Enjoy FAEST on Hardware: Efficient and Novel Hardware Acceleration of Emerging Post-Quantum Digital Signature Scheme FAEST

Brendan Funk and Jiafeng Xie

Department of Electrical and Computer Engineering
Villanova University, Villanova, PA 19087 USA
{bfunk,jiafeng.xie}@villanova.edu

Abstract. The National Institute of Standards and Technology (NIST) Post-Quantum Cryptography (PQC) standardization process has drawn significant attention from the research community recently, especially regarding these PQC schemes' hardware acceleration. Nevertheless, related research on the emerging schemes, e.g., those currently under consideration by the NIST additional digital signature scheme standardization, still needs further exploration. With this background, this paper presents an efficient and novel hardware acceleration for FAEST, one of the promising signatures in the NIST Round 3 additional scheme competition. Overall, we have presented several layers of innovative efforts to obtain the proposed acceleration. First of all, we have conducted detailed algorithmic analysis on the major computationally intensive operations within FAEST and proposed novel design strategies and algorithms for their efficient hardware implementation. Next, we have proposed novel design efforts to construct these major operations into efficient building blocks to finalize the overall accelerator, along with the help of optimized operational phase workflow. The dedicated extension to build the L1-EM-fast accelerator is also given. Finally, we have provided a detailed and thorough evaluation process to showcase the superior performance of the proposed design over the existing complete, Sign part, and single security level MPC-in-the-head signature acceleration works. For instance, at the same security level, the proposed fast variant has at least 98% less equivalent Area-Delay Product (eADP) than the recent complete SDitH design. To the authors' best knowledge, this is the first hardware acceleration work for FAEST, and we hope the outcome of this research will facilitate the ongoing PQC standardization process.

Keywords: FAEST · FPGA · hardware accelerator · MPC-in-the-Head signature · Post-Quantum Cryptography · symmetric-based signature

1 Introduction

Post-Quantum Cryptography (PQC) has drawn significant attention from various communities, as it has been proven that existing public-key cryptosystems like RSA and ECC can be broken by large-scale quantum computers executing Shor's algorithm [Sho94]. In response, the National Institute of Standards and Technology (NIST) launched PQC standardization process in 2016 and has selected five algorithms so far [nis]. In addition, an ongoing digital signature scheme competition has recently selected 9 candidates for consideration in the third round [add] (except that HAWK was recently withdrawn).

Alongside the NIST PQC standardization process, the research community has gradually shifted its attention to implementation, especially on hardware platforms. As a result, a good number of hardware acceleration works have been released on NIST-selected schemes,

e.g., [DHSY24, GJCJ23, TX26], to mention a few. While for the schemes that are under the NIST additional signature standardization process, less exploration has been made, e.g., there is still no hardware acceleration work reported for the third-round additional signature scheme FAEST (a symmetric/MPC-in-the-Head signature) [BBdSG+23].

This work follows this trend to present an efficient and novel hardware acceleration work for FAEST. In particular, we hope the presented work scope not only showcases the superior efficiency of the proposed acceleration framework but also allows us to examine the actual performance of FAEST among its peer schemes, especially those MPC-in-the-Head signatures in the NIST additional digital signature scheme round 3 competition process.

1.1 Prior Works

So far, no hardware acceleration work has been reported for FAEST. Nevertheless, since FAEST can be categorized as both symmetric-based and MPC-in-the-Head signatures [add, 3rd], we briefly discuss prior work on these schemes: (a) the first hardware design for SDitH was presented in [DHSY24]; (b) recently, a resource-efficient SDitH Signature Generation (Sign) accelerator was reported in [HHH+26]; (c) very recently, a hardware accelerator for security level 1 parameters of MQOM was released in [MNKV26].

1.2 Motivation and Challenges

Despite the available hardware acceleration for MPC-in-the-Head signature schemes, it is observed that these designs involve several limitations. (i) The existing design of SDitH [DHSY24] involves very large resource usage, e.g., the SDitH-L5-GF251 design consumes 34,456 LUTs, 31,409 FFs, 472 DSPs, and 521.5 BRAMs while running at 164MHz on NIST-recommended Artix-7 Field-Programmable Gate Array (FPGA). (ii) The design of [HHH+26] only covers the Sign part, and hence, it is not a complete SDitH accelerator. Meanwhile, this design uses an on-the-fly computation strategy to reduce memory usage; thus, it requires an external host to store and feed the needed data for the required operations (but this extra cost is not considered in the final evaluation). (iii) [MNKV26] only covers a specific design for NIST security level 1 parameter sets, while the designs for other security parameter sets remain unexplored. Meanwhile, it also uses the on-the-fly computation strategy to reduce the memory usage but neglects the potential host cost to receive/feed-in needed data; e.g., the secret key in the Key Generation (KeyGen) phase is delivered out, which needs to be fed in during the Signature Verification (Verify) phase.

With these considerations, we aim to develop the hardware acceleration for FAEST with the following objectives: (a) achieve efficient area usage and decent timing, and obtain area-time efficiency over the existing MPC-in-the-Head and related designs; (b) design complete accelerator architecture that contains all phases of FAEST (keyGen, Sign, and Verify), while maintaining standalone execution (no external host support); (c) configurable to all parameter sets while retaining dedicated optimization for specific comparison.

1.3 Contributions

Following these objectives, this work presents a complete Field-Programmable Gate Array (FPGA) implementation framework for all twelve parameter sets of FAEST v2.0 [BBB+a], covering both standard and Even-Mansour (EM) families, the three security categories, and both fast and short-signature configurations. Major contributions of this work include:

Algorithmic analysis and design strategy. We have conducted detailed algorithmic analysis on major computationally intensive operations within FAEST and proposed respective novel design strategies and algorithms for their efficient hardware implementation.

Dedicated strategy/algorithm-to-hardware co-design. We have proposed novel design efforts to construct those major operations into efficient building blocks to finalize the targeted accelerator while taking care of optimized workflow for all operational phases.

We also provide a dedicated extension to build an L1-EM-fast Sign accelerator for a fair comparison with the recently released MQOM work [MNKV26].

Thorough implementation and rigorous comparison. We have conducted a rigorous evaluation process to showcase the superior performance of the proposed accelerator over the existing complete, Sign part, and single security level MPC-in-the-head signature acceleration works. Related work and future directions are also provided at the end.

The rest of the paper is organized as follows. Section 2 gives the preliminaries. Section 3 presents the design strategy for major operations. Section 4 introduces the accelerator. Section 5 gives the evaluation. Sections 6 and 7 present related work and the conclusion.

2 Preliminaries and Algorithm Overview

FAEST signs by proving knowledge of a secret input to a public cipher relation, and we briefly introduce its algorithmic procedures below. More details can be seen at [BBB+a].

Table 1: FAEST v2.0 parameter details [BBB+a]. Lengths ℓ are in bits and signatures in bytes; w_g denotes grinding bits, avoiding confusion with the witness w.

Variants	NIST Security	λ	ℓ	C	τ	w_g	L	T_{open}	Signature
FAEST-128s	L1	128	1280	321	11	7	22,528	102	4,506
FAEST-128f	L1	128	1280	321	16	8	3,072	110	5,924
FAEST-192s	L3	192	2496	641	16	12	40,960	162	11,260
FAEST-192f	L3	192	2496	641	24	8	5,120	163	14,948
FAEST-256s	L5	256	3104	777	22	6	61,440	245	20,696
FAEST-256f	L5	256	3104	777	32	8	7,168	246	26,548
FAEST-EM-128s	L1	128	960	241	11	7	22,528	103	3,906
FAEST-EM-128f	L1	128	960	241	16	8	3,072	112	5,060
FAEST-EM-192s	L3	192	1728	433	16	8	49,152	162	9,340
FAEST-EM-192f	L3	192	1728	433	24	8	5,120	176	12,380
FAEST-EM-256s	L5	256	2688	673	22	6	61,440	218	17,984
FAEST-EM-256f	L5	256	2688	673	32	8	7,168	234	23,476

2.1 Parameters and Cipher Relation

The security parameter $\lambda \in \{128, 192, 256\}$ corresponds to categories 1/3/5 (Table 1). Each category has standard and EM variants, with fast (f) and short-signature (s) settings (total 12 sets). Table 1 also lists the witness length ℓ, constraint count C, number τ of small-VOLE instances, total leaf count L, grinding bits w_g, and opening bound T_{open} ($\oplus$ denotes XOR, $\|$ refers to concatenation, and $[a, b)$ represents indices a through $b-1$).

The public key is (x, y) with $y = F_k(x)$: k is secret, x is public, and F is the selected One-Way Function (OWF). A signature proves knowledge of k without disclosing it. The parameter set selects remaining parameters; fast/short changes signature size.

The seed-expansion PseudoRandom Generator (PRG) is a separate use of AES-CTR, with λ-bit seeds and 128-bit output blocks. SHAKE128 at L1 and SHAKE256 at L3/L5 derive commitments and challenges. Proof arithmetic uses $\mathbb{F}_{2^\lambda}$: addition is XOR and multiplication includes polynomial reduction. Universal hashing uses a 64-bit field path.

2.2 Witnesses, Correlations, and Commitments

The witness w contains the secret and intermediate values needed to check C constraints. VOLE supplies correlations $q_i = v_i + u_i\Delta$, with (u_i, v_i) held by the Signer and (q_i, Δ) by the verifier. The correction $d = w \oplus u_{[0,\ell)}$ gives $q'_i = v_i + w_i\Delta$. Matrices V, Q have λ columns and $\widehat{\ell} = \ell + 3\lambda + 16$ rows for witness tags, proof masks, and consistency checks.

Batch All-but-one Vector Commitment (BAVC) commits to correlation seeds expanded through a Goldreich-Goldwasser-Micali (GGM) tree. XOR accumulation forms the VOLE

instances, and corrections align their u values. The final challenge selects one hidden leaf per vector; leaf opening reveals seeds sufficient to reconstruct the others.

QuickSilver represents each valid-witness constraint as $p_j(z) = a_{0,j} + a_{1,j}z + a_{2,j}z^2$. The Signer computes coefficients, while the verifier evaluates at Δ. ZKHash combines the results with challenge-derived weights and masks into three proof fields.

