HAKE: Efficient Hardware Accelerator for Key Generation of Post-Quantum Signature Scheme PERK

Brendan Funk, Tianyou Bao, Loïc Bidoux, and Jiafeng Xie

***Abstract*—The rapid progress in quantum computing has sparked a new wave of cryptosystem innovation, namely, the development of cryptographic schemes that are resistant to quantum attacks, known as Post-Quantum Cryptography (PQC). Notably, the National Institute of Standards and Technology (NIST) has already initiated the PQC standardization process with several algorithms selected. Meanwhile, an additional round of digital signature scheme competition is on-going. Following the standardization efforts, many investigations in the field have gradually switched to the implementation side (especially on the hardware platform aspect). This paper follows this trend by delivering an efficient Hardware Accelerator for Key Generation of the digital signature scheme PERK (HAKE), which is one of the promising candidates in the NIST additional round of digital signature scheme standardization. Apart from that, we have followed the PERK's recent update to design two versions of Key Generation accelerators, one based on the previous PERK specification and another based on the newly released specification. Overall, we have conducted three major efforts to obtain the proposed accelerators. (i) We have broken down the Key Generation process of PERK into three distinct components through detailed algorithmic analysis, and meanwhile, we have proposed innovative methodologies to reduce these components' hardware design complexities. (ii) We have developed dedicated hardware microarchitectures for these components to construct the Key Generation accelerator (HAKE). (iii) We have conducted detailed implementation and comparison to showcase the efficiency of the proposed accelerator. For instance, it is shown that the proposed accelerator (following the previous PERK specification) is found to be 14.3× faster than the software implemented one and less area-time complexities than other recent NIST-selected SPHINCS$^+$ hardware accelerations. Overall, our design is highly efficient and configurable, and it is the first hardware accelerator for Key Generation of PERK, to the best of our knowledge. This research will be beneficial for the ongoing NIST PQC standardization and hardware acceleration for related schemes, and attract many follow up works in the field.**

***Index Terms*—Digital signature scheme, FPGA, hardware accelerator, Key Generation, PQC, PERK.**

I. Introduction

POST-quantum cryptography (PQC) research has drawn significant attention recently as it aims to address vulnerabilities posed by advancing quantum computational capabilities [1]. The National Institute of Standards and Technology (NIST) has initiated the PQC standardization process and has selected five algorithms: KYBER [2] and HQC [3] for key encapsulation mechanisms, and Dilithium [4], FALCON [5], and SPHINCS$^+$ [6] for digital signature schemes [7]. Meanwhile, an additional round of digital signature scheme competition is still ongoing, and algorithms employing Multi-Party Computation (MPC)-in-the-Head paradigms have emerged as promising candidates [8], which represent competitive options beyond SPHINCS$^+$ [9].

Recent Research. Following the PQC standardization process, recent research efforts have shifted toward hardware design for PQC [10]–[13]. At the time of writing, however, dedicated studies examining PERK hardware design remain unexplored. Nevertheless, there exist a few other (software-based) works that deserve mentioning: Bettaieb et al. have developed a memory footprint reduction technique for PERK [14]; they also addressed permutation operations to improve signature sizes and computational efficiency of PERK [15]; Gazzoni Filho et al. demonstrated PERK's constant-time implementation with runtime code generation methodologies [16].

Motivation. Indeed, the need to build novel hardware accelerators for PERK is at an all-time high. In this paper, we follow the recent research trend to design a novel hardware accelerator for Key Generation of PERK. We chose to build a hardware accelerator for PERK's Key Generation, but not for Signature Generation or Verification, for two main reasons. First, PERK's Key Generation process requires specialized mathematical tasks that are not duplicated in the Signing or Verification phases, and these tasks will be explained later in the paper. Second, there is a standardized separation according to NIST for each portion of post-quantum algorithms. FIPS 203, 204, and 205 treat KeyGen, Signing, and Verification as separate algorithms with distinct APIs (Application Programming Interfaces) to simplify implementation and validation [17]. Thus, a PERK Key Generation only accelerator is a modular contribution that aligns with this standardized split. In fact, the existing literature also reports similar works such as Signature Verification for SLH-DSA and FALCON [18]–[20] as well as Key Generation for Kyber [21].

Following this direction, in this paper, we aim to design an efficient hardware accelerator for Key Generation phase of PERK. In particular, we are interested in comparing its performances with respect to existing hardware works for SPHINCS$^+$ [22]–[24] and other MPC-in-the-head schemes like Syndrome-Decoding-in-the-Head (SDitH) [25].

Major Challenges. Following the NIST PQC standardiza-

Manuscript submitted April 29, 2026, revised XXXX. Corresponding author: J. Xie.

B. Funk, T. Bao, and J. Xie are with the Department of Electrical and Computer Engineering, Villanova University, Villanova, PA, 19085 USA (e-mail: {bfunk,tbao, jiafeng.xie}@villanova.edu).

L. Bidoux is with the Technology Innovation Institute (TII), Abu Dhabi, UAE (e-mail: loic.bidoux@tii.ae).

tion process, PERK specification has been updated recently [26]. As a result, the hardware acceleration of PERK Key Generation encounters several primary computational challenges.

- For the previous specification (version 2.0.0): (i) its Key Generation involves a huge size of matrix-vector multiplication operations that require up to 34,000 field multiplications, and (ii) storage overhead inherent to the MPC-in-the-head paradigm, necessitating substantial memory allocation for signature and key operations.
- The recent specification (version 2.1.1/2.2.0) has a relatively simpler algorithmic procedure compared with the previous one. Nevertheless, it still involves: (a) between 2,300 and 7,000 field multiplications that require 121 AND operations and 230 XOR operations per multiplication, and (b) similarly significant storage overhead from matrix expansion, kernel basis representation and permutation structures.

Contributions. To address the aforementioned challenges, this work presents an efficient **H**ardware **A**ccelerator for **K**ey Generation of P**E**RK (HAKE). We follow the PERK specifications [26], [27] to include two versions of HAKE, and the major contributions include:

- **Algorithmic Analysis and Novel Design Methodologies.** We have conducted detailed algorithmic analysis to break down the Key Generation of PERK into three major components and proposed innovative methodologies to reduce their further hardware design complexities.
- **Dedicated Hardware Accelerator Design.** We have designed dedicated hardware microarchitectures for these major components, along with detailed data flow control to build the accelerator HAKE (for both versions of Key Generation of PERK).
- **Comprehensive Evaluation and Comparison.** We have carried out a systematic performance evaluation across all PERK security categories for HAKE, significantly outperforming the existing microcontroller implementation and recent hardware designs of SPHINCS$^+$ and SDitH.

To the authors' best knowledge, this is the **first** hardware acceleration for PERK, and the results (see Section 5) establish its competitiveness in the NIST additional round of digital signature scheme standardization.

The remaining parts of this paper are structured as follows: Section II presents preliminaries; Section III describes the algorithmic analysis and design methodologies; Section IV presents the dedicated accelerator building; Section V presents the comprehensive evaluation and discussion; and Section VI concludes this work.

II. Preliminaries

PERK is a post-quantum digital signature scheme based on the hardness of the Permuted Kernel Problem (PKP) [26], [28]. The scheme builds on a Zero-Knowledge Proof of Knowledge of a PKP solution that is converted into a signature scheme using the Fiat-Shamir transform.

