

# Area-Efficient Hardware Architecture of the Post-Quantum HQC Cryptosystem with Side-Channel Protection

Kamal Raj<sup>1</sup>, Member, IEEE, Peizhou Gan<sup>2</sup>, Sayan Das<sup>3</sup>, Anupam Chattopadhyay<sup>4</sup>, Senior member, IEEE

**Abstract**—The Hamming-Quasi Cyclic (HQC) is the Post-Quantum Cryptography (PQC) Key Encapsulation Mechanism (KEM) algorithm recently selected by the National Institute of Standards and Technology (NIST) for standardization as an alternative candidate for the KEM category. While the algorithm is mathematically robust, several studies state that the physical implementation of HQC remains vulnerable to power-based side-channel attacks (SCA). Countermeasures for such SCAs primarily focus on masking and shuffling, which severely increase hardware resources, introducing additional performance overhead. To address this problem, this paper presents a resource-efficient, constant-time hardware architecture for HQC that features a novel idle-hardware reuse strategy as an SCA countermeasure. By intentionally reusing the KECCAK hardware to generate dynamic power during decoding, the proposed architecture scrambles secret-dependent power correlations with less than 0.5% area overhead for the countermeasure logic. The complete design is fully parameterized to support all NIST security categories (HQC-128, HQC-192, and HQC-256). A prototype implementation on an Artix-7 FPGA achieves a maximum clock frequency of 139 MHz while consuming 11,489 LUTs, 6,785 FFs, and 4 DSPs. Functional correctness is verified via NIST Known Answer Tests (KATs) across Key Generation, Encapsulation, and Decapsulation operations, and Test Vector Leakage Assessment (TVLA) confirms significant suppression of side-channel leakage.

**Index Terms**—PQC, HQC, FPGA, Side-Channel Attack, Countermeasure

## I. INTRODUCTION

The National Institute of Standards and Technology (NIST) Post-Quantum Cryptography (PQC) standardization process is emerging from theoretical standardization to real-world deployment. Hamming-Quasi Cyclic (HQC) [1] is a code-based cryptography algorithm selected for standardization by NIST as a backup Key-Encapsulation Mechanism (KEM) in early 2025. Although the algorithm is theoretically secure against known quantum attacks such as Shor's [2], its implementation on hardware and embedded devices remains highly vulnerable to Side-Channel Attacks (SCAs). HQC implementations have been shown vulnerable to fault-injection [3], electromagnetic [4], and power side-channel attacks [5], [6], including single-trace [7], [8], [9] and chosen-ciphertext techniques that exploit secret-dependent behavior in polynomial multiplication and decoding. Although authors did not target the hardware implementation, similar attacks may be possible on Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) due to similar physical characteristics.

A critical architectural characteristic of HQC algorithms is the distinct separation of operational phases between public-key arithmetic and error-correction decoding. It uses concatenated Reed-Muller and Reed-Solomon encoders to encode the message during encapsulation and their corresponding decoders during decapsulation. As the decapsulation procedure utilises a repeated RM decoder with small GF(256) arithmetic operations to process the primary secret message during these operations, this becomes a primary target for attackers leveraging side-channel leakage.

There are various hardware implementations of HQC, ranging from high-performance [10] designs to area-efficient designs; however, there is a lack of side-channel-protected designs in the literature. Existing hardware countermeasures against SCA attacks, such as Boolean or arithmetic masking, rely on splitting the intermediate values into multiple random shares. This requires additional hardware resources to split and process the shares and may also require extra memory to store the intermediate random shares. Applying masking techniques to the HQC decoder increases the hardware area, BRAM, and cycles required to decode the message. Additionally, modern SCA attacks target the masked implementation and recover the key [11].

To address this problem, this paper outlines a hardware-efficient SCA protection methodology for HQC's hardware implementation. During secret-dependent decoding, cryptographic blocks such as the KECCAK permutation core remain idle; we leverage this idle hardware to protect secrets from side-channel leakage. From a hardware perspective, the KECCAK core is the largest and most power-hungry module in the high-performance core implementation because of its high-density bitwise operations across a 1600-bit permutation state.

In HQC, KECCAK is treated purely as a computational unit during key generation, encapsulation, and decapsulation, and remains completely deactivated during error-correction decoding, since the next operation cannot proceed until message decoding is finished. However, this strict scheduling leaves a quiet, low-noise, secret-dependent decoder operation easily distinguishable in correlation power attacks. Using the KECCAK core's random switching activity as an active hiding mechanism during these operations is an unexplored opportunity for a resource-efficient side-channel countermeasure.

To bridge the gap between heavy, area-intensive side-channel protections and the low-cost requirements of PQC deployments, this work presents a constant-time, SCA-resistant hardware architecture for HQC. By exploiting the KECCAK core's dynamic power signature during decoding, our design achieves robust leakage suppression without dedicated mask-

The authors are with the College of Computing and Data Science, Nanyang Technological University, Singapore 639798 (e-mail: kamalraj@ieee.org; peizhou001@e.ntu.edu.sg; sayan005@e.ntu.edu.sg; anupam@ntu.edu.sg).

TABLE I: HQC Parameter Set

| Instance | $n_1$ | $n_2$ | $n$    | $w$ | $w_r = w_e$ | $\lambda$ |
|----------|-------|-------|--------|-----|-------------|-----------|
| HQC-128  | 46    | 384   | 17,669 | 66  | 75          | 128       |
| HQC-192  | 56    | 640   | 35,851 | 100 | 114         | 192       |
| HQC-256  | 90    | 640   | 57,637 | 131 | 149         | 256       |

ing logic. The main contributions of this work are as follows:

- 1) We present a novel hardware architecture for the HQC cryptosystem that uses the shared KECCAK core to protect the decoder against power SCA leakage. Our hardware reuse strategy provides robust SCA protection with an overall area overhead of less than 0.5%, at the cost of only a few tens of additional clock cycles.
- 2) By using a single-port BRAM, reusing the memory space for intermediate-value computation via hardware address mapping, we reduced BRAM usage by up to 33.3%.
- 3) The proposed hardware is fully scalable and supports all NIST security categories HQC-128, HQC-196, and HQC-256 without requiring additional hardware changes, and all security categories are functionally verified using the NIST Known Answer Tests (KATs).
- 4) We implemented our design on the SAKURA-X SCA evaluation platform, and the Test-Vector Leakage Assessment (TVLA) confirms mitigation of data-dependent leakage, validating the practical side-channel security of the proposed countermeasure.

The rest of this paper is organized as follows. Section II provides the necessary background on HQC algorithms, related work, and existing side-channel attacks on HQC implementations. Section III details the proposed constant-time hardware architecture and the KECCAK-based hardware-reuse SCA countermeasure. Section IV presents the security analysis and practical side-channel evaluation using Test Vector Leakage Assessment (TVLA). Section V discusses the FPGA synthesis results and compares them with state-of-the-art designs. Finally, Section VI concludes the paper.

## II. BACKGROUND

Hamming-Quasi Cyclic (HQC) [1] is a code-based KEM based on the Quasi Cyclic Syndrome Decoding problem, which is robust to quantum attacks. Like any KEM, it consists of three base algorithms: key generation, encapsulation, and decapsulation. The parameter sets for its different levels of security are stated in Table I. Following subsections provide a brief description of the HQC cryptosystem.