2.3 Major Operational Phases

Algorithms 1–3 follow the FAEST v2.0 specification [BBB+a]. Parameters are fixed by the selected set. Here n_x is the public-input width; $H_\mu, H_r, H_{\text{iv}}, H_1, H_2, H_3$ are the specified domain-separated SHAKE calls. We write $c = (c_1, \ldots, c_{\tau-1})$ for the VOLE corrections, $\mathcal{O}$ for the information needed to open BAVC, and o for the encoded opening.

KeyGen. KeyGen samples a valid secret key and public input, then evaluates the OWF once. Its main computation is the cipher evaluation (line 4); no proof generated.

Algorithm 1 FAEST.KeyGen: sample a valid key and evaluate the OWF.

Require: Selected OWF F and parameters λ, n_x
Ensure: Secret key sk and public key pk
1: **repeat**
2: $\quad k \xleftarrow{\$} \{0,1\}^\lambda$
3: **until** $k[0] \wedge k[1] = 0$ ▷ the first two key bits are not both one
4: $x \xleftarrow{\$} \{0,1\}^{n_x}$; $y \leftarrow F_k(x)$
5: **return** $(\mathsf{sk}, \mathsf{pk}) = (k, (x, y))$

Sign. Sign has three steps: commit to correlations; check their consistency and prove the OWF constraints; then search for an accepted final challenge and open BAVC. Each step depends on the preceding challenge. Correlations must remain available for both VOLEHash and OWFProve, and the tree must remain recoverable for the leaf opening.

Algorithm 2 FAEST.Sign: three challenge-dependent steps.

Require: $\mathsf{sk} = k$, $\mathsf{pk} = (x, y)$, message m
Ensure: Complete signature σ
1. Commitment and correlation generation
1: $\rho \xleftarrow{\$} \{0,1\}^\lambda$; $\mu \leftarrow H_\mu(\mathsf{pk} \| m)$
2: $(r, \mathsf{iv}_{\text{pre}}) \leftarrow H_r(k \| \mu \| \rho)$; $\mathsf{iv} \leftarrow H_{\text{iv}}(\mathsf{iv}_{\text{pre}})$
3: $(\mathsf{com}, \mathcal{O}, c, u, V) \leftarrow \mathsf{VOLECommit}(r, \mathsf{iv})$
4: $\mathsf{chall}_1 \leftarrow H_1(\mu \| \mathsf{com} \| c \| \mathsf{iv}_{\text{pre}})$
2. Consistency checking and proof generation
5: $\widehat{u} \leftarrow \mathsf{VOLEHash}(\mathsf{chall}_1, u)$
6: $\widehat{V} \leftarrow \mathsf{VOLEHash}(\mathsf{chall}_1, V)$ ▷ column-wise
7: $w \leftarrow \mathsf{ExtendWitness}(k, x)$; $d \leftarrow w \oplus u_{[0,\ell)}$
8: $\mathsf{chall}_2 \leftarrow H_2(\mathsf{chall}_1 \| \widehat{u} \| \widehat{V} \| d)$
9: $u^* \leftarrow u_{[\ell, \ell+2\lambda)}$; $V^* \leftarrow V_{[0,\ell+2\lambda),:}$
10: $(\widetilde{a}_0, \widetilde{a}_1, \widetilde{a}_2) \leftarrow \mathsf{OWFProve}(w, u^*, V^*, \mathsf{pk}, \mathsf{chall}_2)$
3. Grinding, bounded opening, and serialization
11: $\mathsf{ctr} \leftarrow 0$
12: **loop**
13: $\quad \mathsf{chall}_3 \leftarrow H_3(\mathsf{chall}_2 \| \widetilde{a}_0 \| \widetilde{a}_1 \| \widetilde{a}_2 \| \mathsf{ctr})$
14: $\quad$ **if** $\mathsf{chall}_3[\lambda - w_{\text{g}}, \lambda) = 0^{w_{\text{g}}}$ **then**
15: $\quad\quad I \leftarrow \mathsf{Decode}(\mathsf{chall}_3[0, \lambda - w_{\text{g}}))$
16: $\quad\quad o \leftarrow \mathsf{BAVC.Open}(\mathcal{O}, I; T_{\text{open}})$
17: $\quad\quad$ **if** $o \neq \perp$ **then**
18: $\quad\quad\quad$ **break**
19: $\quad\quad$ **end if**
20: $\quad$ **end if**
21: $\quad \mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
22: **end loop**
23: **return** $\mathsf{Serialize}(c, \widehat{u}, d, \widetilde{a}_1, \widetilde{a}_2, o, \mathsf{chall}_3, \mathsf{iv}_{\text{pre}}, \mathsf{ctr})$

Verify. Verify reconstructs the commitment and Q from the signature, repeats the column checks, and evaluates the OWF constraints on corrected tags. This recovers the missing proof field $\widetilde{a}_0$ and allows the final hash check. Lastly, Verify checks the supplied grinding counter once to ensure the value provided by matches.

Algorithm 3 FAEST.Verify: reconstruct and check the supplied transcript.

Require: Public key pk, message m, signature σ
Ensure: Accept or reject
Parse σ as $(c, \widehat{u}, d, \widetilde{a}_1, \widetilde{a}_2, o, \mathsf{chall}_3, \mathsf{iv}_{\mathrm{pre}}, \mathsf{ctr})$
$\mu \leftarrow H_\mu(\mathsf{pk} \| m)$; $\mathsf{iv} \leftarrow H_{\mathrm{iv}}(\mathsf{iv}_{\mathrm{pre}})$
if $\mathsf{chall}_3[\lambda - w_{\mathrm{g}}, \lambda) \neq 0^{w_{\mathrm{g}}}$ **then**
    return reject
end if
$(\mathsf{com}, Q) \leftarrow \mathsf{VOLEReconstruct}(o, \mathsf{chall}_3, c, \mathsf{iv})$
if reconstruction fails **then**
    return reject
end if
$\mathsf{chall}_1 \leftarrow H_1(\mu \| \mathsf{com} \| c \| \mathsf{iv}_{\mathrm{pre}})$
$\widehat{Q} \leftarrow \mathsf{VOLEHash}(\mathsf{chall}_1, Q)$ ▷ column-wise
for $j = 0, \ldots, \lambda - 1$ **do**
    $\widehat{V}_{:,j} \leftarrow \widehat{Q}_{:,j} \oplus \delta_j \widehat{u}$ ▷ δ_j: bit j of chall_3
end for
$\mathsf{chall}_2 \leftarrow H_2(\mathsf{chall}_1 \| \widehat{u} \| \widehat{V} \| d)$
$Q^* \leftarrow Q_{[0, \ell + 2\lambda), :}$
$\widetilde{a}_0 \leftarrow \mathsf{OWFVerify}(d, Q^*, \mathsf{pk}, \mathsf{chall}_2, \mathsf{chall}_3, \widetilde{a}_1, \widetilde{a}_2)$
return $\mathsf{chall}_3 = H_3(\mathsf{chall}_2 \| \widetilde{a}_0 \| \widetilde{a}_1 \| \widetilde{a}_2 \| \mathsf{ctr})$

2.4 Computationally Intensive Operations

Overall, we identified four operations that account for substantial hardware implementation complexity in FAEST: PRG/BAVC expansion, VOLE accumulation, universal hashing, and OWF constraint evaluation. Table 2 lists each operation and related hardware cost.

Table 2: Computationally Intensive Operations, Hardware Costs, and Design Strategies.

Operation	Intensive Computation	Hardware Cost	Section 3 Strategy
PRG/BAVC	Expand nodes and leaf seeds	AES calls and expansion state	Process expansion outputs in order (§3.1)
	Commit leaves; recover an opening	Seed memory and tree access logic	Preserve recoverable opening inputs (§3.1)
VOLE	Expand seeds and XOR correlations	Accumulator reads/writes	Accumulate incoming PRG words (§3.2)
	Supply columns, then rows/masks	State across challenges; different access patterns	Keep or regenerate columns and proof rows (§3.2)
VOLEHash/ ZKHash	Multiply and accumulate ordered words	Dependent field operations	Reuse arithmetic within folds (§3.3)
	Hash columns, then proof values and masks	Challenge-dependent reads	Check columns before proving constraints (§3.4)
OWF proof	Evaluate authenticated cipher constraints	Field products and witness reads	Replay one constraint engine (§3.4)
	Combine evaluation streams	Retained results and interpolation	Pair results by constraint index (§3.4)

3 Proposed Algorithms and Design Strategies

Connecting Table 2, by examining their computation, storage requirements, and challenge dependencies, we derive the proposed design principles/algorithms used in our accelerator building, particularly to reduce their further hardware implementation complexity.

3.1 PRG/BAVC

Hardware Design Challenges. VOLECommit expands a root seed into a tree of seeds, commits to the leaves, and supplies seeds for correlation generation (Algorithm 2, line 3). Each expansion requires cipher computation. A straightforward implementation can save all expanded nodes to avoid computing them again, but this requires memory usage, address generation, and logic resources to select the requested node.

The GGM tree is also needed later for BAVC opening, which reveals seeds that allow the verifier to reconstruct selected leaves. The third challenge (chall_3) lies in determining which leaves remain hidden, so leaf opening cannot be completed during the initial expansion. Discarding the tree too early loses information needed to produce the signature; while storing the entire tree would consume more memory for storing intermediate values.

Proposed Strategy. We propose to process expansion results in the order required by commitment and correlation generation, passing each result to its next operation without first storing every expanded vector. Meanwhile, we preserve enough information to recover the later leaf opening. Retention and deterministic re-expansion provide two ways to meet this requirement. Re-expansion reproduces the same values when the seed, initialization value, tweak, and output order are unchanged; the tweak distinguishes expansion contexts. It therefore exchanges additional cipher calls for reduced storage and access logic. Figure 1(a) illustrates this choice (Section 5 confirms its efficiency).

When a trial challenge does not permit a bounded opening, the Signer keeps its proof fields and tries the next counter. It does not repeat proof generation. Verification instead expands the revealed seeds from the supplied opening (Algorithm 3, line 6). Both operations require seed expansion, but they start from different available information.

