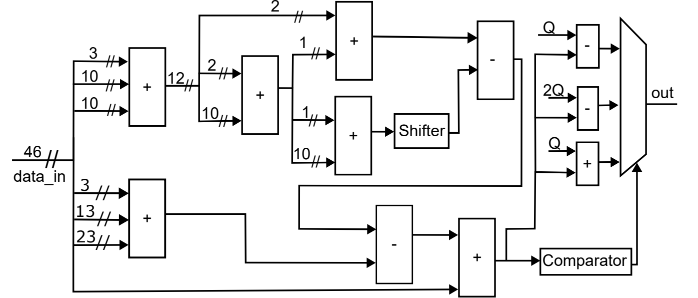

Fig. 3. Modular Reduction Hardware Implementation [15].

modular reduction uses the modulus property of Dilithium recursively, which is $2^{23} = 2^{13} - 1 \pmod{8380417}$ [15]. The equations for modular reduction for Dilithium are given below:

$$a = 2^{23}a[45:23] + a[22:0] \tag{1}$$

Using the modulus property we get:

$$\begin{aligned} &= 2^{13}(a[45:43] + a[42:33] + a[32:23]) \\ &- (a[45:43] + a[45:33] + a[45:23]) + a[22:0] \end{aligned} \tag{2}$$

$$= 2^{13}c - e + a[22:0](\text{ mod q}) \tag{3}$$

Using the same modulus property, we can reduce c as:

$$2^{13}c = 2^{13}(c[11:10] + c[9:0]) - c[11:10] \tag{4}$$

$$= 2^{13}f - c[11:10](\text{ mod q}) \tag{5}$$

Further reduction gives the following equation:

$$= 2^{13}(f[10] + f[9:0]) - (f[10] + c[11:10]) \tag{6}$$

The modulus reduction described above require shift and add operations which is efficient in hardware. Fig. 3 shows the circuit diagram of modular reduction for mod q using the

technique described above. The similar technique is used to calculate mod 2q and mod 4q which are used in masking gadgets, with the cost of increased size of the adder circuit.

C. Psudo Random Number Generator (PRNG)

The masking requires random numbers to refresh the masked coefficients. In our hardware design, we used a Pseudo Random Number Generator (PRNG) for refreshing the variables. We choose the size of PRNG 96-bits. It is enough for masking operation because the design requires a random number at most 47-bit for masking y, and 24-bit for masking the rest of the variables.

D. Masking s_1 and s_2

The s_1 and s_2 variables are used in signature generation algorithms (Line 16 to 18 in Algorithm 2). These variables are first converted to the arithmetic shares and then it is used in the masked signature generation. The circuit for masking s_1 and s_2 are shown in Fig. 4. It takes s_1 or s_2 as an input, and generate two output shares such that (share1 + share2) mod q = (s_1 or s_2). Both the shares are stored separately in memory so that it can be use later in the masked signature generation operation. The shares are refreshed with random number each and every time when the signature generation is performed in the hardware. Refreshing the values with random number makes harder for the attacker to get the values from side-channel attack.

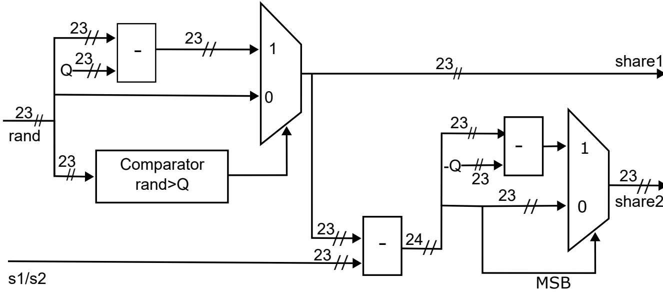

Fig. 4. Circuit For Masking s1 and s2.

E. Masking y

The expandmask function samples y from the SHAKE256 output and it is used to compute signature z. The masked sampler takes input form expandmask hardware, then it makes boolean shares of it using random number and then, it generates two share of y. The architecture for masking y is shown in Fig. 5. The masked sampler uses the boolean to arithmetic conversion gadgets and Shift Mod gadget [14] to generate shares of y.