### A. Key Generation Algorithm

Using the random entropy source, a seed ( $\text{seed}_{\text{KEM}}$ ) is generated, which is further used to compute the randomness ( $\sigma$ ), seeds for generating pseudo-random sparse polynomials ( $x, y$ ), and a dense polynomial ( $h$ ). These polynomials generate the encapsulation key ( $\text{ek}_{\text{KEM}}$ ) via polynomial multiplication and addition. Algorithm 1 presents the HQC key-generation algorithm.

### Algorithm 1 HQC Key Generation Algorithm

**Output:** ( $\text{ek}_{\text{KEM}}, \text{dk}_{\text{KEM}}$ )

- 1:  $\text{seed}_{\text{KEM}} \leftarrow \mathcal{B}^{|\text{seed}|}$
- 2:  $(\text{seed}_{\text{PKE}}, \sigma) \leftarrow \text{XOF}(\text{seed}_{\text{KEM}})$
- 3:  $(\text{seed}_{\text{dk}}, \text{seed}_{\text{ek}}) \leftarrow \text{I}(\text{seed}_{\text{PKE}})$
- 4:  $(x, y) \leftarrow \text{SampleFixedWeightVect}(\text{XOF}(\text{seed}_{\text{dk}}))$
- 5:  $s \leftarrow \text{SampleVect}(\text{XOF}(\text{seed}_{\text{ek}}))$
- 6:  $s \leftarrow x + h \cdot y$
- 7:  $\text{ek}_{\text{KEM}} \leftarrow (\text{seed}_{\text{ek}}, s)$
- 8:  $\text{dk}_{\text{KEM}} \leftarrow (\text{ek}_{\text{KEM}}, \text{seed}_{\text{dk}}, \sigma, \text{seed}_{\text{KEM}})$
- 9: **return** ( $\text{ek}_{\text{KEM}}, \text{dk}_{\text{KEM}}$ )

### B. Encapsulation Algorithm

It takes the  $\text{ek}_{\text{KEM}}$ , and computes the shared secret key ( $K$ ) and ciphertext ( $c_{\text{KEM}}$ ). To perform the operation, three sparse polynomials are sampled from the sampler. A ciphertext  $u$  is computed by multiplying  $h$  with  $r_2$  and adding  $r_1$ . The ciphertext  $v$  is computed by multiplying  $s$  by  $r_2$  and adding  $e$  and the encoded message. Algorithm 2 gives the HQC encapsulation algorithm.

### Algorithm 2 HQC Encapsulation Algorithm

**Input:** ( $\text{ek}_{\text{KEM}}$ )

**Output:** ( $K, c_{\text{KEM}}$ )

- 1:  $(m, \text{salt}) \leftarrow \mathcal{B}^{|k|}, \mathcal{B}^{|\text{salt}|}$
- 2:  $(K, \theta) \leftarrow \text{G}(\text{H}(\text{ek}_{\text{KEM}}) || m || \text{salt})$
- 3:  $h \leftarrow \text{SampleVec}(\text{XOF}.\text{Init}(\text{seed}_{\text{PKE}}))$
- 4:  $(r_2, e, r_1) \leftarrow \text{SampleFixedWeightVect}(\text{XOF}(\theta))$
- 5:  $u \leftarrow r_1 + h \cdot r_2$
- 6:  $v \leftarrow \text{Encode } C(m) + \text{Truncate}(s \cdot r_2 + e)$
- 7:  $c_{\text{KEM}} = (u, v)$
- 8: **return** ( $K, c_{\text{KEM}}$ )

### C. Decapsulation Algorithm

It takes the ciphertext and the secret key as input, recovers the message bytes using the HQC decoder, and computes the shared secret key. The recovered message is re-encrypted, and the recomputed ciphertext is compared with the received ciphertext. The rejection key is returned if there is a mismatch between the computed ciphertext and the received ciphertext. Algorithm 3 presents the HQC decapsulation algorithm.

### D. Existing Hardware Implementations

There are multiple hardware implementations available focusing on different aspects such as performance and specific optimisations. Deshpande et al. [10] implemented a high-performance HQC design on an Artix-7 FPGA. Antognazza et al. [12] implemented a high-performance HQC with optimizations for polynomial multiplication and, in [13], extended their work to separate the key generation, encapsulation, and decapsulation hardware modules. Li et al. [14] implemented HQC-128 on an Artix-7 FPGA with an efficient Reed-Solomon (RS) and Reed-Muller (RM) decoder. Schoffel et

---

**Algorithm 3** HQC Decapsulation Algorithm
 

---

**Input:**  $(dk_{KEM}, c_{KEM})$ **Output:**  $K$ 

```

1:  $y \leftarrow \text{SampleFixedWeightVect}(\text{XOF}(\text{seed}_{\text{PKE.dk}}))$ 
2:  $m' \leftarrow \mathcal{C}.\text{Decode}(v - \text{Truncate}(u.y))$ 
3:  $(K', \theta') \leftarrow G(\text{H}(\text{ek}_{KEM}) || m || \text{salt})$ 
4:  $c'_{PKE} \leftarrow \text{HQC-PKE}.\text{Encrypt}(\text{ek}_{KEM}, m', \theta')$ 
5:  $c'_{KEM} \leftarrow (c'_{PKE}, \text{salt})$ 
6:  $K' \leftarrow J(\text{H}(\text{ek}_{KEM}) || \sigma || c_{KEM})$ 
7: if  $m' = \perp$  or if  $c'_{KEM} \neq c_{KEM}$  then
8:    $K' \leftarrow \bar{K}$ 
9: end if
10: return  $K'$ 

```

---

al. [15] implemented the HQC using a hardware-software co-design method with a RISC-V processor core. Most of these implementations focus on high performance and constant-time execution, thereby lacking side-channel resistance.

### E. Existing Side-Channel Attacks

Side-channel attacks are a serious threat to PQC implemented on physical hardware. Code-based cryptography algorithms are attacked by various SCA attacks in the literature. To date, no side-channel attacks have been reported against hardware implementations of HQC, but side-channel attacks on software implementations suggest that similar attacks can be possible on hardware implementations if secret-dependent operations are not well protected with SCA countermeasures.

1) *Fault-Injection Attack*: Xagawa et. al. in [3] outlined a fault-injection attack on the Round 3 version of HQC in KEM mode. This attack introduced a glitch in the decapsulation routine's equality test  $ct' = ct$ , causing it to always output  $\mathcal{K}(\bar{m}', ct)$ , even for invalid ciphertexts  $ct$ . This lets the attacker use it as a plaintext-checking oracle and, by sending carefully crafted, malformed ciphertexts, obtain enough information to recover the victim's secret key.

2) *Electromagnetic SCA*: Paiva et. al. in [4] performed power and electromagnetic side-channel analysis using a physical EM probe on a version of PQClean's HQC implementation running on an STM32F4 microcontroller. They obtained information about secret codeword bits processed by the `expand_and_sum` and `find_peaks` routines and recovered an HQC secret key.

3) *Power Analysis Attacks*: Goy et al. [5] presented a side-channel attack on the HQC reference software running on an ARM Cortex-M4 microcontroller. They targeted the decapsulation phase, specifically the syndrome computation step in the Reed-Solomon decoder. By applying horizontal correlation attacks using the Hamming weight leakage model, they showed that an attacker could recover intermediate values during decoding and reconstruct the secret key. This attack highlights that the Reed-Solomon syndrome computation is a major side-channel weakness in software implementations of HQC.

