Storing Less in-the-Head

An Area-Efficient Hardware Architecture for SDitH-v2

Stef Halmans[1⋆], Niklas Höher[1⋆], Dina Hesse[1], Sanjay Deshpande[3], Jakub Szefer[3], Tim Güneysu[1,2]

[1] Ruhr University Bochum, Bochum, Germany
firstname.lastname@rub.de
[2] German Research Center for Artificial Intelligence (DFKI), Bremen, Germany
[3] Northwestern University, Evanston, IL, USA
{sanjay.deshpande1, jakub.szefer}@northwestern.edu

Abstract. In 2022, the National Institute of Standards and Technology (NIST) initiated a second call for post-quantum digital signature proposals to broaden algorithmic diversity beyond the already standardized schemes based on structured lattices and hashes. While five out of the 14 second-round entries were built on the MPC-in-the-Head paradigm, only two advanced into the recently announced third round. Despite their theoretical appeal, available results on their implementation costs imply a reduced applicability to resource-constrained embedded devices.

In this work we analyze one of the remaining candidates, the revised second version of the Syndrome Decoding in the Head (SDitH) signature scheme based on the VOLE-in-the-Head framework, and showcase that these issues are not inherent to the family of algorithms. We propose an area-efficient hardware design including highly optimized variants of both Batch Line Commitment (BLC) and Polynomial Interactive Oracle Proof (PIOP) routines. As a result, we reduce memory requirements by a factor of 22 to 52 compared to the approach of the reference implementation and by a factor of 40 to 88 compared to the previous hardware implementation of SDitH-v1, while still achieving competitive runtimes. To the best of our knowledge, this yields the most compact hardware implementation of any MPCitH or code-based signature primitive, with our algorithmic improvements directly transferable to software designs.

During our analysis, we additionally determined nine significant discrepancies between specification and reference implementation of SDitH-v2, with a subset breaking test vector compatibility. Beyond addressing these issues, we suggest specification changes to further reduce resource usage on embedded targets without compromising security assumptions.

Keywords: Post-Quantum Cryptography · Digital Signatures · Hardware Implementation · VOLE-in-the-Head · Code-based Cryptography

⋆ Both authors contributed to this work equally.

1 Introduction

The advent of quantum computing presents a critical threat to classical public-key cryptosystems [1,2,3,4], prompting the U.S. National Institute of Standards and Technology (NIST) to initiate a global standardization effort for Post-Quantum Cryptography (PQC) algorithms [5]. In 2022, NIST announced the first set of future PQC standards: ML-KEM [6] as a key encapsulation mechanism, and ML-DSA [7], SLH-DSA [8], and FN-DSA [9] as digital signature schemes. While these selections provide strong quantum resistance, all except the hash-based SLH-DSA rely on structured lattice problems, limiting the diversity of underlying hardness assumptions. To address this, NIST launched an additional digital signature standardization process, aiming to increase algorithmic diversity and improve performance [10].

This ongoing process, now in its third round, includes nine candidate schemes, two of which are based on the Multi-Party Computation-in-the-Head (MPCitH) paradigm [11], namely MQOM [12] and SDitH [13]. In general, these schemes derive their security from zero-knowledge proofs, converting an interactive MPC protocol into a non-interactive signature by performing all phases *in the head*. This approach enables signatures based on classical symmetric primitives combined with a variety of hardness assumptions (e.g., coding-theoretic problems) as well-established alternatives to algebraic structures like lattices.

Among these candidates, the SDitH signature scheme relies on the Regular Syndrome Decoding (RSD) problem combined with the Vector Oblivious Linear Evaluation-in-the-Head (VOLEitH) framework [14] for efficient commitment generation and the PIOP formalism for construction of zero-knowledge proofs. While a hardware implementation of the round-1 submission exists [15], it unfortunately became obsolete due to comprehensive cryptographic design changes made in the round-2 submission. Thus, a complete reimplementation is required.

As for most asymmetric PQC schemes, the increased inherent algorithmic complexity poses a significant challenge. A diverse set of established cryptographic paradigms and formalisms is combined in novel ways to optimize performance metrics, such as signature size or public key size.

Specifications of such schemes tend to focus heavily on cryptographic clarity and associated security proofs while practical implementation constraints are only partially considered. Thus, implementers are faced with the difficult task of transforming theoretical constructs into efficient code. While existing reference implementations can help to provide additional insights into possible implementation strategies for certain platforms, demands of different real-world deployment scenarios are still not covered. In particular, designs for embedded devices require carefully engineered software and hardware in order to minimize code size, area cost, or runtime with limited available computational resources.

The difference between what is specified and what is efficiently implementable becomes particularly apparent when analyzing designs of MPCitH-based signature schemes in hardware and software. Within these, large internal data structures are commonly generated, sometimes requiring tens of Mbit of memory [15,16,17,18]. While this level of resource utilization might be acceptable for

powerful general-purpose devices, it is infeasible for small microcontrollers or embedded hardware platforms like smart cards. In this work, we therefore aim to find new trade-offs to enable such schemes on resource-constrained devices.

Contribution. We investigate the second version of the SDitH signature scheme as a representative of the class of MPCitH algorithms and develop an area-efficient hardware architecture for the signing procedure of the NIST additional signature candidate. As a result, we provide the following contributions:

(1) We present the first hardware implementation of the signature generation algorithm of the heavily modified second version of the SDitH scheme, available publicly at https://github.com/Chair-for-Security-Engineering/sdith-v2-hw.
(2) As such, we demonstrate a new practical lower bound for the memory requirement of MPCitH-based signature schemes in hardware that improves the state of the art by more than an order of magnitude. Most notably, we reduce memory usage compared to the only hardware implementation of SDitH-v1 by a factor of 40 to 88 while retaining competitive runtimes.
(3) To achieve this, we propose functionally equivalent and Known Answer Test (KAT) compliant improved algorithm designs for all major signature generation phases. Most of these systematically eliminate the need for storing large intermediate data structures in hardware and software implementations through different interleaving and recomputation strategies.
(4) During our analysis of the scheme, we identify nine discrepancies between the specification and the reference implementation, highlighting those that break test vector compatibility along with appropriate fixes.
(5) We further observe that the SDitH-v2 specification implicitly requires storage of large intermediate elements during several algorithmic stages, which the reference implementation mostly follows. We suggest changes to the specification that reduce resource usage without affecting security assumptions.

Related Work. Optimization of MPCitH-based schemes in both hardware and software is a highly active research area. We first focus on other works specifically targeting memory-optimized implementations before highlighting other recently proposed hardware designs for relevant signature schemes.

Concurrent work by Benadjila and Feneuil [18] proposes a memory-optimized software implementation of the MQ on my Mind (MQOM) signature scheme targeting embedded microcontrollers. While the scheme shares some algorithmic similarities with SDitH, e.g., the use of BLC or PIOP routines, their optimizations are not trivially applicable. As noted by the authors, VOLEitH-based schemes are inherently harder to optimize due to higher overall memory usage and complex PIOP structure. Furthermore, while both works rely on an optimized form of GGM tree caching, they choose a significantly different approach by caching not an entire layer at some depth, but a single path through the tree. This would be advantageous for hashing leaves in depth-first order during the initial stages of SDitH signature generation, but since we require non-sequential leaf access at all other stages of the algorithm, it mostly cannot be applied.

Similarly, Bettaieb et al. [17] present a memory-optimized software architecture for the PERK signature scheme. Their approach mainly relies on commitment streamlining and seed tree regeneration. While we also utilize recomputation during certain phases, our optimizations with regard to interleaved and redesigned algorithms go significantly beyond that. Furthermore, their work precedes SDitH-v2 and only considers SDitH-v1, which it excludes from analysis due to signature size and heap-memory usage. While these arguments no longer hold for SDitH-v2, the large structural differences, e.g., PERK not being VOLEitH-based and not using BLC or PIOP, make applicability questionable.

Moving to hardware designs, another concurrent work by Manasidis et al. [16] focuses on a relatively compact but still highly performance-oriented Mirath implementation that combines on-the-fly GGM tree expansion with massively parallel arithmetic units during the PIOP stage to achieve a significantly lower Area-Time Product (ATP) than previous MPCitH implementations. While the authors suggest that their optimizations could be transferable to SDitH-v2, this is largely not viable for our use case. Since we target a purely area-optimized design, choosing a highly parallelized PIOP phase is not feasible. Furthermore, their suggested optimizations for computation of a sibling path during the opening procedure are not efficiently transferable due to the significantly larger tree size of SDitH-v2. Only their GGM tree optimizations, structurally similar to the work on MQOM, remain applicable, which we use during the DFS stage.

Besides the outdated hardware implementation of the SDitH-v1 scheme by Deshpande et al. [15], other works investigated the previous NIST candidate Picnic [19] or the non-MPCitH-based round-2 candidate CROSS [20].

2 Preliminaries

In this section, we briefly introduce the individual stages of the SDitH-v2 signing procedure along with an overview of relevant parameters and intermediate values with their sizes. For more details, we refer to the specification document [13].

2.1 Overview of SDitH-v2 Signature Generation

The second version of the SDitH signature scheme derives its security from the RSD problem: the signer holds a secret witness, encoding a solution to an RSD instance, and produces a non-interactive zero-knowledge proof of knowledge of this witness using the PIOP formalism. To make this proof efficient, SDitH employs a VOLEitH-style commitment scheme in which a GGM tree is expanded using a PRG based on AES-128 or Rijndael-256-256, a soon-to-be-standardized [21] variant of the AES proposal [22]. While the choice depends on the security level, we use the terms interchangeably in this work. The correlated randomness generated by this process is then used in a BLC phase to commit to the witness and auxiliary polynomials. Subsequently, a PIOP phase proves to the verifier that the committed witness satisfies the RSD constraints. Revealing of necessary proof

Table 1: SDitH-v2 parameter sets and intermediate variable sizes.

(a) Parameters across security levels and fast / short signature variants.

RSD	**L1**	**L3**	**L5**
	(identical across fast/short)		
λ	128	192	256
n	10 360	18 396	19 864
$n-k$	432	592	800
w	56	73	104
$(\mu_1, \mu_2, \mu_3, \mu_4)$	$(4,4,4,3)$	$(4,4,4,4)$	$(4,4,4,3)$

VOLEitH	**L1**	**L3**	**L5**
	fast/short	fast/short	fast/short
τ	16 / 11	24 / 16	32 / 21
N	256 / 2048	256 / 4096	256 / 4096
$\kappa = \log_2 N$	8 / 11	8 / 12	8 / 12

$d = 4$, $B = 16$ for all configurations

(b) Approximate variable sizes across all configurations.

Variable	**L1**	**L3**	**L5**
	fast/short	fast/short	fast/short
GGM Tree			
leaf seeds	0.5/2.8 Mb	1.1/12 Mb	2/21 Mb
tree nodes	1/5.5 Mb	2.3/24 Mb	4/42 Mb
commitments	1/5.5 Mb	2.3/24 Mb	4/42 Mb
VOLE Data			
$\mathbf{V}$	143 kb	312 kb	546 kb
$\mathbf{r}_{\mathsf{base}}[e]$	8.9/12.3 kb	13/19.5 kb	17.1/25.6 kb
$\mathbf{u}$	1144 b	1664 b	2184 b
PIOP Data			
γ	512 b	768 b	1024 b
γ'	7 kb	13.7 kb	26 kb
$\mathbf{H}^{[\gamma]}$	1.3 Mb	3.4 Mb	4.9 Mb
$y^{[\gamma]}$	128 b	192 b	256 b

Variable	**L1**	**L3**	**L5**
	fast/short	fast/short	fast/short
Proof Polynomials			
P_{wit}	154 kb	328.5 kb	572 kb
P_{rnd}	96 kb	216 kb	384 kb
P_{α}	640 b	960 b	1280 b
Proof Data			
wit	616 b	876 b	1144 b
$\boldsymbol{\alpha}'_{\mathsf{plain}}$	144 b	208 b	272 b
$\boldsymbol{\alpha}'_{\mathsf{base}}$	18 kb	39 kb	68 kb
$\mathbf{M}$	140.6 kb	295.8 kb	508 kb
$\mathbf{H}'$	4.1 Mb	10.1 Mb	14.5 Mb

data happens in a final opening phase. The individual stages are cryptographically linked through a chain of hash values h_{com}, h_{aux}, h_{lines}, and h_{piop} generated using SHAKE, which is also used for expansion of Fiat–Shamir challenges.

This significantly deviates from the round-1 submission of SDitH which did not utilize the VOLEitH paradigm, operated over $\mathbb{F}_{256}$ or $\mathbb{F}_{251}$ instead of $\mathbb{F}_2$, and did not use Rijndael for the seed trees. As a result, SDitH-v2 offers much smaller signature sizes, more conservative security assumptions, and faster runtimes.

A list of the most notable parameters of the scheme is provided in Table 1a, while Table 1b contains an overview of the sizes of relevant intermediate values during signing to give an intuition regarding the memory requirements when implemented naïvely.

2.2 Goldreich–Goldwasser–Micali (GGM) Trees

A GGM tree is a binary tree that derives its pseudorandom leaf seeds from a single root seed by iteratively applying a Pseudorandom Generator (PRG) at each internal node. In the case of SDitH-v2, Rijndael is used for that. Originally, SDitH-v1 operated over τ independent GGM trees with N leaves each. Following

a proposal by Baum et al. [23], these were merged into a single tree of depth $\log_2(\tau \cdot N)$ for the round-2 submission. As noted in Table 1b, the size of the full tree can exceed 40 Mbit, making it a significant contributor to memory usage.

2.3 Batch Line Commitment (BLC)

The BLC phase expands the pseudorandom leaf seeds of the GGM tree into correlated random vectors (a VOLE correlation) that enable the prover to commit to its witness without revealing it in the following PIOP stage. This expansion of existing seeds is performed via AES in counter mode and results in the generation of a vector $\mathbf{u}$ and a matrix $\mathbf{V}$, which are used to derive the vector polynomials P_{wit} and P_{rnd} required as input to PIOP. A consistency check involving a random matrix $\mathbf{M}$ additionally verifies that all τ committed intermediate polynomials, used to create P_{wit} and P_{rnd}, share the same leading term, which is required by the VOLEitH framework (see [13], Section 3.2.5). Lastly, generation of hash values h_{aux} and h_{lines} ties this step into the full Fiat–Shamir chain.

2.4 Polynomial Interactive Oracle Proof (PIOP)

The PIOP protocol plays a central role in the SDitH signature generation as it is used to construct a zero-knowledge proof of knowledge of a witness $\mathsf{wit} \in \mathbb{F}_2^{|\mathsf{wit}|}$ that encodes the solution $\mathbf{x} = (e_0 \,||\, ... \,||\, e_{w-1})$ of the corresponding RSD instance, where $e_0, ..., e_{w-1}$ are elementary vectors of size $m = n/w$. To construct this witness, each of the elementary vectors is represented as a tensor product of $d = 4$ smaller elementary vectors with arities $\mu_1, ..., \mu_d$ and $\mu_1 \cdot ... \cdot \mu_d \geq m$. Afterward, the last coordinate of each individual vector is removed to further compress the witness to size $|\mathsf{wit}| = w \cdot \sum_{j=1}^{d}(\mu_j - 1)$. We refer to Table 1 for an overview of vector arities and resulting witness sizes for different security levels.

More precisely, this phase proves that wit satisfies a set of degree-d polynomial constraints f_j over $\mathbb{F}_2$ which ensure that (1) $\mathbf{H} \cdot \mathbf{x} - \mathbf{y} = 0$, and (2) each witness chunk of size m is an elementary vector.

Polynomial encoding. In order to allow embedding into the PIOP formalism, the previously committed vector polynomial $P_{\mathsf{wit}} \in (\mathbb{F}_{2^\lambda}[X])^{|\mathsf{wit}|}$ embeds each witness bit w_j into the constant term of a randomly sampled degree-1 polynomial. Additionally, the vector polynomial P_{rnd} contains $(d-1)\lambda$ degree-1 polynomial entries which are used to construct a masking polynomial of degree $(d-1)$ as

$$P_0(X) := \sum_{i=0}^{d-2} \left(\sum_{j=0}^{\lambda-1} \xi^j \cdot P_{\mathsf{rnd},i,j}(X) \right) \cdot X^i,$$

where $(\xi^0, \xi^1, ..., \xi^{\lambda-1})$ is an $\mathbb{F}_2$ basis of $\mathbb{F}_{2^\lambda}$.

