FAEST for Memory-Constrained Devices

Diego F. Aranha, Johan Degn, Jonathan Eilath, Kent Nielsen and Peter Scholl

Aarhus University, Aarhus, Denmark

Abstract. We introduce a new compact and constant-time implementation of the FAEST v1.1 signature scheme that allows it to run in resource-constrained Arm Cortex-M4 microcontrollers under 190M cycles for signing or verifying at level 1 security. The main technique for reducing the memory footprint is a new abstraction to reuse or recompute VOLEs on demand, which reduces memory consumption by at least an order of magnitude.

Keywords: FAEST · VOLE-in-the-Head · Embedded · Memory Footprint

1 Introduction

As quantum computing continues to advance, many traditional cryptographic schemes will become vulnerable to efficient cryptanalysis. To address this concern, NIST started a project in 2016 to develop standards for post-quantum cryptography, which culminated in the standardization of hash-based and efficient lattice-based signature schemes. A notable entry to this first process was Picnic 3 [CDG+20], built from the Multi-Party-Computation-in-the-Head paradigm (MPCitH), and selected as a Round 3 alternate candidate. As an attempt to diversify the hardness assumptions, the initial effort by NIST was followed by the call for additional signature schemes in 2022[1].

FAEST [BBdSG+23] is one of the signature schemes that recently reached Round 2 of the follow-up process. It is the only candidate building on the non-interactive Vector Oblivious Linear Evaluation (VOLE) construction: VOLE-in-the-Head (VOLEitH) [BBD+23], which allows for using the efficient VOLE-based zero-knowledge (ZK) proof system QuickSilver [YSWW21]. FAEST is particularly compelling because it relies solely on the security of well-established cryptographic primitives, specifically AES encryption and the SHA-3 hash function, which are widely regarded as conservative assumptions. FAEST is also built in a modular way, enabling the replacement of its underlying cryptographic primitives, the VOLEitH protocol, or the ZK proof system if any are shown to be insecure.

While certain quantum-safe algorithms can be appealing from a design perspective, their practical deployment may present some obstacles. As standardized post-quantum signature schemes will be deployed on hardware with all kinds of limitations, it is important to develop implementations that are efficient on resource-constrained devices. Many post-quantum signature schemes have already been ported to embedded devices, within the `pqm3`[2], `pqriscv`[3] and `pqm4` [KRSS19] frameworks. However, this task can be much more challenging for schemes with a large memory footprint and heavy dynamic allocation. For example, the FAEST reference implementation requires at least 466 KB of memory to compute a signature, and 14 MB in the worst case.

E-mail: dfaranha@cs.au.dk (Diego F. Aranha), johantdegn@gmail.com (Johan Degn), eilath@cs.au.dk (Jonathan Eilath), keni@cs.au.dk (Kent Nielsen), peter.scholl@cs.au.dk (Peter Scholl)

[1] https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures

[2] https://github.com/mupq/pqm3

[3] https://github.com/mupq/pqriscv

This work is licensed under a "CC BY 4.0" license.

Date of this document: 2026-06-29.

Developing an embedded implementation of a post-quantum signature scheme comes with additional challenges. Interoperability with the reference implementation is crucial, since changing the parameter set and breaking compatibility would be a major drawback in practice. Ensuring the compatibility ensure that the security of the embedded implementation can be directly reduced to the security of the FAEST signature scheme. This paper raises and answers the following question affirmatively: *is it possible to make* FAEST *run efficiently on embedded microcontrollers?*

Contributions. We introduce a new implementation of the FAEST signature scheme that is tailored towards the Arm Cortex-M4 microcontrollers and compatible with reference FAEST signatures. This new implementation comes with a new construction, deemed VOLE Black Box, which enables the on-demand computation of only part of a VOLE, in an oblivious manner to the signing and verification algorithms. The VOLE Black Box is a VOLEitH implementation that is optimized for low memory consumption, at the cost of only a logarithmic increase in computational cost. This new implementation reduces the memory consumption to 52.1 KB of memory for level 1 parameters, including the signature, and runs in constant-time with respect to secret data. To the best of our knowledge, this is the first implementation of a VOLEitH scheme on an embedded device with a resulting latency in the main variant of 187 million cycles to compute a signature, and 179 million cycles to verify. Due to an order of magnitude lower memory requirement, this implementation additionally increases the performance of the reference code by 1.40x for signing and 1.55x for verification on *a standard office laptop from 2020* due to better utilization of the memory hierarchy, effectively replacing the original reference implementation.