The shares of y is computed such a way that, (y_share1 + y_share2) mod q = y. After this, both the shares are made in the range of q using the center hardware which subtracts q if the value of the share is more than q. This module also refreshes the shares of y using random number each and every time the module computes y.

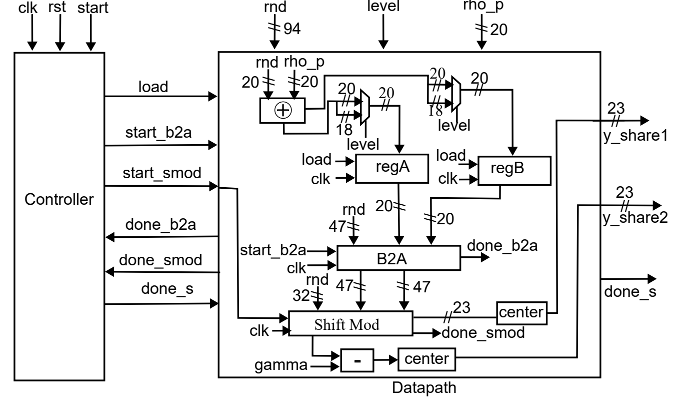

Fig. 5. Masked Sampler Architecture.

F. Masked Decompose

The masked decompose function computes HighBits and LowBits of w as w_0 and w_1. It takes two shares of w as an input and computes two shares of w_0 and unmasks the $w1$ variable because it can be skipped for masking. The architecture of masked decompose is shown in Fig. 6. This

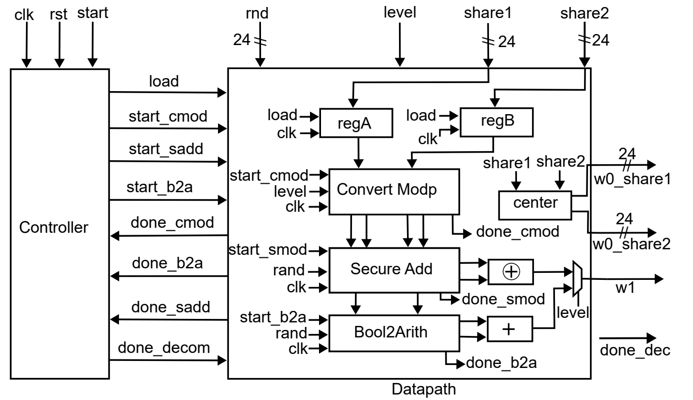

Fig. 6. Masked Decompose Architecture.

module uses the boolean to arithmetic conversion which is also used in the masked sampler and other masking gadgets to compute the share of w_1. This module also refreshes the values using the random number each and every time the masked w_1 is computed.

G. Masked Rejection

The signature generation algorithm rejects the signature if it leaks the information. If the signature is rejected, it computes y again with the next nonce and all the computation is repeated till there is no rejection. It is also masked using the masked gadgets with is used in previous masked decompose module. It takes two shares of z and r and 24-bit random number to refresh the values and set control signal to reject if there is rejection. The architecture of masked rejection sampling is shown in Fig. 7.

H. Unmasking

If there is no rejection, the signature is unmasked. It is computed only after the completion of rejection sampling.

TABLE II
PERFORMANCE COMPARISON OF ML-DSA IMPLEMENTATION ON FPGA (KINTEX-7) SYNTHESIZED IN AMD VIVADO 2024.1

Hardware Module	LUT	FF[1]	BRAM[1] (36 Kb)	DSP[1]	Freq[1] (MHz)	Keygen Cycles[2] ($\times 10^3$)	Keygen Time (μs)	Sign Cycles ($\times 10^3$)	Sign Time (μs)	Verify Cycles[3] ($\times 10^3$)	Verify Time (μs)
Dilithium 3.1 [13]	54668	28199	29	16	111.8	14.030	125.49	24.846	222.24	14.635	130.9
ML-DSA Ref	55040	28746	29	16	106.2	14.038	125.56	24.857	222.33	14.647	131.01
ML-DSA Opt	55156	28625	29	16	125.9	14.038	111.5	24.857	197.43	14.647	116.34
ML-DSA Masked	61629	33861	47	21	122.5	14.038	114.59	1151.294	9398.32	14.647	119.56

[1] Same design constraint and synthesis strategy are used for fair comparison. [2] Cycle count is for ML-DSA-88's best case sign generation.