Constraint batching. In order to increase the computational efficiency, both types of constraints are batched into a single polynomial P_α of degree d. While we refer to the specification document for additional mathematical details, this polynomial is computed using subroutines MuxTree and UnitaryCheck as

$$\begin{aligned} P_\alpha(X) = P_0(X) \cdot X \\ + \underbrace{\left(\sum_{i=0}^{w-1} \mathsf{MuxTree}\left(\left(h_{i\cdot m}^{[\gamma]}, ..., h_{(i+1)\cdot m-1}^{[\gamma]} \right), \left\{ P_{w_{i,j,k}}(X) \right\}_{j,k} \right) \right) - y^{[\gamma]}}_{\text{RSD constraint } \mathbf{H}\cdot\mathbf{x}-\mathbf{y}=0} \\ + \underbrace{\sum_{i=0}^{w-1} \gamma'_i \cdot \sum_{j=1}^{d} \xi^{32(j-1)} \cdot \mathsf{UnitaryCheck}\left(\mu_j, \left\{ P_{w_{i,j,k}}(X) \right\}_k \right)}_{\text{Ensure that all chunks are elementary vectors}}, \end{aligned}$$

where $\gamma \in \mathbb{F}_{2^\lambda}^{\lceil \frac{n-k}{\lambda} \rceil}$ and $\gamma' \in \mathbb{F}_{2^\lambda}^{w}$ are generated from SHAKE and act as randomizers, which would correspond to verifier challenges in the interactive variant of the zero-knowledge proof. Values $h^{[\gamma]}$ refer to columns $\mathbf{h}$ of the public key matrix $\mathbf{H}$, which have been batched into $\mathbb{F}_{2^\lambda}$ using the block-wise field-embedding isomorphism $\Phi : \mathbb{F}_2^{n-k} \to \mathbb{F}_{2^\lambda}^{\lceil (n-k)/\lambda \rceil}$ as $\boldsymbol{\gamma}^T \cdot \Phi(\mathbf{h}_j)$. Similarly, $y^{[\gamma]}$ corresponds to the field-embedding $\gamma^T \cdot \Phi(\mathbf{y})$ of the RSD syndrome $\mathbf{y}$.

MuxTree subroutine. Intuitively, the MuxTree is used to sum up all (preprocessed) columns of $\mathbf{H}$ corresponding to an index where the witness bit is 1, thus computing the constant coefficient of the resulting polynomial in a way which is directly correlated to the matrix-vector multiplication result $\mathbf{H} \cdot \mathbf{x}$. The first layer of nodes in the multiplexer tree is instantiated with the raw values $h^{[\gamma]}$, while nodes in the subsequent layers j with $1 \leq j \leq d$ are computed as

$$\mathsf{node}_{j,h} := \mathsf{node}_{j-1,\mu_j \cdot h} + \sum_{k=0}^{\mu_j - 2} P_{w_{i,j,k}}(X) \cdot \left(\mathsf{node}_{j-1,\mu_j \cdot h+(1+k)} - \mathsf{node}_{j-1,\mu_j \cdot h} \right).$$

The degree of the resulting polynomial nodes therefore increases in each layer as nodes of degree $j-1$ are multiplied by witness polynomials of degree 1. In the final layer, degree d is reached for the root $\mathsf{node}_{d,0}$ of the tree.

UnitaryCheck subroutine. In order to additionally verify that a given tensor factor with arity μ_j of the witness has at most one non-zero entry, a polynomial $D_{i,j} \in \mathbb{F}_2[Y]$ is computed from the witness bits for $1 \leq i \leq w$ and $1 \leq j \leq d$ as

$$D_{i,j}(Y) := \left(\sum_{k=0}^{\mu_j - 2} Y^k \cdot w_{i,j,k} \right) \cdot \left(\sum_{k=0}^{\mu_j - 3} Y^{(\mu_j - 1)\cdot k} \cdot w_{i,j,k} \right) - \left(\sum_{k=0}^{\mu_j - 3} Y^{\mu_j \cdot k} \cdot w_{i,j,k} \right),$$

such that $D_{i,j}(Y) = 0$ if and only if $w_{i,j,k} \cdot w_{i,j,k'} = 0$ for all $k \neq k'$. All resulting polynomials are subsequently batched into P_α so that all chunks can be verified simultaneously. If all chunks are valid, the batched constant term is again 0.

Output handling. The output of the PIOP phase consists of the coefficients of the degree-d polynomial P_α, which are part of the final signature and also hashed into h_{piop} together with the public key and the message. However, since $P_\alpha(0) = 0$ by construction for all honest provers, the constant term of the polynomial is omitted from both future computations and the signature.

2.5 Opening of a Random Evaluation

The final signing phase uses SHAKE to expand a set $\boldsymbol{\delta}$ consisting of τ leaf indices from the previously generated Fiat–Shamir transcript h_{piop} that should remain hidden as part of the Batched All-but-One Vector Commitment (BAVC) opening routine. The subsequently computed BAVC proof π_{BAVC} then includes seeds for the smallest set of GGM tree nodes required to recompute all non-hidden leaves without leaking information about the hidden leaves, for which the commitments are included instead. During verification, this enables efficient recomputation of all leaf commitments and thus generation of the commitment hash h_{com}.

Alongside $\boldsymbol{\delta}$, a grinding value $v_{\mathsf{pow}} \in \mathbb{F}_2^{w_{\mathsf{pow}}}$ is sampled, which is used as a proof-of-work to exponentially increase the soundness of the scheme. A candidate $\boldsymbol{\delta}$ can only be accepted if $v_{\mathsf{pow}} = 0$. Otherwise, a counter is incremented and new indices are sampled, therefore significantly increasing the expected amount of hashing that needs to be performed. A candidate is furthermore rejected if all sampled indices are zero, i.e., $\boldsymbol{\delta} = \mathbf{0}$, or if the required number of opened GGM tree nodes exceeds a pre-defined threshold T_{open}. In the following, we denote these conditions as (1) *grinding*, (2) *invertibility*, and (3) *compactness*.

3 Review of Specification and Reference Implementation

To enable our area-efficient hardware design of SDitH, we analyze the specification and its corresponding reference implementation and find several discrepancies between both, spanning from unspecified details to intentional design choices in the implementation that deviate from the algorithm description. In general, we decide to follow the reference code to ensure compatibility with KATs. Nonetheless, for each discrepancy, we systematically evaluate all observed variants to identify the approach best suited to low-memory hardware and software designs.

3.1 Inconsistencies in Matrix and Vector Ordering and Sampling

Matrix generation order. The specification assumes column-wise generation of the consistency-check matrix $\mathbf{M}$ from the SHAKE output, while the reference implementation squeezes it row-wise. For the subsequent computation of $\boldsymbol{\alpha}'_{\mathsf{base}} = [I_{n-k} \mid \mathbf{M}] \cdot \mathbf{V}$, row-wise sampling aligns naturally with the matrix-vector multiplication. Since $\mathbf{V}$ is generated column-wise, each produced row of $\mathbf{M}$ can be immediately consumed, avoiding the need to store parts of the matrix in memory. We therefore advise to adopt the row-wise approach.

Processing order of $\boldsymbol{\alpha}'_{base}$. The reference implementation generates the matrix $\boldsymbol{\alpha}'_{\mathsf{base}}$ in row-major order and writes the vector $\mathbf{r}_{\mathsf{base}}$ in column-major order. This combination is efficient in software, assuming that the full matrix can reside in memory. If this is not the case, however, this mismatch creates a bottleneck: since each entry of $\mathbf{r}_{\mathsf{base}}$ contributes only a partial row of $\boldsymbol{\alpha}'_{\mathsf{base}}$, the matrix cannot be consumed on the fly and must instead be stored in larger chunks before downstream processing can begin. While we follow the reference implementation for consistency, we note that a column-major processing order would resolve this limitation. With this tweak, each complete column could be consumed immediately upon generation, substantially reducing the memory footprint.

Sampling in PIOP. The expansion of batching challenges for the PIOP protocol (γ and γ') also relies on SHAKE, seeded with h_{lines}. While the specification describes sampling $w \cdot d'$ field elements from $\mathbb{F}_{2^\lambda}$ for γ', only w elements are actually used in the subsequent PIOP evaluation.

Moreover, the specification draws γ' before γ, despite the fact that γ is consumed first during protocol execution. The reference implementation corrects both inefficiencies: Sampling of γ is performed first and only w field elements are extracted for γ'. Since both vectors are squeezed sequentially from the same SHAKE XOF state, any change in sampling order or number of drawn elements alters the resulting challenge values, breaking compatibility across implementations. Thus, we adopt the more sensible variant used by the reference implementation. This choice further enables a fully streaming datapath in which each challenge entry from γ is produced by the SHAKE core on-demand when it is needed by the downstream PIOP logic.

3.2 Discrepancies in GGM Tree Handling

GGM leaf commitment order. The specification defines that leaves are committed and hashed in the same order as they are processed during BLC.Commit (see [13, Algorithm 8]). This allows leaf commitments to be computed directly within the main loop of the respective algorithm, eliminating an additional pass over the leaf set. The reference implementation, however, hashes them in chronological order instead, which corresponds to the sequence in which seeds are expanded during initial tree construction. In our design, we again follow the reference implementation to maintain compatibility with existing KATs. This requires an additional depth-first full tree traversal, as detailed in Section 4.3. We remark that following the specification would be beneficial in this case.

ExpandSeed and CommitSeed. For both seed expansion and commit generation, two AES calls are used to generate left and right parts of the output (see [13, Section 3.2.2 and Section 3.2.3]). While the specification defines the plaintext for generation of the right child as $\mathsf{salt} \oplus (x+1)$, where x refers to a small node counter, the reference implementation computes $(\mathsf{salt} \oplus x) + 1$ instead. This distinction matters in hardware, because the former only requires a small adder corresponding to the bit width of the tweak value x, while the latter necessitates

an adder of width λ. We adopt the version from the reference implementation for compatibility reasons, but suggest to consistently follow the specification.

3.3 Observed Differences in PIOP Definitions

Instantiation of $P_{rnd}(X)$. A major algebraic discrepancy between the specification and the reference implementation concerns the definition of $P_{\mathsf{rnd}}(X)$. The specification assigns each entry of the vector polynomial a single-bit constant term from $\mathbf{u}$ and a linear term consisting of an element from $\mathbb{F}_{2^\lambda}$, taken from rows of the matrix $\mathbf{V}$, yielding $P_{\mathsf{rnd}}(X) \leftarrow \mathbf{r}_{\mathsf{rnd}} + \mathbf{r}_{\mathsf{base,rnd}} \cdot X$. In the reference implementation, however, this assignment is reversed and $\mathbf{r}_{\mathsf{rnd}}$ serves as the source of the linear coefficients while $\mathbf{r}_{\mathsf{base,rnd}}$ is used for the constant terms. While analysis of the security implications of either choice is beyond the scope of this work, we note that both variants produce verifiable signatures, given that prover and verifier use the same convention. Intuitively, the version described in the specification seems to more accurately match the VOLEitH paradigm.

Bit ordering. The order in which bits are extracted from $\mathbf{V}$ and $\mathbf{u}$ for the random coefficients of both PIOP vector polynomials varies between both variants. While the specification extracts bits for $P_{\mathsf{wit}}(X)$ first, the reference implementation starts with $P_{\mathsf{rnd}}(X)$. Similarly, the specification defines generation of the parity check matrix $\mathbf{H}$ as $\mathbf{H} = [\mathbf{H}' \mid \mathbf{I}_{n-k}]$, while the reference implementation uses $\mathbf{H} = [\mathbf{I}_{n-k} \mid \mathbf{H}']$. As before, we follow the reference implementation. Neither ordering has a relevant impact on performance or memory utilization.

Possible omission of $y^{[\gamma]}$. We further note that the batched syndrome $y^{[\gamma]}$ within the PIOP phase only contributes to the constant term of $P_\alpha(X)$, which is zero by construction for an honest prover and does not appear in future computations or the final signature. Thus, it can be omitted entirely during signing without changing the resulting output. We utilize this observation within our design.

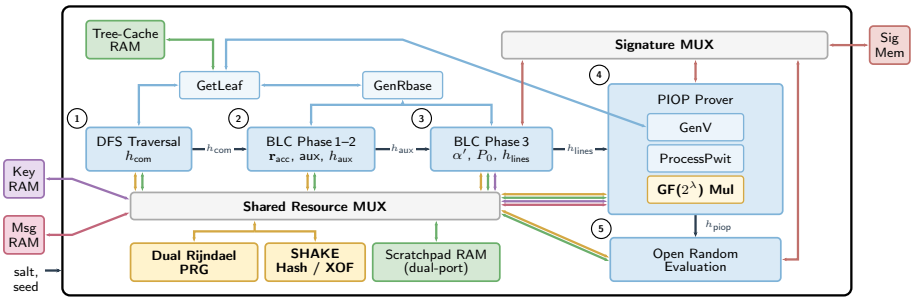
4 Hardware Architecture and Algorithm Design

As highlighted in Table 1b, several important intermediate values, like the GGM tree, the VOLE matrix $\mathbf{V}$, or the batched matrix $\mathbf{H}^{[\gamma]}$, dramatically exceed practical memory limits for embedded use cases. Thus, a multi-pass approach relying on partial computation and storage of relevant results is required, which differs significantly from the software reference implementation, where many values are stored in full (see comparison in Table 3).

In this section, we discuss details of our area-optimized hardware implementation and highlight the algorithmic improvements that are applied to reduce memory usage during signing. An overview of the individual modules and their interactions is provided in Figure 1, with the main stages of the signature algorithm being highlighted in blue and captioned with ① to ⑤.

We start by detailing the required symmetric primitives in Section 4.1 and general memory topology in Section 4.2. Afterward, the individual steps of the algorithm are discussed, starting with efficient GGM tree handling for DFS traversal and partial caching of intermediate nodes ① in Section 4.3. The following BLC stage is split into hardware modules ② and ③, responsible for execution up to h_{aux} and h_{lines} respectively, as discussed in Section 4.4. Efficient realization of the PIOP protocol ④ is described in Section 4.5, before concluding with an analysis of the random evaluation opening procedure ⑤ in Section 4.6.

Fig. 1: Hardware architecture of the SDitH-v2 signature algorithm.



4.1 Symmetric Primitives

The SDitH scheme relies heavily on the use of Rijndael for sampling of the parity check matrix $\mathbf{H}$, generation of seeds and commitments within the GGM tree, and expansion of leaf seeds into VOLE data. As a result, the cipher calls dominate the execution time of the signing procedure. Since our design requires multiple leaf traversals and repeated resampling of the VOLE data, we instantiate two round-based Rijndael instances, requiring 11 cycles for AES-128 and 15 for Rijndael-256-256. While this results in a certain area overhead, it massively reduces the time spent waiting for cipher outputs, effectively halving the total execution time compared to the instantiation of a single encryption core.

The SHAKE instance for hashing and expansion of Fiat–Shamir challenges is less critical since it is used less often and not during parts affected by our recomputation-based approach. Thus, we instantiate a standard design executing one Keccak permutation iteration per clock cycle, requiring 24 cycles for a full update of the 1600-bit internal state. A 64-bit streaming interface is used to easily interact with the chosen memory architecture and the module is shared across all relevant algorithm stages.

4.2 Memory Layout

In line with similar implementations, e.g., the original SDitH-v1 hardware implementation [15], we assume that input and output data of the algorithm, i.e.,

secret key, message, and signature, are stored in external memories, supporting read and write access on a single 64-bit wide data port.

Additionally, we instantiate two internal memories for (1) storing parts of the GGM tree to enable efficient recomputation of leaves (→ tree-cache memory), and (2) storing intermediate results during execution (→ scratchpad memory).

For the tree-cache memory, which stores a full layer of the GGM tree at configurable depth κ_c (see Section 4.3), we opt for a single-port configuration with 64-bit data width and fixed capacity of 32 kbit. This way, the memory always maps to a single Block RAM (BRAM) instance when targeting common Xilinx Field-Programmable Gate Array (FPGA) platforms. However, a discussion of potential performance gains by increasing the capacity is provided in Section 5.