Related work. Few VOLEitH-based signature schemes comparable to FAEST exist, but we expect our contributions to help make others viable on embedded devices. The original FAEST authors have proposed a new set of schemes called FAESTER, KuMQuat, and MandaRain [BBM+24]. The first scheme utilizes the ability of QuickSilver to prove statements over higher degree polynomials, increasing both speed and communication complexity. The latter two are designed under different hardness assumptions, namely the multivariate quadratic (MQ) problem and the one-way function candidate Rain. The KuMQuat scheme was developed concurrently with another MQ-based signature scheme [Bui24], both significantly reducing the size of the signature while keeping the computational cost competitive.

Our results are more easily comparable to other benchmarks at 128-bit security from the `pqm4` testing framework. The standardized scheme Dilithium [LDK+22] computes a signature in 7.3 million cycles, and a new version of SPHINCS+ called SPHINCS-alpha [YCZ24] requires 382 million cycles. For more directly comparable schemes, an embedded implementation of Picnic 3 [ABE+21] computes a signature in 302 million cycles, and MQOM [BFR23] takes 149 million cycles. When compared to other schemes relying on symmetric primitives, our embedded implementation is competitive in terms of speed: only 1.25x slower than MQOM, while being 1.61x faster than Picnic and around 2x faster than SPHINCS-alpha. In terms of memory consumption, signing requires 52 KB for Dilithium, 280 KB for SPHINCS-alpha, and 32 KB for Picnic. Our embedded FAEST implementation uses 42.7 KB of RAM, again demonstrating competitive performance.

Disclaimer. This latest revision omits our masked implementation [ADE+25], after a reviewer of our submission to IACR Communications in Cryptology found an attack against the seed masking for which we could not find an efficient countermeasure. We thank the anonymous reviewer for finding the flaw. Since that submission, subsequent work has employed a variant of the low-memory tree-walk algorithm presented here, but

in the context of correlated trees in the MQOM signature scheme [BF26]; and adapted it to the one-tree setting used by FAEST and other MPCitH algorithms.

Organization. The rest of the paper is organized as follows: Section 2 gives an overview of the VOLE-in-the-Head framework and the FAEST signature scheme. Section 3 introduces our embedded implementation and the VOLE Black Box technique, together with other optimizations. We conclude the paper in Section 5, including a discussion about how our techniques can be adapted to newer versions of FAEST.

2 Preliminaries

2.1 Notation

For $n \in \mathbb{N}$ we denote $[n]$ to be the set $\{1, .., n\}$ and for $a, b \in \mathbb{N} : a \leq b$ we denote $[a, b]$ as the set $\{a, ..., b\}$ and $[a, b)$ as the set $\{a, ..., b-1\}$.

Vectors are column vectors, and we denote them using bold letters, as in $\mathbf{x} \in \{0,1\}^n$. For matrices we use a capital bold letter, $\mathbf{X} \in \{0,1\}^{n \times m}$. We use the following notation for indexing:

- $\mathbf{x}[i]$ denotes the i-th element of the vector.
- $\mathbf{X}_i$ denotes the i-th row of a matrix.
- $\mathbf{X}^j$ denotes the j-th column of a matrix.
- $\mathbf{x}||\mathbf{y}$ denotes the concatenation of two vectors.
- $[\mathbf{X}\ \mathbf{Y}]$ denotes the concatenation of two matrices of equal number of rows into a single matrix.

We use the notation $x \leftarrow S$ to sample values uniformly random from a set or distribution, and say that x is sampled from S.