Velek et al. [6] presented a Simple Power Analysis (SPA) attack targeting the polynomial multiplication step in HQC. By

observing the power consumption profile of a single execution of the polynomial multiplier, they showed that secret key bits could be directly distinguished because of operational differences in the underlying arithmetic logic. This work shows that without constant-time execution and side-channel countermeasures, even core arithmetic operations such as polynomial multiplication remain vulnerable to straightforward single-trace power analysis.

4) *Single-Trace Attacks*: Goy et al. [7] presented a single-trace side-channel attack on the HQC decapsulation algorithm targeting the decoding process. Using Soft Analytical Side-Channel Attacks (SASCA), they exploited side-channel leakage from a single decoder execution to recover the shared secret key. This work demonstrated that HQC decoding is highly vulnerable to single-trace power analysis, underscoring the critical need for hardware-level countermeasures during decoding.

Dong et al. [8] presented a single-trace key recovery attack on HQC by combining power side-channel analysis with valid and invalid ciphertexts. By feeding constructed ciphertexts into the decapsulation process and observing power leakage during execution, they showed that an adversary can exploit subtle operational differences during decoding to recover the secret key from a single trace. This demonstrated that HQC remains vulnerable when power leakage is combined with chosen-ciphertext strategies, further underscoring the need to secure the decoding architecture against side-channel analysis.

Lai et al. [9] presented a ciphertext-independent, single-trace passive side-channel attack on the HQC decapsulation algorithm. Their approach demonstrates that an attacker need not even manipulate input ciphertexts to extract secret key information; observing power leakage during a single standard decapsulation execution is sufficient to recover the secret key. This result highlights that passive side-channel leakage during the decoding phase poses an immediate threat, reinforcing the need for continuous, autonomous hardware countermeasures.

5) *Attack on Masked HQC*: Jeon et al. [11] demonstrated that applying codeword masking to HQC can actually introduce new side-channel vulnerabilities under replay attacks. By analyzing power leakage during the decapsulation phase when identical or constructed ciphertexts are replayed, they showed that attackers can bypass intermediate masked states in the decoding pipeline to recover secret information. This work highlights that traditional masking techniques for HQC are not only resource-intensive but can also fail under active replay scenarios, further underscoring the advantage of physical-layer dynamic noise generation over pure algorithmic masking.

As the side-channel attacks above show, the HQC must decode the message before completing decapsulation. This decoding phase is mathematically and operationally distinct from the rest of the cryptosystem because it relies on concatenated Reed-Muller (RM) and Reed-Solomon (RS) decoders that perform fine-grained GF(256) arithmetic operations. Since these specific operations are highly vulnerable to side-channel power leakage, we propose a low-cost, hardware-efficient countermeasure. The following section presents the details of our proposed architecture and countermeasure.

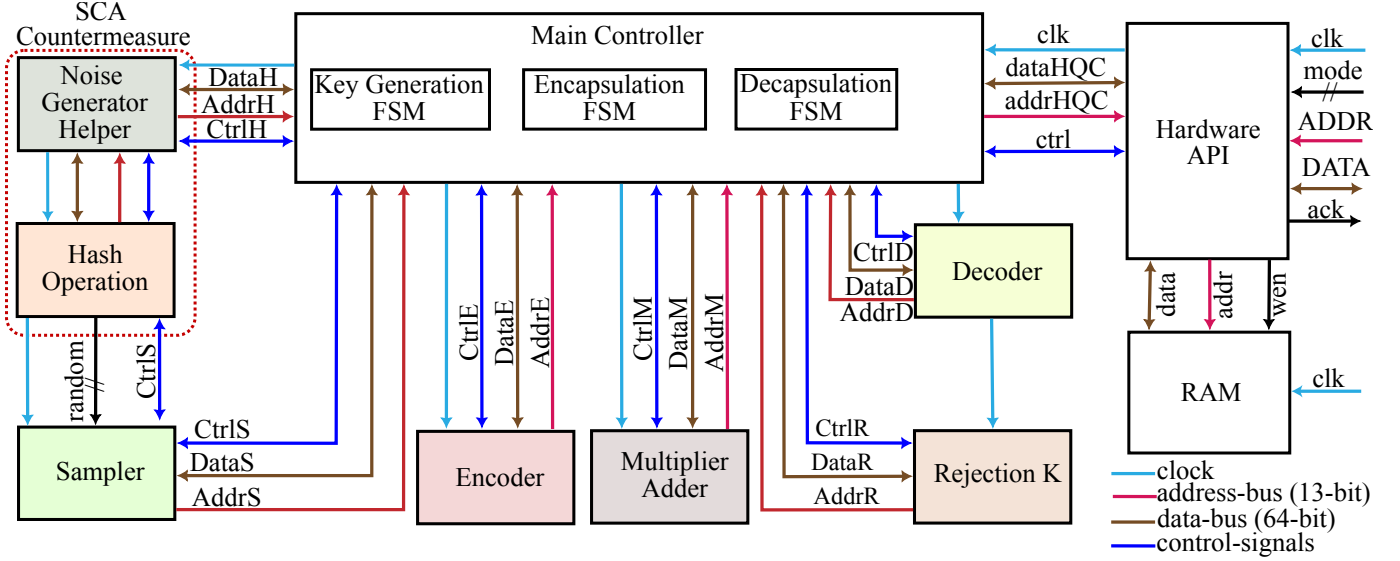


Fig. 1: Proposed HQC Hardware Architecture With Low-Cost Side-Channel Attack Countermeasure.

### III. PROPOSED HARDWARE ARCHITECTURE

To reduce side-channel leakage risk and meet area constraints in low-area applications, we adopted the hardware-and-memory-ressharing approach. Additionally, to maintain scalability across all parameter sets, we added dedicated hardware modules that take the parameters as input, process the HQC algorithms, and generate the desired output. By using single-port RAM to store intermediate values, hardware resource sharing, dedicated hardware address mapping, and reusing the KECCAK core to generate noise, we designed a low-area, power SCA-protected architecture. Fig. 1 shows the proposed design's hardware architecture with power SCA protection.

#### A. Hardware Modules

To design the scalable hardware, we identified the operations that can be computed with hardware modules, designed hardware modules that accept parameter inputs, and interconnected them with the central controller. The central controller generates the desired constants parameters based on the parameters sets. The different hardware modules used in the proposed hardware architecture are given below:

1) *Hardware API*: The algorithm requires desired inputs before processing, such as seed, public key, secret key, and ciphertext. These data need to be transferred to the core's internal RAM. The hardware API uses a multiplexer to connect the external address and data buses to the internal address and data buses, allowing the desired data to be transferred. We used the custom hardware API to speed up function verification, with an option to migrate to a standard interface like AXI. By using the low-cost, UART-based custom hardware API, we reduce the significant effort required for function verification after FPGA implementation.