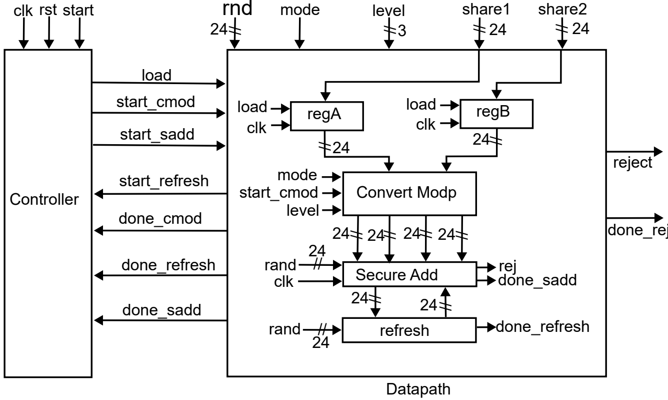

Fig. 7. Masked Rejection Sampling Architecture.

Unmasking is basically the addition of the arithmetic shares of the signature z and the variable r followed by mod Q operation such that z or r = (share1 + share2) mod q. Since we have implemented first order masking, we can use subtraction circuit instead of mod circuit, which is less expensive in hardware. The unmasking circuit is shown in Fig. 8.

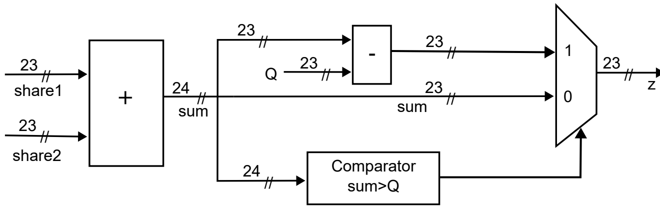

Fig. 8. Circuit For Unmasking Signature z.

V. RESULTS

In this section, we present the hardware synthesis, performance and side channel analysis results of masked ML-DSA hardware implementation. The masking gadget design is described in Verilog HDL and it is integrated with the existing hardware implementation of Dilithium 3.1. The synthesis report of masking gadget hardware is given in Table III. These masking gadget are integrated with the improved ML-DSA hardware which is verified with the Known Answer Test (KAT) generated from an updated ML-DSA software implementation. The synthesis report of complete system is given in Table II.

A. Hardware Performance and Area Analysis

At present, there is no hardware implementation of protected ML-DSA known to us, so we compared our proposed protected ML-DSA design with the unprotected design to compare the area and computational overhead. The performance comparison of the proposed design is given in the TABLE II. The intermediate values of masked variables need to be stored separately which increases the BRAM count in the proposed design. The expandmask, decompose, and rejection sampler modulus are replaced with a new masked design which also increased the LUT count. Also, the finite state machine of the top module is updated so that it can compute and process masked variables, which also increased the total LUT count. The computation of masked variables requires more clock cycles as given in TABLE III that increased the execution time of signature generation. Other operations like NTT multiplication, addition, and subtraction to the masked variable are computed twice, which further increases the execution time of signature generation.

B. Challenges of Implementing Masking Countermeasure

Masking implementation requires the redesign of the targeted module which computes the sensitive variable. The design needs to be integrated with the unmasked design by replacing the unprotected module with the protected one. The integration is also challenging since the design has various handshaking signals that need to be taken care of during design integration. Since the masked variable needs to be computed twice and more constraints need to be added in the design since some part of the computation is unmasked, it also requires the redesign of the finite state machine.