We define the finite fields $\mathbb{F}_{2^8}, \mathbb{F}_{2^{64}}, \mathbb{F}_{2^{128}}, \mathbb{F}_{2^{192}}$, and $\mathbb{F}_{2^{256}}$ as polynomials over $\mathbb{F}_2$ taken modulo an irreducible polynomial. These polynomials are given in the FAEST v1.1 specification [BBdSG+23]. Lastly, we define λ to be the security parameter.

2.2 FAEST and VOLE-in-the-Head

We briefly review the main components and design of FAEST, which uses the VOLE-in-the-head paradigm [BBD+23] for zero-knowledge proofs.

2.2.1 VOLE

An Oblivious Linear Evaluation (OLE) over a field $\mathbb{F}$ is an interactive protocol between a sender and a receiver. The sender provides two inputs $u, v \in \mathbb{F}$ and the receiver gives a single input $\Delta \in \mathbb{F}$. The interactive protocol computes and outputs q to the receiver such that:

$$q = u\Delta - v. \tag{1}$$

The ideal functionality of the OLE protocol states that the receiver only learns Δ, q and nothing about u, v. Likewise, the sender learns nothing about Δ, q.

A Vector OLE (VOLE) is an extension of an OLE, where instead of a single correlation, a long series of correlations is constructed with the same scalar Δ. In a VOLE, the inputs

of the sender are a vector, $\mathbf{u}$, of length n and a matrix, $\mathbf{V}$, of size $n \times \lambda$, and the output of the receiver is a matrix $\mathbf{Q} \in \mathbb{F}^{n \times \lambda}$, such that:

$$\mathbf{Q} = \mathbf{u}\Delta - \mathbf{V}$$

We refer to this as a VOLE relation, or *VOLE correlation* if the inputs $\Delta, \mathbf{u}$ and $\mathbf{V}$ are random. We call Δ the global key, $\mathbf{V}$ for the VOLE tag and $\mathbf{Q}$ for the VOLE key. A VOLE protocol produces triples of the form (u_i, v_i, q_i) such that (1) is satisfied for each triple. The tag v_i can be seen as an information-theoretic MAC on the sender's value u_i, under the key (q_i, Δ) known to the receiver. As long as these MACs are under the same global key Δ, they are linearly homomorphic.

2.2.2 VOLE-in-the-Head

VOLE-in-the-Head (VOLEitH) [BBD+23] is a recent protocol that enables VOLE correlations to be generated non-interactively in a publicly verifiable manner. The protocol is a relaxation of the VOLE functionality, whereby the sender (or signer) eventually learns the random input Δ of the receiver (or verifier).

All-But-One Vector Commitments. A key component of VOLEitH is a form of vector commitment, where the signer can commit to N messages, and then later open $N-1$ of them to the verifier. When the committed messages are only required to be pseudorandom, this can be done with an opening size of only $O(\lambda \log N)$ bits using a puncturable pseudorandom function (PRF) based on a GGM tree [GGM86] with a length-doubling pseudorandom generator (PRG) and a hash function.

For $N = 2^d$, the construction works by computing a balanced binary tree of height d, level by level using the PRG, where the master key is the root node of the tree, and each node's children are obtained by applying the length-doubling PRG. Each leaf of the tree is then hashed, using a hash function on the form $\{0,1\}^\lambda \to \{0,1\}^\lambda \times \{0,1\}^{2\lambda}$ to obtain a pair consisting of a message $\mathsf{sd}_{i\in[N]}$ and commitment $\overline{\mathsf{com}}_{i\in[N]}$. To optimize communication, the leaf commitments are hashed together to get a single commitment com for all messages.

From the construction, it is clear that it provides N random messages and a commitment com to these messages. After sending com, the signer can later open all messages except the j-th one, by first defining the punctured key consisting of all d siblings on the path from the root to the leaf containing sd_j. These d nodes suffice to reconstruct the other $N-1$ leaves. By additionally including the commitment $\overline{\mathsf{com}}_j$ in the opening, the receiver is able to verify the opening of the commitment by reconstructing com.