We follow the recent trend in the PQC field that more attentions have been made on the implementation side, especially on the hardware acceleration [13], to develop an efficient accelerator for PERK's Key Generation. As mentioned above, i) PERK's Key Generation involves unique operations that are not duplicated in other operational phases; ii) there is a standardized separation according to NIST for each operational phase of PQC. With this background, we decided to build a Key Generation only accelerator for PERK in this paper.

Recently, the PERK team has updated the specification reducing the signature sizes of the scheme and improving its security. In this paper, we refer PERK-I as the previous version (version 2.0.0) [27], [29] and PERK-II as the new one (version 2.2.0) [26], [28], and will cover both versions to present the accelerator HAKE.

Algorithm 1: Key Generation Procedure for PERK-I

Input : None (generates public/private keypair)
Output: Public key `pk`, Private key `sk`

// Step 1: Sample randomness
$sk.\texttt{seed} \leftarrow \texttt{random_bytes}(\lambda)$;
$pk.\texttt{seed} \leftarrow \texttt{random_bytes}(\lambda)$;
// Step 2: Generate secret permutation
$sk.\pi \leftarrow \texttt{PRG}(sk.\texttt{seed})$ from S_n;
// Step 3: Generate public parameters
$(pk.H,\ pk.x[1 \dots t]) \leftarrow$ $\texttt{PRG}(pk.\texttt{seed})$ from $\mathbb{F}_q^{m \times n} \times (\mathbb{F}_q^n)^t$;
// Step 4: Compute permuted syndromes
for $j \leftarrow 1$ **to** t **do**
  $pk.y[j] \leftarrow pk.H\pi[pk.x[j]]$;
end
// Step 5: Output keypair
$(sk,\ pk) \leftarrow (sk.\texttt{seed},\ (pk.\texttt{seed},\ pk.y[1 \dots t]))$;
return `SUCCESS`

Key Generation of PERK. Key Generation in PERK achieves compactness and efficiency through deterministic expansion of short seeds into complete keys using pseudorandom generators (PRGs). The major distinction between PERK-I and PERK-II lies in the dimensionality of their underlying algebraic structures: PERK-II employs a single-instance PKP (Permuted Kernel Problem) based on one secret vector, while PERK-I generalizes this to a multi-instance variant utilizing multiple linearly independent secret vectors.

• Key Generation in PERK-I. PERK-I operates with multiple secret vectors, accommodating t linearly independent secret vectors forming the foundation of its algebraic structure. The secret key consists of a uniformly random seed $sk_{\text{seed}} \in \{0,1\}^\lambda$, while the public key derives from an independent seed $pk_{\text{seed}} \in \{0,1\}^\lambda$. The entire Key Generation for PERK-I is shown in Algorithm 1.

• Key Generation in PERK-II. PERK-II simplifies this framework through reduction to a classical single-vector PKP instance parameterized by (q, m, n) and security parameter λ. The Key Generation procedure of PERK-II can be seen in Algorithm 2. Interested readers can refer to the original documentation [26], [28] for more details.

Parameter Sets. Apart from the above mentioned differences in algorithmic procedures, the two versions of Key

Algorithm 2: Key Generation Algorithm for PERK-II

Input : Public parameters $\texttt{paramPKP} = (q, m, n)$ and security parameter λ
Output: Public key pk, Private key sk

// Step 1: Sample randomness
1 $H_{\text{seed}} \leftarrow \texttt{random_bytes}(\lambda)$;
2 $\texttt{ker_seed} \leftarrow \texttt{random_bytes}(\lambda)$;
3 $\texttt{perm_seed} \leftarrow \texttt{random_bytes}(\lambda)$;
// Step 2: Construct PKP instance
4 $M \leftarrow \texttt{ExpandMatrixM}(H_{\text{seed}})$ // $M \in \mathbb{F}_q^{m\times(n-m)}$;
5 $H \leftarrow [I_m \mid M]$ // $H \in \mathbb{F}_q^{m\times n}$;
6 $x' \leftarrow \texttt{ExpandKernelVector}(\texttt{ker_seed}, H)$ // $x' \in \ker(H)$;
7 $\pi \leftarrow \texttt{ExpandPermutation}(\texttt{perm_seed})$ // $\pi \in S_n$;
8 $x \leftarrow \pi^{-1}(x')$;
// Step 3: Output keypair
9 $(pk,\ sk) \leftarrow ((H_{\text{seed}},\ x),\ \texttt{perm_seed})$;
10 **return** SUCCESS

Generation for PERK also involve different parameter sets (which we have listed in Table I). The following algorithmic analysis and accelerator design are based on the parameters.

TABLE I: PERK-I & -II Key Generation Parameter Sets

SL	Version	PRG	Matrix (H)	Vector Size
128	PERK-I	SHAKE128	79 x 35	79
192	PERK-I	SHAKE256	112 x 54	112
256	PERK-I	SHAKE256	146 x 75	146
128	PERK-II	SHAKE128	37 x 27	64
192	PERK-II	SHAKE256	49 x 43	92
256	PERK-II	SHAKE256	59 x 59	118

SL: Security Level (NIST standard).

III. Algorithmic Analysis and Design Methodologies

In this section, we first analyze the major operations in the Key Generation of the two versions of PERK, and then propose novel design methodologies for their efficient hardware implementation. Following this principle, we break down the Key Generation of both versions of PERK into components based on the features of their inherent algorithmic procedures.

A. Algorithmic Analysis for PERK-I

Following the algorithmic procedure in Algorithm 1, PERK-I's Key Generation can be decomposed into three major components: Value Generation, Sorting, and Arithmetic Computation. Each component addresses distinct computational requirements within the Key Generation process, enabling modular optimization and independent performance analysis.

• Component-I: Value Generation (Lines 1-4 and 8). This component perform deterministic pseudorandom expansion from compact seeds: Lines 1-2 sample uniform random seeds $sk_{\text{seed}}, pk_{\text{seed}} \in \{0,1\}^{\lambda}$; Line 3 generates the secret permutation $\pi \in S_n$ from sk_{seed}; Lines 4 generate the public matrix $H \in \mathbb{F}_q^{m\times n}$ and vectors $\{x_j\}_{j=1}^{t} \in \mathbb{F}_q^n$ from pk_{seed}; and Line 8 initializes storage for output vectors $\{y_j\}_{j=1}^{t}$. This component is memory and bandwidth-intensive, relying on PRG expansion (e.g., SHAKE) with minimal arithmetic cost.

• Component-II: Sorting (Line 6). This component performs a reordering of vector elements according to the secret permutation mapping π. This operation applies the permutation π to each value x_j, producing a permuted vector $\text{tmp} = \pi(x_j)$, where the relationship between indices is defined as $\text{tmp}_{\pi(k)} = x_{j,k}$ for all positions k. Effectively, this mapping creates a bijective rearrangement of the original vector entries, ensuring that every element is relocated to a unique position without loss or duplication of data. The transformation occurs only at the index level, remapping positions rather than modifying the values themselves. As a result, the transformation introduces data-dependent memory access patterns that are driven by the secret permutation π, increasing implementation complexity and reducing predictability in access behaviors. This step helps "hide" the structural relationships within the vector prior to arithmetic computation, helping to mask correlations and prevent data leakage.