2) *Main Controller*: The main controller module connects all the modules with a central interconnection with the shared BRAM and the finite state machine (FSM) scheduler. This

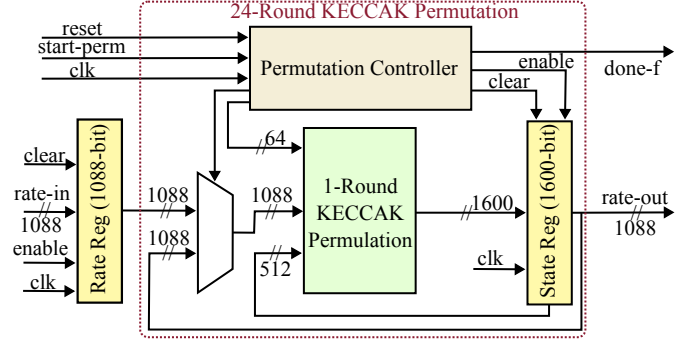


Fig. 2: KECCAK function-f architecture for hash function module design for HQC hardware implementation.

module also generates all the constants required for the different parameter sets listed in TABLE I. By separating the main controller from the other modules and keeping it as the central module, we made it easier to centralize configuration for updating the hardware with the different parameter sets, and this also allows us to reuse the pre-designed hardware like SHA-3 hardware, with minor changes, resulting in a significant reduction in the time required to design the overall hardware.

3) *Hash Function*: The HQC algorithm uses the KECCAK SHAKE-XOF to sample random vectors across all three algorithms. It also uses the SHA-3 operation to compute the ciphertext and secret key. The proposed design uses the KECCAK permutation module from the official KECCAK hardware implementation. This permutation module has a 1600-bit input-output port, and it computes one permutation per clock cycle. This module serves as the base for designing SHAKE-XOF and SHA modules, as well as the proposed noise generator for SCA protection.

As shown in Fig. 2, the permutation module takes input from the rate register, feeds the remaining bytes from the state register back to the permutation circuit, and uses the

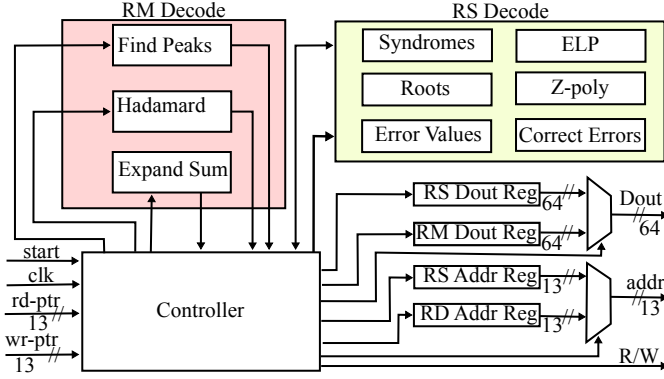


Fig. 3: Concatenated Reed-Muller and Reed Solomon Hardware Architecture.

permutation controller to enable or reset the state register. By using the additional input state register designed using the 64-bit rotation shift register, we designed this module to accept and process the variable-length message digest.

4) *Sparse Polynomial Sampler*: To sample the sparse polynomial across all three main algorithms, we designed a sampler that takes input directly from the hash function module, enabling partial parallelism by avoiding writing and reading random numbers to generate the sparse polynomial indices. To maintain full compatibility with the official HQC specification, the sampler incorporates three distinct operating modes: fixed-weight sampling, uniform random sampling, and dense sampling, allowing a single, hardware-efficient module to serve all sampling operations across the cryptosystem.

5) *Polynomial Multiplier Adder*: The HQC multiplication operation is performed for polynomials. This operation is one of the most time-consuming operations. Its optimization can lead to a substantial reduction in the overall execution time. The official software implementation uses the Karatsuba multiplier to compute the large polynomial of size  $n$ . However, HQC uses both sparse and dense polynomials in the multiplication operation, and using only dense multipliers can increase the number of clock cycles. Even though a sparse polynomial multiplier can increase multiplication speed, it can make the multiplier data-dependent and a non-constant-time operation if not carefully designed, potentially exposing timing side-channel vulnerabilities.

We designed the sparse multiplier, which takes the index of the sparse polynomial rather than the sparse input, thereby eliminating the need for a data-dependent comparison to determine whether the inputs are zero and skip the operation. To make it constant time, we decoupled multiplication and reduction operation so that multiplication takes the same time regardless of the sparse polynomial index.

6) *RS-RM Encoder*: The HQC algorithm uses Reed-Muller and Reed-Solomon encoders to generate the ciphertext. The operations of both encoders are concatenated: the message is first encoded with the Reed-Solomon encoder, then with the Reed-Muller encoder. In our proposed design, both encoders are designed separately, and the operations are concatenated by the controller so that the encoder module generates the encoded message for the next encryption operation.

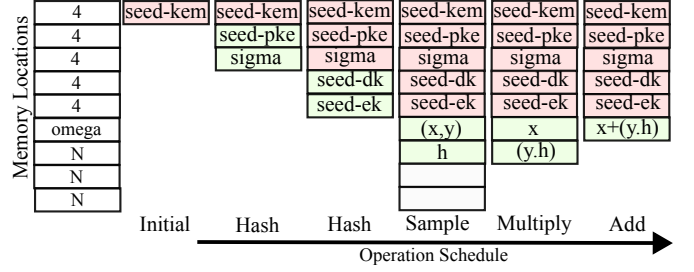


Fig. 4: Keygen Scheduling and Memory Mapping.

The RS-Encoder takes the message bytes and generates the codeword for the RM encoder. It requires the RS constant polynomials, finite-field multiplier, and ex-or operations. Since HQC uses a small RS encoder, the constants are directly implemented in the logic gates. Also, the finite field size is small, so we implemented the multiplier using a carry-less array multiplier, which requires only XOR chains. The HQC uses the repeated RM-Encoder, which is small and requires only shift and XOR operations to encode the codeword output by the RS encoder.

7) *RM-RS Decoder*: The proposed RM decoder design is based on the HQC reference software implementation, which uses find peaks, the Hadamard transform, and expand-and-sum operations to decode the encoded message. All the modules are designed separately and connected to the controller, which generates an enable/read-write signal and an address to locate and write data.

The proposed RS decoder is also based on the HQC reference software implementation; it consists of six operations: compute syndromes, compute the error locator polynomial and its roots, compute the Z-poly, find error values, and correct errors. We use behavioral modeling to describe the RM decoder design, enabling us to model the hardware directly from the algorithm. Additionally, we replaced the FFT-based root computation with the Chien search algorithm because the FFT-based computation increases the complexity of designing and modifying the root computation module with a fixed memory structure.

8) *Rejection K*: To perform the final rejection-key operation during decapsulation, we designed a separate module to format the data per the HQC specification, then compare the ciphertext with the recomputed ciphertext so the hash-function module can generate the final shared secret key.

## B. Scheduling and Memory Mapping (SMM)

Because PQC algorithms require significantly more memory than current public-key cryptosystems, reducing the memory footprint is critical for low-area hardware, as the memory needed to store intermediate values can exceed the hardware core's capacity. Therefore, careful memory planning can significantly reduce the on-chip memory footprint. We use a 64-bit data bus width to balance area and performance requirements, and we use a fixed memory mapping for each parameter, along with a memory-sharing method, to reduce the overall BRAM requirement. The details of the memory mapping are given as:

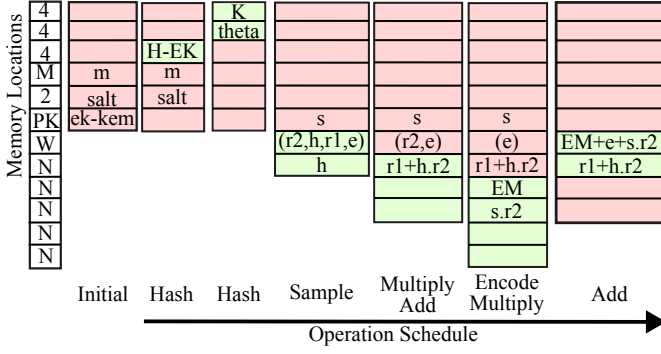


Fig. 5: Encap Scheduling and Memory Mapping.