Note that we also have extended the proposed accelerator to an optimized FAEST sign design to have a fair comparison with the existing work of [MNKV26], where we have proposed to use a "regenerate when needed" strategy, that is, we only store values that are absolutely necessary to recover the seeds (Fig. 1(b)).

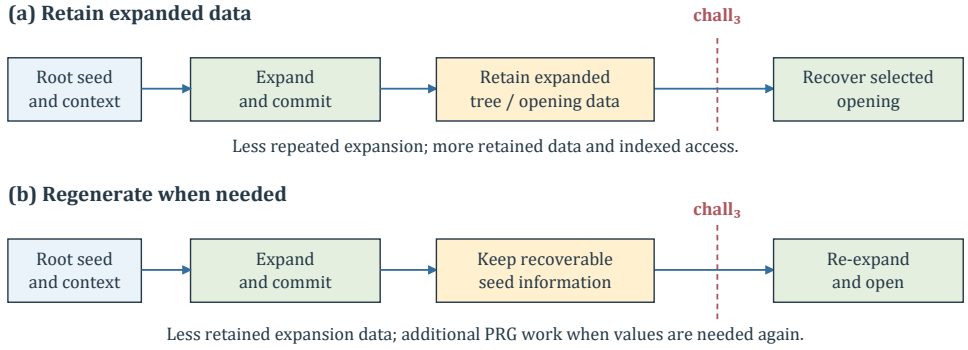

Figure 1: Proposed PRG/BAVC storage choices. (a) Retaining expanded tree and opening data avoids some repeated expansion; (b) retaining recoverable seed information instead allows deterministic re-expansion after chall_3 selects the opening.

3.2 VOLE

Hardware Design Challenges. VOLE generation expands leaf seeds into words and combines them by XOR. Although each XOR is inexpensive, memory logic must read and write the correct accumulated value. Expanding all vectors into temporary storage before combining them adds memory usage and another round of reads.

The resulting matrix also has two access patterns. VOLEHash reads columns after chall_1 (Algorithm 2, line 6); OWFProve later needs rows and masks after chall_2 (Algorithm

2, line 10). A row holds the complete tag associated with one value, whereas a column holds one bit position across many tags. An arrangement that supplies columns efficiently may need extra selection logic to assemble rows. A complete matrix occupies $\lambda\widehat{\ell}$ bits, and the corrections require another $(\tau-1)\widehat{\ell}$ bits.

Proposed Strategy. We propose to accumulate PRG words as they become available, updating the required XOR sums rather than storing every expanded leaf vector. The sums are unchanged, but temporary expansion data can be released after its contributions have been applied. We then propose to schedule correlation access around its two uses: column checking first, followed by proof rows and masks. Completing a column check does not make the underlying correlation information disposable. The correlation data must remain stored or reproducible until the proof has used it. Similarly, corrections must remain available for challenge hashing and final signature output. Our strategy preserves the order while allowing each implementation to balance stored data against regeneration work. The same distinction between column checks and proof-row reads applies to the verifier's matrix Q. Figure 2 shows the accumulation step and the two later accesses.

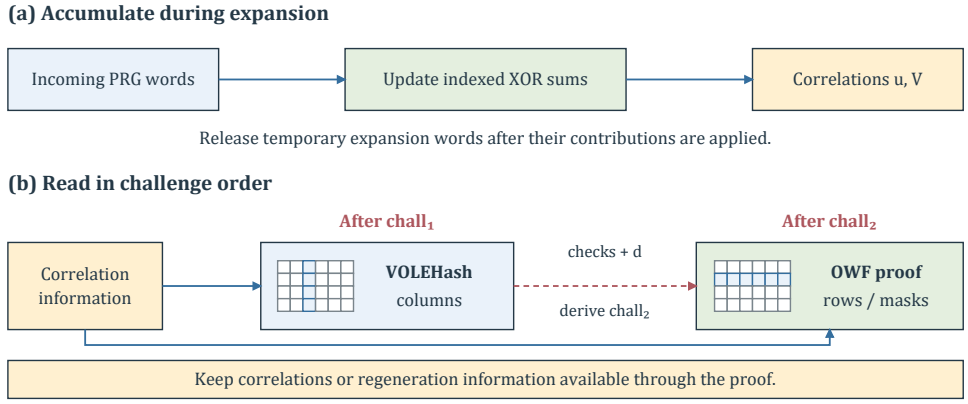

Figure 2: Proposed VOLE accumulation and data lifetime. (a) Incoming PRG words update the required XOR sums without first storing every expanded vector. (b) The same correlation information supports column checks after chall_1 and proof rows and masks after chall_2.

3.3 VOLEHash/ZKHash

Hardware Design Challenges. VOLEHash and ZKHash compress long sequences into short checks using field arithmetic. VOLEHash checks the correlations; ZKHash combines the OWF proof values. Both contain ordered folds: a running value h is updated as $h \leftarrow sh + x_i$, where x_i is the next input and s comes from a challenge. Each multiplication needs the preceding result. Adding arithmetic units alone therefore does not remove the dependency within a fold.

The two hashes become active at different times. VOLEHash needs the first challenge and reads correlation columns using λ-bit and 64-bit arithmetic. ZKHash needs the second challenge and processes proof coefficients with their masks. Reading either sequence before its challenge is available cannot complete the hash. Conversely, overwriting masks or advancing the sequence before an update finishes can change the result.

Proposed Strategy. We propose to reuse arithmetic within each hash and keep its running values until the ordered updates finish. The next dependent input advances only when its current multiplication and addition are complete. This preserves the same polynomial evaluation while trading replicated operators for sequential work. Local reuse does not require VOLEHash and ZKHash to share one physical multiplier. Figure 3

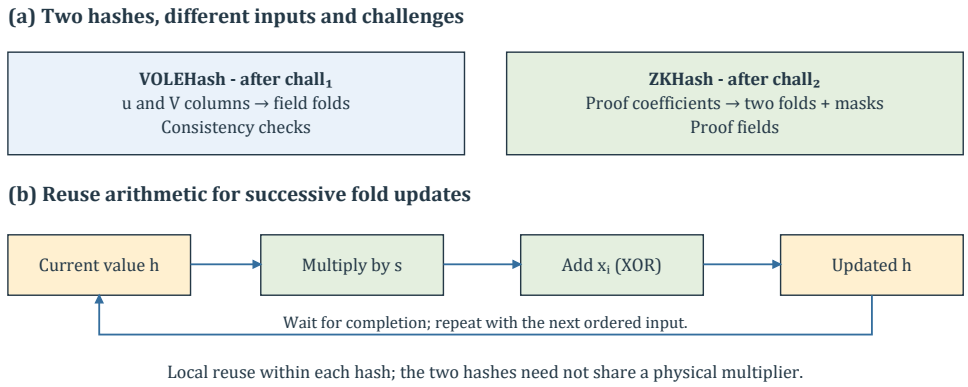

Figure 3: Proposed ordered arithmetic reuse in VOLEHash and ZKHash. (a) The hashes operate on different data after different challenges. (b) A fold retains its running value and reuses arithmetic for $h \leftarrow sh + x_i$, advancing after the current update completes.

separates their challenge dependencies and illustrates the reusable fold.

For ZKHash, each coefficient sequence updates two running values, $h_0 \leftarrow sh_0 + x_i$ and $h_1 \leftarrow th_1 + x_i$. After the last coefficient, the result is $r_0h_0 + r_1h_1 + \eta$, where s, t, r_0, r_1 come from chall_2 and η is the corresponding proof mask. We propose to retain that mask through final mixing and preserve the constraint order established by OWFProve.

3.4 OWF Proof

Hardware Design Challenges. KeyGen evaluates the cipher relation once. OWFProve must also show that the hidden intermediate values satisfy the cipher's equations, called constraints. Masking these values turns constraint evaluation into polynomial arithmetic over the proof field. The prover needs polynomial coefficients, whereas the verifier checks evaluations at its challenge value. Computing the required evaluations with separate copies of the constraint engine would replicate field arithmetic and witness-access logic.

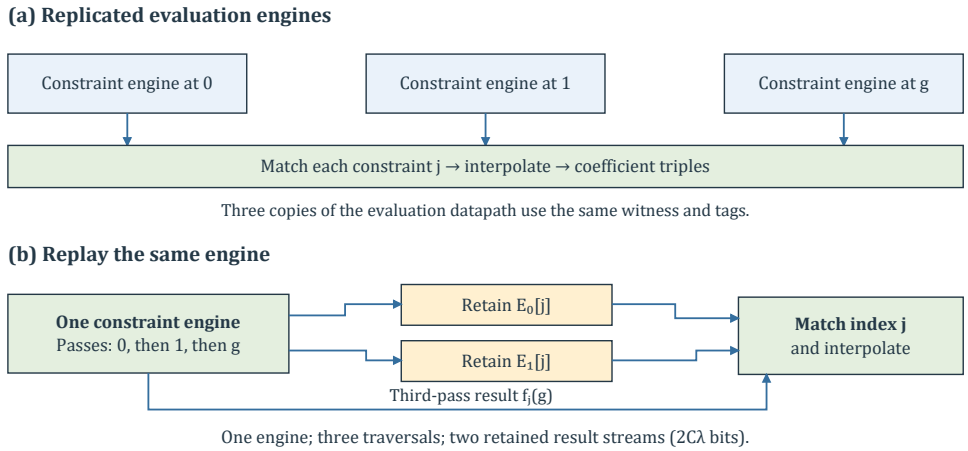

Figure 4: Proposed constraint replication versus replay. (a) One design evaluates the three points with separate engines. (b) Replay evaluates the same constraints in three sequential passes, retaining $E_0[j]$ and $E_1[j]$ until the matching third-pass value arrives.

Proposed Strategy and Algorithm. We propose to reuse one constraint engine at three distinct field points, 0, 1, and g, with $g \notin \{0, 1\}$. For a valid witness, a constraint

polynomial has degree at most two, so these evaluations determine its coefficients exactly. Figure 4 contrasts this sequential schedule with a design containing three engines.