TABLE III
SYNTHESIS RESULT OF MASKING GADGETS IMPLEMENTED ON KINTEX-7 FPGA

Module	LUT	FF	DSP	Cycles *
PRNG	88	192	0	1
Masked s1 and s2	60	0	0	1
Masked Sampler	2256	1587	3	58
Masked Decompose	1664	1107	2	678,132**
Masked Rejection	854	1233	0	166
Unmask	75	0	0	1

* It is the number of clock cycle required to process one coefficient.
** Boolean to arithmetic conversion is not required for ML-DSA-65 and ML-DSA-87.

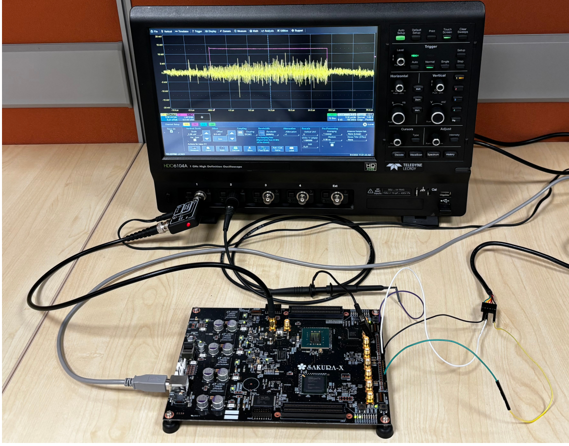

Fig. 9. Experimental Setup for Side-Channel Attack Resistant Evaluation.

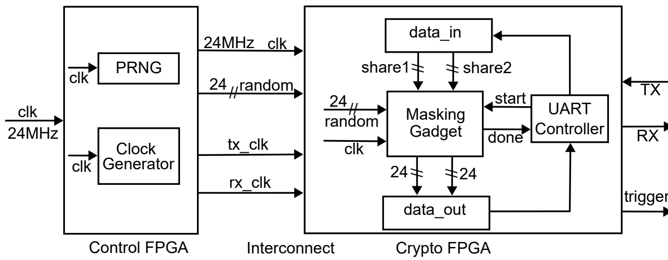

Fig. 10. Block Diagram of Side-Channel Attack Trace Collection Method.

C. Side-Channel Attack Analysis

1) Experimental Setup: The masking gadgets is evaluated for Side-Channel attack (SCA) resistant using standard side channel attack testing FPGA board (SAKURA X). The experimental setup is shown in Figure 9. This FPGA board have two FPGAs Kintex-7 (to implement cryptographic design) and Spartan-6 (to implement non-cryptographic design). The target hardware (masking gadget design) is implemented on the Kintex-7 FPGA which is running on 24MHz clock frequency. To eliminate the noise effect generated from parallel hardware, other hardware module like UART clock generator and PRNG are implemented on the control FPGA. Fig. 10 shows the block diagram of different hardware modules implemented on control and crypto FPGAs to perform side-channel attack experiment. For trace collection, High Definition digital storage oscilloscope with the sample size of 100 Mega samples per second, a low noise amplifier, and 200 MHz internal digital filter are used.

2) Testing Methodology: We used the similar technique used by the authors in [18] in which each masked module is tested separately for the experiment. Although, they used the simulation based power result which is only limited to 10 thousand traces, we used actual power consumption trace from FPGA implementation beyond 10 thousand traces. The masked share, which is two shares of original value is generated from the computer and transferred it to the FPGA through UART. The UART controller enables the masking gadgets once both of the shares are received. Once the masked operation are performed, the UART controller sends the result to the PC. The

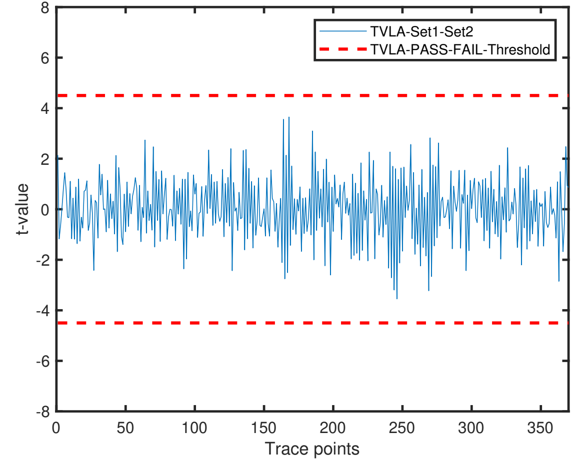

Fig. 11. Masked Sampler t-test Result (1 million Traces).