By contrast, the size and required port configuration of the scratchpad memory depend on the security level and parameter set with all values provided in Table 2. This is done to avoid even higher numbers of recomputation passes for larger security levels as the size of critical intermediate values like the VOLE matrix $\mathbf{V}$ increases dramatically. As a result of this high variability, a single fixed scratchpad memory size for all configurations is infeasible. To allow for an additional area–latency trade-off that can be tuned independently for each parameter set (as shown in Table 4), we support two design points, referred to as HP0 (HALF_POLY=0) and HP1 (HALF_POLY=1), which control the number of recomputation passes required during the PIOP protocol.

While significant parts of the modified algorithms are designed around the availability of a 64-bit dual-port scratchpad memory, this constraint can be relaxed for certain address ranges where no concurrent read and write is required. Such a heterogeneous port configuration increases implementation complexity, but allows us to achieve a more efficient mapping to FPGA primitives since 64-bit dual port memories always necessitate the combination of two BRAMs, while a single port configuration can be implemented using just a single instance. Similarly, the port width can be reduced to 32 bits for security levels which have a lower memory demand to again avoid inefficient mapping to FPGA resources.

We provide a visualization of scratchpad memory utilization during different phases of the algorithm for one sample configuration in Figure 2. For HP0 variants, the BLC routine (denoted as Haux and Hlines) forms the memory bottleneck, while it shifts to the subsequent PIOP phase for the faster HP1 designs. Additional details on memory usage are included in the following sections.

4.3 Tree Handling and Efficient Partial Caching

Handling of the GGM tree in hardware is implemented using two distinct algorithms, namely Tree.InitCache (Algorithm 1) for initial leaf traversal, i.e., computation of h_{com}, as well as population of the tree-cache memory, and Tree.GetLeafPair (Algorithm 2) for efficiently computing a pair of leaf nodes from the cached data. Both routines share the two Rijndael cores described in Section 4.1 as well as the tree-cache and scratchpad memories.

Algorithm 1 traverses the tree using a Depth-First Search (DFS) approach starting from the root to hash all individual leaf commitments in the correct

Variant	Data Width	# Dual-port addr.	# Single-port addr.	# BRAMs
L1-f (HP0)	32 bit	462	—	1
L1-s (HP0)	32 bit	512	7	1.5
L1-f/s (HP1)	64 bit	937	—	2
L3-f/s (HP0)	64 bit	980	—	2
L3-f/s (HP1)	64 bit	1024	507	3
L5-f (HP0)	64 bit	1024	395	3
L5-s (HP0)	64 bit	1563	—	4
L5-f/s (HP1)	64 bit	2048	485	5

Table 2: Scratchpad memory data width and capacity for all configurations.

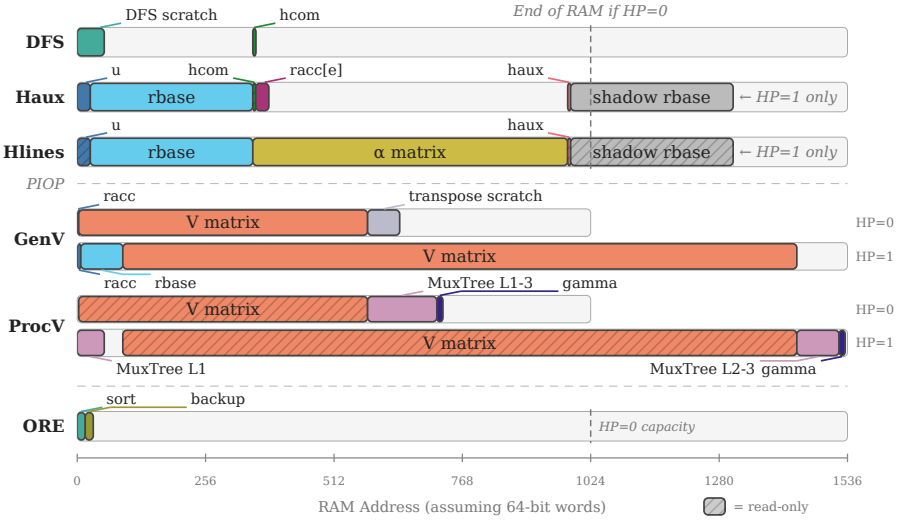


Fig. 2: Scratchpad memory layout across different execution stages for security level 3 short in both HP0 and HP1 configurations.

order. This traversal is non-trivial because the number of leaves $\tau \cdot N$ is not guaranteed to be a power of two, leading to leaves potentially sitting at two different depths. In order to configure the caching behavior, this algorithm takes a layer index κ_c as a synthesis-time parameter, which we set to 8 for security level 1, and to 7 for all others. During the descent towards the leaves, each node at that layer is inserted into tree-cache memory, so that the full layer is cached after the algorithm completes without any runtime overhead. Since computation of a single path towards a leaf generates a set of additional sibling nodes across all layers (produced by the ExpandSeed subroutine), we also store these in the scratchpad memory to avoid some recomputations during DFS backtracking. To determine where the DFS should resume after reaching a leaf, we count the trailing zeros of its index; this yields the nearest ancestor whose right subtree has not yet been visited. If the next leaf is a direct sibling of the current one, its seed is already available. Otherwise, a previously stored sibling at a higher layer

serves as the starting point for the next descent. The path from any intermediate node to a target leaf is determined directly from the binary representation of the leaf index, where each bit selects the left or right child at the corresponding tree level. Throughout the traversal, memory writes are interleaved with subsequent Rijndael computations to hide a portion of the cipher latency.

The TREE.GETLEAFPAIR algorithm highlights how the cached tree layer is used to accelerate retrieval of a leaf pair $(\mathsf{leaf}_1, \mathsf{leaf}_2)$. Similar to the previously described approach, the individual bits of the leaf index are used to determine the correct path through the tree. In this stage, only the child on the required path is computed without building up a sibling cache like during the DFS phase. Using the set of shared Rijndael cores, two leaves can be retrieved in parallel. Prefetching data from memory is additionally interleaved with Rijndael execution whenever applicable to further reduce latency.

Algorithm 1 TREE.INITCACHE(root seed rseed, salt)

Output: h_{com}, $\mathsf{cache}[0 \dots 2^{\kappa_c}-1]$ populated ▷ κ_c: cache depth

1: `SHAKE.Init(0x01)` ▷ h_{com} domain prefix
2: $p \leftarrow \mathsf{rseed}$; $\mathsf{idx} \leftarrow 1$; $\mathsf{depth} \leftarrow 0$
3: **for** each leaf t in DFS order **do**
  // Descend from current node
4: **while** $\mathsf{depth} < \mathsf{depth}_{\mathsf{leaf}}(t)$ **do**
5: $(c_L,\ c_R) \leftarrow \mathtt{ExpandSeed}(p,\ \mathsf{idx},\ \mathsf{salt})$ ▷ See [13], Alg. 5
6: $p \leftarrow$ child on path to t; $\mathsf{sib} \leftarrow$ other child
7: $\mathsf{idx} \leftarrow 2\,\mathsf{idx} + \mathtt{dir}(t, \mathsf{depth})$; $\mathsf{depth} \leftarrow \mathsf{depth} + 1$
8: **if** $\mathsf{depth} = \kappa_c$ **then**
9: $\mathsf{cache}[\mathsf{idx}-2^{\kappa_c}] \leftarrow p$; $\mathsf{cache}[(\mathsf{idx}\oplus 1)-2^{\kappa_c}] \leftarrow \mathsf{sib}$
  // Commit leaf
10: $(\mathsf{com}_L,\ \mathsf{com}_R) \leftarrow \mathtt{CommitSeed}(p,\ t,\ \mathsf{salt})$ ▷ See [13], Alg. 5
11: `SHAKE.Absorb`$(\mathsf{com}_L \parallel \mathsf{com}_R)$
  // Backtrack: Load sibling at appropriate depth for next leaf
12: $(p,\ \mathsf{idx},\ \mathsf{depth}) \leftarrow \mathtt{Backtrack}(t)$
13: $h_{\mathsf{com}} \leftarrow$ `SHAKE.Squeeze`(2λ)
14: **return** h_{com}

Algorithm 2 TREE.GETLEAFPAIR(leaf indices $(\mathsf{idx}_1, \mathsf{idx}_2)$, salt)

Output: $(\mathsf{leaf}_1, \mathsf{leaf}_2)$

1: **for** $x \in \{1, 2\}$ **do** ▷ AES_1 and AES_2 in parallel
2: $p_x \leftarrow \mathsf{cache}[\mathtt{ancestor}(\mathsf{idx}_x,\ \kappa_c)]$
3: $\mathsf{depth} \leftarrow \kappa_c$
4: **while** $\mathsf{depth} < \mathsf{depth}_{\mathsf{leaf}}(\mathsf{idx}_x)$ **do**
5: $p_x \leftarrow \mathtt{ExpandSeed}_{\mathsf{single}}(p_x,\ \mathsf{idx}_x,\ \mathsf{depth},\ \mathsf{salt})$ ▷ Compare [13], Alg. 5
6: $\mathsf{depth} \leftarrow \mathsf{depth} + 1$
7: **return** (p_1, p_2)

4.4 Multi-Pass Implementation of Batch Line Commitment

When analyzing the $\mathsf{BLC.Commit}$ algorithm as originally specified (see [13, Algorithm 8]), it is evident that both computation of h_{aux}, via the described folding phase, and generation of h_{lines} directly or indirectly depend on the random values $\mathbf{r}_{\mathsf{rnd}}$ which are expanded using the Rijndael-based PRG and serve as the basis for the auxiliary values aux and VOLE data $\mathbf{u}$ and $\mathbf{V}$.

However, as discussed previously, it is not feasible to store all of $\mathbf{V}$ in memory. Furthermore, the computation of h_{lines} from these values depends on the previous hash value h_{aux} through the consistency check matrix $\mathbf{M}$. Thus, interleaving of both phases to avoid storing $\mathbf{V}$ is not possible. As a result, multiple passes involving the randomness generation procedure are needed for a memory-optimized implementation. To apply this concept, we split the algorithm into two separate stages, involving only the computation of h_{aux} and h_{lines} respectively. Since both passes rely on the same inner randomness generation loop for computing a single entry of $\mathbf{r}_{\mathsf{base}}$ or aux, the shared logic is moved into a dedicated module, called GenerateRbase (Algorithm 3). In the following, we will discuss both stages of the commitment routine, as well as this sub-algorithm.

Shared generation of aux and $\mathbf{r}_{base}$ entries. The complete vectors aux and $\mathbf{r}_{\mathsf{base}}$ consist of $\tau - 1$ and τ entries respectively. In order to avoid storing all at once, this module computes just one of the entries, corresponding to one *repetition* e, which requires iterating over all N leaves of the GGM sub-tree with index e to expand and accumulate the correlated VOLE randomness.

The generation of leaves uses the previously described Tree.GetLeafPair module (Algorithm 2) to utilize both Rijndael cores. One of the resulting leaves is immediately processed, while the other one is buffered until the current leaf iteration is finished. Similarly, expansion of $\mathbf{r}_{\mathsf{rnd}}$ uses both cipher instances in parallel to speed up operation, which is feasible due to the use of counter mode.

While accumulation into the vector $\mathbf{r}_{\mathsf{acc}} \in \mathbb{F}_2^L$, with $L = |\mathsf{wit}| + (d-1)\lambda + (\lambda + B)$, happens via a trivial XOR between the cipher outputs and corresponding chunks in memory, generation of the matrix $\mathbf{r}_{\mathsf{base}} \in \mathbb{F}_2^{L \times \kappa}$ is more complex. If implemented naïvely according to [13, Algorithm 8], this algorithm involves accumulating the cipher outputs into all columns where the corresponding bits of the Gray code value of the leaf index $\phi_{\mathsf{Gray}}(i)$ are 1. Therefore, multiple columns are updated per leaf iteration. To improve this, we rearrange the formula as

$$\mathbf{r}_{\mathsf{base}}[e] = \sum_{i=0}^{N-1} \mathbf{r}_{\mathsf{rnd},i} \cdot \phi_{\mathsf{Gray}}(i) = \sum_{i=0}^{N-1} \mathbf{r}_{\mathsf{acc},i} \cdot (\phi_{\mathsf{Gray}}(i) \oplus \phi_{\mathsf{Gray}}(i+1)) \,,$$

which follows the optimization technique detailed in [13, Section 3.2.5]. This way, only one column needs to be accessed per iteration since $\mathsf{HW}(\phi_{\mathsf{Gray}}(i) \oplus \phi_{\mathsf{Gray}}(i+1)) = 1$ by definition. The index of this column can be efficiently determined by extracting the position of the single 1-bit by counting the number of trailing zeros, which is a routine we abbreviate as ctz in the following sections.

Each column of $\mathbf{r}_{\mathsf{base}}[e]$ is then logically split into two regions, corresponding to the first $\lambda + B$ bits, which remain unchanged during processing of the

consistency check as $\boldsymbol{\alpha}'_{\mathsf{base}} = [\mathbf{I}_{\lambda+B} \mid \mathbf{M}] \cdot \mathbf{V}$, and the remaining bits used for multiplication with $\mathbf{M}$. Since $\lambda+B$ is not divisible by 64, we insert some padding to ensure that both partitions start at a memory word boundary, which significantly simplifies later word-based processing during the h_{lines} stage. Without this, extracting a single 64-bit chunk for the matrix multiplication during the consistency check would require multiple memory reads.

Lastly, during all computation steps, cipher executions are interleaved with read-modify-write operations in order to hide parts of the Rijndael latency.

Generation of h_{aux}. The hash value h_{aux} is computed by Algorithm 4, which is invoked after the initial DFS traversal has finished producing h_{com} and the tree-cache memory is therefore already populated. h_{aux} then depends on the $\tau - 1$ aux values, each obtained by XOR-ing the first accumulated share $\mathbf{r}_{\mathsf{acc}}[0]$, also referred to as $\mathbf{u}$, with a subsequent share $\mathbf{r}_{\mathsf{acc}}[e]$. To avoid regenerating $\mathbf{u}$ on every iteration, the value produced by the initial call to GenerateRbase (Algorithm 3) is stored persistently in scratchpad memory (see Section 4.2). A simple loop then invokes GenerateRbase once per remaining share to produce the corresponding $\mathbf{r}_{\mathsf{acc}}[e]$ and, by extension, the auxiliary value $\mathsf{aux}[e]$. Each of these values is written to signature memory and fed into a running SHAKE instance that also absorbed the domain-separation byte 0x02 and h_{com}, yielding h_{aux} upon finalization.

To avoid redundant tree traversals during the subsequent h_{lines} phase, we fill all remaining scratchpad capacity with the first $E_{\mathsf{pre}} + 1$ entries of $\mathbf{r}_{\mathsf{base}}$. Under the HP0 configuration, $E_{\mathsf{pre}} = 0$ for all parameter sets. Under HP1, E_{pre} is set to 2 for security levels 1 and 5, and to 1 for security level 3.