We propose to retain the first two evaluation streams, E_0 and E_1, and combine them with each matching result from the third pass. The constraint index j identifies which evaluations belong together, preserving the coefficient order expected by ZKHash. Replay requires three traversals and $2C\lambda$ bits for the retained streams, so its benefit is reduced datapath replication rather than fewer constraint evaluations. Verification evaluates at the supplied challenge and does not need these interpolation passes. Section 4.5 shows the detailed replay hardware architecture design and related datapath.

Algorithm 4 Proposed coefficient-streaming OWF proof generation.

Require: Ordered OWF arithmetic traversal, its input triples, chall_2, masks η_0, η_1, η_2
Ensure: Proof fields $\widetilde{a}_0, \widetilde{a}_1, \widetilde{a}_2$

Obtain s, t, r_0, r_1 from chall_2 as in ZKHash
$h_{0,k}, h_{1,k} \leftarrow 0$ for $k = 0, 1, 2$
for each arithmetic operation in one OWF traversal **do**
  if the operation adds triples A and B **then**
    $R_k \leftarrow A_k + B_k$ for $k = 0, 1, 2$
  else if the operation multiplies triples A and B **then**
    $p_d \leftarrow \sum_{i+j=d} A_i B_j$ for $d = 0, \ldots, 4$, with $i, j \in \{0, 1, 2\}$
    $R_0 \leftarrow p_0$
    $R_1 \leftarrow p_1 + g p_3 + (g^2 + g) p_4$
    $R_2 \leftarrow p_2 + (g + 1) p_3 + (g^2 + g + 1) p_4$
  end if
  Keep R for subsequent operations that need it
  if R completes the next constraint in the prescribed order **then**
    for $k = 0, 1, 2$ **do**
      $h_{0,k} \leftarrow s h_{0,k} + R_k$; $h_{1,k} \leftarrow t h_{1,k} + R_k$
    end for
    Finish these updates before accepting the next constraint
  end if
end for
$\widetilde{a}_k \leftarrow r_0 h_{0,k} + r_1 h_{1,k} + \eta_k$ for $k = 0, 1, 2$
return $(\widetilde{a}_0, \widetilde{a}_1, \widetilde{a}_2)$

Note that the below strategy was proposed specifically to compete with the [MNKV26] Signer design: **coefficient-streaming OWF proof generation.** In brief, we propose carrying the three coefficients through the OWF computation itself, then feeding each completed constraint directly into ZKHash. Algorithm 4 replaces the OWFProve step in Algorithm 2; it is an alternative to replay, not an additional pass.

Each intermediate value is represented by a triple (a_0, a_1, a_2) for $a_0 + a_1 z + a_2 z^2$, where z is a formal variable and the coefficients belong to the proof field. We use the same distinct points $0, 1, g$ as replay. Addition acts on matching coefficients. Multiplication first forms a product of degree at most four, then reduces it modulo $m(z) = z(z + 1)(z + g)$. This gives the exact substitutions $z^3 \longmapsto gz + (g + 1)z^2$, $z^4 \longmapsto (g^2 + g)z + (g^2 + g + 1)z^2$.

Thus high-degree terms contribute to the retained coefficients instead of being discarded. The traversal follows the original OWF additions and multiplications in dependency order. The input triples represent the same masked inputs as the original computation; constants use $(c, 0, 0)$ and z uses $(0, 1, 0)$.

Reduction preserves the evaluations at $0, 1, g$, because m is zero at each point. Applying it after every product therefore preserves the three scalar computations. For a valid witness, each completed constraint has degree at most two, so its retained triple is exactly the triple recovered by interpolation. Keeping the original constraint order, challenge weights, and masks then preserves the proof fields.

This organization removes the two replay-result streams and their interpolation stage, but still requires coefficient working state and sequential field arithmetic. The multiplier

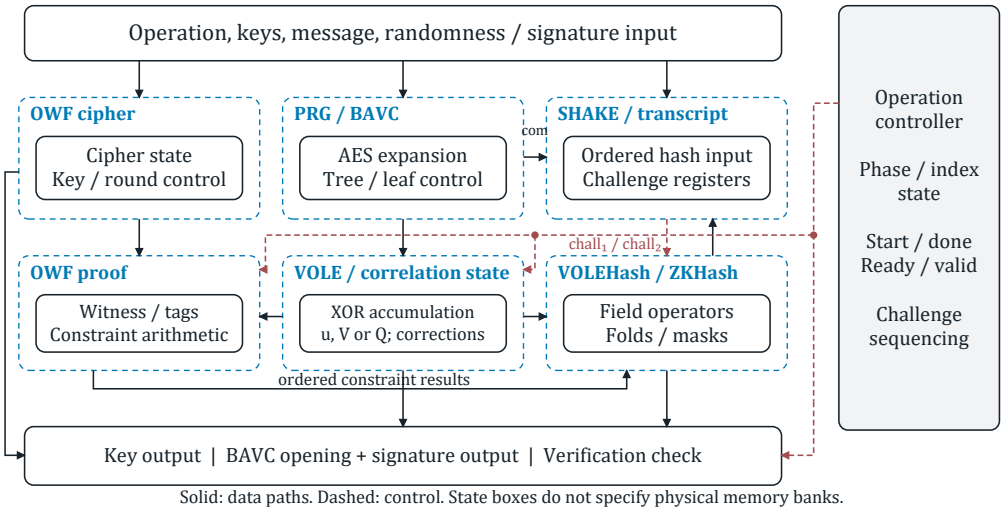

Figure 5: Complete FAEST accelerator organization.

uses 4, 6, and 9 field products for degree-1 $\times$ 1, 2 $\times$ 1, and 2 $\times$ 2 products, respectively; the algorithm does not claim fewer field multiplications. Its benefit is one traversal with coefficients passed directly to proof hashing.

3.5 Brief Summary

Together, these four strategies address repeated expansion, correlation storage, dependent hashing, and proof arithmetic. Their common principle is to keep data only as long as it is needed or can be recovered, and to reuse computation without changing the order required by FAEST. Retention, regeneration, and arithmetic sharing move costs between memory, control logic, and execution time. Section 4 describes how the hardware implements these choices, including the SHAKE sequencing that connects them through the three challenges.

4 Proposed Accelerator

4.1 Overview

Figure 5 connects the OWF cipher, PRG/BAVC, VOLE, constraint engine, field hashes, and SHAKE to implement KeyGen, Sign, and Verify (a complete accelerator). An operation controller starts each block when its inputs are ready; parsing, bounded opening, and output logic complete the operation paths. We first describe the four datapaths implementing Section 3, then trace each operation, and finally the separately optimized L1-EM-fast sign design. In particular, we propose dedicated strategy/algorithm-to-hardware co-design efforts to build these important building blocks and also the whole accelerator. Meanwhile, we take care of the accelerator architectural layout that it covers all parameter sets and does not need external host support for standalone acceleration.

Control and configuration. Start/done signals delimit operations; valid/ready signals transfer indexed words. A stalled block holds its result and index. Field width sets operand sizes, $\widehat{\ell}$ sets column length, and τ sets small-VOLE loops and corrections. Constraint and tree bounds set iteration limits; signature length sets output offsets. Thus, configuration changes both stored data and controller schedules of a specific accelerator.

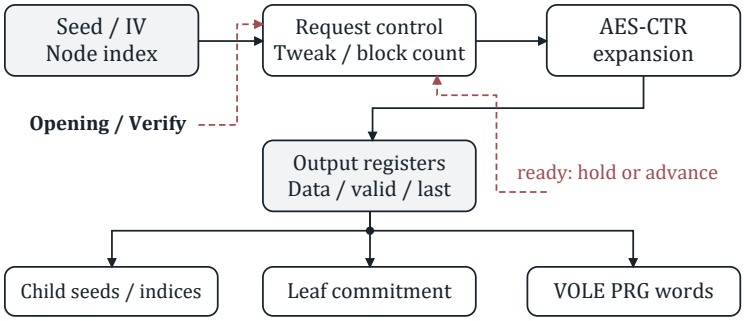

Figure 6: PRG/BAVC request and output organization.

4.2 PRG/BAVC Datapath

The OWF cipher evaluates the public-key relation and supplies witness intermediates. The PRG instead expands seeds for the tree, leaf processing, and VOLE. Figure 6 separates its requests, cipher execution, and output registers, implementing the idea of Section 3.1.

Expansion interface. A request supplies a seed, IV, tweak, and block count. The tweak identifies the expansion context; a counter selects successive AES inputs. Each output carries data, valid, and last-block indicators. Backpressure holds the pending block and counter. Expanding a $\widehat{\ell}$-bit correlation vector requires $\lceil \widehat{\ell}/128 \rceil$ blocks, using only the specified bits of the last block. Configuration therefore changes the number of transfers without changing their 128-bit width.

Tree and leaf control. For an internal node, the controller supplies the parent seed and index-derived tweak. Child seeds retain their node indices through output stalls; the reconstruction expansion block collects them into a response before the walker advances. Leaf processing separately produces commitment information and correlation seed material. Indices preserve the commitment order required by H_1 as well as identifying destinations; counting completed AES calls alone would not preserve that order.

Leaf opening waits for the hidden-leaf indices from chall_3. Retained or reproducible seed information then supplies the bounded BAVC opening. Verification instead expands revealed seeds from the signature. Both paths use the same deterministic seed, IV, and tweak interpretation.

4.3 VOLE Datapath and Correlation Storage

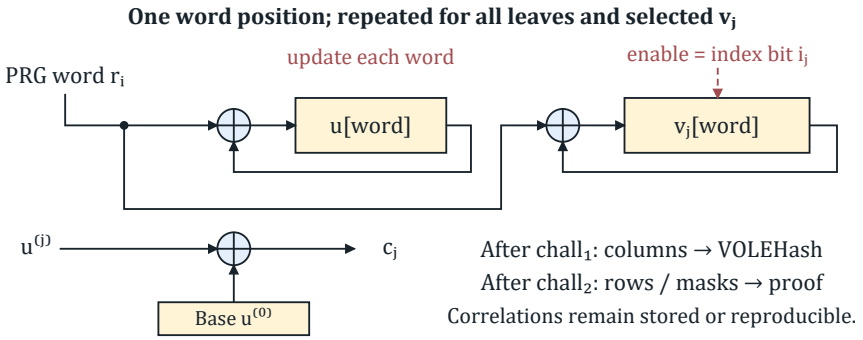

Figure 7: Architectural deatils of wordwise VOLE accumulation.