Vector Commitments to VOLE. An $(N-1)$-out-of-N vector commitment can be converted into a VOLE over $\mathbb{F}_{2^d}$ between the signer and verifier. Let the unopened index be Δ, which we view as a d-bit integer mapped into $\mathbb{F}_{2^d}$. The messages $\mathsf{sd}_{i\in[N]}$ can be converted into a VOLE of arbitrary length, by using a PRG to expand each seed into a length-ℓ string $\mathbf{r}_i = \mathrm{PRG}(\mathsf{sd}_i) \in \mathbb{F}_2^\ell$, and having the signer define its VOLE outputs as:

$$\mathbf{u} = \sum_{i\in[N]} \mathbf{r}_i, \mathbf{V} = \sum_{i\in[N]} \mathbf{r}_i \cdot i \tag{2}$$

Here, each index i is viewed in $\mathbb{F}_2^d$, and multiplied coordinate-wise with every bit of $\mathbf{r}_i$. The verifier, meanwhile, does not know sd_Δ, but can compute:

$$\mathbf{Q} = \sum_{i\in[N]\setminus\Delta} \mathbf{r}_i \cdot (\Delta - i) = \sum_{i\in[N]} \mathbf{r}_i \cdot (\Delta - i) = \mathbf{u}\Delta - \mathbf{v}$$

Algorithm 1: $\mathsf{FAEST.Sign}(\mathsf{msg}, \mathsf{pk} = (x, y), sk)$

$\mu = \mathsf{H}_1(\mathsf{pk}||\mathsf{msg})$;
$(r, \mathsf{iv}) = \mathsf{H}_3(\mathsf{sk}||\mu||\rho)$ // ρ is a random nonce ;
$(h_{\mathsf{com}}, \mathsf{decom}_0, ..., \mathsf{decom}_{\tau-1}, c_1, ..., c_{\tau-1}, \mathbf{u}, \mathbf{V}) = \mathsf{VOLECommit}(r, \mathsf{iv})$;
$\mathsf{chall}_1 = \mathsf{H}_2^1(\mu, ||h_{\mathsf{com}}||c_1||...||c_{\tau-1}||\mathsf{iv})$;
$(\tilde{\mathbf{u}}, h_V) = \mathsf{VOLEHash}(\mathsf{chall}_1, \mathbf{u}, \mathbf{V})$;
$\mathbf{w} = \mathsf{ExtendWitness}(\mathsf{sk}, x)$;
$\mathbf{d} = \mathbf{w} \oplus \mathbf{u}$;
$\mathsf{chall}_2 = \mathsf{H}_2^2(\mathsf{chall}_1||\tilde{\mathbf{u}}||h_V||\mathbf{d})$;
$(\tilde{\mathrm{a}}, \tilde{\mathrm{b}}) = \mathsf{AESProve}(\mathbf{w}, \mathbf{u}, \mathbf{V}, \mathsf{pk}, \mathsf{chall}_2)$;
$\mathsf{chall}_3 = \mathsf{H}_2^3(\mathsf{chall2}||\tilde{\mathrm{a}}||\tilde{\mathrm{b}})$;
$(\mathsf{pdecom}_0, ..., \mathsf{pdecom}_{\tau-1}) = \mathsf{VC.Open}(\mathsf{decom}_0, ..., \mathsf{decom}_{\tau-1}, \mathsf{chall}_3)$;
return $\sigma = ((c_i)_{i\in[1..\tau)}, \tilde{\mathbf{u}}, \mathbf{d}, \tilde{\mathrm{a}}, (\mathsf{pdecom}_i)_{i\in[0..\tau)}, \mathsf{chall}_3, \mathsf{iv})$

Figure 1: Condensed description of the FAEST signing algorithm, taking as input the public key, the secret key, and the message to sign, and returning the signature.