• Component-III: Arithmetic Computation (Lines 5-7). This component computes syndrome vectors $y_j = H_\pi \cdot x_j \in \mathbb{F}_q^m$ for each $j \in [1, t]$ performing finite-field operations over $m \times n$ entries. This section is classified as the algorithm's dominant arithmetic workload due to several factors: (a) it executes element-wise multiplications and accumulations across a substantial $m \times n$ matrix space, requiring significant resources to maintain timing constraints; it produces a new object y_i derived from the public data and secret domain inputs, which represents a shift from simple expansion or reordering; (c) it marks the boundary from pseudo-random generation and structural manipulation and the final stage of arithmetic execution. As a result, Line 6 encapsulates the core computational engine of the algorithm, orchestrating arithmetic operations that define the syndrome generation process. This stage ensures the security of the scheme is realized through the transformation of secret-space inputs into public-space syndromes that are infeasible to invert.

B. Algorithmic Analysis for PERK-II

The procedure in Algorithm 2 follows a single-vector architecture, emphasizing kernel mathematics over the multi-vector processing in PERK-1. This procedure can be decomposed into three primary functional components: Value Generation, Arithmetic Computation, and Permutation Application.

• Component-I: Value Generation (Lines 1-5, Algorithm 2). This component establishes the cryptographic workspace by deriving structured objects from several entropy sources. The algorithm samples three independent λ-bit seeds: H_{seed}, $perm_{seed}$, and ker_{seed}. These are the roots upon which the keypair's unpredictability depends. Using H_{seed}, the algorithm expands the public matrix $H \in \mathbb{F}_q^{m\times n}$ via $\text{ExpandMatrixM}(H_{\text{seed}})$. The resulting public matrix H is instantiated in Reduced Row Echelon Form (RREF) as $H = [I_m|M]$. This defines defines the linear constraints and helps facilitate kernel basis extraction.

• Component-II: Permutation Application (Lines 7-8). This component obscures the values through index manipulation rather than arithmetic. The seed $perm_{seed}$ is expanded into a secret permutation $\pi \in S_n$ via EXPANDPERMUTATION. In hardware-efficient implementations, this typically utilizes a constant-time sorting network to prevent timing side-channels. The algorithm computes and applies the inverse mapping π^{-1} to x', producing the vector x where $x[j] = x'[\pi^{-1}(j)]$. This rearrangement is the primary layer of defense, hiding the relationship between the public matrix and the kernel vector without modifying the underlying values.

• Component III: Arithmetic Computation (Line 6). The arithmetic component begins the algebraic construction, generating the secret. Using ker_{seed} and the previously expanded H, EXPANDKERNELVECTOR derives a kernel basis $k_1, ..., k_{n-m}$ from the rows of $(M^T|I_{n-m})$. The structured vector $x' \in ker(H)$ is produced via the linear combination $x' = \sum_{i=1}^{n-m} c_i \cdot k_i$, where $c_i \in \mathbb{F}_q$ is derived from the kernel seed. This satisfies the public constraint $Hx' = 0$.

Note there are additional expansion functions used in PERK-II:

(i) EXPANDMATRIXM$(H_{\text{seed}}) \rightarrow \mathbb{F}_q^{m\times(n-m)}$: Sample M uniformly via PRG with H_{seed} (domain separation `0x00`); set $H = [I_m \mid M]$.

(ii) EXPANDKERNELVECTOR$(\text{ker}_{\text{seed}}, H) \rightarrow \text{ker}(H)$: Derive kernel basis $\{k_i\}$ from rows of $(M^T \mid I_{n-m})$; sample $c_i \in \mathbb{F}_q$ via PRG with ker_{seed} (domain separation `0x10`); output $x' = \sum_i c_i k_i$.

(iii) EXPANDPERMUTATION$(\text{perm}_{\text{seed}}) \rightarrow S_n$: Set $v = (0, \ldots, n-1)$; sample $e = (e_0, \ldots, e_{n-1}) \in (\mathbb{F}_2^{16})^n$ via PRG with $\text{perm}_{\text{seed}}$ (domain separation `0x20`); form p from high bits of e and low bits of v; sort p in constant time and extract π from low bits.

C. Design Challenges and Novel Methodologies for PERK-I

• Challenge related to Component-I. The direct implementation of Value Generation relies on sequential seeding and system-level entropy sources. Hardware implementation requires continuous seed expansion for generating matrix elements and vector computers, which leads to memory bottlenecks. Variable-length hash outputs conflict with fixed-width register register architectures as well.

Design Methodology. We propose an integrated Value Generator with an on-chip entropy pool coupled to a KECCAK component for seed, permuation, matrix and vector generation, and hashing. Along with the KECCAK and Value Generator, the design will employ domain separators for correct memory address computation and will write the results to BRAM.

• Challenge related to Component-II. PERK-I's direct implementation takes advantage of a constant time sorting algorithm library which is unavailable to the hardware implementation so easily.

Design Methodology. We thus propose to use a new Fisher–Yates shuffle with a precomputed lookup table (for modular reductions) coupled with an odd-even brick transposition brick sorting component to eliminate sequential dependencies and variable-width operations within sorting, while maintaining constant-time operation. This will also allows us to sort the vector data in place to ensure no underlying values are changed, just indexes.

• Challenge related to Component-III. Matrix–vector multiplication in direct implementations is much easier due to memory management and scaling. In direct implementations, matrix rows can be loaded into buffers to mask memory latency and they are equipped to handle irregular memory access patterns. In hardware implementations, standard BRAM primitives offer a limited number of access ports and calculate memory addresses in real time. Also, scaling hardware implementation multipliers up from 2 to 4 to 8 can increase datapath complexity exponentially.

Design Methodology. We propose to use a configurable arithmetic component with parallel Karatsuba multipliers (and Barrett reduction units), an accumulator for post-Barret result addition, and a multi-ported BRAM for fast processing.

D. Design Challenges and Novel Methodologies for PERK-II

• Challenge related to Component-I. SHAKE expansion in direct implementations create completed output streams directly into buffers before element extraction. Hardware implementations are constrained by field element boundaries misaligning with fixed-width datapaths. Domain separators introduce context-switching overhea; while variable-lenth output requirements conflict with fixed memory port widths. Higher security levels increase absorpotion cycles and squeezing iterations for equivalent output lengths.

Design Methodology. We propose a unified KECCAK core that can handle all SHAKE operations and value generations, while eliminating datapath misalignment and context-switching penalties. Our KECCAK core will output multiple values at once to maintain an acceptable output speed.

• Challenge related to Component-II. Similar to PERK-I, PERK-II's sorting network also involves dynamic changes in size. Hardware design requires generating and inverting the entire permutation, and applying it demands indirect memory addressing access.

Design Methodology. We propose a hardware methodology for generating cryptographically secure permutations using a modified Fisher-Yates algorithm [30] with bounded sampling iterations. The permutation array is initialized sequentially, and each shuffle iteration performs a swap between a current index and a random index derived in a fixed, deterministic manner. Conditional branches and data-dependent memory accesses are eliminated, and all iterations consume the same number of clock cycles, ensuring constant-time execution. The inverse permutation π^{-1} is computed on demand from π to avoid storing a separate copy, reducing memory overhead. The resulting permutation is output sequentially over a fixed number of cycles, mitigating timing-based side-channel attacks while remaining scalable to varying permutation sizes.