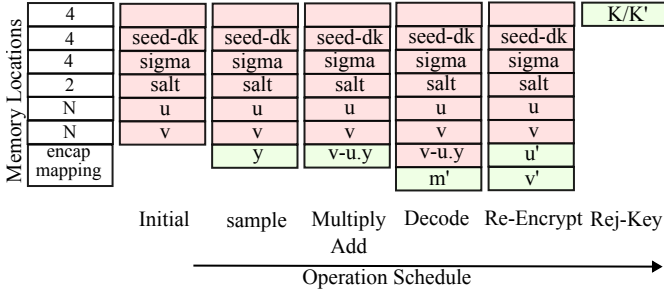


Fig. 6: Decap Scheduling and Memory Mapping.

1) *Keygen SMM*: The keygen module requires the seed input so that it can expand it and generate the required encryption and decryption key. It consumes only 4 memory locations. The hashing operation expands the key, which consumes an additional 16 locations. The sampler module uses  $\omega$  locations to store the sparse polynomial indices,  $N((n/64)+1)$  locations to store the dense polynomial, and the next  $2N$  locations are used by the multiplier to store the temporary output.

Overall, the design uses a maximum of 2854 BRAM memory locations across all three security levels during key generation operation. Fig. 4 illustrates the memory mapping and scheduling operation.

2) *Encap SMM*: The encapsulation module requires the message, salt, and public key inputs. The message uses 4 memory locations, the salt uses 2 memory locations, and the public key uses  $(4 + N)$  locations, which contain the seed and the vector  $s$ . The sampler module uses  $\omega$  locations to store the sparse polynomial indices,  $N = ((n/64) + 1)$  locations to store the dense polynomial, and the next  $4N$  locations are used by the encoder and the multiplier to store the temporary outputs and partial results of the ciphertext  $(u, v)$ . Overall, the design uses up to 5428 BRAM memory locations across all three security levels during encapsulation. Fig. 5 illustrates the memory mapping and scheduling operation.

3) *Decap SMM*: The decapsulation module requires the decryption key, sigma, salt, and the ciphertext. It reuses the encapsulation address mapping to decode and compute the message. Once the message bytes are recovered, it uses the same encapsulation address mapping to re-encrypt and generate the ciphertext  $(u'v')$ . Except for the memory locations required for encapsulation, it requires 1816 locations. Overall,

the design uses up to 7044 BRAM memory locations across all three security levels during decapsulation. Fig. 6 illustrates the memory mapping and scheduling operation.

### C. Proposed SCA Countermeasure

For the timing SCA countermeasure, we implemented the hardware to run in constant time. To achieve this, we avoided all data-dependent operations; for example, we implemented a constant-time sparse polynomial multiplier instead of a data-dependent one. Furthermore, we implemented the mod operation in the sampler using Barrett reduction.

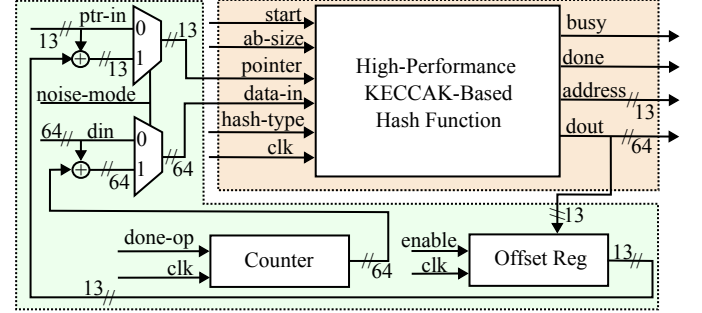


Fig. 7: Proposed architecture for noise generation using KECCAK hardware.

The proposed power side-channel countermeasure exploits structural microarchitectural features to defeat modern power-based attacks without dedicated hardware overhead. The defense relies on three core architectural mechanisms:

1) *Power Obfuscation via High-Entropy Diffusion*: The KECCAK hardware core operates on a 1,600-bit internal state using dense, wide-state permutations  $(\theta, \rho, \pi, \chi, \iota)$ . Activating the KECCAK core in parallel with HQC decoding forces massive multi-bit switching activity on every clock edge. Due to KECCAK's near-ideal avalanche effect ( $\sim 50\%$  flip probability per cycle), it injects high-amplitude dynamic power noise. This drastically reduces the global Signal-to-Noise Ratio (SNR), preventing single-trace attack algorithms [7] and Soft-Information Set Decoding (Soft-ISD) matrices [8] from achieving the high-fidelity prior probabilities required for statistical convergence.

2) *State Register Randomization*: By reseeding the KECCAK core with a unique random state prior to each decapsulation run, the dynamic power profile varies non-deterministically from cycle to cycle within a single execution trace. This non-stationary background noise mitigates Horizontal Side-Channel Attacks (HSCA) [7] by eliminating stable temporal baselines. Furthermore, the non-deterministic power floor degrades consistent hardware signatures that passive, ciphertext-independent profiling models rely on [9].

3) *Architectural Parallelism and Temporal Collapse*: Unlike software implementations that execute decoding sequentially on CPU microcontrollers, our FPGA architecture leverages fully parallelized pipeline structures for Reed-Solomon (RS) syndrome computation and polynomial arithmetic. Simultaneous execution collapses sequential Points of Interest (POIs) into compressed temporal windows, preventing

an adversary from isolating individual secret-dependent state transitions.

Having established the microarchitectural implementation and the low-cost countermeasure, we next evaluate the proposed architecture's security resilience.

#### IV. SECURITY ANALYSIS

This section describes the methodologies used to evaluate side-channel leakage in the proposed hardware implementation, specifically the HQC decoder, and to validate the proposed countermeasure. The evaluation covers both pre-silicon dynamic power analysis and post-silicon empirical measurements on the target FPGA platform. We formally assess side-channel leakage suppression using Welch's t-test-based TVLA under fixed-versus-random trace testing.

##### A. Threat Model

We assume a physical adversary with access to the target hardware device and its external communication interface. The attacker can execute Chosen-Ciphertext Attacks (CCA) by feeding arbitrary ciphertexts into the hardware API. Concurrently, the adversary performs passive non-invasive Side-Channel Analysis (SCA) by capturing power consumption traces along the primary power supply rail. To isolate evaluation of our microarchitectural countermeasure, we consider physical invasive attacks (e.g., direct probing, scan-chain extraction, or memory readout) out of scope.

##### B. Pre-Silicon Leakage Analysis

Our pre-silicon analysis utilizes behavioral simulation to identify data-dependent operations within the Finite State Machine (FSM). To reduce leakage, we modify these operations to be constant-time and data-independent. Building on this foundation, we implemented the design on a Kintex-7 FPGA and used the AMD Vivado Power Analysis tool to estimate the dynamic power consumption of various modules, including our proposed noise generator at a 100 MHz clock frequency. Finally, using these estimates, we calculated the Signal-to-Noise Ratio (SNR) for each module, as detailed in Table II.