Indexed XOR accumulation. Figure 7 implements the design strategy proposed in Section 3.2. Let r_i be the vector expanded from leaf i. Every r_i contributes to u, and it

contributes to v_j when index bit i_j is one. For each PRG word, the word index selects an accumulator position and the leaf-index bits enable the appropriate XOR updates. Expanded words can be released after these updates, avoiding storage of all leaf vectors.

The accumulator requests a seed, completes its expansion, and advances to the next leaf. It emits the accumulated words with their vector and word indices. The correction block retains the first instance's u and XORs it with each later instance's corresponding words to form c_j. These corrections remain available for hashing and signature output.

Column and row access. VOLEHash requests $\widehat{\ell}$-bit columns after chall_1; the proof requests λ-bit rows and masks after chall_2. Finishing column checks therefore does not release the correlations: the proof still needs them, either stored or regenerated. These accesses occur in sequence and do not require simultaneous column and row reads.

Each request identifies a column or row, and the VOLE arithmetic unit waits for its matching valid response. Depending on the representation, a row can be read directly or assembled from several words. Eliminating duplicate storage can therefore add read traffic and selection logic; logical capacity and mapped BRAM savings are different quantities. Lifetimes also differ: proof masks remain through final mixing, whereas the witness correction d must survive both H_2 and final serialization.

4.4 VOLEHash/ZKHash Datapaths

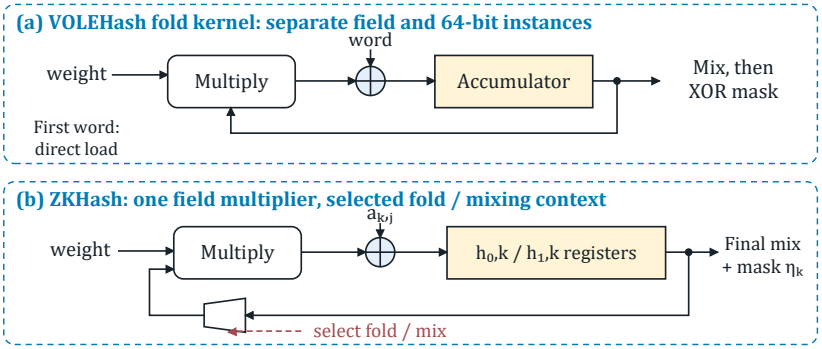

Figure 8: Registered multiply–XOR folds and local arithmetic reuse.

VOLEHash. The column wrapper realizes Section 3.3: request a column, capture the valid response, start its hash, and return the result with its index. It retains chall_1 throughout the checks. Figure 8 details the involved multiplication, XOR, and feedback registers.

The VOLEHash engine reads the folded input as proof-field words and as 64-bit words, using separate arithmetic units and accumulators. The first word initializes each accumulator; later words multiply the previous value by its challenge weight and XOR the next word. Final mixing reuses the proof-field multiplier for four products, embedding the 64-bit result into that field. A temporary register holds the first product of each pair. The first mixed value and the specified short part of the second are concatenated and XORed with the retained column mask. Only then is the indexed hash released to H_2.

ZKHash. In the replay proof wrapper, one field multiplier serves interpolation, six folds, and final mixing. Operand selectors and a controller sequence these requests; registers hold coefficients, intermediate products, and two accumulators per coefficient stream. For coefficient $a_{k,j}$, they update $h_{0,k} \leftarrow sh_{0,k} + a_{k,j}$ and $h_{1,k} \leftarrow th_{1,k} + a_{k,j}$, for $k \in \{0,1,2\}$. The first coefficient initializes both accumulators.

After the last constraint, the multiplier forms $r_0 h_{0,k} + r_1 h_{1,k} + \eta_k$. Weights come from chall_2; masks η_k remain until this step. The controller waits for each product before updating its target register, and accepts the next coefficient only when the current folds

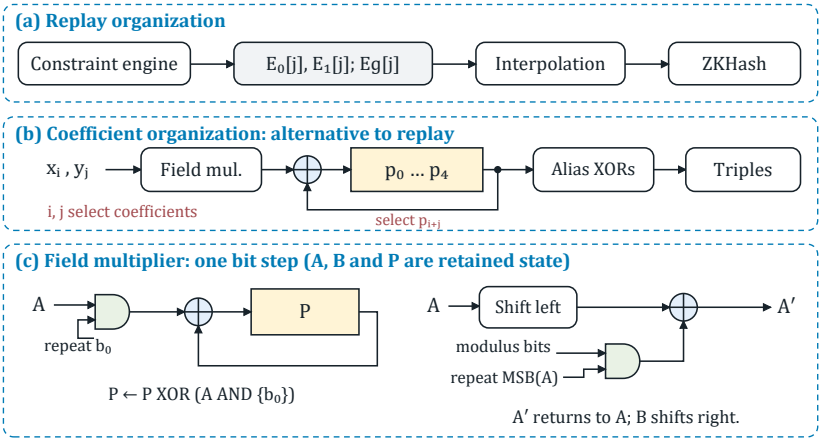

Figure 9: OWF proof organizations and one bit step digit-serial field multiplier.

finish. All three final fields enter H_3; the signature transmits $\widetilde{a}_1, \widetilde{a}_2$, while verification reconstructs $\widetilde{a}_0$.

4.5 OWF Proof Datapaths

Scalar replay. The datapath in Figure 9 implements Section 3.4 with one constraint engine, a pass controller, and two indexed result stores. It traverses the witness and tags at 0, 1, and g, completing each pass before changing. The first two passes store $E_0[j]$ and $E_1[j]$; the third pairs $E_g[j]$ with those values for interpolation and ZKHash. The stores require $2C\lambda$ bits for C constraints, and witness/tag data remain available across all passes.

Constraint generation. The engine checks authenticated intermediate values rather than performing an ordinary encryption. Its controller fetches witness rows, retains partial expressions, and issues dependent field operations. Separate sequences generate the initial constraint, applicable key-schedule constraints, and encryption constraints. The latter check nonlinear inverse/norm relations and links between cipher intermediates. Public inputs and outputs enter at fixed positions. Each completed equation produces a field value and constraint index; a stalled proof holds up this stream without changing its order.

Field multiplication. The documented shared multiplier retains two operands and an accumulated product. Each bit step conditionally XORs the multiplicand into the product, shifts the operands, and conditionally XORs the reduction constant into the shifted multiplicand. Figure 9 shows these AND/XOR paths. A configured number of bit steps forms one digit-serial iteration; completion returns the product to the requesting controller. Constraint latency also includes operand reads, result capture, and stalls.

Coefficient-streaming alternative. Algorithm 4 represents each working value by three field coefficients. Matching coefficients are XORed for addition. For multiplication, two indices select operand coefficients for one field multiplier; each returned product is XORed into one of five partial-product registers, according to its degree. Shift/reduction and XOR logic then applies the exact aliases in Section 3.4, folding degrees three and four into the three output coefficients. The caller holds the operand triples stable. Degree combinations require four, six, or nine field products.

Working values and indexed constraint results still need storage. In particular, nonlinear and linkage calculations finish in a different order from the constraint sequence, so their results are stored and read back in ZKHash order. The wrapper accepts a triple, completes its six fold updates, and then accepts the next constraint. This organization removes the two replay-result stores and interpolation, but retains ordering storage, witness/key state, weights, accumulators, and masks. It is an alternative to three scalar passes; its evaluated variant is identified in Section **??**.

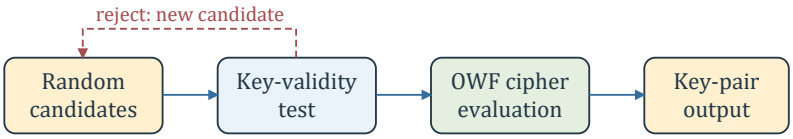

Figure 10: KeyGen workflow through the accelerator.

Larger proof fields widen operands and reduction logic; longer witnesses and additional constraints arithmetic steps. Proof-tag arithmetic remains distinct from the cipher's byte arithmetic. Section 5 relates these differences to the measured configurations.

4.6 Key Generation

The workflow presented in Figure 10 follows the procedure of Algorithm 1. The controller tests random key candidates, retains the accepted key and public input, and starts the OWF cipher. Where the relation requires multiple cipher blocks, earlier results remain stored while later blocks execute. Block and round bounds follow the selected relation; completion releases the key pair only after the full public output is available.

4.7 Signature Generation

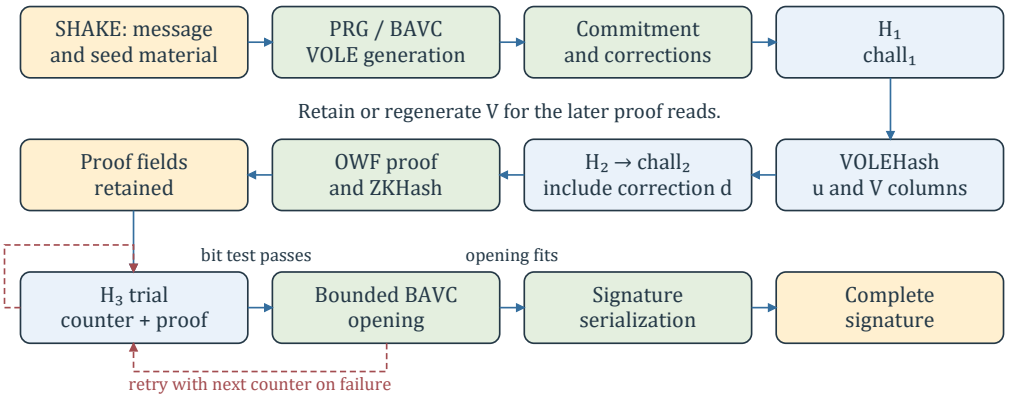

Figure 11: Signature generation, with challenge boundaries and the grinding retry path.

Figure 11 maps the procedure of Algorithm 2 (Sign) onto the four datapaths.