This gives a length-ℓ VOLE correlation over $\mathbb{F}_{2^d}$. The signer and verifier each need to perform $O(N) = O(2^d)$ PRG evaluations, so using this directly to create a large VOLE over $\mathbb{F}_{2^\lambda}$, for security parameter λ, is infeasible. Instead, a better approach described by [Roy22] is used in FAEST where $\tau = \lambda/k$ independent vector commitment instances are constructed, each of size $N = 2^k$, creating τ independent VOLE correlations. After sending the commitment of each VOLE correlation, the signer additionally sends $\tau - 1$ correction values, called c_i, to correct the verifier's part of the VOLE correlation so that each VOLE uses the same $\mathbf{u}$ vector. This allows the signer (resp. verifier) to concatenate all its τ $\mathbf{V}$ (resp. $\mathbf{Q}$) matrices to form a single VOLE over $\mathbb{F}_{2^\lambda}$ with length ℓ.

The VOLEitH protocol combines these two constructions to create a VOLE correlation between the sender and the receiver. By applying the Fiat–Shamir transform [FS87], the VOLE correlations can be generated non-interactively and made publicly verifiable by selecting the unopened index Δ as the output of a hash function which input is the transcript of the protocol.

2.2.3 FAEST Overview

FAEST builds a signature following the Schnorr paradigm, via a publicly verifiable zero-knowledge proof of knowledge based on VOLEitH. The key pair $(\mathsf{pk}, \mathsf{sk})$ is defined as $\mathsf{pk} = (x, y)$ and $\mathsf{sk} = k$ such that $E_k(x) = y$, where x is the plaintext, y is the ciphertext, k is the secret and E is a block cipher. In FAEST, the block cipher is instantiated using AES.

A condensed description of the FAEST signing algorithm is shown in Algorithm 1. The signing algorithm works in 4 phases, each phase divided by a challenge created by a Fiat-Shamir transform. The first phase is the VOLE commitment phase, discussed in the previous section, where the signer computes its VOLE outputs $\mathbf{u} \in \mathbb{F}_2^{\hat{\ell}}, \mathbf{V} \in \mathbb{F}_2^{\hat{\ell}\times\lambda}$ alongside the commitment, h_{com}. Here, the VOLE outputs are of length $\hat{\ell}$, a parameter slightly longer than the witness length ℓ, to allow for some additional mask values that are needed. In the second phase, the challenge chall_1 is used to check the consistency of the VOLE with the VOLEHash linear universal hash function, which ensures the signer did not cheat in the c_i correction values. Then, the signer uses the VOLE to authenticate the AES witness $\mathbf{w}$, replacing $\mathbf{u}$. In the third phase, the signer runs the QuickSilver zero-knowledge proof system [YSWW21] to prove a set of constraints on the committed witness, and combines these into the result $(\tilde{a}, \tilde{b})$ using the next challenge, chall_2. In the final phase, chall_3 defines

the verifier's VOLE input $\Delta \in \{0,1\}^\lambda$, which is divided into τ substrings, each used to open one of the all-but-one vector commitments. This allows the verifier to compute its VOLE output and verify the entire zero-knowledge proof.

Details on AES Expanded Witness. In the second phase of the FAEST signing algorithm, the secret key is expanded to define the witness used in the ZK proof system. Concretely, this witness has the form:

$$\mathbf{w} = (\mathsf{sk} \| \texttt{non_lin_word_bits} \| \texttt{shiftrow_bits})$$

Here, $\mathsf{sk} \in \{0,1\}^\lambda$ is the original AES secret key. This is followed by the value of every word that is an output of the non-linear SubWord operation run during the key scheduling of AES. Then, the witness includes hints about the state during the AES encryption routine, in particular the state after each ShiftRows operation is included, for every round except the last. Since ShiftRows is linear and immediately follows the S-box computation, this suffices to give enough information to verify the full AES execution in zero-knowledge.