##### C. Theoretical Defense Analysis

1) *Attack by Goy et al.*: [7] **Defeated**: KECCAK core injects high-entropy noise directly into this execution window. Parallel, randomized bit-flipping masks the power signature of the iterative GF multiplication operation, thereby erasing the predictable point-of-interest (POI) loop patterns.

2) *Attack by Dong et al.*: [8] **Defeated**: High-entropy parallel bit-flipping from the KECCAK core introduces massive amplitude white noise. This drops the Signal-to-Noise Ratio (SNR) near zero, preventing the Soft-ISD algorithm from gathering the high-fidelity soft information required to seed the MDPC decoding matrix.

3) *Attack by Lai et al.*: [9] **Defeated**: Re-seeding the KECCAK core with a unique random state per execution destroys the static physical profile of the decapsulation routine. The passive footprint changes completely every run, rendering ciphertext-independent profiling impossible.

TABLE II: Estimated Dynamic Power Consumption and SNR Of Modules used in HQC Design

| Module                                                            | LUT  | FF   | Clock MHz | Power ( $P_d$ ) mW | SNR ( $P_d/P_n$ ) |
|-------------------------------------------------------------------|------|------|-----------|--------------------|-------------------|
| RM-Decoder                                                        | 820  | 626  | 100       | 30                 | 0.093             |
| RS-Decoder                                                        | 2162 | 1009 | 100       | 3                  | 0.009             |
| RS-Encoder                                                        | 316  | 89   | 100       | 41                 | 0.127             |
| RM-Encoder                                                        | 239  | 296  | 100       | 24                 | 0.070             |
| Multiplier-Adder                                                  | 959  | 634  | 100       | 40                 | 0.124             |
| Hash Function Estimated Dynamic Power @ 100MHz ( $P_n$ ) = 321mW. |      |      |           |                    |                   |

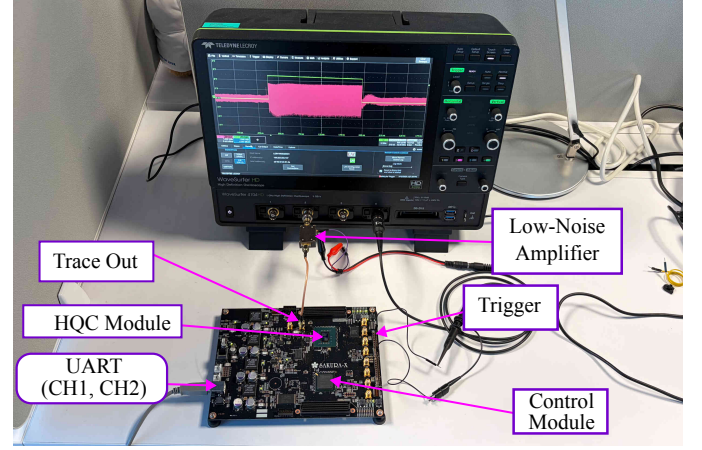


Fig. 8: Experimental Setup For HQC Hardware Design SCA Evaluation.

4) *Attack by Jeon et al.*: [11] **Defeated**: The KECCAK core runs concurrently throughout the entire decapsulation window, flooding the power lines with 1600-bit-wide chaotic switching current. This drops the SNR near zero, completely masking the data-dependent signatures of the re-encryption arithmetic blocks.

5) *Attack by Velek et al.*: [6] **Defeated**: Parallel architecture removes the need for sequential software lookup table loops. Additionally, the KECCAK noise core floods this initial window with chaotic bit-flipping, reducing the SNR to the point where fine-grained, data-dependent SPA signatures are completely buried.

##### D. Post-Silicon Leakage Analysis

For post-silicon validation using Welch's t-test (TVLA), we implemented the proposed HQC architecture on the SAKURA-X standard SCA evaluation platform. Fig. 8 illustrates the complete experimental measurement setup. The top-level HQC core is instantiated on the Kintex-7 target FPGA (XC7K160T). To capture high-fidelity power traces, an inline low-noise amplifier (LNA) conditions the signal from the onboard power measurement SMA connector, while a dedicated GPIO trigger line precisely frames the sensitive decoding execution window during trace acquisition.

The UART is used as a hardware API to connect to the HQC hardware core, allowing us to send desired inputs to the HQC core's BRAM and trigger the hardware to start or restart the desired operation. To minimize external noise, the control FPGA (Spartan-6) generates the UART module clock and connects it to the Crypto FPGA via the interconnects.

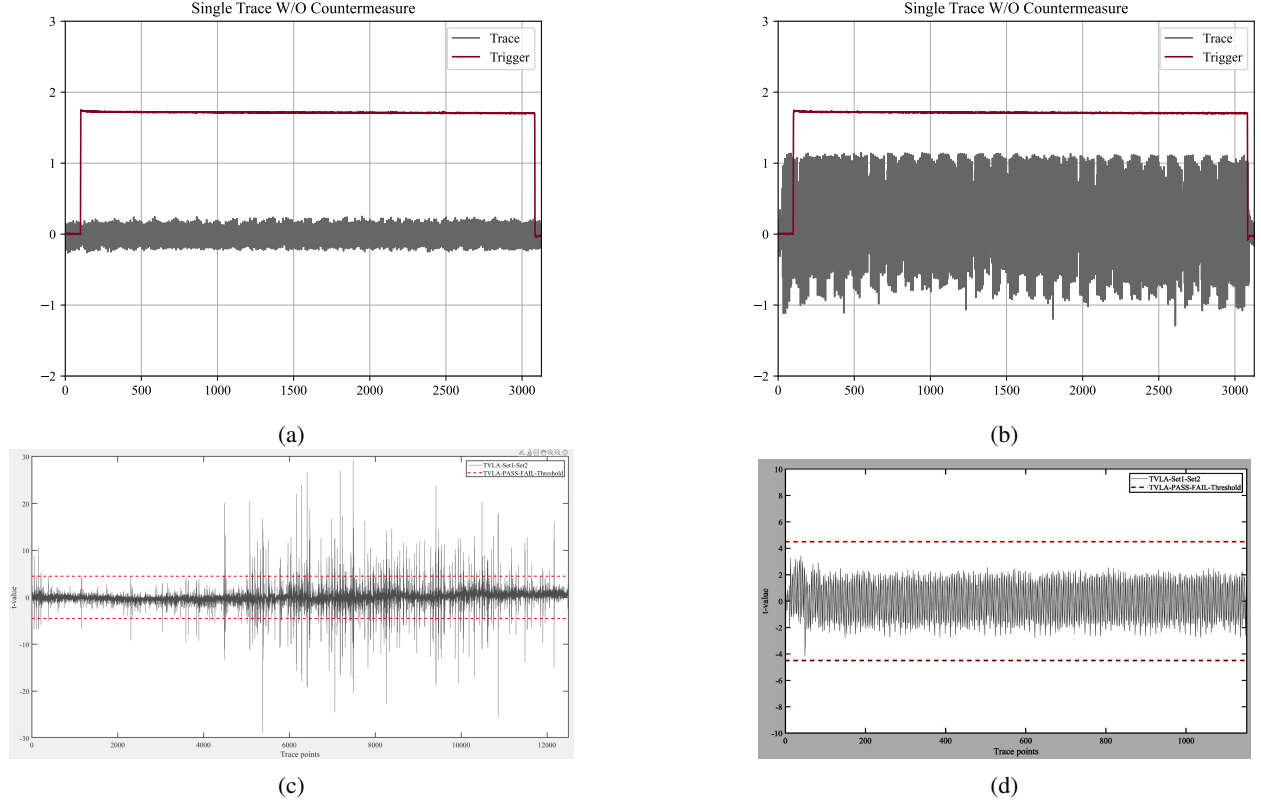


Fig. 9: Comparison of power traces and TVLA of RM-Decoder under different conditions: (a) Countermeasure is turned off, (b) Countermeasure is turned on, (c) TVLA under SCA disabled, and (d) TVLA under SCA enabled.