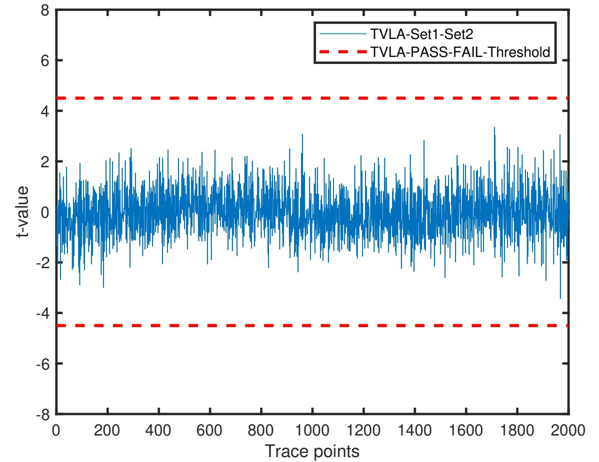

Fig. 12. Masked Decompose t-test Result (1 million Traces).

trigger signal is used to locate the masking operation since we used asynchronous communication protocol. The trigger signal is just ex-or operation of start signal and done signal.

After the trace collection, the computed result is checked in the PC. If the output is unexpected, the collected trace is rejected and it is re-run again for the same sample data until computed operation is correct and expected. We have done this to avoid any false data collected which is generated by unexpected glitching the hardware which can led to false positive or false negative result in the t-test plot. Then, 1 million traces are logged in the system, and t-test plot is generated for first 500K traces as first-set and next 500K traces as second-set. The t-test plot is plotted for masked decompose, sampler and rejection sampler and are shown in Figure 12, Figure 11 and Figure 13 respectively. As shown in the plot, there is no leakage in the design for 1 million traces.

VI. Conclusion

In this paper, we presented improved ML-DSA hardware implementation with the addition of masking countermeasures

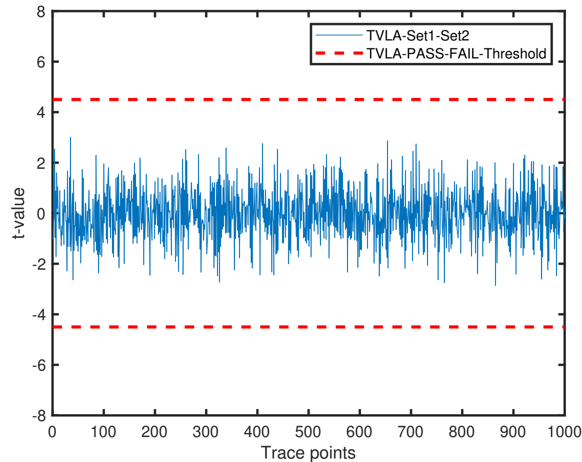

Fig. 13. Masked Rejection Sampling t-test Result (1 million Traces).

to protect the hardware implementation from the side-channel attack threat. The Proposed hardware design is implemented as per ML-DSA final standard which was published on 13 August 2024. It is verified with the Known Answer Tests (KATs), which are generated from an updated ML-DSA software implementation. After designing and verifying ML-DSA hardware, masking gadgets are designed and integrated with the ML-DSA hardware by replacing expandmask, decompose, and rejection sampling module with the masked module which is functionally equivalent to these modules. These masked modules generate two shares of sensitive variables to protect these variables from the side-channel attack. Finally, the side-channel attack experiments are performed to test the masked module to see if it is leaking any information or not. The experimental results show that the proposed masked design is resistant to side-channel attacks.