• Challenges related to Component-III. PERK-II's RREF matrix construction requires Gaussian elimination and dynamic memory allocation for row operations. Hardware state machines have to serialize read-modify-write cycles with iteration counts depending on matrix rank. Division operations in

hardware implementations require large amounts of resources. Hardware must explicitly instantiate multiplier circuits along with accumulation and adder units to complete full matrix multiplications. Another challenge is that EXPANDKERNELVECTOR computes $x' = \sum_i c_i k_i$, requiring $(n - m)$ scalar–vector products over $\mathbb{F}_{2^{11}}$. As a result, the direct hardware implementation of this component will require several parallel multipliers processed simultaneously to produce the result (the potential involved hardware resource usage will be high).

Design Methodology. We propose a three-stage pipelined multiplier over GF(2^{11}) using the irreducible polynomial $x^{11} + x^2 + 1$. The first stage generates 11 parallel partial products by conditionally shifting the multiplicand according to each multiplier bit. The second stage combines these partial products via XOR operations to form the unreduced 21-bit polynomial product. In the third stage, modular reduction is performed iteratively using XOR to fold high-order terms into the lower 11 bits. Each stage registers its outputs with validity signals, allowing one multiplication per clock cycle once the pipeline is filled, enabling high-throughput operation for HAKE-2.

IV. HAKE: ACCELERATOR BUILDING DETAILS

This section follows the proposed design methodologies given in Section III to present complete design details for the proposed HAKE, covering dedicated component microarchitecture design to upper-level accelerator layout.

A. HAKE-1 (Based on PERK-I)

The top-level diagram of HAKE-1 is shown in Fig. 1, where the whole accelerator consists of three components (connecting Section 3): (i) Value Generation; (ii) Sorting; and (iii) Arithmetic Core.

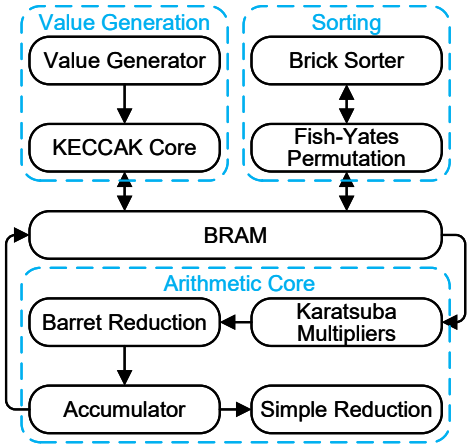

Fig. 1: Overview diagram of HAKE-1.

Value Generation. The Value Generator component produces all structured random values needed for signature operations, as illustrated in Fig. 2. The module collects entropy from several hardware unit sources, i.e., a 16-bit ring oscillator chain with XOR feedback taps at positions 1 and 7, an 8-bit metastability flip-flop chain that toggles and self-XORs each cycle, a 32-bit jitter counter that free-runs and contributes its lower 8 bits as thermal noise, and a bank of four 16-bit LFSRs (Linear Feedback Shift Registers) with different feedback accumulating them via XOR mixing into a 512-bit entropy pool over multiple clock cycles. The diversity of these entropy sources ensures that no single point of failure or predictable bias can compromise the quality of the collected randomness, and their contribution to the shared pool promotes entropy accumulation. Once 512 bits of entropy are collected, a KECCAK wrapper processes different 64-bit chunks of the entropy pool to deterministically generate two master seeds (primary and secret), invoking the hash core multiple times to build up each seed across several 64-bit output chunks. A Finite State Machine (FSM) then progresses through generation phases using these seeds: key generation (public and private keys), permutation generation (a new Fisher-Yates shuffle, see Algorithm 3, using modulo table lookups), matrix generation (row-by-row), and three vector generation phases. The sequential structure enforces a strict ordering of operations, preventing any phase from consuming values that have not yet been fully derived. The design employs domain separation by XORing the secret seed with unique combinations of hash counters and element indices for each generation phase. Domain separation makes sure that values produced for different cryptographic contexts are independent from one another, eliminating cross-phase leakage. All generated cryptographic materials are written to BRAM upon completion.

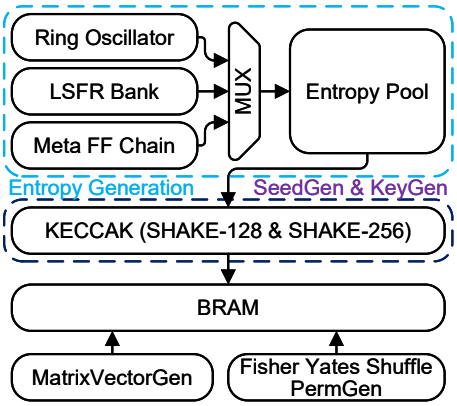

Fig. 2: Internal structure of the Value Generation component.

Algorithm 3: New Fisher-Yates Shuffle

Input : Unshuffled vector V, array A containing values [0, ..., vector length - 1]
Output: Shuffled vector

$cnt \leftarrow$ vector length $- 1$;
for $i \leftarrow 0$ **to** cnt **do**
  generate random 8-bit number a;
  $a \leftarrow a \bmod (78 - cnt)$;
  `// 78 is the highest "slot" a value can be shuffled into`
  swap $A[cnt]$ and $A[a]$;
end
return A;

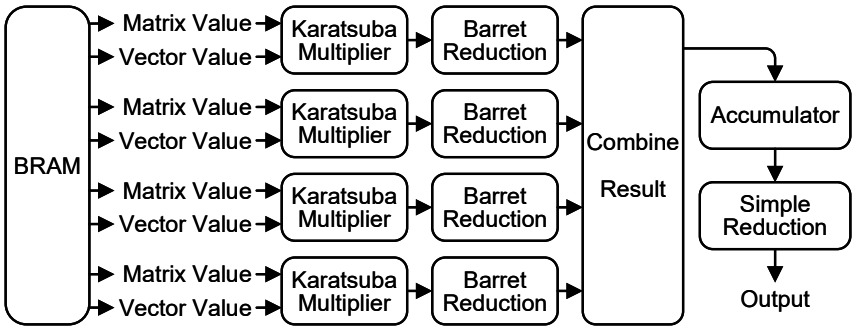

Fig. 3: Diagram of Arithmetic Core Data Flow.

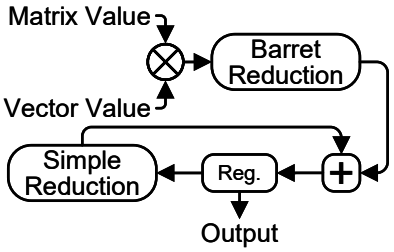

Fig. 4: Diagram of the Arithmetic Core.