### E. TVLA Validation

Before trace collection, the hardware API is tested with Known Answer Tests (KATs) to verify that the hardware produces the correct output, confirming that the implementation works as expected and that the design contains no hidden logic minimization.

For the TVLA evaluation, fixed ciphertexts are sent to the hardware via the hardware API, and 40,000 traces are collected. The ciphertexts are read back at fixed intervals to check whether the inputs change due to glitches, which is common with asynchronous communication hardware. After collecting the fixed ciphertext traces, we randomly change the ciphertexts and send them to the hardware to collect traces with random inputs. The t-test results are then plotted as shown in Figure 9.

When the countermeasure is disabled, significant side-channel leakage during the decoding operation is observed, with the absolute t-value exceeding the standard threshold ( $|t| > 4.5$ ) and peaking at +30. In contrast, enabling the proposed KECCAK-based dynamic noise generator suppresses statistical power correlations across the entire execution trace, maintaining ( $|t| < 4.5$ ) throughout the sensitive decoding phase and confirming the physical security of the proposed countermeasure.

## V. RESULTS

This section presents the synthesis results, performance analysis, comparison with the state-of-the-art designs, and

functional correctness validation, including the constant-time validation methodology.

### A. Synthesis Results

The Verilog HDL source code was synthesized for an Artix-7 FPGA using AMD Vivado 2025 with the default synthesis constraints. TABLE IV shows the synthesis results. The Hash function module occupies the most area due to the high-performance KECCAK core. By using the sparse polynomial multiplier, we significantly reduced the multiplier's area. To balance area and performance, we used single-index processing instead of multiple indices, keeping overall area low while leveraging sparse polynomial multiplication.

### B. Performance Analysis

Although we used the sparse polynomial multiplier, it remains the primary bottleneck because of the dense polynomial's large size. To further improve speed, increasing the datapath size and the BRAM data bus can significantly reduce execution time. Aside from this operation, key generation is bottlenecked by fixed-weight sampling, which was recently added to the HQC reference software implementation. Adding this sampling forces index computation to be sequential because each index must be checked for duplicates.

### C. Comparison with the state-of-the-art Designs

TABLE III compares our proposed design with the state-of-the-art hardware implementation of HQC. Compared with

TABLE III: Comparison of Proposed Design With The State-of-the-art Designs.

| Design                                          | SCA Protection | LUT   | FF    | DSP | BRAM | Freq. | Cycles $10^3$ | time ( $\mu$ s) | Cycles   | time ( $\mu$ s) | Cycles    | time ( $\mu$ s) |
|-------------------------------------------------|----------------|-------|-------|-----|------|-------|---------------|-----------------|----------|-----------------|-----------|-----------------|
|                                                 |                |       |       |     | 36Kb | MHz   | Keygen        |                 | Encap    |                 | Decap     |                 |
| [16]                                            | ×              | 33701 | 21806 | 0   | 148  | -     | 3108.737      | -               | 6302.661 | -               | 11095.034 | -               |
| [13]                                            | ×              | 26561 | 13636 | 0   | 28   | 143   | 5.557         | 39              | 11.440   | 82              | 18.304    | 128             |
| [13]*                                           | ×              | 18310 | 10989 | 0   | 27.5 | 151   | 4.530         | 30              | 11.939   | 79              | 17.969    | 119             |
| [10]*                                           | ×              | 15214 | 7293  | 8   | 24   | 178   | 16.020        | 90              | 23.140   | 130             | 37.380    | 210             |
| Proposed HQC Design Without SCA Countermeasure  |                |       |       |     |      |       |               |                 |          |                 |           |                 |
| HQC-128                                         | ×              | 11449 | 6707  | 4   | 16   | 139   | 93.442        | 672.245         | 211.819  | 1523.878        | 414.802   | 2984.187        |
| HQC-192                                         | ×              | 11449 | 6707  | 4   | 16   | 139   | 271.070       | 1950.144        | 632.555  | 4550.755        | 1043.357  | 7506.165        |
| HQC-256                                         | ×              | 11449 | 6707  | 4   | 16   | 139   | 549.451       | 3952.885        | 1279.065 | 9201.906        | 2107.811  | 15164.108       |
| Proposed HQC-128 Design With SCA Countermeasure |                |       |       |     |      |       |               |                 |          |                 |           |                 |
| HQC-128P                                        | ✓              | 11489 | 6785  | 4   | 16   | 139   | 93.442        | 672.245         | 211.819  | 1523.878        | 414.864   | 2984.633        |

\* Considered maximum resource utilization for fair comparison.

TABLE IV: Breakdown of FPGA hardware resource usage by modules after post-implementation on an Artix-7 FPGA.

| Module                 | LUT          | FF          | DSP      | BRAM      |
|------------------------|--------------|-------------|----------|-----------|
| Hash Function          | 4993         | 2747        | 0        | 0         |
| Multiplier-Adder       | 959          | 634         | 0        | 0         |
| Encoder                | 555          | 385         | 0        | 0         |
| Decoder                | 2982         | 1635        | 0        | 0         |
| Sampler                | 728          | 333         | 4        | 0         |
| Rejection K            | 474          | 347         | 0        | 0         |
| Noise Generator Helper | 40           | 78          | 0        | 0         |
| Main Controller        | 675          | 626         | 0        | 0         |
| Hardware API           | 83           | 0           | 0        | 16        |
| <b>Total</b>           | <b>11489</b> | <b>6785</b> | <b>4</b> | <b>16</b> |

the lowest-area designs, the proposed design uses 24.75% fewer LUTs, 8% fewer FFs, 50% fewer DSPs, and 33.3% fewer BRAMs. Although we reduce area, execution time increases. For key generation, clock cycles are  $4.6\times$  higher, encapsulation is  $7.7\times$  higher, and decapsulation is  $9.7\times$  higher than the lowest-area design. The primary factors contributing to this increased clock cycles are discussed below:

1) *Updated HQC Specification Compliance*: Recent algorithmic and reference software implementation updates, such as updates to the fixed-weight sampler, introduced additional computational steps. To meet the latest specification and ensure protocol compliance, these updates inherently added extra cycles to the overall execution.

2) *Architectural Choice*: To achieve the smallest possible hardware footprint, we prioritized single-port RAM over dual-port or multi-port RAM. While dual-port RAMs allow parallel read/write to overcome the read-write bottleneck and reduce cycle count, they add significant silicon and routing area overhead. By restricting our design to single-port RAM, we achieved substantial BRAM utilization. However, the memory read-write bottleneck directly contributed to the observed clock cycle count.