Commitment and correlations. SHAKE derives the message digest and seed material. PRG/BAVC expands the tree and computes leaf commitments; VOLE accumulates correlations and corrections. The commitment, corrections, message digest, and IV information complete H_1. Only then does chall_1 enable column checking. Seed information remains recoverable for leaf opening.

Consistency and proof. VOLEHash checks u and the correlation columns. Its outputs and the witness correction d complete H_2 with chall_1, producing chall_2. The OWF engine reads witness rows and sends ordered constraints to the proof wrapper. ZKHash folds these results and applies the retained masks to produce the three proof fields.

Grinding, opening, and output. Registers hold chall_2, the proof fields, and a trial counter for H_3. A failed zero-bit test increments the counter and repeats the hash with unchanged proof fields. A passing test supplies hidden-leaf indices to bounded BAVC opening; an oversized opening also retries with the next counter. Success enables serialization of the corrections, consistency output, d, transmitted proof fields, opening, accepted challenge, IV information, and counter in specification order.

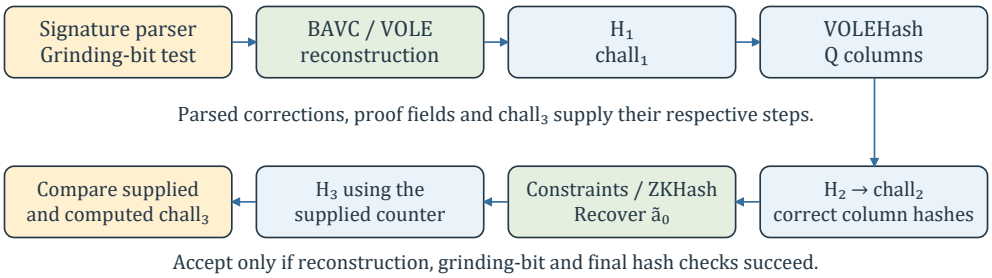

Figure 12: Verify reconstructs correlations and checks the supplied counter and challenge.

SHAKE interface and transcript order. Each request specifies a domain, ordered input bytes, and output length. Field and byte counters track the concatenations above; partial-word state and a final-byte count exclude unused bits. Hash completion releases the next challenge. Fields needed again remain stored, particularly the unchanged H_3 inputs across grinding trials and signature fields through serialization.

4.8 Verification

Figure 12 begins with parsing and the grinding-bit test. BAVC expands the revealed seeds, while supplied commitments account for the hidden leaves. Expansion, corrections, and XOR accumulation reconstruct the commitment and verifier matrix Q, without recovering hidden seeds. The commitment supplies H_1. VOLEHash checks the reconstructed columns, combining their hashes with the supplied hash of u using the challenge bits. Those results and d reproduce chall_2.

The proof evaluator reads rows of Q, applies d, and evaluates constraints at Δ from chall_3. Its ordered stream enters proof hashing. Field multiplication and XOR recover $\widetilde{a}_0 = \widetilde{q} + \widetilde{a}_1\Delta + \widetilde{a}_2\Delta^2$ from the hash result $\widetilde{q}$ and transmitted fields. SHAKE recomputes H_3 with the supplied counter. Acceptance requires successful reconstruction, zero grinding bits, and agreement with the supplied third challenge. The verifier checks the Signer's counter once; it does not try other counters.

4.9 Optimized L1-EM-fast Signing Path

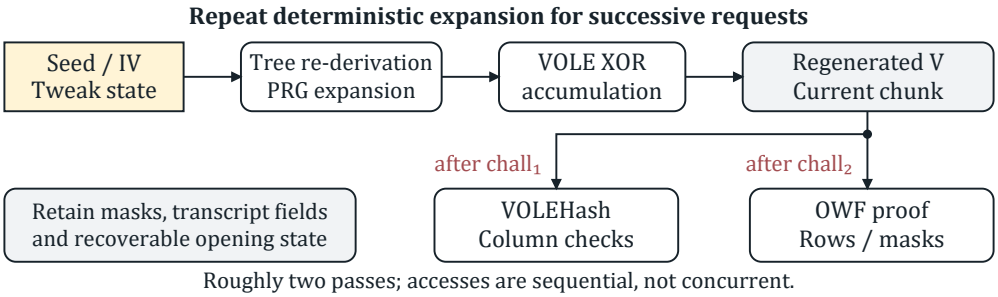

Figure 13: Optimized EM signing path: regenerate correlations for successive checks while retaining seeds, masks, and transcript/opening state.

The optimized complete L1-EM-fast design (Figure 13) changes signing storage: deterministic tree re-derivation replaces the approximately 64 KB retained leaf layer, and V is regenerated in chunks over roughly two passes.

Initial expansion still produces the commitment and corrections for chall_1. Later requests reproduce correlations for VOLEHash and the proof using the same seeds, IV, and tweaks. Seed information remains recoverable for opening, masks survive through proof mixing, and transcript/signature fields remain until their last use. This preserves the challenge order while exchanging memory and its access logic for additional cipher work. Section 5.7 evaluates that tradeoff for the complete design.

4.10 Brief Summary

Overall, the proposed design is complete (covers all 3 phases) and is configurable for all 12 parameter sets for FAEST. Meanwhile (except for the L1-EM-fast Sign design, for a fair comparison with [MNKV26]), our design also stores all needed data within the accelerator, so there is no need for external support or storage (can realize standalone operation).

5 Evaluation: Implementation and Comparison

5.1 Experimental Setup

The proposed FEAST accelerator is coded in VHDL, and synthesized and implemented on the AMD Artix-7 (XC7A200T-3) FPGA using Vivado 2024.2. We have verified the functional correctness of the accelerator against the NIST FAEST reference implementation for all parameter sets. The performance results are obtained after place and route.

5.2 Performance

Table 3: Results of complete FAEST family (this work). Times are ms; total delay is KeyGen + Sign + Verify. L1/L3/L5 denote categories 1/3/5; f/s denote fast/short.

Security	Design	LUT	FF	BRAM	DSP	eSlice	MHz	KeyGen	Sign	Verify	Total
L1	128f	32,940	17,420	16.5	0	14,247	138.6	0.00009	1.51	2.55	4.06
	128s	31,764	17,112	15.5	0	13,682	137.5	0.00009	9.43	15.92	25.35
	EM-128f	29,716	16,324	15	0	12,956	138.3	0.00009	1.33	2.24	3.57
	EM-128s	28,228	15,624	14	0	12,264	138	0.00009	6.65	11.23	17.88
L3	192f	43,820	24,180	25.5	0	19,904	125.9	0.0016	3.08	4.92	8.00
	192s	42,012	23,464	24.5	0	19,130	124.7	0.0016	18.24	29.14	47.38
	EM-192f	41,268	23,308	24	0	18,808	125.4	0.0016	1.92	3.07	4.99
	EM-192s	39,524	22,452	23.5	0	18,149	124.6	0.0016	10.78	17.23	28.01
L5	256f	54,760	31,880	34.5	0	25,693	115.4	0.0024	4.83	8.05	12.88
	256s	52,708	30,748	33	0	24,690	114.1	0.0024	31.24	52.07	73.31
	EM-256f	52,636	30,268	33	0	24,612	115	0.0024	4.23	7.05	11.28
	EM-256s	49,516	29,324	32.5	0	23,598	113.3	0.0024	25.65	42.76	68.41

Equivalent Slice (eSlice) = LUT/4 + FF/8 + DSP×102.4 + BRAM×232.4, following [AMD25].

Table 3 gives resource usage and related timing for all 12 complete designs. To have a comprehensive understanding of the overall resource usage of the proposed accelerator, we have used metric, equivalent Slice (eSlice), to represent the weighted overall area (footnote of Table 3), following [AMD25]. We also provide the resource breakdown for the proposed design for FAEST-128f (Table 4), for a better understanding of the overall area complexity.

Observation: configuration effects. Standard FAEST design proves AES with a secret key schedule: one block at L1 and two at L3/L5. EM variant design uses a public schedule and widens Rijndael to 192/256-bit blocks at L3/L5, omitting secret-schedule proof constraints. Short settings keep these widths but increase tree work: L1 leaf count rises from 3,072 to 22,528. Table 3 shows the resulting tradeoff across all 6 fast/short pairs: slightly smaller hardware, but longer Sign and Verify. EM variant reduces both times relative to standard FAEST at each matched category/setting.

Table 4: Module-level resource utilization of the complete standard FAEST L1-128f design at 138.6 MHz. Blocks correspond to those presented in Figure 5.

Module	**LUT**	**FF**	**BRAM**
OWF cipher	5,800	4,100	1
PRG / BAVC	7,900	2,800	3
SHAKE / transcript	6,100	3,300	1
VOLE / correlation state	2,900	1,500	9
OWF proof	6,500	3,500	2
VOLEHash / ZKHash	2,600	1,500	0.5
Operation controller and interfaces	1,140	720	0
Total	**32,940**	**17,420**	**16.5**

BRAM stores all required data for the standalone operation of our accelerator.

5.3 Comparison Consideration

To further evaluate the efficiency of the proposed accelerator, we consider the following factors: (i) FAEST is listed as one of the MPC-in-the-Head signature candidates in the NIST additional digital signature 3rd round standardization process [add], and hence, we need to compare its performance with its fellow signatures SDitH and MQOM; (ii) FAEST claims its design style follows the pattern of Picnic [KRR+19], except FAEST uses AES where Picnic uses LowMC as its OWF (in this case, we also need to include existing Picnic implementations into comparison); (iii) NIST expects the candidate from the additional digital signature scheme to be more efficient than the standardized SLH-DSA [add], and thus, we also include one existing SLH-DSA design (covers all parameter sets and all operational phases) for comparison; (iv) lastly, though there exist other digital signature hardware works like MAYO [HSK+23], we do not include them as they belong to different family of signature schemes and the direct comparison is out of the scope of this work.

Table 5: Complete designs: recent MPC-in-the-head SDitH (Tables 15–16 of [DHSY24], XC7A200T-3) and FAEST/FAEST-EM (this work). Times are ms; eADP uses weighted area; time: KeyGen + Sign + Verify. SDitH BRAM counts are used as reported. FAEST f/s denote fast/short. †Source Table 16 gives 149.99; Table 17 gives 117.76.