Algorithm 4: Odd–Even Transposition Sort

Input : Unsorted array A of length n
Output: Sorted array A

$sorted \leftarrow False$;
while $\neg sorted$ **do**
  $sorted \leftarrow True$;
  // Even phase: compare (0,1), (2,3), ...
  for $i \leftarrow 0$ **to** $n-2$ *step* 2 **do**
    if $A[i] > A[i+1]$ **then**
      swap $A[i]$ and $A[i+1]$;
      $sorted \leftarrow False$;
    end
  end
  // Odd phase: compare (1,2), (3,4), ...
  for $i \leftarrow 1$ **to** $n-2$ *step* 2 **do**
    if $A[i] > A[i+1]$ **then**
      swap $A[i]$ and $A[i+1]$;
      $sorted \leftarrow False$;
    end
  end
end
return A;

Sorting. The Sorting Component reorders cryptographic vector data according to a permutation sequence generated by the new Fisher-Yates shuffle (Algorithm 3) by bridging BRAM storage to an odd-even transposition brick sorting accelerator shown in Algorithm 4. A FSM progresses through stages: loading the permutation array (index mappings) from BRAM at address 0x064, loading one of three selectable vectors (starting at 0xB00, 0xB92, or 0xC24 based on a 2-bit selector), triggering the brick sorter with both arrays, and storing sorted results back to the original vector addresses. The 2-bit selector mechanism allows the same sorting hardware to serve all three vectors without replication, reducing resource utilization while maintaining the flexibility to independently process each vector as required. The module handles arrays of up to n elements with 7/8-bit permutation keys and 11-bit vector values (0-1020), packing data into concatenated bit vectors for the brick sorter input and unpacking sorted outputs for BRAM storage. This packing scheme allows the brick sorter to operate on wide parallel data words in a single pass, minimizing the number of memory accesses required and keeping the critical path latency predictable across varying array sizes. All vector data is sorted in place at the original locations, with the interface managing data movement and synchronization between shared memory and the sorting core.

Arithmetic Core. Figs. 3 and 4 show the Arithmetic Core component performing matrix-vector multiplication over $GF(1021)$ with configurable parallelism (2, 4, or 8 multipliers). A FSM progresses through stages: loading matrix and vector elements based on the parallel mode setting, performing parallel Karatsuba multiplications (splitting 10-bit inputs into 5×5 half-multiplications), applying Barrett reduction to each 20-bit product using a precomputed constant to obtain modular results, combining reduced results through summation, accumulating the sum with previous partial results for the current row, performing final simple modular reduction and storing completed row elements. The choice of Karatsuba multiplication over schoolbook multiplication reduces the number of partial products required for each 10-bit operand pair, which yields a meaningful reduction in combinational logic depth and contributes to a shorter critical path across each parallel channel. The Barrett reducer pipelines the reduction across multiple cycles, and the design handles remaining elements when the number of columns is not evenly divisible by the number of multipliers with a counter. This remainder-handling counter ensures correctness for arbitrary matrix dimensions without requiring padding of the input data or the addition of dummy computation cycles that would otherwise inflate latency. Each parallel channel has dedicated memory interfaces with address generation computing matrix addresses and vec-

tor addresses. The separation of address generation per channel avoids structural hazards that would arise from a shared address bus, allowing all active multipliers to issue memory requests simultaneously and sustain the full throughput of the parallel datapath. The module instantiates up to 8 parallel Karatsuba multipliers (can be reconfigured with 2, 4, or 8 multipliers) and Barrett reduction units, allowing the hardware to trade between throughput and resource utilization.

System-Level Setup. The control unit ensures that the BRAM is accessed by the active component based on the active process. Generally, the order of operations is Value Generation, Sorting, and then Arithmetic, with the Value Generator taking precedence. This ordering is not arbitrary but reflects the data dependencies inherent to PERK: the Sorting Component cannot begin reordering vectors until the permutation array and vector data have been fully written to BRAM by the Value Generator, and the Arithmetic Core cannot begin matrix-vector multiplication until the sorted vectors are at their final BRAM addresses. The Value Generartor acts as the root producer in a strict pipeline, with downstream components remaining idle and relinquishing BRAM access until the `done` signal from the preceding stage confirms that all material has been committed to memory.

B. HAKE-2 (Based on PERK-II)

Connecting Section 3, we follow the design methodologies introduced to design HAKE-2 (the architecture can be seen in Fig. 5), which also has: (i) Value Generator, (ii) Permutation Generator, and (iii) Arithmetic Core.

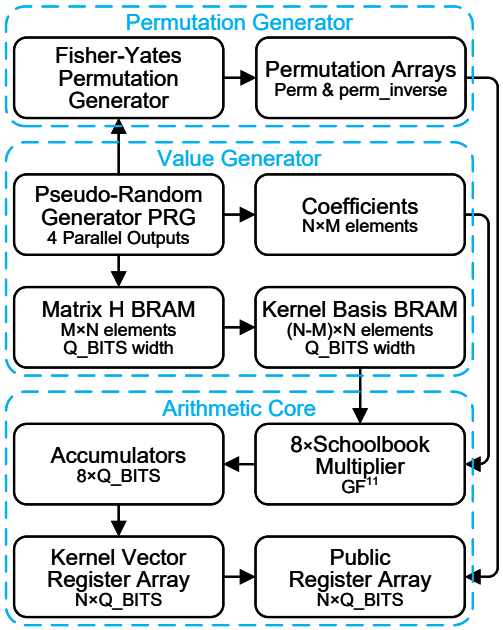

Fig. 5: Top-level diagram of the proposed HAKE-2 (Q_Bits is 11 bits based on the Q value and N is the length of the SVS).

Value Generator (PRG). The PERK PRG component generates field elements for $GF(2^{11})$ using either SHAKE128 or SHAKE256 as the underlying hash function. The selection between these two variants allows the module to support different security levels within the same hardware footprint, with SHAKE128 offering a 128-bit security target and SHAKE256 providing a 256-bit security target, both sharing the same underlying KECCAK permutation and differing only in their rate and capacity parameters. A finite state machine progresses through states: loading a 128-bit seed and 8-bit domain separator (17 bytes total) into input RAM, releasing the KECCAK wrapper from reset to trigger absorption and squeezing, waiting for the hash operation to complete via the `done` signal, collecting output data from output RAM into a buffer, and extracting groups of four 11-bit elements per cycle by shifting the buffer after each extraction. The design uses dual-port RAM arrays to interface with the KECCAK wrapper, with the hash function variant selected via a 2-bit `instance_type` signal. The module calculates the required output length as $\lceil (num_elements \times 11)/8 \rceil$ bytes before starting the hash, then extracts elements by loading 64-bit words from output RAM until the buffer contains at least 44 bits (four 11-bit elements), masking each element to the $GF(2^{11})$ range (0–2047), and shifting the buffer after each extraction. Pre-computing the required output length prior to initiating the hash avoids the need for on-the-fly length adjustments mid-absorption, simplifying the FSM control logic and ensuring that the KECCAK wrapper is configured with the correct squeeze length before it is released from reset. The bit-packing approach of extracting four 11-bit elements per 44-bit window is notably efficient, as it avoids byte-boundary alignment complications by treating the output stream as a flat bitfield and advancing the buffer pointer uniformly after each group extraction. Timing control includes stabilization delays after loading the seed and delays after hash completion to ensure proper RAM writes before extraction begins.

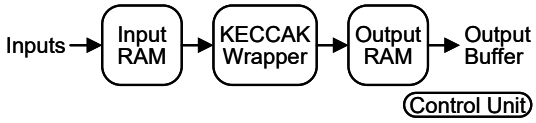

Fig. 6: Diagram of HAKE-2 Value Generator.