References

[1] W. S. Admass, Y. Y. Munaye, and A. A. Diro, "Cyber security: State of the art, challenges and future directions," *Cyber Security and Applications*, vol. 2, p. 100031, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2772918423000188

[2] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*, 2nd ed. Chapman & Hall/CRC, 2014.

[3] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997, _eprint: https://doi.org/10.1137/S0097539795293172. [Online]. Available: https://doi.org/10.1137/S0097539795293172

[4] C. Paquin, D. Stebila, and G. Tamvada, "Benchmarking Post-quantum Cryptography in TLS," in *Post-Quantum Cryptography*, J. Ding and J.-P. Tillich, Eds. Cham: Springer International Publishing, 2020, pp. 72–91.

[5] T. Wang, C. Zhang, P. Cao, and D. Gu, "Efficient Implementation of Dilithium Signature Scheme on FPGA SoC Platform," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 9, pp. 1158–1171, 2022.

[6] S. Mathew, S. Satpathy, V. Suresh, M. Anders, H. Kaul, A. Agarwal, S. Hsu, G. Chen, and R. Krishnamurthy, "340 mV–1.1 V, 289 Gbps/W, 2090-Gate NanoAES Hardware Accelerator With Area-Optimized Encrypt/Decrypt GF(2 4) 2 Polynomials in 22 nm Tri-Gate CMOS," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 4, pp. 1048–1058, 2015.

[7] Shay Gueron, "Intel® Advanced Encryption Standard (AES) New Instructions Set," 2020. [Online]. Available: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[8] X. Hou and J. Breier, "Side-Channel Analysis Attacks and Countermeasures," in *Cryptography and Embedded Systems Security*. Cham: Springer Nature Switzerland, 2024, pp. 205–352. [Online]. Available: https://doi-org.remotexs.ntu.edu.sg/10.1007/978-3-031-62205-2_4

[9] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[10] N. I. of Standards and Technology, "Module-lattice-based digital signature standard," *(Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 204*, 2024.

[11] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium algorithm specifications and supporting documentation," 2021. [Online]. Available: https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf

[12] Z. Qiao, Y. Liu, Y. Zhou, J. Ming, C. Jin, and H. Li, "Practical Public Template Attacks on CRYSTALS-Dilithium With Randomness Leakages," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 1–14, 2023.

[13] 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.

[14] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun, "Improved Gadgets for the High-Order Masking of Dilithium," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 4, pp. 110–145, Aug. 2023. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/11160

[15] N. Gupta, A. Jati, A. Chattopadhyay, and G. Jha, "Lightweight Hardware Accelerator for Post-Quantum Digital Signature CRYSTALS-Dilithium," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 8, pp. 3234–3243, 2023.

[16] X. Li, J. Lu, D. Liu, A. Li, S. Yang, and T. Huang, "A High Speed Post-Quantum Crypto-Processor for Crystals-Dilithium," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 71, no. 1, pp. 435–439, 2024.

[17] A. Aikata, A. C. Mert, M. Imran, S. Pagliarini, and S. S. Roy, "KaLi: A Crystal for Post-Quantum Security Using Kyber and Dilithium," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 2, pp. 747–758, 2023.

[18] V. Migliore, B. Gérard, M. Tibouchi, and P.-A. Fouque, "Masking Dilithium Efficient Implementation and Side-Channel Evaluation," in *Applied Cryptography and Network Security*, R. H. Deng, V. Gauthier-Umaña, M. Ochoa, and M. Yung, Eds. Cham: Springer International Publishing, 2019, pp. 344–362.

[19] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, and Shaojun Wei, "Highly Efficient Architecture of NewHope-NISTon FPGA using Low-Complexity NTT/INTT," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 2, pp. 49–72, Mar. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8544

[20] F. Yaman, A. C. Mert, E. Öztürk, and E. Savaş, "A Hardware Accelerator for Polynomial Multiplication Operation of CRYSTALS-KYBER PQC Scheme," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1020–1025.