Cat.	Field/setting	LUT	FF	BRAM	DSP	MHz	KeyGen	Sign	Verify	eADP
SDitH [DHSY24]										
L1	GF256	16,592	8,778	164.5	0	164	0.34	41.04	52.98	4,102,306
	GF251	17,423	16,336	164.5	196	164	0.35	130.77	142.71	17,716,240
L3	GF256	22,569	13,881	356.0	0	164	0.28	47.17	66.85	10,299,776
	GF251	29,961	25,794	356.0	382	164	0.29	149.99	149.99†	39,805,503
L5	GF256	23,323	14,962	520.5	0	164	0.51	106.60	152.10	33,351,306
	GF251	34,456	31,409	521.5	472	164	0.53	276.07	183.57	83,782,933
FAEST / FAEST-EM (this work)										
L1	128f	32,940	17,420	16.5	0	138.6	0.00009	1.51	2.55	57,845
	128s	31,764	17,112	15.5	0	137.5	0.00009	9.43	15.92	346,845
	EM-128f	29,716	16,324	15	0	138.3	0.00009	1.33	2.24	46,252
	EM-128s	28,228	15,624	14	0	138	0.00009	6.65	11.23	219,274
L3	192f	43,820	24,180	25.5	0	125.9	0.0016	3.08	4.92	159,261
	192s	42,012	23,464	24.5	0	124.7	0.0016	18.24	29.14	906,401
	EM-192f	41,268	23,308	24	0	125.4	0.0016	1.92	3.07	93,883
	EM-192s	39,524	22,452	23.5	0	124.6	0.0016	10.78	17.23	508,380
L5	256f	54,760	31,880	34.5	0	115.4	0.0024	4.83	8.05	330,985
	256s	52,708	30,748	33	0	114.1	0.0024	31.24	52.07	2,056,958
	EM-256f	52,636	30,268	33	0	115	0.0024	4.23	7.05	277,679
	EM-256s	49,516	29,324	32.5	0	113.3	0.0024	25.65	42.76	1,614,362

5.4 Comparison with Existing Complete SDitH Design [DHSY24]

As the only complete SDitH design, Deshpande et al. implemented hypercube MPC-in-the-head SDitH [DHSY24]. Nevertheless, as shown in Table 5, the proposed FAEST

significantly outperforms all SDitH results when the same security level is considered. Applying the area weighting metric to the published counts shows that the standard FAEST-fast designs involve 98.45–99.67% lower eADP at corresponding categories. While the FAEST-EM-fast variants are 98.87– 99.7% lower eADP than [DHSY24] when the same security level is considered. These results sufficiently demonstrate the superior performance of the proposed design strategy and accelerator building.

5.5 Comparison with Existing SDitH Sign Design [HHH+26]

Table 6: Dedicated Sign part: SDitH-v2 [HHH+26] and FAEST/FAEST-EM (this work). HP0/HP1 denote HALF_POLY = 0/1. Delay is $1000C_{\mathrm{M}}/f$ ms. Area is Signer-only and is not comparable to the complete-design eADP in Table 7.

Scheme	Design	LUT	FF	BRAM	DSP	eSlice	MHz	Mcycles	Delay (ms)	eADP
SDitH-v2 [HHH+26]	L1-f HP0	24,851	9,373	2	1	7,952	145.3	3.58	24.6	195,916
	L1-f HP1	24,073	9,225	3	1	7,971	129.9	2.75	21.2	168,747
	L1-s HP0	24,324	9,524	2.5	1	7,955	119.6	15.19	127.0	1,010,326
	L1-s HP1	24,442	9,456	3	1	8,092	130	9.89	76.1	615,622
	L3-f HP0	32,201	12,001	3	1	10,350	135.9	7.46	54.9	568,144
	L3-f HP1	32,184	11,579	4	1	10,525	121.8	6.26	51.4	540,959
	L3-s HP0	32,844	11,615	3	1	10,462	143.3	58.05	405.1	4,238,288
	L3-s HP1	33,749	11,935	4	1	10,961	120.2	39.94	332.3	3,642,158
	L5-f HP0	39,331	15,026	4	1	12,743	126.7	11.03	87.1	1,109,355
	L5-f HP1	37,719	15,032	6	1	12,806	114.7	9.46	82.5	1,056,151
	L5-s HP0	39,793	14,853	5	3	13,274	124.7	81.93	657.0	8,721,291
	L5-s HP1	37,858	15,721	6	3	13,131	117.6	58.07	493.8	6,484,101
FAEST (this work)	L1-f	28,360	14,920	13.5	0	12,092	144.8	1.48	10.2	123,596
	L1-s	27,184	14,612	12.5	0	11,528	143.7	9.17	63.8	735,610
	L3-f	36,480	19,360	20.5	0	16,304	130.6	3.02	23.1	377,019
	L3-s	34,672	18,644	19.5	0	15,530	129.4	17.72	136.9	2,126,715
	L5-f	45,980	25,740	26.5	0	20,871	119.6	4.82	40.3	841,126
	L5-s	43,928	24,608	25	0	19,868	118.3	30.84	260.7	5,179,452
	L1-EM-s	23,648	13,124	11	0	10,109	144.2	6.49	45.0	454,971
	L1-EM-f	25,136	13,824	12	0	10,801	144.5	1.3	9.0	97,170
	L3-EM-s	32,184	17,632	18.5	0	14,549	129.2	10.46	81.0	1,177,916
	L3-EM-f	33,928	18,488	19	0	15,209	130.1	1.88	14.5	219,771
	L5-EM-s	40,736	23,184	24.5	0	18,776	117.4	25.13	214.1	4,019,045
	L5-EM-f	43,856	24,128	25	0	19,790	119.2	4.21	35.3	698,959

The Sign part of the proposed accelerator still stores all required data in BRAMs for standalone operation, while the existing design of [HHH+26] needs external host support to receive and feed in needed data for related operation and on-the-fly computation.

As shown in Table 6, SDitH HP1 uses extra scratchpad memory to reduce recomputation and has lower eADP than HP0 in every matched setting [HHH+26]. Against HP1, standard FAEST Signer's eADP (this work) is 26.8%, 30.3%, and 20.4% lower for L1-, L3-, and L5-fast, respectively. For short parameter settings, the proposed design is 19.5% higher at L1, but 41.6% and 20.1% lower at L3 and L5, respectively, than [HHH+26] (the larger eSlice at L1 comes from the fact that the proposed design stores all needed data in the accelerator to execute all necessary operations while the design of [HHH+26] needs external host support to feed in needed data for Sign operation). Therefore, from the perspective of both performance and design completeness, our design significantly outperforms [HHH+26].

5.6 Comparison with Existing L1 Security MQOM Design [MNKV26]

For a fair comparison with the existing single-security (L1) MQOM design of [MNKV26], we have specifically designed a FAEST-L1-EM-fast design (as presented in Section 4.9) to compare against it, following a similar on-the-fly computation strategy. Again, when a similar situation is applied, the proposed design significantly outperforms that of [MNKV26], as shown in Table 7. **Note** we have followed the exact same method of calculating the weighted area in [MNKV26], namely A_{LUT} (footnote of Table 7). The proposed design has lower ATP (Area-Time Product used in [MNKV26]) than all competing ones. This

Table 7: Results of L1 hardware: FAEST-EM-fast (proposed) and MQOM [MNKV26].

Design	LUT	FF	BRAM	DSP	A_{eq}	MHz	Total (ms)	eADP	A_{LUT}	ATP
FAEST-L1-EM-fast	19,780	13,939	4.5	0	7,733	334	0.6583	5,091	21,130	13.91
gf16-fast-r5-2	18,912	10,168	4.5	0	7,045	260	0.7515	5,294	20,262	15.23
gf16-fast-r5-1	15,812	9,384	4.5	0	6,172	260	0.8719	5,381	17,162	14.96
gf16-fast-r3-1	15,704	11,265	4.5	0	6,380	260	0.9427	6,014	17,054	16.08
gf16-fast-r3-2	19,578	11,825	4.5	0	7,418	260	0.8196	6,080	20,928	17.15
gf16-fast-r5-4	25,918	11,917	4.5	0	9,015	260	0.6897	6,218	27,268	18.81
gf16-fast-r3-4	26,077	12,968	4.5	0	9,186	260	0.7588	6,970	27,427	20.81
gf2-fast-r3-4	23,979	13,397	4.5	0	8,715	260	1.0963	9,554	25,329	27.77

Total delay is KeyGen + Sign + Verify (ms). $A_{\text{LUT}} = \#\text{LUT} + 300 \times \#\text{BRAM}$, following [MNKV26]. ATP = $(A_{\text{LUT}} \times \text{Total delay})/1000$, following [MNKV26].

showcases the efficiency of the proposed design strategy and accelerator building. Besides that, we would like to mention that the proposed FAEST-L1-EM-fast design is an optimized version derived from our complete acceleration work while [MNKV26] is a specific design that only covers the L1 security (the extension to other security levels remain unexplored). From this perspective, the proposed design also outperforms [MNKV26].

Meanwhile, MQOM, by nature, involves lower computational complexity (offers stronger performance) than FAEST, as indicated in the NIST report [3rd]. However, the performance listed in Table 7 demonstrates that FAEST outperforms MQOM, which again demonstrates the superior efficiency of the proposed acceleration strategy.

5.7 Comparison with Existing Picnic Designs [LJWJ24]

Table 8: Picnic Signers: Picnic [KRR+19] and Picnic3 [LJWJ24]. [KRR+19]'s DSP count is not supplied with its Signer resource rows, so its weighted area and eADP are omitted. [LJWJ24]'s MHz/delay preserve the source's rounding; derived eADP uses its cycle count and clock period (5.019/5.893 ns for L1/L5). A dash means unreported or inapplicable.