Algorithm 3 GenerateRbase(e, build_rbase, output targets ($\mathbf{r}_{\mathsf{acc}}$, $\mathbf{r}_{\mathsf{base}}$))

$\mathbf{r}_{\mathsf{acc}} \leftarrow 0$
$\mathsf{leaf}_{\mathsf{buf}} \leftarrow \perp$
for i from 0 to $(N-1)$ **do**
  $\mathsf{idx} \leftarrow i \cdot \tau + e$
  if $\mathsf{leaf}_{\mathsf{buf}} \neq \perp$ **then**
    $\mathsf{leaf} \leftarrow \mathsf{leaf}_{\mathsf{buf}}$; $\mathsf{leaf}_{\mathsf{buf}} \leftarrow \perp$
  else
    $(\mathsf{leaf}, \mathsf{leaf}_{\mathsf{buf}}) \leftarrow$ Tree.GetLeafPair(idx, $\mathsf{idx} + \tau$)
    if $i + 1 \geq N$ **then**
      $\mathsf{leaf}_{\mathsf{buf}} \leftarrow \perp$
  $\mathbf{r}_{\mathsf{rnd}} \leftarrow$ AES-CTR($\mathsf{key} = \mathsf{leaf}$, $\mathsf{ctr} = 0$, $\mathsf{len} = L$) ▷ Uses both AES cores
  $\mathbf{r}_{\mathsf{acc}} \leftarrow \mathbf{r}_{\mathsf{acc}} \oplus \mathbf{r}_{\mathsf{rnd}}$ ▷ Accumulate in scratchpad memory
  if build_rbase **then**
    $j \leftarrow \mathtt{ctz}(\phi_{\mathsf{Gray}}(i) \oplus \phi_{\mathsf{Gray}}(i{+}1))$
    $\mathbf{r}_{\mathsf{base}}.\mathsf{col}[j] \leftarrow \mathbf{r}_{\mathsf{base}}.\mathsf{col}[j] \oplus \mathbf{r}_{\mathsf{acc}}$ ▷ Accumulate in scratchpad memory

Algorithm 4 BLC.Commit.Haux(h_{com})

GenerateRbase(0, **true**, $\mathbf{u}$, $\mathbf{r}_{\mathsf{base}}[0]$) ▷ $\mathbf{u}$ persists for all iterations
for e from 1 to $\tau - 1$ **do**
  if $e \leq E_{\mathsf{pre}}$ **then**
    GenerateRbase(e, **true**, $\mathbf{r}_{\mathsf{acc,tmp}}$, $\mathbf{r}_{\mathsf{base}}[e]$) ▷ Precompute for h_{lines}
  else
    GenerateRbase(e, **false**, $\mathbf{r}_{\mathsf{acc,tmp}}$, $\perp$)
  $\mathsf{aux}[e] \leftarrow \mathbf{u} \oplus \mathbf{r}_{\mathsf{acc,tmp}}$ ▷ $\mathbf{r}_{\mathsf{acc,tmp}}$ is overwritten each iteration
$h_{\mathsf{aux}} \leftarrow \mathtt{SHAKE}(\mathtt{0x02} \parallel h_{\mathsf{com}} \parallel \mathsf{aux}[1] \parallel \cdots \parallel \mathsf{aux}[\tau-1])$

Generation of h_{lines}. After computation of h_{aux}, the second phase of the BLC process (Algorithm 5) produces the plain consistency check result $\boldsymbol{\alpha}'_{\mathsf{plain}}$ and the masked witness Δ_{wit} which are direct parts of the resulting signature. Along with the consistency check output matrix $\boldsymbol{\alpha}'_{\mathsf{base}}$, these values form the inputs for generation of the h_{lines} hash using the shared SHAKE instance.

Algorithm 5 BLC.Commit.Hlines(h_{aux}, $\mathbf{u}$, $\mathbf{r}_{\mathsf{base}}[0..E_{\mathsf{pre}}]$)

$\Delta_{\mathsf{wit}} \leftarrow \mathbf{u}_{\mathsf{wit}} \oplus \mathsf{wit}$ ▷ $\mathbf{u}_{\mathsf{wit}}$: wit part of $\mathbf{u}$, Δ_{wit} stored in signature memory
InitP0($\mathbf{u}_{\mathsf{rnd}}$) ▷ $\mathbf{u}_{\mathsf{rnd}}$: P_0 part of $\mathbf{u}$
UpdateP0($\mathbf{r}_{\mathsf{base}}[0]$, 0)
ProcessRbase($\mathbf{r}_{\mathsf{base}}[0]$, $\mathbf{u}$, 0, h_{aux})
for e from 1 to $\tau - 1$ **do**
  if $e \leq E_{\mathsf{pre}}$ **then** ▷ Precomputed, no leaf retrieval
    UpdateP0($\mathbf{r}_{\mathsf{base}}[e]$, e)
    ProcessRbase($\mathbf{r}_{\mathsf{base}}[e]$, $\perp$, e, h_{aux})
  else ▷ Leaf retrieval required
    GenerateRbase(e, **true**, $\mathbf{r}_{\mathsf{acc,tmp}}$, $\mathbf{r}_{\mathsf{base}}[e]$)
    UpdateP0($\mathbf{r}_{\mathsf{base}}[e]$, e)
    ProcessRbase($\mathbf{r}_{\mathsf{base}}[e]$, $\perp$, e, h_{aux})
$h_{\mathsf{lines}} \leftarrow \mathtt{SHAKE}(\mathtt{0x03} \parallel h_{\mathsf{aux}} \parallel \boldsymbol{\alpha}'_{\mathsf{plain}} \parallel \boldsymbol{\alpha}'_{\mathsf{base}} \parallel \Delta_{\mathsf{wit}})$

The specification prescribes computing the PIOP input polynomials $P_{\mathsf{wit}}(X)$ and $P_{\mathsf{rnd}}(X)$ from $\mathbf{V}$, $\mathbf{u}$, as well as the witness wit during this phase and storing them for later use. This approach is infeasible in a memory-optimized design because the vector polynomials are prohibitively large (see Table 1b). We therefore defer the generation of $P_{\mathsf{wit}}(X)$ to the PIOP computation itself, where it is produced incrementally during a chunked pass over $\mathbf{V}$. We do, however, advance the computation of the PIOP masking polynomial $P_0(X)$ into this stage. Since $P_0(X)$ depends only on $P_{\mathsf{rnd}}(X)$, and hence on $\mathbf{V}$ and $\mathbf{u}$, it can be evaluated during the first generation pass over $\mathbf{V}$. The resulting coefficients are written directly into the $P_\alpha(X)$ slots in signature memory, performing the initialization that would otherwise occur during the PIOP phase (in [13, line 4 of Algorithm 17]). This reordering eliminates the need to either store or recompute $P_{\mathsf{rnd}}(X)$.

In our hardware implementation, we first compute Δ_{wit} from the value $\mathbf{u}$, residing in scratchpad memory, and the value wit, which is retrieved from the

external key memory. The result is immediately written to signature memory. Afterward, the consistency check matrix multiplication $\boldsymbol{\alpha}'_{\mathsf{base}} = [\mathbf{I}_{\lambda+B} \mid \mathbf{M}] \cdot \mathbf{V}$ is performed in a loop that processes κ columns of $\mathbf{V}$, i.e., one value $\mathbf{r}_{\mathsf{base}}[e]$, at a time using the ProcessRbase (Algorithm 6) subroutine. Within this algorithm, each row of $\mathbf{M}$, which is expanded from SHAKE, is multiplied with each of the κ columns of $\mathbf{r}_{\mathsf{base}}[e]$ to produce κ output columns of the matrix $\boldsymbol{\alpha}'_{\mathsf{base}}$. Since the first $E_{\mathsf{pre}} + 1$ entries of $\mathbf{r}_{\mathsf{base}}$ are already stored in scratchpad memory, they do not need to be recomputed. All other entries are generated analogously to the h_{aux} phase using GenerateRbase. To avoid having to store the full matrix $\mathbf{M}$, it is re-expanded from SHAKE in each of the ProcessRbase iterations. One additional regeneration of $\mathbf{M}$ is prevented, by folding the computation of $\boldsymbol{\alpha}'_{\mathsf{plain}} = [\mathbf{I}_{\lambda+B} \mid \mathbf{M}] \cdot \mathbf{u}$ into the first ProcessRbase call.

In our hardware design, each inner product with a row of $\mathbf{M}$ is computed incrementally in 64-bit chunks as SHAKE outputs are progressively squeezed, avoiding full row buffering. Using both scratchpad memory read ports, two $\mathbf{r}_{\mathsf{base}}$ words can be read simultaneously, further reducing per-row processing latency.

In parallel, the entries $\mathbf{r}_{\mathsf{base}}[e]$ are consumed for on-the-fly generation of $P_0(X)$ using the InitP0 and UpdateP0 routines (Algorithm 7). This is, however, not trivially possible without rearranging of the involved equations. As detailed in Section 3, the reference implementation does not follow the specification regarding construction of the polynomials $P_{\mathsf{rnd},i}$. As implemented, each entry in the vector polynomial consists of a constant term in $\mathbb{F}_2^\lambda$ taken from rows of the matrix $\mathbf{V}$, while the linear term is populated from the vector $\mathbf{u}$. Since we generate $\mathbf{V}$ column-wise and not row-wise, we write its contribution to $P_0(X)$ as:

$$
\begin{aligned}
P_0^{(\mathbf{V})}(X) &= \sum_{i=0}^{d-2} \left(\sum_{j=0}^{\lambda-1} \xi^j \cdot P_{\mathsf{rnd},i,j}(0) \right) \cdot X^i \\
&= \sum_{i=0}^{d-2} \left(\sum_{j=0}^{\lambda-1} \xi^j \cdot \left(\sum_{e=0}^{\tau-1} \sum_{t=0}^{\kappa-1} \mathbf{V}[\mathsf{cchk} + i \cdot \lambda + j][e \cdot \kappa + t] \cdot \xi^{e \cdot \kappa + t} \right) \right) \cdot X^i \\
&= \sum_{i=0}^{d-2} \left(\sum_{j=0}^{\lambda-1} \xi^j \cdot \left(\sum_{e=0}^{\tau-1} \sum_{t=0}^{\kappa-1} \mathbf{r}_{\mathsf{base}}[e][\mathsf{cchk} + i \cdot \lambda + j][t] \cdot \xi^{e \cdot \kappa + t} \right) \right) \cdot X^i \\
&= \sum_{e=0}^{\tau-1} \sum_{t=0}^{\kappa-1} \xi^{e \cdot \kappa + t} \cdot \left(\sum_{i=0}^{d-2} \left(\sum_{j=0}^{\lambda-1} \xi^j \cdot \mathbf{r}_{\mathsf{base}}[e].\mathsf{col}[t][\mathsf{cchk} + i \cdot \lambda + j] \right) \cdot X^i \right)
\end{aligned}
$$

We begin by substituting $P_{\mathsf{rnd},i,j}(0)$ with its bit-level definition, expressing each term as a field-element extraction from the rows of $\mathbf{V}$, where cchk refers to the offset of the beginning of the P_{rnd} section of the matrix. Next, we exploit the fact that $\mathbf{V}$ is the concatenation of the matrices $\mathbf{r}_{\mathsf{base}}[e]$, each comprising κ columns, to decompose the expression across sub-matrices. Finally, we reorder the summations so that the computation proceeds column by column, yielding Algorithm 7. We note that multiplications by the polynomial basis element ξ reduce to a sequence of computationally cheap shifts and XOR operations.

Algorithm 6 ProcessRbase($\mathbf{r}_{\mathsf{base}}$, $\mathbf{u}$, e, h_{aux})

```
SHAKE.Absorb(h_aux)                                   ▷ Re-initialized each iteration
if e = 0 then
    α'_plain ← 0;   α'_base ← 0
for row from 0 to λ + B − 1 do
    M[row] ← SHAKE.Squeeze(ℓ)                          ▷ ℓ = L − (λ + B)
    for k from 0 to κ − 1 do
        α'_base[row, e·κ+k] ← r_base.col[k][row] ⊕ M[row] · r_base.col[k][λ+B : L−1]
    if e = 0 then                  ▷ Fold α'_plain computation into first iteration
        α'_plain[row] ← [ I_{λ+B} | M[row] ] · u
```

Algorithm 7 P_0 Computation

```
procedure InitP0(u_rnd ∈ {0,1}^{(d−1)λ})
    P_0[0] ← 0                          ▷ Implicitly zero; not stored in signature
    for j = 1 to d − 1 do
        P_0[j] ← Σ_{t=0}^{λ−1} u_rnd[(j−1)λ + t] · ξ^t
procedure UpdateP0(r_base, repetition index e)
    for t = 0 to κ − 1 do
        for i = 0 to d − 2 do
            s ← Σ_{j=0}^{λ−1} r_base.col[t][cchk + iλ + j] · ξ^j        ▷ Polynomial basis
            P_0[i] ← P_0[i] ⊕ s · ξ^{e·κ+t}
```

4.5 Memory-Optimized Chunked PIOP

In both specification and reference software implementation, BLC and PIOP phases happen strictly sequentially. Thus, the full matrix $\mathbf{V}$ (see Section 4.4) needs to be stored since it is required for subsequent generation of the P_0 and P_{wit} polynomials. As detailed previously, this is not feasible for memory-constrained embedded devices so we opt for a multi-pass approach instead. Due to the early $P_0(X)$ computation within the BLC stage, the P_α coefficient accumulators in signature memory are already initialized accordingly. The subsequent generation of P_{wit} for MuxTree and UnitaryCheck evaluation relies on a chunked recomputation of $\mathbf{V}$ during the PIOP phase. While the individual computational paths and differences between HP0 and HP1 are described in the following sections, a general overview of the PIOP implementation is provided in Algorithm 8.

Algorithm 8 Interleaved PIOP.Prover.Run (replaces Algorithm 17, [13])

▷ BLC.Commit.Hlines writes P_0 to P_α slot of signature memory

1: **for** each $\mathbf{V}$ chunk p **do**
2: GenerateV(index $= p$, mode $=$ HP) ▷ Generate subset of $\mathbf{V}$ rows
3: TransposeV ▷ Convert from column-major to row-major order
4: ProcessPwit(index $= p$) ▷ Evaluate MUX instances and update P_α
5: $h_{\mathsf{piop}} \leftarrow \mathtt{SHAKE}(\mathtt{0x04} \parallel \mathsf{pk} \parallel h_{\mathsf{lines}} \parallel \alpha_1, ..., \alpha_d \parallel \mathsf{msg})$

The intermediate transpose step is required since the matrix $\mathbf{V}$ is sampled column-wise from the Rijndael-based PRG but is consumed row-wise to construct the P_{wit} polynomials. Implementation of this operation varies between the different configurations with HP0 relying on an in-place transpose algorithm after generation itself has finished, while HP1 efficiently interleaves it with matrix generation. The latter is necessitated by the increased size of the stored chunk of $\mathbf{V}$, which would require a larger amount of scratch space in order to remain reasonably efficient during transposition.

Generation of P_{wit} Polynomials In order to iteratively generate all degree-1 P_{wit} polynomials from rows of the matrix $\mathbf{V}$ and the respective witness bits, Algorithm 9 is used. In both HP0 and HP1 cases, it generates a subset of consecutive matrix rows from the Rijndael-based PRG and stores them in scratchpad memory. These directly form the linear terms of the corresponding polynomials.

In order to achieve the lowest possible memory footprint, HP0 generates at most λ rows at once. This ensures that the matrix chunk fits into the existing memory, whose size is determined by the amount of storage required for $\boldsymbol{\alpha}'_{\mathsf{base}}$ during generation of h_{lines} (see Figure 2). However, this smaller chunk size results in multiple GenerateV iterations being required, which heavily increases execution time since each iteration requires full GGM leaf traversal and subsequent Rijndael-based randomness expansion. To mitigate some of the runtime overhead, both Rijndael cipher cores operate in parallel for randomness expansion and memory read–write operations are interleaved with cipher invocations.

In HP1 configurations, the chunk size is increased to half of the required rows for P_{wit}, therefore resulting in a fixed number of two iterations at the cost of higher memory consumption. In this case, the PIOP stage becomes the peak-memory phase of the signing algorithm (see Figure 2), translating to additional BRAM instances on FPGA targets (see Section 5).