3) *Intentional Decoupling Operations*: Rather than parallelizing or pipelining interdependent sub-operations, we intentionally decoupled major processing in hardware because it can generate overlapping power signatures, including combined power and EM side-channel leakage; this flattens peak dynamic power consumption and reduces joint side-channel information leakage, at the cost of additional execution time.

4) *Intentional Sequential Operation*: A crucial factor in our execution-time trade-off is the serialized subroutines in the decoder operation, since decoding is the primary target

for side-channel analysis in HQC. This reduces peak dynamic power consumption per cycle, increasing the proposed noise generator's effectiveness in suppressing noise at the cost of a higher observed clock-cycle count.

In constrained hardware accelerators, memory footprint is the largest area consumer because PQC algorithms require large amounts of memory, and multi-port RAM configurations consume significant area and contribute to routing congestion. Compared with traditional hardware-based hiding or masking countermeasures, which require additional dedicated hardware such as heavy noise generators that typically significantly increase area, or masking-only hardware, which can double or triple it, our proposed countermeasure increases area by less than 0.5%. By adopting our architectural choices and operation scheduling, we achieved an optimal balance among security, area, and performance trade-offs.

#### D. Constant-Time Validation

Since constant-time execution is required to mitigate side-channel attacks, we ensured that the operations are strictly data-independent. To validate the constant-time behavior of the hardware implementation, we use both simulation-based validation and FPGA implementation. Our validation techniques include initializing memory states with random numbers and setting uninitialized memory states to all don't-cares or all zeros. Then we run the simulation and the FPGA hardware and observe the time or clock cycles required to complete the operation. We observe that the simulation produces the overall done signal at the expected number of clock cycles or time, regardless of the BRAM's initial states or previously stored values.

#### E. Functional Verification Through NIST KATs

We functionally verified the proposed hardware implementation using the NIST KATs API from the official software implementation. We first generated the KATs, converted them to the desired memory-map format, and then verified them across all three security levels using the automated verification scripts. To speed up the KAT verification process, which is bottle-necked by simulation, we also used the physical FPGA implementation to verify the KATs.

## VI. CONCLUSION AND FUTURE WORK

In this paper, we presented an area-efficient, side-channel attack-protected architecture of the HQC algorithm. The proposed design supports key generation, encapsulation, and decapsulation for NIST category HQC-128, HQC-196, and HQC-256. We leveraged the reusable KECCAK hardware core to generate random noise and protect the sensitive variable during decoding. In the future, We'll investigate the side-channel leakage in other parts of the operation and use the proposed method to protect the secret-dependent variable from side-channel leakage. We'll also explore fault attacks on the HQC cryptosystem and investigate low-cost countermeasures.

## REFERENCES

- [1] Philippe Gaborit, Carlos Aguilar-Melchor, Nicolas Aragon, and Others, "Hamming Quasi-Cyclic (HQC)," 2025. [Online]. Available: [https://pqc-hqc.org/doc/hqc\\_specifications\\_2025\\_08\\_22.pdf](https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf)
- [2] 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. [Online]. Available: <https://doi.org/10.1137/S0097539795293172>
- [3] K. Xagawa, A. Ito, R. Ueno, J. Takahashi, and N. Homma, "Fault-Injection Attacks Against NIST's Post-Quantum Cryptography Round 3 KEM Candidates," in *Advances in Cryptology – ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 33–61. [Online]. Available: [https://doi.org/10.1007/978-3-030-92075-3\\_2](https://doi.org/10.1007/978-3-030-92075-3_2)
- [4] T. B. Paiva, P. Ravi, D. Jap, and Others, "Et tu, Brute? Side-Channel Assisted Chosen Ciphertext Attacks Using Valid Ciphertexts on HQC KEM," in *Post-Quantum Cryptography*. Cham: Springer Nature Switzerland, 2025, pp. 294–321.
- [5] Guillaume Goy, "Estimating the strength of horizontal correlation attacks in the hamming weight leakage model: A side-channel analysis on HQC KEM," in *WCC 2022*. CEA Scientific production in open Access, 2022. [Online]. Available: <https://cea.hal.science/cea-04176652/>
- [6] Pavel Velek, Tomáš Rabas, and Jiří Buček, "Simple Power Analysis of Polynomial Multiplication in HQC." [Online]. Available: <https://arxiv.org/abs/2601.07634>
- [7] G. Goy, J. Maillard, P. Gaborit, and A. Loiseau, "Single trace HQC shared key recovery with SASCA," 2023, published: Cryptology ePrint Archive, Paper 2023/1590. [Online]. Available: <https://eprint.iacr.org/2023/1590>
- [8] H. Dong, Q. Guo, and D. Nabokov, "Single-Trace Key Recovery Attacks on HQC Using Valid and Invalid Ciphertexts," 2025, published: Cryptology ePrint Archive, Paper 2025/1987. [Online]. Available: <https://eprint.iacr.org/2025/1987>
- [9] Z. Lai, R. Zhang, Z. Zhang, J. Hermelink, M. Schwarz, V.-T. Pham, and U. Parampalli, "You Only Decapsulate Once: Ciphertext-Independent Single-Trace Passive Side-Channel Attacks on HQC," 2025, published: Cryptology ePrint Archive, Paper 2025/2162. [Online]. Available: <https://eprint.iacr.org/2025/2162>
- [10] S. Deshpande, C. Xu, M. Nawaz, K. Nawaz, and J. Szefer, "Fast and Efficient Hardware Implementation of HQC," in *Selected Areas in Cryptography – SAC 2023: 30th International Conference, Fredericton, Canada, August 14–18, 2023, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2023, pp. 297–321. [Online]. Available: [https://doi.org/10.1007/978-3-031-53368-6\\_15](https://doi.org/10.1007/978-3-031-53368-6_15)
- [11] J. Jeon, Y. Park, J. Lee, S. Lee, D. Kim, and Y.-S. Kim, "Codeword Masking Can Be Harmful Under Replay Attacks on HQC," 2026, published: Cryptology ePrint Archive, Paper 2026/071. [Online]. Available: <https://eprint.iacr.org/2026/071>
- [12] F. Antognazza, A. Barengi, G. Pelosi, and R. Susella, "A High Efficiency Hardware Design for the Post-Quantum KEM HQC," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2024, pp. 431–441.
- [13] F. Antognazza, A. Barengi, and G. Pelosi, "An Efficient and Unified RTL Accelerator Design for HQC-128, HQC-192, and HQC-256," *IEEE Transactions on Computers*, vol. 74, no. 7, pp. 2306–2320, 2025.
- [14] C. Li, S. Song, J. Tian, Z. Wang, and C. K. Koc, "An Efficient Hardware Design for Fast Implementation of HQC," in *2023 IEEE 36th International System-on-Chip Conference (SOCC)*, 2023, pp. 1–6.
- [15] M. Schöffel, J. Feldmann, and N. Wehn, "Code-based Cryptography in IoT: A HW/SW Co-Design of HQC," in *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*, 2022, pp. 1–7.
- [16] A. Dolmeta, S. Di Matteo, E. Valea, M. Carmona, A. Loiseau, M. Martina, and G. Masera, "TYRCA: A RISC-V Tightly-Coupled Accelerator for Code-Based Cryptography," in *2025 Design, Automation & Test in Europe Conference (DATE)*, 2025, pp. 1–7.