Arithmetic Core. Our Arithmetic Core performs multiplication over the Galois Field $GF(2^{11})$ using the modular reduction by the irreducible polynomial $x^{11} + x^2 + 1$ (Fig. 7). The design adopts a three-stage pipeline architecture. In the first stage, it generates 11 parallel partial products by conditionally shifting the multiplicand left based on each bit of the multiplier, creating terms that represent the polynomial product expansion. In the second stage, it combines all partial products through XOR operations to form the unreduced 21-bit polynomial product. The intermediate 21-bit width is a natural consequence of multiplying two degree-10 polynomials, whose product can reach degree 20, and retaining all bits at this stage ensures no information is discarded before the reduction step that follows. In the third stage, it performs modular reduction by systematically replacing each high-order term x^k (where $k \geq 11$) with its equivalent lower-order representation $x^{k-9} + x^{k-11}$, according to the reduction polynomial relationship $x^{11} = x^2 + 1$. This process is applied iteratively through XOR operations to fold the upper bits down until only

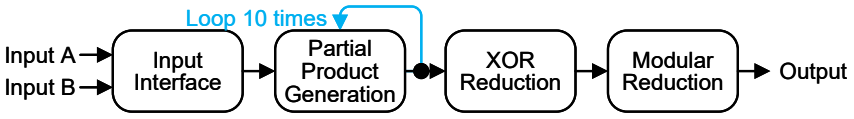

Fig. 7: Diagram of the Arithmetic Core.

an 11-bit result remains. Because each substitution introduces terms only at positions strictly below k, the folding process is guaranteed to terminate after processing all bits from position 20 down to position 11, producing a fully reduced result in a fixed and predictable number of XOR steps regardless of the input values. Each pipeline stage operates on registered data from the previous stage and maintains validity signals that propagate through the pipeline, allowing a continuous throughput of one multiplication per clock cycle once the pipeline is filled. Such throughput is essential for HAKE-2 to obtain high-performance operation.

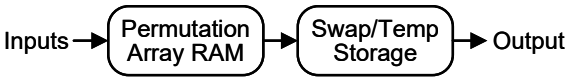

Fig. 8: Diagram of Permutation Generator.

Permutation Generator. Our Permutation Generator component (Fig. 8) executes the Fisher–Yates shuffle of [30] for generating random permutations using an external random source with constant-time index reduction to avoid modulo bias. The design employs an FSM that progresses through several stages: it first initializes an array with sequential values from 0 to $n-1$ (n is the length of the vector) and then iteratively performs the shuffle by starting from the last position and working backward. For each position, it requests a random byte from the external source and extracts the lower $PermBits$ to form a candidate swap index, clamping any value exceeding the current position to the current position itself in a single constant-time comparison, thereby avoiding the variable-latency retry loop of classical rejection sampling. The clamping strategy trades a slight, bounded non-uniformity at the top of each range for a fully predictable and data-independent execution time. Once a valid swap index is determined, a two-cycle swap operation reads both the current element and the target element into temporary registers simultaneously before writing them back in the following cycle, ensuring the exchange is always performed unconditionally even when the two indices coincide, which is itself a constant-time property that prevents observable branching on secret data. Once all $n-1$ swap operations are complete, the permuted array is packed sequentially into a flat output vector across n output cycles and signaled as valid, providing cryptographically sound random permutations suitable for HAKE-2 that requires unbiased index shuffling. The `busy` signal remains asserted throughout all FSM phases and is deasserted only upon entering the done state, giving the surrounding system a reliable occupancy indicator without requiring inspection of internal state.

System-Level Setup. The control unit monitors the BRAM and active components to ensure they use the BRAM at the correct time, based on the FSM state. The Value Generator accesses it first, followed by the Permutation Generator, then the Arithmetic Core, with the Value Generator taking the bulk of the operations. This ordering reflects the data dependencies that flow through the system: the Arithmetic core cannot begin its multiplications until the field elements produced by the PERK PRG component are in BRAM, and the Permutation generator cannot begin its Fisher-Yates shuffle until the array has been committed to memory. The PERK PRG component itself contributes to this initial phase by expanding the seed to produce the field elements that fill BRAM ahead of arithmetic processing.The `busy` and the `done` signals of each component allow transitions between ownerships and require no additional handshake logic.

V. Evaluation and Discussion

In this section, a comprehensive evaluation and comparison have been conducted to demonstrate the efficiency of the proposed HAKE-1 and HAKE-2.

Experimental Setup. The experimental setup is as follows: (i) the two accelerators are coded in VHDL (functionality verified through ModelSim) and implemented by the AMD Vivado 2022.2, Artix-7 xc7a200t device; (ii) the resource usage, including the number of LUTs, FFs, slices, DSPs, and BRAMs are reported along with maximum frequency (Fmax) and latency; (iii) we have used Equivalent Number of Slices (ENS) to calculate the Equivalent Area-Time Product (eATP) for a comprehensive calculation of the overall complexities (the calculation of ENS and eATP are shown at the footnote of Tables II and III); (iv) we have also obtained the performance results for both HAKEs under different multiplier configurations (see Table II). The Artix-7 xc7a200t was selected as the target device as it represents a mainstream, cost-accessible FPGA platform that is widely used as a benchmark device in the post-quantum cryptography hardware literature, allowing the reported results to be directly situated within the broader landscape of existing PQC accelerator designs.

Performance Evaluation and Discussion. One can see that both HAKE-1 and HAKE-2 obtain relatively better area-time complexities when the number of multipliers being executed in the Arithmetic Core increases, i.e., the eATP is the smallest when the number of multipliers is set to 8. Although the area usage of both HAKE-1 and HAKE-2 increases slightly, the latency of both accelerators drops significantly, leading to a higher overall efficiency when the number of multipliers reaches 8. Note that we do not recommend increasing the multiplier configuration beyond 8, since a large number of multipliers will only lead to a significant increase in area with disproportional gain in timing (large area may deteriorate frequency/latency). The three-stage pipelined Karatsuba structure of the Arithmetic Core means that adding multipliers beyond

TABLE II: HAKE-1 and HAKE-2 Performance based on Different Multiplier Configurations (Artix-7 Device)

Design	#Mults	SL	LUT/FF/Slice/DSP/BRAM	ENS[a]	Fmax (MHz)	CCs/Latency (K)/(ms)	eATP[b] (K)
HAKE-1	2	128	6,481/4,665/3,283/2/6	3,803	169	36.517/0.216	520
		192	8,362/4,610/4,258/2/11	5,423	169	73.242/0.433	1,243
		256	11,243/4,765/5,919/2/20	8,259	169	106.280/0.629	2,271
	4	128	7,815/5,168/4,144/4/6	4,403	169	21.774/0.128	385
		192	9,487/5,020/5,055/4/11	5,965	169	41.142/0.243	828
		256	12,174/5,117/6,654/4/20	8,741	169	61.405/0.363	1,485
	8	128	9,977/6,027/5,461/8/6	5,461	169	14.318/0.085	346
		192	11,763/5,885/7,052/8/11	7,052	169	25.407/0.151	679
		256	14,517/5,752/9,815/8/20	9,815	169	36.942/0.218	1,127
HAKE-2	2	128	10,112/7,933/4,436/0/4	4,449	170	14.982/0.088	310
		192	10,713/8746/6,231/7/7.5	6,231	170	26.652/0.157	705
		256	11,400/9,413/6,835/7/9	6,835	170	37.828/0.223	1,058
	4	128	10,101/7,933/4,446/0/4	4,446	170	14.390/0.085	299
		192	10,848/8,856/6,279/7/7.5	6,279	170	25.525/0.150	680
		256	11,557/9,560/6,893/7/9	6,893	170	36.088/0.212	1,018
	8	128	10,337/8,111/4,528/0/4	4,528	170	14.094/0.082	295
		192	11,142/9,112/6,384/7/7.5	6,043	170	24.962/0.147	632
		256	11,825/9,765/6,985/7/9	6,985	170	35.218/0.207	1,013

SL: Security level.
#Mults. refers to the number of multipliers used in the Arithmetic Core.
[a] Slices, DSPs and BRAMs are converted to ENS (Equivalent Number of Slices) using $\mathrm{ENS} = \mathrm{LUT}/4 + \mathrm{FF}/8 + \mathrm{DSP} \times 102.4 + \mathrm{BRAM} \times 232.4$, following [22].
[b]: eATP=ENS$\times$Latency.