Algorithm 9 GenerateV(index, mode)

for $e = 0, \ldots, \tau - 1$ **do**
  Clear $\mathbf{r}_{\mathsf{acc}}$ and $\mathbf{r}_{\mathsf{base}}[0\,..\,\kappa{-}1]$ in memory
  for $k = 0, \ldots, N/2 - 1$ **do**
    Compute leaf pair $(\mathsf{seed}_L, \mathsf{seed}_R)$ using GGM cache
    for $h \in \{0, 1\}$ **do**
      // Extract correct chunk size of bits from the PRG stream
      $\mathsf{prg} \leftarrow$ AES-CTR(seed_h, start offset for chunk p)
      $\mathsf{col} \leftarrow \mathrm{CTZ}(2k + h + 1)$ ▷ Gray-code via count-trailing-zeros
      $\mathbf{r}_{\mathsf{acc}} \oplus{=} \mathsf{prg}$
      $\mathbf{r}_{\mathsf{base}}[\mathsf{col}] \oplus{=} r_{\mathsf{acc}}$
  Write $\mathbf{r}_{\mathsf{base}}[0\,..\,\kappa{-}1]$ to $\mathbf{V}$ region for repetition e in memory

Algorithm 10 ProcessPwit(index)

if index $= 0$ **then** ▷ First chunk of $\mathbf{V}$: Initialize persistent state
  Squeeze $\gamma \in \mathbb{F}_{2^\lambda}^{\lceil (n-k)/\lambda \rceil}$ from $\mathsf{SHAKE}(h_{\mathsf{lines}})$ ▷ Stored in memory
  Initialize H' column stream (AES-CTR from $\mathsf{seed}_{\mathsf{pk}}$)

for each witness chunk i in the currently processed range of $\mathbf{V}$ **do**
  Squeeze $\gamma'_i \in \mathbb{F}_{2^\lambda}$ from XOF ▷ On-demand, not precomputed
  Extract witness bits $\boldsymbol{b}_i = (\boldsymbol{b}_{i,1}, \ldots, \boldsymbol{b}_{i,d})$ from key memory ▷ $\boldsymbol{b}_{i,j} \in \mathbb{F}_2^{\mu_j - 1}$
  // Layer-1 MuxGate with on-the-fly $h^{[\gamma]}$ generation
  for $h = 0, \ldots, \lceil m/\mu_1 \rceil - 1$ **do**
    Generate and store $h_c^{[\gamma]}$ for columns $c \in [i{\cdot}m{+}\mu_1 h,\ i{\cdot}m{+}\mu_1(h{+}1){-}1]$
    MuxGate($j{=}1$, $h^{[\gamma]}$ values, $\mathbf{V}$, $\boldsymbol{b}_{i,1}$) ▷ Degree-1 outputs
    // Depth-first approach: Trigger deeper MuxGates on-demand
    if μ_2 layer-1 output nodes are available **then**
      MuxGate($j{=}2$, layer-1 nodes, $\mathbf{V}$, $\boldsymbol{b}_{i,2}$) ▷ Degree-2 outputs
    if μ_3 layer-2 output nodes are available **then**
      MuxGate($j{=}3$, layer-2 nodes, $\mathbf{V}$, $\boldsymbol{b}_{i,3}$) ▷ Degree-3 outputs
    if μ_4 layer-3 output nodes are available **then**
      $\mathsf{root} \leftarrow$ MuxGate($j{=}4$, layer-3 nodes, $\mathbf{V}$, $\boldsymbol{b}_{i,4}$) ▷ Degree-d root
  $P_\alpha[1\,..\,d] \oplus{=} \mathsf{root}[1\,..\,d]$ ▷ Accumulate to sig. memory; $P_\alpha[0]$ is skipped
  // UnitaryCheck for each elementary vector of the current witness chunk
  $\mathsf{coeff} \leftarrow \gamma'_i$
  for $j = 1, \ldots, d$ **do**
    $P_\alpha[1\,..\,2] \oplus{=} \mathsf{coeff} \cdot \mathrm{UnitaryCheck}(\mu_j, \mathbf{V}, \boldsymbol{b}_{i,j})$ ▷ Alg. 13, [13]
    $\mathsf{coeff} \leftarrow \mathsf{coeff} \cdot \xi^{32}$ ▷ $\xi = 2 \in \mathbb{F}_{2^\lambda}$

Processing of P_{wit} Polynomials After generating an appropriately sized chunk of witness polynomial coefficients and ensuring that they are stored in row-major format, the ProcessPwit subroutine (Algorithm 10) is applied. Both sub-algorithms MuxGate (Algorithm 11) and UnitaryCheck closely follow their counterparts from the SDitH-v2 specification, with the latter remaining fully unchanged (see [13, Algorithm 13]). Inputs $\mathbf{V}$ and $\boldsymbol{b}_{i,j}$ are equivalent to $\left\{P_{w_{i,j,k}}(X)\right\}_k$ as given in the specification. The number of outer loop iterations $0 \leq i \leq w-1$ (see Section 2.4 and [13, lines 5–7 of Algorithm 17]) that are performed on each pass are trivially determined by the chunk size of the matrix generation, with each iteration processing $\sum_{j=1}^{d}(\mu_j - 1)$ witness bits. In order to adapt the PIOP implementation to resource-constrained devices, several structural changes and optimizations are required, which are discussed individually.

Algorithm 11 MuxGate(layer index j, input nodes $\mathbf{a}$, $\mathbf{V}$ entries, control bits $\boldsymbol{b}$)

Input: μ_j input polynomial nodes $a_0, \ldots, a_{\mu_j - 1}$ of degree $\delta = j - 1$, and the corresponding $\mathbf{V}$ entries $v_1, \ldots, v_{\mu_j - 1}$

1: $\mathsf{res}[0\,..\,\delta] \leftarrow a_0$; $\mathsf{res}[\delta{+}1] \leftarrow 0$
2: **for** $k = 1, \ldots, \mu_j - 1$ **do**
3: **for** $c = 0, \ldots, \delta$ **do**
4: $\mathsf{diff} \leftarrow a_k[c] \oplus a_0[c]$
5: $\mathsf{res}[c] \oplus\!= b_k \cdot \mathsf{diff}$ ▷ Free since $b_k \in \mathbb{F}_2$
6: $\mathsf{res}[c{+}1] \oplus\!= v_k \otimes \mathsf{diff}$ ▷ One field multiplication
7: **return** $\mathsf{res}[0\,..\,\delta{+}1]$ ▷ Polynomial node of degree $\delta{+}1$

On-demand generation of γ'. Both the specification and the reference implementation precompute the set of all batching challenges $\gamma' \in \mathbb{F}_{2^\lambda}^{w}$ and $\gamma \in \mathbb{F}_{2^\lambda}^{\lceil (n-k)/\lambda \rceil}$ at the start of the PIOP phase and store them in memory. While this makes sense for γ since all entries of the vector are needed in every outer loop iteration to generate the MuxTree inputs, i.e., the batched column values $h^{[\gamma]}$, it is not required for the challenges γ'. In each loop iteration, a distinct entry is used, which is generated using the SHAKE instance seeded with h_{lines}. Instead of storing these values, amounting to $\lambda \cdot w$ bits (see Table 1b), we squeeze them on-demand from the persistent SHAKE state during each iteration, storing only a single element from $\mathbb{F}_{2^\lambda}$ at a time. Since no other operation within PIOP uses SHAKE, reseeding is not required and runtime is not negatively impacted.

On-the-fly generation of $h_j^{[\gamma]}$. Similarly, both specification and reference implementation precompute and store all n batched challenge values $h_j^{[\gamma]} \in \mathbb{F}_{2^\lambda}$ generated through the BatchLinearEquations routine ([13, Algorithm 19]) before executing the main PIOP loop. This requires $n \cdot \lambda$ bits of memory (see Table 1b), which is clearly not practical for a memory-efficient implementation. To circumvent this issue, we expand each column of the parity check matrix $\mathbf{H}$ on-the-fly during the first layer of the MuxTree procedure. Depending on the column index, the value $\mathbf{h}_j \in \mathbb{F}_2^{n-k}$ either corresponds to an elementary vector if it is located in the identity part of the matrix, i.e., if $j < n - k$, or to a random value generated

using the Rijndael-based PRG. Since expansion of $\mathbf{H}'$ from the PRG operates column-wise, generation of individual values $\mathbf{h}_j$ is efficiently possible.

Batching is then implemented by multiplying each generated λ-sized chunk of a column with the corresponding part of the stored challenge γ and accumulating accordingly, requiring $\lceil (n-k)/\lambda \rceil$ field multiplications.

Depth-first MuxTree traversal. If following the specification, the MuxTree evaluation happens in a breadth-first manner, i.e., by evaluating all nodes in the first layer of the tree before continuing with the second layer. Since the degree of the polynomial nodes increases by one in each layer while the number of nodes is reduced by μ_j, the peak memory usage of this process occurs in the first layer, resulting in $\lceil m/\mu_1 \rceil \cdot 2\lambda$ bits (up to 24 kbit for security level 5). In order to reduce this, we opt for a depth-first approach instead, computing only a μ_2-sized subset of all layer-1 outputs before continuing with the layer-2 evaluation. The same procedure is applied to deeper layers, resulting in at most $\mu_2 + (\mu_3 - 1) + (\mu_4 - 1)$ nodes needing to be stored at once, reducing the memory requirement to approximately 6.4 kbit for the highest security level.

Removal of explicit $y^{[\gamma]}$ and $P_\alpha(0)$ terms. As per the specification, the constant term of the PIOP output polynomial P_α is initialized to $y^{[\gamma]}$, which corresponds to the batched syndrome of the RSD instance. While this makes sense mathematically, considering that the constant term is supposed to encode a batched representation of $\mathbf{H} \cdot \mathbf{x} - \mathbf{y}$, it is not required for functional correctness of the scheme. For an honest prover, the constant term of P_α will always be 0 since the witness is a valid solution to the syndrome decoding problem. The specification accounts for this by only including the four non-constant coefficients of P_α in the subsequent generation of h_{piop} as well as in the resulting signature. Nonetheless, the value $y^{[\gamma]}$ is still computed by the reference implementation using $\lceil (n-k)/\lambda \rceil$ field multiplications. In order to improve computational efficiency of the PIOP phase, we omit this and implicitly set $P_\alpha(0) = 0$.

Usage of a single shared $\mathbb{F}_{2^\lambda}$ multiplier. While the evaluation of a single MuxTree layer could theoretically be parallelized using multiple arithmetic units, we opt for a single serial-parallel multiplier instead, which processes one of the λ-sized operands in chunks of 16 bits. As a result, one field multiplication requires $\lambda/16$ clock cycles. This multiplier instance is used for all polynomial arithmetic within PIOP, therefore reducing overall area requirement.

As an additional optimization, we reduce the number of full $\mathbb{F}_{2^\lambda}$ multiplications by observing that each degree-1 witness polynomial has a constant term from $\mathbb{F}_2$, i.e., the corresponding witness bit. Multiplication by $P_{\mathsf{wit}_{i,j,k}}(0)$ within the MuxTree thus corresponds to a cheap multiplexing operation.

4.6 Evaluation Point Selection and Commitment Opening

While the implemented hardware architecture of the OpenRandomEvaluation routine (see Algorithm 12 and Section 2.5) largely follows the specification, there are some notable design choices and optimizations, which we will discuss next.

Algorithm 12 OpenRandomEvaluation($h_{\mathsf{piop}}, \mathsf{salt}, \mathsf{rseed}$)

ctr $\leftarrow 0$
repeat ▷ Proof-of-work loop
  ($\boldsymbol{\delta}$, v_{pow}) $\leftarrow$ ExpandEvaluationChallenge($h_{\mathsf{piop}}, \mathsf{ctr}$); ctr += 1
  if $v_{\mathsf{pow}} \neq \mathbf{0}$ **or** $\boldsymbol{\delta} = \mathbf{0}$ **then continue** ▷ Grinding + invertibility
  Compute absolute tree indices $\boldsymbol{l}$ from $\boldsymbol{\delta}$ and sort in-place
  $t_{\mathsf{open}} \leftarrow$ GetOpeningPath($\boldsymbol{l}$, salt, rseed, **false**)
  if $t_{\mathsf{open}} > T_{\mathsf{open}}$ **then continue**
until valid $\boldsymbol{\delta}$ found

for $k = 0, \ldots, \tau - 1$ **do** ▷ Hidden-leaf commitment generation
  $C_k \leftarrow$ SeedAndCommitGen(rseed, salt, l_k, **commit**)
  Write C_k to signature memory

GetOpeningPath($\boldsymbol{l}$, salt, rseed, **true**) ▷ Sibling-path seed reveal
Zero-pad remaining $T_{\mathsf{open}} - t_{\mathsf{open}}$ sibling slots
Write ctr$^\star \leftarrow$ ctr $- 1$ to signature memory

After extracting a set of hidden leaf index candidates $\boldsymbol{\delta}$ from the seeded SHAKE instance, the τ per-subtree indices must be converted to absolute GGM node indices. Since all subtrees at the leaf layer are interleaved within the aggregate tree, the k-th of τ total subtrees occupies every τ-th leaf starting at offset k. As a result, index conversion happens as $l_k = \delta^{(k)} \cdot \tau + k + \tau \cdot 2^\kappa$, where $\delta^{(k)}$ denotes the k-th element of $\boldsymbol{\delta}$ and $\tau \cdot 2^\kappa$ corresponds to the offset of the first leaf in the tree. The resulting indices are stored in the shared scratchpad memory and then need to be sorted to allow for efficient computation of the shortest *opening path* (see Algorithm 6, [13]). Since the number of elements to sort is small with $\tau \leq 32$, we select an in-place insertion sort algorithm, performing at most 496 comparisons. Due to its simplicity, this is a more suitable choice for hardware architectures compared to the quicksort algorithm used in the reference implementation.

Dual-mode t_{open} estimation and tree opening. The set of t_{open} many intermediate nodes that need to be revealed during tree opening is determined by the structure of the hidden-leaf set $\{l_0, ..., l_{\tau-1}\}$ relative to the binary-tree topology. To compute this, a queue-based bottom-up merge procedure GetOpeningPath (Algorithm 13) is applied which starts with the hidden-leaf set and repeatedly checks whether the following entry is a sibling of the queue head, i.e., whether it also needs to stay hidden. If yes, both entries are removed from the queue. If not, the sibling of the queue head needs to be revealed and its index is added to the set of opened nodes. In both cases, the parent of the queue head is appended to indicate that it must not be revealed. The algorithm terminates once the root of the GGM tree has been reached. Intuitively, the queue contains the set of nodes that definitely need to remain hidden, at the current stage of the algorithm.

To efficiently implement verification of the compactness constraint $t_{\mathsf{open}} \leq T_{\mathsf{open}}$ during grinding (see Section 2.5) and generation of opened GGM seeds

without redundant control logic, we implement a single module which, depending on a mode bit, either computes just t_{open} or interfaces with an external SeedAndCommitGen module to populate all required seeds along the path. Since the hidden leaf indices in memory are overwritten with other node indices during path generation, a backup is created beforehand to enable subsequent processing steps (see Section 4.2). The queue data structure itself is implemented through a circular buffer in scratchpad memory with dedicated head and tail pointers.

Algorithm 13 GetOpeningPath($\boldsymbol{l}$, salt, rseed, reveal)

Input: Sorted hidden-leaf indices $\boldsymbol{l} = (l_0, \ldots, l_{\tau-1})$, reveal: mode flag
Output: t_{open}: number of nodes to reveal; if reveal: seeds are written to RAM

$t \leftarrow 0$; $Q \leftarrow$ Queue.Init($\boldsymbol{l}$) ▷ Enqueue leaf indices in decreasing order
while Q.Head() $\neq 1$ **do**
  $u \leftarrow Q$.Dequeue()
  if $|Q| \geq 1$ **and** $u \oplus Q$.Head() $= 1$ **then**
    Q.Dequeue() ▷ Sibling also hidden? Consume both
  else
    $t \leftarrow t + 1$ ▷ Otherwise, add sibling to opening set
    if reveal **then**
      seed $\leftarrow$ SeedAndCommitGen(rseed, salt, $u \oplus 1$, **seed**)
      Write seed to signature memory
  Q.Enqueue($u \gg 1$) ▷ Propagate upward: Parent inherits hidden status
return t

Stateless GGM seed and commitment regeneration. While the specification implicitly assumes that the full GGM tree is cached and individual nodes can trivially be retrieved, this is not feasible in hardware, as shown in Table 1b. Therefore, a recomputation-based approach is taken in the dual-purpose SeedAndCommitGen module instead, which computes a commitment or seed, depending on the selected mode, for a given node index starting from the root seed. The depth of the target node can be determined as $\lfloor \log_2 \mathsf{idx} \rfloor$ while the path through the tree can be identified by looking at the individual bits of the index. The combination of both seed and commitment generation again reduces control path overhead. As a result of these optimizations, no additional memory is required for the OpenRandomEvaluation implementation. Similar to previously described modules, parts of the Rijndael latency are utilized to prefetch the next node index during each processing iteration in order to increase performance.

5 Evaluation

In order to evaluate the performance and efficiency of the developed hardware designs on FPGA platforms, we perform synthesis for a Xilinx Artix-7 100T target using Vivado 2024.2 and provide an overview of the resulting metrics in Table 4. This is augmented by a comparison with other hardware implementations of additional PQC signature candidates as well as already standardized

signature schemes. Input and output memories, i.e., key memory, message memory, and signature memory, are not included in the area results. Maximum clock frequency is determined by systematically increasing the target clock frequency during synthesis until timing closure cannot be achieved. We further ensured that our designs are functionally identical to the reference implementation through execution of 100 of the official KATs.