Scheme	Design	LUT	FF	BRAM	DSP	eSlice	MHz	Mcycles	Delay (ms)	eADP
Picnic [KRR+19]	L1-FS	76,472	21,061	52.5	–	–	125	≈ 0.0313	0.250	–
	L5-FS	149,456	30,441	98.5	–	–	125	≈ 0.1545	1.236	–
Picnic3 [LJWJ24]	L1 v2	134,964	42,556	65.5	0	54,283	199	0.009423	0.047	2,567
	L5 v2	198,186	65,594	159.5	0	94,814	170	0.024792	0.146	13,852

Both Picnic3 [LJWJ24] and Picnic [KRR+19] used the Kintex-7 FPGA board.

Table 8 compares Signer-only hardware with the existing Picnic designs. As seen from Table 6, standard FAEST L1-fast uses 28,360 LUTs and 13.5 BRAM. Picnic [KRR+19] uses 76,472 LUTs and 52.5 BRAM, while Picnic3 [LJWJ24] uses 134,964 LUTs and 65.5 BRAM. FAEST takes 10.2 ms to sign (Artix-7 device), versus 0.250 ms for Picnic and 0.047 ms for Picnic3 (Kintex-7). At L5-fast, FAEST likewise uses fewer resources than Picnic3. Overall, considering that the existing designs used advanced FPGA (Kintex-7) to obtain better frequency and the reported area usage cannot fit into the NIST-recommended Artix-7 device, we conclude that our design strategy is more balanced and efficient than those ones in [LJWJ24, KRR+19].

5.8 Comparison with Existing SLH-DSA Design [Saa24]

SLotH [Saa24]'s complete programmable system uses fewer LUTs than the complete FAEST designs, but includes 32 BRAM and lacks reported FFs, preventing weighted-area comparison [Saa24]. At L1-fast, it signs in 49.040 ms and verifies in 4.406 ms; standard FAEST takes 1.51 ms and 2.55 ms, respectively. At L1-short, SLotH verifies faster (1.796 versus 15.92 ms) but signs slower (1,023.467 versus 9.43 ms). As a result, the proposed FAEST design outperforms [Saa24]. Note that the other works of SLH-DSA like [DLK+25] and [HLL+25], either don't include the KeyGen or only cover the fast variant, we don't include them here for comparison as they do not cover all parameters and phases.

Table 9: SLotH complete system, SLH-DSA-SHAKE [Saa24]. All configurations share one system (RISC-V core, hash accelerators, 128 KiB main memory), so LUT, BRAM, and clock are listed once; DSP is 0. FF counts are unreported, so weighted area cannot be computed. Delays (ms) are derived from published cycles at 100 MHz.

Design	LUT	BRAM	MHz	KeyGen	Sign	Verify
128f	14,428	32	100	1.766	49.040	4.406
128s				111.806	1,023.467	1.796
192f				2.842	105.962	7.114
192s				180.389	2,631.008	2.898
256f				8.156	236.602	8.571
256s				130.037	2,962.655	4.700

5.9 Discussions

Overall, the above comparison exhibits higher area-time efficiency of the proposed design than the existing MPC-in-the-Head and related post-quantum signature schemes. Meanwhile, this hardware acceleration for FAEST also possesses other advantages such as covers all operational phases and can be configured to all parameter sets. The proposed design stores all needed data in the accelerator that it can operate in a standalone format.

The presented results also strengthen the promising future of FAEST in the NIST additional digital signature scheme competition as NIST also views implementation performance as an important factor towards a specific scheme's potential deployment [3rd].

6 Related Work and Future Directions

The FAEST construction [BBdSG+23] evolved through v2.0 [BBB+a] and FAESTer [BBB+26]. Memory-constrained software recovers correlations deterministically [ADE+25]; the latest software implementation results for FAEST are also available on the FAEST website [fae]. Improvements on FAEST signing and VOLE optimizations have been shown on software [LKS+26]. There may exist other works and we do not explicitly mention here.

Attacks against masked FAEST software show that cipher protection alone does not protect the proof [JD25]. Masking, leakage testing, and fault-injection resistance remain future work; smaller storage and repeated computation are not countermeasures.

The FAEST v3.0 [BBB+b] specification introduces changes that open several new and interesting research directions. First, proving AES with degree-7 constraints and a meet-in-the-middle strategy calls for a new constraint-generation datapath, whose area and latency trade-offs against the degree-3 design of v2.0 remain to be explored. Second, v3.0 replaces the commitment to the λ-bit VOLE masks with a Chinese Remainder Theorem (CRT)-based construction, motivating the design of an efficient, dedicated CRT unit. Third, we can extend our design strategy to other NIST third round additional signature MPC-in-the-Head candidates. Finally, side-channel and fault-attack countermeasures for the secret-dependent AES witness computation and the grinding loop are future work.

7 Conclusion

A complete and efficient hardware acceleration work for FAEST is proposed in this paper, covering KeyGen, Sign, and Verify for all twelve FAEST v2.0 parameter sets. With rigorous algorithmic analysis, we propose novel design strategies and algorithms for the identified computationally intensive operations with low hardware implementation complexity. Then, novel design efforts have been proposed to transfer these major operations into dedicated building blocks and also the whole accelerator while taking care of optimized workflow for all operational phases (including a dedicated extension to build an L1-EM-fast Sign accelerator). Finally, thorough evaluation has demonstrated the superior performance of the proposed work over the existing complete, Sign part, and single security level MPC-in-the-head signature acceleration designs. This is the first hardware acceleration work for FAEST, and we hope this work will attract many follow-up explorations.

References

[3rd] NIST IR 8610 Status Report on the Second Round of the Additional Digital Signature Schemes for the NIST Post-Quantum Cryptography Standardization Process. https://nvlpubs.nist.gov/nistpubs/ir/2026/NIST.IR.8610.pdf.

[add] Post-Quantum Cryptography: Additional Digital Signature Schemes. https://csrc.nist.gov/projects/pqc-dig-sig.

[ADE+25] Diego F. Aranha, Johan Degn, Jonathan Eilath, Kent Nielsen, and Peter Scholl. FAEST for memory-constrained devices. Cryptology ePrint Archive, Paper 2025/1261, 2025. Revised June 29, 2026.

[AMD25] AMD. 7 Series FPGAs Configurable Logic Block User Guide (UG474) (v1.9). https://docs.amd.com/r/en-US/ug474_7Series_CLB/7-Series-FPGA-CLB-Resources, April 2025.

[BBB+a] Carsten Baum, Ward Beullens, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. FAEST v2: Algorithm specifications. Version 2.0, NIST Round 2 submission.

[BBB+b] Carsten Baum, Ward Beullens, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. FAEST v3: Algorithm specifications. Version 3.0, NIST Round 3 submission.

[BBB+26] Carsten Baum, Ward Beullens, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. Shorter, tighter, FAESTer: Optimizations and improved (QROM) analysis for VOLE-in-the-head signatures. Cryptology ePrint Archive, Paper 2026/164, 2026. Revised version of the CRYPTO 2025 publication.

[BBdSG+23] Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head. Cryptology ePrint Archive, Paper 2023/996, 2023.

[DHSY24] Sanjay Deshpande, James Howe, Jakub Szefer, and Dongze Yue. Sdith in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2), 2024.

[DLK+25] Sanjay Deshpande, Yongseok Lee, Cansu Karakuzu, Jakub Szefer, and Yunheung Paek. Sphincslet: An area-efficient accelerator for the full sphincs+ digital signature algorithm. *ACM Transactions on Embedded Computing Systems*, 24(5):1–19, 2025.

[fae] FAEST. https://faest.info/.

[GJCJ23] Naina Gupta, Arpan Jati, Anupam Chattopadhyay, and Gautam Jha. Lightweight hardware accelerator for post-quantum digital signature crystals-dilithium. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(8):3234–3243, 2023.

[HHH+26] Stef Halmans, Niklas Höher, Dina Hesse, Sanjay Deshpande, Jakub Szefer, and Tim Güneysu. Storing less in-the-head: An area-efficient hardware architecture for sdith-v2. *Cryptology ePrint Archive*, 2026.

[HLL+25] Tianze Huang, Jiahao Lu, Dongsheng Liu, Zhixiang Luo, Chi Cheng, Aobo Li, Lei Chen, Shuo Yang, Jiaming Zhang, and Xiang Li. An efficient and reconfigurable post-quantum crypto-processor for sphincs+. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 72(5):2252–2262, 2025.

[HSK+23] Florian Hirner, Michael Streibl, Florian Krieger, Ahmet Can Mert, and Sujoy Sinha Roy. Whipping the MAYO signature scheme using hardware platforms. Cryptology ePrint Archive, Paper 2023/1267, 2023.

[JD25] Sönke Jendral and Elena Dubrova. Side-channel and fault injection attacks on VOLEitH signature schemes: A case study of masked FAEST. Cryptology ePrint Archive, Paper 2025/378, 2025. Revised January 20, 2026; publication record distinguishes the side-channel and revised fault-attack papers.

[KRR+19] Daniel Kales, Sebastian Ramacher, Christian Rechberger, Roman Walch, and Mario Werner. Efficient FPGA implementations of LowMC and Picnic. Cryptology ePrint Archive, Paper 2019/1368, 2019. Full version of the CT-RSA 2020 paper.

[LJWJ24] Guoxiao Liu, Keting Jia, Puwen Wei, and Lei Ju. High-performance hardware implementation of MPCitH and Picnic3. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):190–214, 2024.

[LKS+26] Seung-Won Lee, Ha-Gyeong Kim, Min-Ho Song, Si-Woo Eum, and Hwa-Jeong Seo. Accelerating faest signatures on arm: Neon simd aes and parallel vole optimization. *Applied Sciences*, 16(8), 2026.

[MNKV26] Stelios Manasidis, Quinten Norga, Suparna Kundu, and Ingrid Verbauwhede. Mq on my hardware: Performance analysis of mqom on fpga. *Cryptology ePrint Archive*, 2026.

[nis] Post-Quantum Cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography.

[Saa24] Markku-Juhani O. Saarinen. Accelerating SLH-DSA by two orders of magnitude with a single hash unit. In *Advances in Cryptology – CRYPTO 2024*, volume 14920 of *Lecture Notes in Computer Science*, pages 276–304. Springer, 2024.

[Sho94] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[TX26] Yazheng Tu and Jiafeng Xie. Leah: Lightweight and efficient hardware accelerator for code-based pqc scheme hqc. In *2026 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 34–44. IEEE, 2026.