8 also places increased pressure on the Barrett reduction units and the accumulation logic, which become bottlenecks that limit the frequency achievable by the synthesizer and erode the latency improvements that additional multipliers would otherwise provide.

Comparison Consideration. To further validate the efficiency of the proposed HAKE accelerators, we have conducted a comparison with the following considerations: (a) since there is no hardware design works for PERK, we just use the existing microcontroller implementation of PERK [31] (based on PERK-I) for comparison; (b) we also consider NIST's priority in the additional round of PQC digital signature scheme competition that the targeted candidate needs to have better performance than the NIST-selected SPHINCS$^+$ (we hence list some of the recent SPHINCS$^+$ hardware designs for comparison); (iii) we also include another MPC-in-the-head scheme SDitH's hardware design [25] in Table III (note [25] only reports the Key Generation module without KECCAK, we have implemented our HAKE-2 without KECCAK instance for a fair comparison).

Comparison Analysis. As demonstrated in Table III, our proposed HAKE-1 and HAKE-2 involve significantly better area-time complexities than the competing designs (at the same security level). Key information includes:

(i) When compared with the microcontroller implementation of [31] (based on PERK-I), the proposed HAKE-1 is at least 58.33x faster under the same security level setup (at most 94.88x faster). This substantial speedup stems from the fundamental architectural advantage of dedicated hardware over sequential software execution: operations that a microcontroller must perform iteratively across many clock cycles, such as the modular multiplications and permutation generation that dominate PERK's runtime, are handled in the HAKE designs through parallel datapaths and pipelined execution that amortize their cost across far fewer cycles.

(ii) When compared with the recent hardware acceleration of SPHINCS$^+$ (SLH-DSA [9]), we consider the fact that SPHINCS$^+$ is mostly built on the hash computations, and the major involved resource usage comes from the KECCAK core (for SHAKE256) and memory usage. Meanwhile, the Key Generation phase of SPHINCS$^+$ won't require most of the BRAMs, we calculate the existing designs' eATPs without BRAMs for a fair comparison. Again, as listed in Table III, the proposed HAKE-1 and HAKE-2 involve significantly less area-time complexities than these SPHINCS$^+$ works, e.g., HAKE-1 and HAKE-2 have at most 37.9% and 49.6% less eATP than the best existing design of [22], respectively.

(iii) When compared with another MPC-in-the-head signature scheme SDitH design [25] that is also being considered by the NIST additional round of PQC standardization, our HAKE-2 (without KECCAK) significantly outperforms the SDitH design (Key Generation module without SHAKE functions), i.e., 88.0%, 73.0%, and 82.9% less eATP than the GF256 variant with respect to the three security levels (compared with the GF251 variant of [25], 94.9%, 86.5%, and 89.9% less eATP, respectively). The superior performance presented underscores a promising status of PERK (if not the best) among all MPC-in-the-head schemes currently under consideration by NIST.

Other Discussion. The proposed accelerators for PERK overall are efficient in performance. While there exist other signature scheme designs, such as Dilithium [32] and FALCON [33], we don't list them here, following NIST's additional PQC standardization recommendation that the additional signature schemes be mainly better than SPHINCS$^+$ [8].

Future Works. Future work will include expanding this research into the Signature Generation and Verification parts of the PERK scheme. Beyond that, building everything together (with all three operational phases) into one unified accelerator

TABLE III: Comparison with Existing PERK Microcontroller Implementation and Other Signature Scheme Hardware Works

Design	Device	SL	LUT/FF/Slice/DSP/BRAM	ENS	Fmax (MHz)	CCs/Latency (K)/(ms)	eATP (K)
PERK-I							
HAKE-1	Artix-7	128	9,977/6,027/5,461/8/6	5,461	169	14.3/0.085	346
		192	11,763/5,885/7,052/8/11	7,052	169	25.4/0.151	679
		256	14,517/5,752/9,815/8/20	9,815	169	36.9/0.218	1,127
[31][a]	ARM Cortex M4	128	-	-	120	595/4.958	-
		192	-	-	120	1.422/11.850	-
		256	-	-	120	2.482/20.683	-
PERK-II							
HAKE-2	Artix-7	128	10,337/8,111/4,528/0/4	4,528	170	14.094/0.082	295
		192	11,142/9,112/6,384/7/7.5	6,384	170	24.962/0.147	682
		256	11,825/9,765/6,985/7/9	6,985	170	35.218/0.207	1,013
HAKE-2[b]	Artix-7	128	769/1,810/1,348/0/4	1,348	213	10.812/0.050	21
		192	969/1,830/2,214/0/7.5	2,214	213	21.131/0.099	47
		256	999/1,828/2,570/0/9	2,570	213	33.700/0.158	76
SPHINCS$^+$/SLH-DSA (Fast Signature Mode, KECCAK Version, eATP excludes BRAMs)							
[22]	Artix-7	128	29,410/14,090/10,043/0/4	10,043	180	14.7/0.082	744
		192				21.4/0.120	1,084
		256				56.6/0.316	2,866
[24]	HW/SW	128	8,605/-/9,588/0/32	-	100	176.6/1.76	3,799
		192				284.2/2.84	6,114
		256				815.6/8.15	17,546
[23]	ZCU-102	128	12,049/10,542/6,712/0/20.5	4,331	200	-/0.42	1,819
		192	13,275/11,581/7,148/0/20.5	4,766	200	-/0.59	2,812
		256	14,383/12,547/7,546/0/20.5	5,165	200	-/0.79	4,080
SDitH (Another NIST MPC-in-the-head scheme, NO KECCAK unit involved)							
[25][c]	Artix-7	128	1.8K/1.1K/632/0/8	2,447	238	55/0.231	565
		192	2.4K/1.4K/770/0/14.5	4,145	234	46/0.197	815
		256	2.5K/1.5K/856/0/24.5	6,506	230	84/0.365	2,376
[25][d]	Artix-7	128	2.3K/1.0K/799/16/9.5	4,546	196	57/0.291	1,322
		192	3.3K/1.2K/1,132/22/14.5	6,598	194	48/0.247	1,632
		256	3.4K/1.3K/1,163/22/25.5	9,192	197	86/0.437	4,013

Note: eATP=ENS$\times$Latency. Slices, DSPs and BRAMs are converted to ENS using $\mathrm{ENS} = \mathrm{LUT}/4 + \mathrm{FF}/8 + \mathrm{DSP} \times 102.4 + \mathrm{BRAM} \times 232.4$, following [22].
[a]: PERK-I-fast-3 variant evaluated on a Nucleo-L4R5ZI evaluation board.
[b]: HAKE-2 without KECCAK.
[c]: GF256 variant.
[d]: GF251 variant.

for PERK, as well as other MPC-in-the-Head algorithms, are all possible paths for future research.