Most notably, in terms of the estimated number of required slices [20], which combines utilization of all available FPGA primitives into a single area metric, our implementations significantly outperform all other analyzed designs. However, this expectedly comes at the cost of higher latency than most other schemes except for the first version of SDitH, which we still manage to beat depending on the configuration and chosen trade-off. While part of the improvement could be attributed to algorithmic changes for the second-round submission, we argue that the majority of area savings, particularly with focus on memory, are the result of our optimizations. To support this claim, we point to Table 3b, which showcases the memory requirements of the reference implementation of both versions of the SDitH scheme. Notably, memory usage improved by less than 40% across all security levels for the second-version of the scheme, which is orders of magnitude lower than our reported improvement.

Another observation is that our designs are notably more LUT-heavy than both the SDitH-v1 and Mirath implementations. A major part of this can be attributed to our recomputation-based approach to area reduction requiring a significantly more complex control path and large amounts of wide multiplexing logic. Depending on the configuration, up to 40% of the total LUTs are located in the chunked PIOP implementation. We refer to Table 5a for a detailed per-submodule resource breakdown for the HP0 fast configurations.

An overview of memory usage compared to the SDitH-v2 reference implementation and other low-memory software implementations is provided in Table 3. While the NIST submission also includes an optimized software implementation, it is algorithmically identically to the reference implementation and only differs in the usage of AVX instructions for some subroutines to improve execution time. Therefore, memory requirements are identical across both variants. The results showcase an order-of-magnitude improvement over the reference implementation as well as significantly lower memory requirements than the optimized PERK implementation by Bettaieb et al. [17]. While the optimized MQOM implementation [18] achieves even lower peak memory usage, it is also not a completely fair comparison as the authors themselves note that VOLEitH schemes inherently require more memory and are harder to optimize.

In order to decrease the latency of our design, there are multiple trade-offs to explore. About 50% of the execution time is spent in the re-generation of the matrix $\mathbf{V}$ during the PIOP stage (see Table 5b). Since full traversal of all GGM leaves already happens during h_{com}, h_{aux}, and h_{lines} phases, redundant computations can be eliminated by storing additional intermediate results. Generating $\mathbf{V}$ only once during BLC.Commit significantly reduces overall runtime, but is incompatible with our optimization target due to the large matrix size. Similarly,

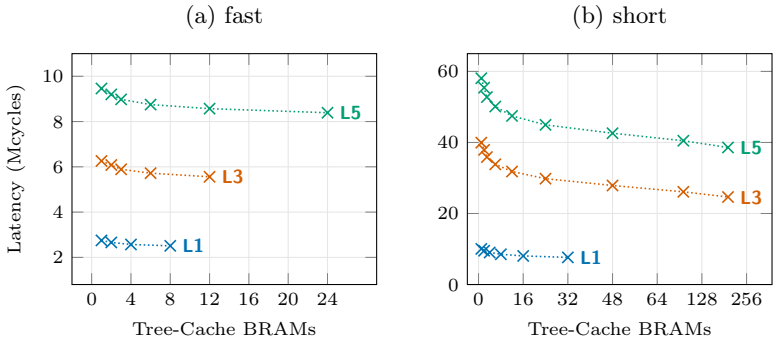


Fig. 3: Effect of increasing GGM tree cache depth on overall signing latency for the HP1 variant, assuming Xilinx 36 kbit BRAM primitives. Each additional cache level doubles the stored GGM tree nodes. The BRAM-count scale is linear until 64 and logarithmic beyond.

Table 3: Analysis of memory usage compared to other software implementations.

(a) Stored variables of our work compared to the SDitH-v2 reference implementation.

Variable	VOLE Data			PIOP Data				Proof Polynomials			Proof Data			
	$\mathbf{V}$	$\mathbf{r}_{\text{base}}$	$\mathbf{u}$	γ	γ'	$\mathbf{H}^{[\gamma]}$	$y^{[\gamma]}$	P_{wit}	P_{rnd}	P_{α}	$\boldsymbol{\alpha}'_{\text{plain}}$	$\boldsymbol{\alpha}'_{\text{base}}$	$\mathbf{M}$	$\mathbf{H}'$
SW [13]	✓	–	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓	–
HW [this]	–	✓	✓	✓	–	–	–	–	–	✓	✓	✓	–	–

(b) Total memory usage compared to other MPCitH software implementations in kB.

Level	This Work[a]		SDitH-v2 Ref.[b][13]		SDitH-v1 Ref.[c][24]		MQOM†[18]		PERK†[17]	
	fast	short	fast	short	$\mathbb{F}_{256}$	$\mathbb{F}_{251}$	fast	short	fast	short
L1	9.1	11.2	246.4	245.6	370	371	5.8	6.0	24.0	27.8
L3	13.4	13.4	610.8	609.0	859	861	–	–	47.7	51.3
L5	17.8	21.8	925.7	922.7	1500	1500	15.8	16.4	80.3	82.3

[a] Computed as $(\#\text{BRAM} \times 32\,768 + \#\text{FF})/8\,192$ for HP0 configurations from Table 4.
[b] Size of a single pre-allocated scratch buffer used for all intermediate signing variables.
[c] RAM usage of the hypercube variants as reported in the round-1 specification.
† Lowest-memory configuration reported in each work.

deeper layers of the GGM tree can be stored in order to reduce the leaf generation time in BLC.Commit. However, the amount of additional memory required increases exponentially as the number of nodes doubles with each tree layer. As a result, this trade-off is likely not feasible beyond a certain degree, particularly for the fast parameter set, as visualized in Figure 3.

Since this work focused solely on establishing a lower bound for the area requirements of MPCitH-based signature schemes in hardware, more balanced trade-offs were only partially evaluated and not practically implemented.

Table 4: Post-implementation resource utilization and performance of the SDitH-v2 designs compared to related work, targeting an Artix-7 FPGA platform.

Scheme	Variant	Resource Usage					Freq.	Latency	Delay
		LUT	FF	BRAM	DSP	Slices[a]	(MHz)	(Mcycles)	(ms)
SDitH-v2 **[this]**	L1-f (HP0)	24 851	9373	2	1	6772	145.3	3.58	24.6
	L1-f (HP1)	24 073	9225	3	1	6790	129.9	2.75	21.2
	L1-s (HP0)	24 324	9524	2.5	1	6746	119.6	15.19	127.0
	L1-s (HP1)	24 442	9456	3	1	6882	130.0	9.89	76.1
	L3-f (HP0)	32 201	12 001	3	1	8822	135.9	7.46	54.9
	L3-f (HP1)	32 184	11 579	4	1	9029	121.8	6.26	51.4
	L3-s (HP0)	32 844	11 615	3	1	8982	143.3	58.05	405.2
	L3-s (HP1)	33 749	11 935	4	1	9421	120.2	39.94	332.2
	L5-f (HP0)	39 331	15 026	4	1	10 816	126.7	11.03	87.0
	L5-f (HP1)	37 719	15 032	6	1	10 837	114.7	9.46	82.5
	L5-s (HP0)	39 793	14 853	5	3	11 414	124.7	81.93	657.1
	L5-s (HP1)	37 858	15 721	6	3	11 142	117.6	58.07	493.6
Related work – MPCitH and code-based additional signature candidates									
SDitH-v1 [15]	L1-GF256	5000	4200	99	0	22 238	240.0	6.70	28.0
	L1-GF251	8400	9100	99	120	39 288	172.0	21.40	124.0
	L3-GF256	8600	7100	246	0	54 302	244.0	7.70	31.0
	L3-GF251	13 100	15 300	246	240	87 827	171.0	24.50	143.0
	L5-GF256	9100	7800	353	0	77 111	241.0	17.40	72.0
	L5-GF251	15 900	18 900	353	300	119 311	174.0	45.20	260.0
Mirath† [16]	1b-fast	15 400	9445	42	0	12 754	254.0	0.55	2.2
	5b-fast	30 415	18 223	96	0	27 956	218.0	1.37	6.3
Picnic‡ [19]	L1	76 472	21 061	52.5	0	30 248	125.0	0.03	0.3
	L5	149 456	30 441	98.5	0	58 246	125.0	0.15	1.2
CROSS† [20]	RSDP-1-f	26 741	11 678	45	0	16 226	118.0	0.06	0.5
	RSDP-1-s	27 779	11 990	111.5	0	30 583	114.0	0.19	1.7
	RSDP-3-f	29 096	13 761	97	0	27 838	108.0	0.12	1.1
	RSDP-3-s	29 742	14 067	148	0	38 812	109.0	0.28	2.6
	RSDP-5-f	32 085	15 537	151	0	40 034	113.0	0.19	1.7
	RSDP-5-s	34 868	15 815	280	0	68 077	105.0	0.49	4.6
Related work – NIST standardized signature schemes									
ML-DSA†*+ [25]	44							0.02	0.1
	65	54 942	29 309	29	16	25 707	230.0	0.03	0.1
	87							0.03	0.2
SLH-DSA [26]	128f	10 197	7357	8	0	5165	150.0	2.89	19.3
	128s	10 255	7410	8	0	5186	150.0	54.45	363.0
	192f	10 333	7590	8	0	5228	150.0	4.66	31.1
	192s	10 644	7643	8	0	5312	150.0	94.74	631.6
	256f	10 416	7672	8	0	5259	150.0	9.26	61.7
	256s	10 802	7698	8	0	5358	150.0	83.55	557.0
Falcon [27]	512	79 065	46 389	45	226	65 615	65.0	0.16	2.5

[a] Estimated slices computed as $\max(\lceil \#\text{LUTs}/4 \rceil, \lceil \#\text{FFs}/8 \rceil) + 212 \cdot \#\text{BRAM} + 135 \cdot \#\text{DSP}$ [20].
† Unified design supporting KeyGen, Sign, and Verify.
* Supports all parameter sets at run-time.
‡ Targets Kintex-7 instead of Artix-7.
+ Targets Ultrascale+ instead of Artix-7.

Table 5: Performance breakdowns of the implemented HP0 fast configurations.

(a) Resource usage of each hardware submodule.

Submodule	L1		L3		L5	
	LUTs	FFs	LUTs	FFs	LUTs	FFs
Dual AES / Rijndael	3284	558	5076	1070	7023	2502
SHAKE / Keccak	4553	2964	4446	2914	4451	2708
Memory Management	172	0	300	0	797	8
Signing Datapath	16 855	5851	22 391	8017	27 076	9808
GGM Tree Traversal	1510	399	2094	533	2956	662
BLC Commit	4972	1332	6061	1875	6318	1865
PIOP Prover	8925	3069	11 895	4135	14 926	5169
Witness Generation	2141	841	2720	983	2483	1101
PIOP Processing	6784	2228	9175	3152	12 443	4068
GF(2^λ) Multiplier	3279	517	5145	792	7278	1048
Open Random Evaluation	1448	1051	2341	1474	2876	2112
Total	24 864	9373	32 213	12 001	39 347	15 026

(b) Latency of each signature generation phase.

Phase	L1		L3		L5	
	Cycles	%	Cycles	%	Cycles	%
Tree-Cache Population	131 617	3.7	264 618	3.5	377 382	3.4
BLC Haux	370 563	10.3	710 981	9.5	1 114 670	10.1
BLC Hlines	799 421	22.3	1 421 165	19.0	2 694 839	24.4
PIOP Prover	2 218 668	61.9	4 924 649	66.0	6 528 402	59.2
Open Random Evaluation	61 322	1.7	140 427	1.9	312 605	2.8
Total	3 581 595	**100.0**	7 461 845	**100.0**	11 027 904	**100.0**

6 Conclusion and Future Work

With this work, we advance the design space exploration of on-ramp PQC signature candidates by analyzing the edge case of highly area-efficient MPCitH hardware implementations with SDitH-v2 as a relevant case study. As a result, we proposed the first hardware implementation of that specific primitive and, to the best of our knowledge, the most compact hardware implementation of any MPCitH- or code-based PQC signature scheme to date, reducing SRAM usage by more than an order of magnitude. The practically demonstrated results can serve as a new baseline for the minimum area requirement for deployment in resource-constrained embedded cryptographic devices. Consequently, they can also provide a starting point for future development of more balanced designs, potentially with the assistance of automated design space exploration frameworks, like HADES [28]. Since the algorithmic optimizations are not specific to hardware designs, application to an optimized software implementation of SDitH-v2 remains an additional interesting direction for further research.

While we notified the SDitH submission team about the identified set of specification errors and proposed appropriate fixes, the general problems resulting from the large size of intermediate variables remain. These findings highlight

the importance of considering different deployment scenarios during the design process of new cryptographic primitives. We therefore urge all researchers involved in ongoing specification efforts to consider implementation aspects not as an afterthought or subordinate to common metrics such as signature size and public key size, but as an essential part of algorithm design.

Acknowledgments. The work described in this paper has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. The authors have additionally received funding from the German Federal Ministry of Research, Technology and Space (BMFTR) through the projects EASEPROFIT (16KIS2126), POST (16ME2157), and DI-OWAS (16ME0967). We furthermore thank Andreas Hülsing, Nicolas Gama, Thibauld Feneuil, and James Howe for helpful discussions on the SDitH-v2 scheme. Generative AI tools by Anthropic were used to assist in the development of the proof-of-concept SystemVerilog implementation.

References

1. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on. pp. 124–134. IEEE (1994)
2. Devoret, M.H., Schoelkopf, R.J.: Superconducting circuits for quantum information: an outlook. Science **339**(6124), 1169–1174 (2013)
3. Kelly, J., Barends, R., Fowler, A.G., Megrant, A., Jeffrey, E., White, T.C., Sank, D., Mutus, J.Y., Campbell, B., Chen, Y., Chen, Z., Chiaro, B., Dunsworth, A., Hoi, I.C., Neill, C., O'Malley, P.J.J., Quintana, C., Roushan, P., Vainsencher, A., Wenner, J., Cleland, A.N., Martinis, J.M.: State preservation by repetitive error detection in a superconducting quantum circuit. Nature **519**, 66–69 (2015)
4. Mosca, M.: Cybersecurity in an Era with Quantum Computers: Will We Be Ready? (2018), http://eprint.iacr.org/2015/1075
5. National Institute of Standards and Technology (NIST): Post-quantum cryptography standardization project (2017), http://csrc.nist.gov/groups/ST/post-quantum-crypto/
6. FIPS 203 (Initial Public Draft): Module-Lattice-Based Key-Encapsulation Mechanism Standard. National Institute of Standards and Technology, NIST FIPS PUB 203, U.S. Department of Commerce (Aug 2023)
7. FIPS 204 (Initial Public Draft): Module-Lattice-Based Digital Signature Standard. National Institute of Standards and Technology, NIST FIPS PUB 204, U.S. Department of Commerce (Aug 2023)
8. FIPS 205 (Initial Public Draft): Stateless Hash-Based Digital Signature Standard. National Institute of Standards and Technology, NIST FIPS PUB 205, U.S. Department of Commerce (Aug 2023)
9. Prest, T., Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: FALCON. Tech. rep., National Institute of Standards and Technology (2022), available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
10. National Institute of Standards and Technology: NIST announces additional digital signature candidates for the PQC standardization process. https://csrc.nist.gov/News/2023/additional-pqc-digital-signature-candidates (Jul 2023), accessed: 2026-04-18
11. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing. p. 21–30. STOC '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1250790.1250794, https://doi.org/10.1145/1250790.1250794
12. Feneuil, T., Rivain, M.: MQOM — MQ on my Mind. Tech. rep., National Institute of Standards and Technology (2024), available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures
13. Aguilar-Melchor, C., Bettaieb, S., Bidoux, L., Feneuil, T., Gaborit, P., Gama, N., Gueron, S., Howe, J., Hülsing, A., Joseph, D., Joux, A., Kulkarni, M., Persichetti, E., Randrianarisoa, T.H., Rivain, M., Yue, D.: The syndrome decoding in the head (sd-in-the-head) signature scheme. https://www.sdith.org/docs/sdith-v2.0.pdf (2025), NIST Post-Quantum Cryptography Round-2 submission
14. Baum, C., Braun, L., de Saint Guilhem, C.D., Klooß, M., Orsini, E., Roy, L., Scholl, P.: Publicly verifiable zero-knowledge and post-quantum signatures from vole-in-the-head. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology – CRYPTO 2023. pp. 581–615. Springer Nature Switzerland, Cham (2023)

15. Deshpande, S., Howe, J., Szefer, J., Yue, D.: Sdith in hardware. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2024**(2), 215–251 (2024). https://doi.org/10.46586/TCHES.V2024.I2.215-251
16. Manasidis, S., Norga, Q., Kundu, S., Verbauwhede, I.: Mpspeed: Implementing and optimizing mpc-in-the-head digital signatures in hardware. IACR Cryptol. ePrint Arch. **2026**, 206 (2026), https://eprint.iacr.org/2026/206
17. Bettaieb, S., Bidoux, L., Budroni, A., Palumbi, M., Perin, L.P.: Enabling PERK and other mpc-in-the-head signatures on resource-constrained devices. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2024**(4), 84–109 (2024). https://doi.org/10.46586/TCHES.V2024.I4.84-109, https://doi.org/10.46586/tches.v2024.i4.84-109
18. Benadjila, R., Feneuil, T.: Breaking the myth of mpcith inefficiency: Optimizing MQOM for embedded platforms. IACR Cryptol. ePrint Arch. **2026**, 78 (2026), https://eprint.iacr.org/2026/078
19. Kales, D., Ramacher, S., Rechberger, C., Walch, R., Werner, M.: Efficient FPGA implementations of lowmc and picnic. In: Topics in Cryptology - CT-RSA 2020. pp. 417–441. Lecture Notes in Computer Science, Springer (2020). https://doi.org/10.1007/978-3-030-40186-3_18
20. Karl, P., Antognazza, F., Barenghi, A., Pelosi, G., Sigl, G.: High-performance FPGA accelerator for the post-quantum signature scheme CROSS. IACR Cryptol. ePrint Arch. **2025**, 1161 (2025), https://eprint.iacr.org/2025/1161
21. National Institute of Standards and Technology: NIST proposes to standardize a wider variant of AES. https://csrc.nist.gov/news/2024/nist-proposes-to-standardize-wider-variant-of-aes (Dec 2024), accessed: 2026-05-11
22. Daemen, J., Rijmen, V.: AES proposal: Rijndael. https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf (1999), NIST AES Competition Proposal
23. Baum, C., Beullens, W., Mukherjee, S., Orsini, E., Ramacher, S., Rechberger, C., Roy, L., Scholl, P.: One tree to rule them all: Optimizing ggm trees and owfs for post-quantum signatures. In: Chung, K.M., Sasaki, Y. (eds.) Advances in Cryptology – ASIACRYPT 2024. pp. 463–493. Springer Nature Singapore, Singapore (2025)
24. Aguilar-Melchor, C., Feneuil, T., Gama, N., Gueron, S., Howe, J., Joseph, D., Joux, A., Persichetti, E., Randrianarisoa, T.H., Rivain, M., Yue, D.: The syndrome decoding in the head (sd-in-the-head) signature scheme. https://sdith.org/docs/sdith-v1.0.pdf (2023), NIST Post-Quantum Cryptography Round-1 submission (version 1.0)
25. Norga, Q., Kundu, S., Verbauwhede, I.: Ml-dsa-osh: An efficient, open-source hardware implementation of ml-dsa. In: 2026 Design, Automation & Test in Europe Conference (DATE). pp. 1–3 (2026). https://doi.org/10.23919/DATE69613.2026.11539720
26. Deshpande, S., Lee, Y., Karakuzu, C., Szefer, J., Paek, Y.: Sphincslet: An area-efficient accelerator for the full sphincs+ digital signature algorithm. ACM Trans. Embed. Comput. Syst. **24**(5) (Sep 2025). https://doi.org/10.1145/3728469, https://doi.org/10.1145/3728469
27. Ouyang, Y., Zhu, Y., Zhu, W., Yang, B., Zhang, Z., Wang, H., Tao, Q., Zhu, M., Wei, S., Liu, L.: Falconsign: An efficient and high-throughput hardware architecture for falcon signature generation. IACR Transactions on Cryptographic Hardware and Embedded Systems **2025**(1), 203–226 (Dec 2024). https://doi.org/10.46586/tches.v2025.i1.203-226, https://tches.iacr.org/index.php/TCHES/article/view/11927

28. Buschkowski, F., Land, G., Höher, N., Richter-Brockmann, J., Sasdrich, P., Güneysu, T.: HADES: automated hardware design exploration for cryptographic primitives. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2025**(4), 1–45 (2025). https://doi.org/10.46586/TCHES.V2025.I4.1-45, https://doi.org/10.46586/tches.v2025.i4.1-45

A Additional Results

A.1 Signing Datapath Resource Breakdown

While Table 5a already provides a per-submodule resource breakdown for the HP0 fast configurations, Tables 6 and 7 extend this to all possible designs. Submodule resources may not sum exactly to the datapath total due to top-level control logic not attributed to any submodule after synthesis.

A.2 Signing Latency Breakdown

Table 5b in the main body shows the per-phase latencies for HP0 fast configurations. Tables 8 and 9 decompose the major signature generation phases into smaller suboperations across all twelve configurations and highlight their contribution to overall latency. The results particularly showcase the impact of the short parameter sets on the duration of GGM leaf regeneration and the speedup resulting from the half-polynomial pipeline.

A.3 Integrated h_{com} Latency

In the baseline design, the DFS phase performs a full GGM tree traversal to compute h_{com}. Alternatively, this could be integrated into computation of h_{aux}, where each leaf is already fetched for $\mathbf{r}_{\mathsf{base}}$ generation, reducing DFS to a cache-tree BFS (about 3–5K cycles). Table 10 compares the resulting signing latencies and Table 11 provides an additional per-phase breakdown.

A.4 Memory Analysis

Table 12 extends the discussion on scratchpad memory topology (see Table 2) with additional information for each possible configuration. Additionally, Table 13 shows the per-stage peak memory usages that eventually determine the overall RAM requirements. To achieve better comparability, Table 14 provides a breakdown of per-variable scratch space usage in the software reference implementation. Lastly, Figures 4–9 showcase our utilized memory layout for all parameter sets (cf. Figure 2 in the main text).

A.5 Cache Depth Sweep

We extend the visualization of the effect of GGM tree cache depth on signing latency from Figure 3 by providing exact cycle counts for both pipeline variants across all parameter sets in Table 15. Each additional cache level doubles the stored tree nodes, reducing the number of AES calls during the descent to a leaf at the cost of additional memory. The cache BRAMs column reflects the tree-cache only; the scratchpad uses separate BRAM instances (see Table 13).

Table 6: Submodule resource breakdown of the HP0 signing datapath.

Submodule	L1				L3				L5			
	fast		short		fast		short		fast		short	
	LUTs	**FFs**	**LUTs**	**FFs**	**LUTs**	**FFs**	**LUTs**	**FFs**	**LUTs**	**FFs**	**LUTs**	**FFs**
GGM Tree Traversal	1510	399	1629	422	2094	533	2252	565	2956	662	3093	690
BLC Haux	829	167	602	159	538	47	286	41	612	102	720	102
Generate $\mathbf{r}_{\mathsf{base}}$	661	155	774	161	1233	91	621	100	553	104	577	118
BLC h_{lines}	2671	663	2784	666	3597	987	3588	754	3993	902	4150	902
BLC $\mathbf{r}_{\mathsf{acc}}$	434	330	264	330	520	718	1514	722	1095	717	1095	717
PIOP Prover	8925	3069	8200	3059	11 895	4135	12 563	4131	14 926	5169	15 325	5184
Control	351	81	341	82	635	90	562	81	283	81	283	81
Gen. P_{wit} (Chunk)	828	193	863	203	1125	321	1603	343	1214	440	1855	492
Transpose P_{wit}	965	567	376	551	965	572	958	572	988	580	866	561
PIOP Processing	6784	2228	6622	2223	9175	3152	9442	3135	12 443	4068	12 324	4050
GF(2^λ) Multiplier	3279	517	3532	517	5145	792	5483	792	7278	1048	7201	1050
Open Random Evaluation	1448	1051	1812	1062	2341	1474	2068	1499	2876	2112	2933	1878
Datapath Total	16 855	5851	16 067	5872	22 391	8017	22 958	7837	27 076	9808	27 963	9630

Table 7: Submodule resource breakdown of the HP1 signing datapath.

Submodule	L1				L3				L5			
	fast		short		fast		short		fast		short	
	LUTs	**FFs**	**LUTs**	**FFs**	**LUTs**	**FFs**	**LUTs**	**FFs**	**LUTs**	**FFs**	**LUTs**	**FFs**
GGM Tree Traversal	1571	404	1961	418	2056	540	3139	569	2677	661	2211	708
BLC Haux	726	101	591	96	315	42	465	41	705	102	797	101
Generate $\mathbf{r}_{\mathsf{base}}$	602	99	675	104	589	99	655	108	594	100	654	126
BLC h_{lines}	2999	623	3110	635	3205	749	4053	772	4134	914	4390	923
BLC $\mathbf{r}_{\mathsf{acc}}$	514	330	464	330	927	718	489	721	850	717	1053	717
PIOP Prover	8845	3073	8852	3272	12 331	4151	12 829	4402	15 143	5174	15 213	5323
Control	335	83	327	86	653	89	493	94	284	105	337	92
Gen. P_{wit} (Half)	1876	770	1792	959	2570	902	3153	1155	1704	991	2064	1166
PIOP Processing	6637	2220	6734	2227	9108	3160	9183	3153	13 158	4078	12 812	4065
GF(2^λ) Multiplier	3554	517	3521	517	5317	792	5101	790	7354	1048	7267	1050
Open Random Evaluation	1555	1051	1557	1061	2527	1475	2185	1504	3249	1831	2831	1877
Datapath Total	16 849	5701	17 282	5934	22 020	7793	23 892	8153	27 399	9544	27 210	9812

Table 8: Per-phase and per-suboperation latencies of the HP0 signing pipeline.

Phase / Sub-op.	**Fast**						**Short**					
	L1		**L3**		**L5**		**L1**		**L3**		**L5**	
	Cycles	**%**	**Cycles**	**%**	**Cycles**	**%**	**Cycles**	**%**	**Cycles**	**%**	**Cycles**	**%**
Tree-Cache Gen.	131 617	3.7	264 618	3.5	377 382	3.4	721 493	4.7	2 818 456	4.9	3 957 369	4.8
BLC Haux	370 563	10.3	710 981	9.5	1 114 670	10.1	2 385 025	15.7	9 332 532	16.1	14 091 490	17.2
BLC Hlines†	799 421	22.3	1 421 165	19.0	2 694 839	24.4	2 917 864	19.2	10 589 945	18.2	17 102 188	20.9
gen_rbase	437 730	12.2	804 198	10.8	1 315 857	11.9	2 586 580	17.0	10 015 590	17.3	15 837 580	19.3
proc_rbase	358 631	10.0	611 230	8.2	1 369 521	12.4	328 563	2.2	569 230	1.0	1 256 341	1.5
PIOP Prover†	2 218 668	61.9	4 924 649	66.0	6 528 402	59.2	8 873 826	58.4	33 803 309	58.2	46 308 679	56.5
gen_pwit	1 020 684	28.5	2 085 330	27.9	2 930 090	26.6	7 674 756	50.5	30 966 330	53.3	42 712 395	52.1
transpose	27 354	0.8	50 795	0.7	89 865	0.8	27 186	0.2	50 795	0.1	89 745	0.1
proc_pwit	1 170 563	32.7	2 788 437	37.4	3 508 353	31.8	1 171 817	7.7	2 786 097	4.8	3 506 445	4.3
Open Rand. Eval.	61 322	1.7	140 427	1.9	312 605	2.8	293 847	1.9	1 501 800	2.6	472 951	0.6
Total	3 581 595	**100.0**	7 461 845	**100.0**	11 027 904	**100.0**	15 192 059	**100.0**	58 046 047	**100.0**	81 932 683	**100.0**

† Listed suboperations exclude minor steps (e.g., P_0 computation, Fiat–Shamir hashing).

Table 9: Per-phase and per-suboperation latencies of the HP1 signing pipeline.

Phase / Sub-op.	Fast						Short					
	L1		L3		L5		L1		L3		L5	
	Cycles	%	Cycles	%	Cycles	%	Cycles	%	Cycles	%	Cycles	%
Tree-Cache Gen.	131 617	4.8	264 618	4.2	377 382	4.0	721 493	7.3	2 818 456	7.1	3 957 369	6.8
BLC Haux	379 272	13.8	728 021	11.6	1 138 676	12.0	2 430 990	24.6	9 422 692	23.6	14 345 574	24.7
BLC Hlines†	552 713	20.1	1 376 207	22.0	2 601 226	27.5	2 167 046	21.9	9 952 191	24.9	15 569 038	26.8
gen_rbase	318 468	11.6	699 336	11.2	1 188 516	12.6	1 946 360	19.7	9 347 884	23.4	14 253 838	24.5
proc_rbase	231 335	8.4	671 134	10.7	1 403 249	14.8	218 115	2.2	599 182	1.5	1 306 933	2.3
PIOP Prover†	1 622 216	59.1	3 754 591	59.9	5 027 956	53.2	4 281 152	43.3	16 248 111	40.7	23 728 716	40.9
gen_pwit	451 590	16.4	966 070	15.4	1 519 510	16.1	3 109 272	31.4	13 461 930	33.7	20 222 178	34.8
proc_pwit	1 170 559	42.6	2 788 434	44.5	3 508 352	37.1	1 171 813	11.8	2 786 094	7.0	3 506 444	6.0
Open Rand. Eval.	61 322	2.2	140 427	2.2	312 605	3.3	293 847	3.0	1 501 800	3.8	472 951	0.8
Total	2 747 144	**100.0**	6 263 869	**100.0**	9 457 851	**100.0**	9 894 532	**100.0**	39 943 255	**100.0**	58 073 654	**100.0**

† Listed suboperations exclude minor steps (e.g., P_0 computation, Fiat–Shamir hashing).

Table 10: Signing latencies for different configurations assuming h_{com} generation during a separate DFS stage or integrated into the h_{aux} procedure.

Config	HP0			HP1		
	h_{com} in DFS	h_{com} in BLC	Δ (%)	h_{com} in DFS	h_{com} in BLC	Δ (%)
L1-f	3 581 595	3 678 585	2.7	2 747 144	2 844 134	3.5
L1-s	15 192 059	14 367 179	−5.4	9 894 532	9 069 652	−8.3
L3-f	7 461 845	7 187 138	−3.7	6 263 869	5 989 162	−4.4
L3-s	58 046 047	54 709 045	−5.7	39 943 255	36 606 253	−8.4
L5-f	11 027 904	10 698 923	−3.0	9 457 851	9 128 870	−3.5
L5-s	81 932 683	79 528 770	−2.9	58 073 654	55 669 741	−4.1

Table 11: Per-phase latency with h_{com} computed during the h_{aux} stage.

(a) HP0 pipeline variant.

Phase	L1		L3		L5	
	fast	short	fast	short	fast	short
Tree-Cache Generation	5098	5098	3426	3426	3806	3806
BLC Haux	371 942	2 386 252	714 023	9 335 595	1 119 297	14 095 538
BLC Hlines	799 421	2 917 864	1 421 165	10 589 945	2 694 839	17 102 188
PIOP Prover	2 218 668	8 873 826	4 924 649	33 803 309	6 528 402	46 308 679
Open Random Evaluation	283 452	184 135	123 870	976 765	352 573	2 018 553
Total	3 678 585	14 367 179	7 187 138	54 709 045	10 698 923	79 528 770

(b) HP1 pipeline variant.

Phase	L1		L3		L5	
	fast	short	fast	short	fast	short
Tree-Cache Generation	5098	5098	3426	3426	3806	3806
BLC Haux	380 651	2 432 217	731 063	9 425 755	1 143 303	14 349 622
BLC Hlines	552 713	2 167 046	1 376 207	9 952 191	2 601 226	15 569 038
PIOP Prover	1 622 216	4 281 152	3 754 591	16 248 111	5 027 956	23 728 716
Open Random Evaluation	283 452	184 135	123 870	976 765	352 573	2 018 553
Total	2 844 134	9 069 652	5 989 162	36 606 253	9 128 870	55 669 741

Table 12: Physical BRAM topology of scratchpad memory across all configurations, assuming Xilinx 36 kbit primitives and 64-bit word size.