VI. Conclusion

This work has presented a novel hardware accelerator design for the Key Generation phase of PERK, a promising post-quantum digital signature scheme candidate in the NIST additional standardization round. The overall efforts address the computational challenges inherent in PERK through proposed novel design methodologies along with dedicated component and accelerator-level building. The proposed accelerator (for both versions of PERK) demonstrates significant performance improvements over the competing designs. This is the first hardware work for PERK, and future research can focus on accelerating the entire scheme and side-channel analysis.

References

[1] P. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, (Santa Fe, NM, USA), pp. 124–134, IEEE, 1994-11, 1994.

[2] Kyber. https://pq-crystals.org/kyber/.

[3] HQC (Hamming Quasi-Cyclic). https://pqc-hqc.org/.

[4] Dilithium. https://pq-crystals.org/dilithium/.

[5] Falcon. https://falcon-sign.info/.

[6] SPHINCS+. https://sphincs.org/.

[7] I. T. L. Computer Security Division, "Round 4 Submissions - Post-Quantum Cryptography." https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions, Jan. 2017.

[8] I. T. L. Computer Security Division, "Round 2 Additional Signatures - Post-Quantum Cryptography: Additional Digital Signature Schemes." https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures, Oct. 2024.

[9] N. I. o. S. and Technology, "Stateless Hash-Based Digital Signature Standard," Tech. Rep. Federal Information Processing Standard (FIPS) 205, U.S. Department of Commerce, Aug. 2024.

[10] 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, pp. 747–758, Feb. 2023.

[11] Y. Lee, J. Youn, K. Nam, H. H. Jung, M. Cho, J. Na, J.-Y. Park, S. Jeon, B. G. Kang, H. Oh, and Y. Paek, "An Efficient Hardware/Software Co-Design for FALCON on Low-End Embedded Systems," *IEEE Access*, vol. 12, pp. 57947–57958, 2024.

[12] S. Deshpande, C. Xu, M. Nawan, 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), pp. 297–321, Springer-Verlag, Aug. 2023.

[13] J. Xie, W. Zhao, H. Lee, D. B. Roy, and X. Zhang, "Hardware circuits and systems design for post-quantum cryptography—a tutorial brief," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 71,

no. 3, pp. 1670–1676, 2024.
[14] S. Bettaieb, L. Bidoux, A. Budroni, M. Palumbi, and L. P. Perin, "Enabling PERK and other MPC-in-the-Head Signatures on Resource-Constrained Devices," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, pp. 84–109, Sept. 2024.
[15] S. Bettaieb, A. Budroni, M. Palumbi, and D. L. G. Filho, "Quasi-optimal Permutation Ranking and Applications to PERK," in *Progress in Cryptology - AFRICACRYPT 2024* (S. Vaudenay and C. Petit, eds.), (Cham), pp. 49–65, Springer Nature Switzerland, 2024.
[16] D. L. Gazzoni Filho, R. G. Flores e Silva, A. Budroni, M. Palumbi, and G. Adj, "Runtime Code Generation for Constant-Time Secret-Indexed Array Accesses: Applications to PERK and NTRU," in *Progress in Cryptology – LATINCRYPT 2025* (D. Escudero and I. Damgård, eds.), (Cham), pp. 450–487, Springer Nature Switzerland, Oct. 2025.
[17] Post-Quantum Cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography.
[18] H. Liu, L. Wu, L. Li, Y. Yang, and X. Zhang, "Design for hardware acceleration of signature verification in PQC stateless hash-based digital signature algorithm," in *2025 10th International Conference on Integrated Circuits and Microsystems (ICICM)*, pp. 937–942, IEEE, 2025.
[19] S. Coulon, J. Xiong, and J. Xie, "LEAF: Lightweight and efficient hardware accelerator for signature verification of Falcon," in *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2025.
[20] H. Hu, Y. Chu, G. Du, X. Wang, Y. Song, D. Zhang, and Z. Li, "A Falcon signature verification accelerator using area-efficient NTT architecture with simplified Barrett modular multiplier," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–12, 2025.
[21] Q. Liao and L. He, "Lightweight key generation circuits for crystals-kyber," in *2024 3rd International Conference on Electronics and Information Technology (EIT)*, pp. 209–213, 2024.
[22] T. Huang, J. Lu, D. Liu, Z. Luo, C. Cheng, A. Li, L. Chen, S. Yang, J. Zhang, and X. Li, "An Efficient and Reconfigurable Post-Quantum Crypto-Processor for SPHINCS+," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 72, pp. 2252–2262, May 2025.
[23] V. T. D. Le, A. K. Pham, H. L. Pham, T. H. Vu, and Y. Nakashima, "PESA: Power-Efficient SPHINCS+ Accelerator for Multi-Domain Security Applications on FPGA SoC," in *2024 Twelfth International Symposium on Computing and Networking (CANDAR)*, (Naha, Japan), pp. 231–237, IEEE, Nov. 2024.
[24] M.-J. O. Saarinen, "Accelerating SLH-DSA by Two Orders of Magnitude with a Single Hash Unit," in *Advances in Cryptology – CRYPTO 2024*, (Santa Barbara, CA, USA), pp. 276–304, Springer, Cham, 2024.
[25] S. Deshpande, J. Howe, J. Szefer, and D. Yue, "SDitH in Hardware," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, pp. 215–251, Mar. 2024.
[26] "PERK v2.2.0," 2025.
[27] "PERK v2.0.0," 2025.
[28] S. Bettaieb, L. Bidoux, P. Gaborit, and M. Kulkarni, "Modelings for generic PoK and Applications: Shorter SD and PKP based Signatures," *Cryptology ePrint Archive, report 2024/1668*, 2024.
[29] S. Bettaieb, L. Bidoux, V. Dyseryn, A. Esser, P. Gaborit, M. Kulkarni, and M. Palumbi, "PERK: Compact signature scheme based on a new variant of the permuted kernel problem," *Designs, Codes and Cryptography*, vol. 92, pp. 2131–2157, Aug. 2024.
[30] R. Durstenfeld, "Algorithm 235: Random permutation," *Commun. ACM*, vol. 7, p. 420, 1964.
[31] M. J. Kannwischer, M. Krausz, R. Petri, and S.-Y. Yang, "Pqm4: Benchmarking NIST Additional Post-Quantum Signature Schemes on Microcontrollers," 2024.
[32] X. Carril, C. Kardaris, J. Ribes-GonzáLez, O. Farràs, C. Hernandez, V. Kostalabros, J. U. González-Jiménez, and M. Moreto, "Hardware acceleration for high-volume operations of CRYSTALS-kyber and CRYSTALS-dilithium," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 17, no. 3, pp. 1–26, 2024.
[33] Y. Lee, J. Youn, K. Nam, H. H. Jung, M. Cho, J. Na, J.-Y. Park, S. Jeon, B. G. Kang, H. Oh, *et al.*, "An efficient hardware/software co-design for Falcon on low-end embedded systems," *IEEE Access*, vol. 12, pp. 57947–57958, 2024.