Config	Width (bits)	Peak (words)	Capacity (words)	BRAMs	Topology
L1-f HP0	32	465	512	1	1×lo/hi pair[c]
L1-s HP0	32	522	519	1.5	1×lo/hi pair[c] + 1×BRAM18[a]
L1-f/s HP1	64	940	1024	2	1×lo/hi pair
L3-f/s HP0	64	984	1024	2	1×lo/hi pair
L3-f/s HP1	64	1531	1536	3	1×lo/hi pair + 1×single[b]
L5-f HP0	64	1424	1536	3	1×lo/hi pair + 1×single[b]
L5-s HP0	64	1568	2048	4	2×lo/hi pair
L5-f/s HP1	64	2533	2560	5	2×lo/hi pair + 1×single[b]

[a] $\boldsymbol{\alpha}$ tail and h_{aux} exceed the BRAM36; a BRAM18 stores the overflow using both TDP ports.
[b] Single BRAM uses both TDP ports for lo/hi halves (no dual-port access for upper addresses).
[c] Pair of RAM blocks with 32-bit data width for accessing low/high halves of a 64-bit word.

Table 13: Scratchpad memory usage (64-bit words) per pipeline stage.

(a) HP0 variant.

Config	Words per stage								
	DFS	Haux	Hlines	GenV	ProcV	ORE	Peak	Bottleneck	BRAMs
L1-f	174	465	465	322	350	32	465	Haux	1
L1-s	235	253	522	322	350	22	522	Hlines	1.5
L3-f	248	876	876	643	729	48	876	Haux	2
L3-s	356	984	984	643	729	32	984	Haux	2
L5-f	331	1424	1424	1092	1212	64	1424	Haux	3
L5-s	475	1568	1568	1092	1212	42	1568	Haux	4

(b) HP1 variant.

Config	Words per stage								
	DFS	Haux	Hlines	GenV	ProcV	ORE	Peak	Bottleneck	BRAMs
L1-f	174	921	921	685	741	32	921	Haux	2
L1-s	231	940	940	700	756	22	940	Haux	2
L3-f	248	1524	1524	1407	1503	48	1524	Haux	3
L3-s	356	1308	1308	1435	1531	32	1531	ProcV	3
L5-f	331	2288	2288	2385	2497	64	2497	ProcV	5
L5-s	475	2432	2432	2421	2533	42	2533	ProcV	5

Table 14: Per-variable scratch space breakdown of the software reference implementation (fast/short configurations differ by <1%).

Variable	Notation	L1 (Bytes)	L3 (Bytes)	L5 (Bytes)
	PIOP Challenge Data			
`chall_a_H`	$\mathbf{H}^{[\gamma]}$	165 760	441 504	635 648
`chall_unitary`	γ'	896	1752	3328
`chall_a`	γ	64	96	128
`chall_a_y`	$y^{[\gamma]}$	16	24	32
	VOLE Data			
`v`	$\mathbf{V}$	18 304	39 936	69 888
`v2`	$\mathbf{V}$ (copy)	18 304	39 936	69 888
`u`	$\mathbf{u}$	2288	4992	8736
`cchk_res_v`		2304	4992	8704
	Circuit Data			
`inputs_f`	circuit inputs	19 712	42 048	73 216
`cchk_matrix`	$\mathbf{M}$	18 000	37 856	65 008
`rvp_f_cz`		128	192	256
`res_f`		80	120	160
`mux_f`		80	120	160
	GGM Tree			
`ggm_tree`		1288	2024	2664
`hidden_leaves_idx`		64	96	128
`root_seed`		16	24	32
	Hash State & Proof			
`hash_com`	h_{com}	32	48	64
`hash_aux`	h_{aux}	32	48	64
`hash_lines`	h_{lines}	32	48	64
`delta0`		24	32	40
`delta1`		16	24	32
Local variables subtotal		**247440**	**615912**	**938240**
Sub-function scratch reservation		4752	9384	9536
Alignment padding		96	112	96
Total scratch space		**252288 (246.4 KB)**	**625408 (610.8 KB)**	**947872 (925.7 KB)**

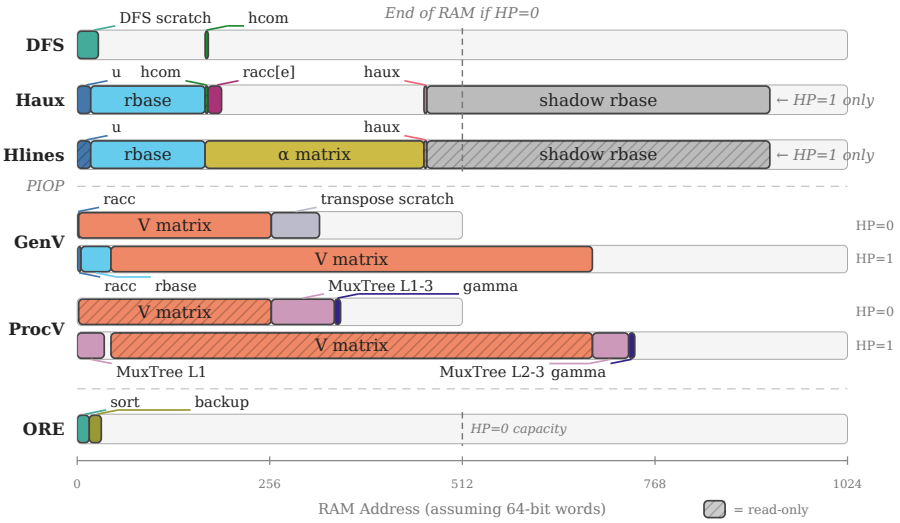


Fig. 4: Scratchpad memory layout for security level 1 fast.

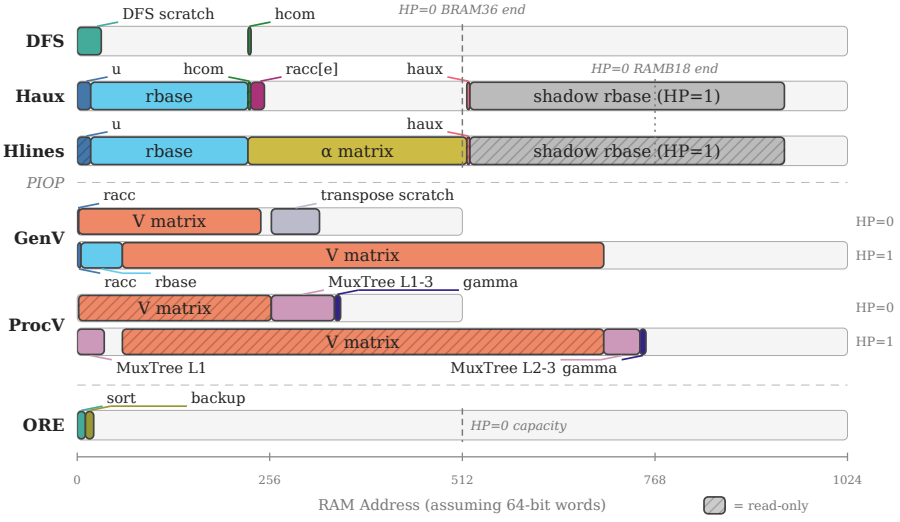


Fig. 5: Scratchpad memory layout for security level 1 short. Note: the HP0 variant uses a single BRAM36 in 32-bit-wide mode (512 words); the tail of the $\boldsymbol{\alpha}$ matrix and the h_{aux} region overflow into an additional BRAM18.

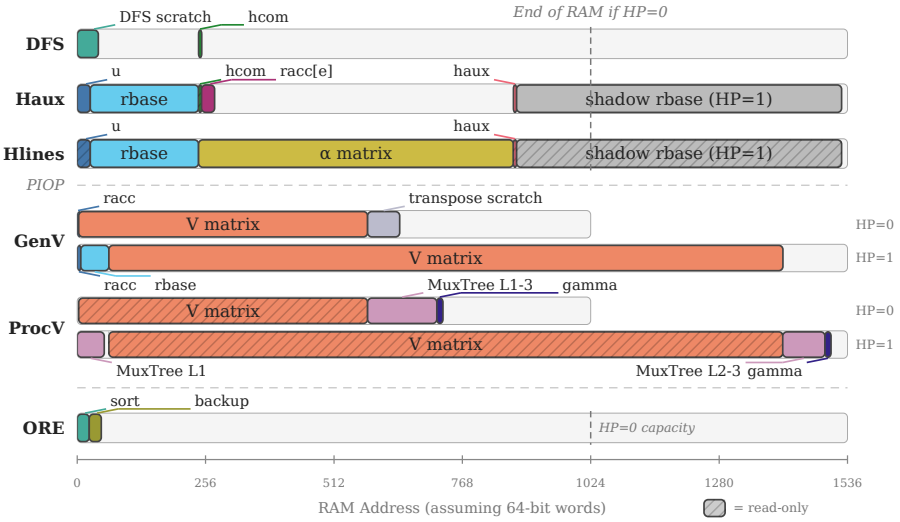


Fig. 6: Scratchpad memory layout for security level 3 fast.

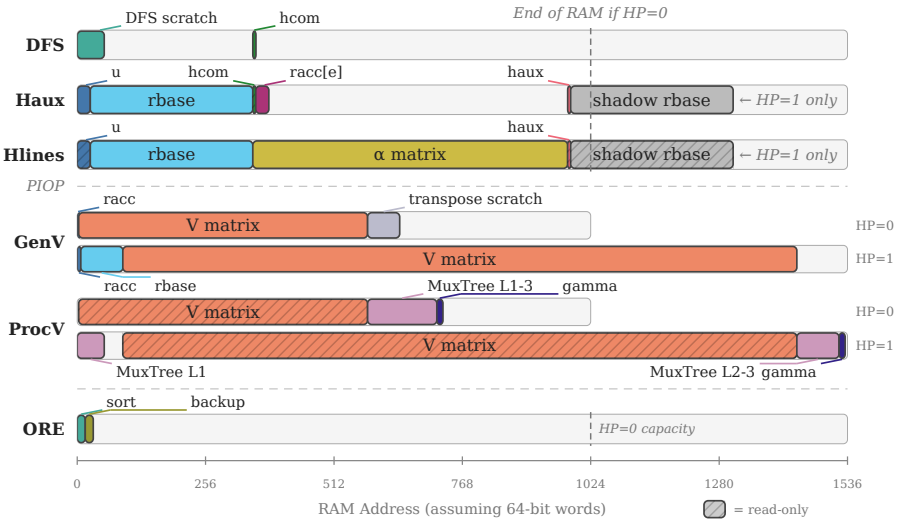


Fig. 7: Scratchpad memory layout for security level 3 short.

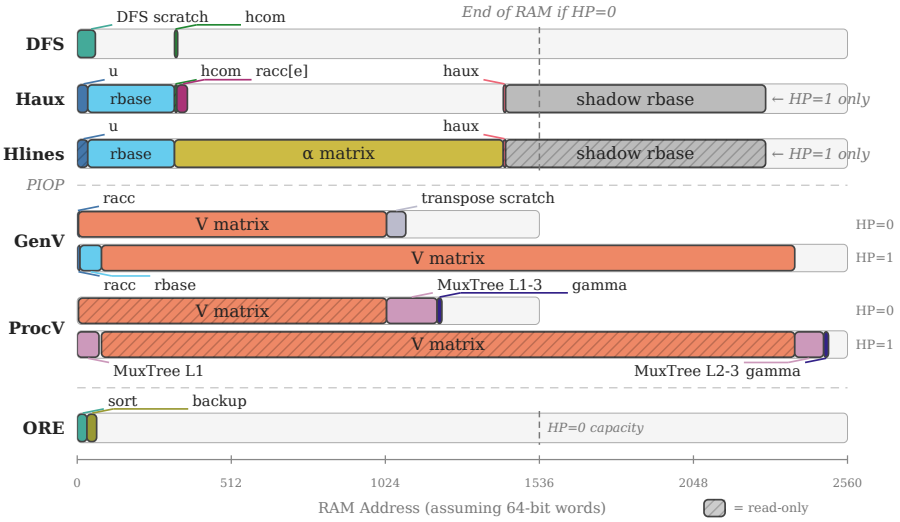


Fig. 8: Scratchpad memory layout for security level 5 fast.

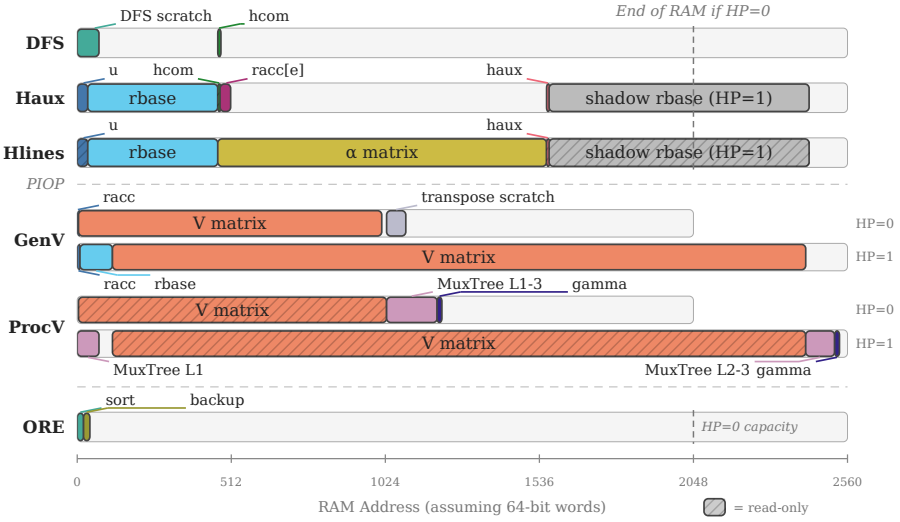


Fig. 9: Scratchpad memory layout for security level 5 short.

Table 15: Absolute latencies and resulting speedup factors for different GGM tree cache depths across all parameter sets.

Config	Cache Depth	Cache BRAMs	HP0		HP1	
			Cycles	Speedup	Cycles	Speedup
L1-f	8*	1	3 581 595	1.00×	2 747 144	1.00×
	9	2	3 390 875	1.06×	2 655 528	1.03×
	10	4	3 219 979	1.11×	2 574 488	1.07×
	11	8	3 051 339	1.17×	2 509 144	1.09×
L1-s	8*	1	15 192 059	1.00×	10 027 991	1.00×
	9	2	14 123 515	1.08×	9 524 695	1.05×
	10	4	13 085 513	1.16×	9 021 999	1.11×
	11	8	12 065 343	1.26×	8 520 503	1.18×
	12	16	11 072 818	1.37×	8 046 179	1.25×
	13	32	10 147 133	1.50×	7 642 776	1.31×
L3-f	7*	1	7 461 845	1.00×	6 263 869	1.00×
	8	2	7 120 213	1.05×	6 075 837	1.03×
	9	3	6 781 797	1.10×	5 891 261	1.06×
	10	6	6 453 685	1.16×	5 717 517	1.09×
	11	12	6 145 005	1.21×	5 563 085	1.13×
L3-s	7*	1	58 046 047	1.00×	39 943 255	1.00×
	8	2	54 409 183	1.07×	37 912 023	1.05×
	9	3	50 772 703	1.14×	35 881 175	1.11×
	10	6	47 136 991	1.23×	33 851 095	1.18×
	11	12	43 502 815	1.33×	31 822 583	1.25×
	12	24	39 871 711	1.46×	29 797 623	1.34×
	13	48	36 344 831	1.60×	27 876 887	1.43×
	14	96	32 960 543	1.76×	26 098 743	1.53×
	15	192	29 897 519	1.94×	24 640 823	1.62×
L5-f	7*	1	11 027 904	1.00×	9 457 851	1.00×
	8	2	10 571 712	1.04×	9 204 411	1.03×
	9	3	10 140 192	1.09×	8 975 963	1.05×
	10	6	9 712 864	1.14×	8 752 411	1.07×
	11	12	9 325 280	1.18×	8 568 603	1.10×
	12	24	8 943 072	1.23×	8 388 891	1.13×
L5-s	7*	1	81 932 683	1.00×	58 073 654	1.00×
	8	2	77 149 067	1.06×	55 419 958	1.05×
	9	3	72 365 963	1.13×	52 766 774	1.10×
	10	6	67 583 883	1.21×	50 114 992	1.16×
	11	12	62 803 851	1.30×	47 465 552	1.22×
	12	24	58 162 696	1.41×	44 954 989	1.29×
	13	48	53 674 221	1.53×	42 597 106	1.36×
	14	96	49 427 128	1.66×	40 479 765	1.43×
	15	192	45 398 008	1.80×	38 579 557	1.51×

* Default cache depth used in Table 4.