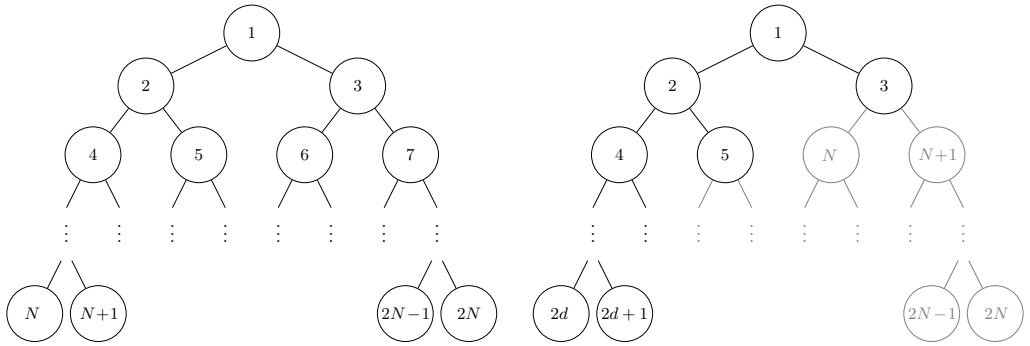
Details of the ZK Proof. In the third phase of the FAEST signing algorithm, the ZK proof of AES is computed. Using the linearly homomorphic commitment to the expanded witness, the verifier can locally emulate all internal steps of AES except the non-linear component of the S-box, which is a multiplicative inverse in $\mathbb{F}_{2^8}$. As the expanded witness contains the state after each S-box, the verifier can compute the output of the previous S-box by a backwards computation through the AES round and the input to the next S-box by a forward computation through the AES round. To ensure that some input x to the multiplicative inverse is consistent with the output, y, the signer and the verifier use the QuickSilver proof system to verify the constraint $xy = 1$ in $\mathbb{F}_{2^8}$[4]. To check these constraints in zero-knowledge, the signer computes a linear combination of the constraints and hides it with some additional VOLE commitments. This allows the verifier to do the same and check that the linear combination of its constraints satisfies the VOLE correlation. This way, the consistency of all S-boxes is verified, convincing the verifier of the signature.

Variants of FAEST. The FAEST signature scheme comes in a series of variants based on the encryption function used, and the parameters of the VOLEitH construction. For each security level $\lambda \in \{128, 192, 256\}$, four variants exits. The variants λf setting $d = 8$ in the VOLEitH construction, and λs setting $d \in \{11, 12\}$ in the VOLEitH construction both use AES as the encryption function. The variants EM-λf and EM-λs use the Even-Mansour variant with Rijndael$\lambda_x(k)$ as the encryption function. Here, the encryption key x is public, and the plaintext k is the secret key.

3 Embedded Implementation

This section describes the techniques used to reduce the memory consumption of FAEST in our embedded implementation. This includes a general approach that also applies to other signature schemes based on the VOLEitH paradigm, such as [OTX24, BBGK24]. We also describe additional optimizations which are specific to FAEST on memory-constrained devices. The performance results of the embedded FAEST implementation can be seen in Section 4.

[4] To rule out the case that $x = y = 0$, the key generation in FAEST v1.1 uses rejection sampling.



(a) FAEST reference implementation (b) Memory optimized tree walk

Figure 2: Illustrated computation order of the memory-efficient tree walk algorithm. Each node is numbered according to the order of computation. Diagram (a) shows the reference implementation computing layer by layer, where diagram (b) shows the preorder tree walk computing leftmost leaf first. Nodes with gray labels are not stored in memory.

3.1 The VOLE Black Box

The most memory-consuming part of FAEST is by far the computation of the VOLE matrix $\mathbf{V}$ (for the signer) or $\mathbf{Q}$ (for the verifier), which is described in Section 2.2.2. We introduce an abstraction for accessing these matrices in a memory-friendly manner, which we call the VOLE Black Box (VBB), since the signing and verification algorithms remain oblivious to the state of the VBB and how the VOLE is actually computed. This allows for the VBB to be used for all VOLEitH-based protocols, enabling the option to run on resource-constrained devices. The VBB consists of a high-level interface for the signing and verification algorithms to access the VOLE, which uses two memory-optimized algorithms: a tree-walk algorithm for vector commitments (VC), and an in-place vector commitments to VOLE computation (VC2VOLE).

3.1.1 Vector Commitment Tree-Walk

Implementing the GGM-based vector commitments according to the FAEST specification [BBD+23] gives a memory requirement of approximately $5\lambda N$ bits, where $N = 2^d$. First, observe that the output messages $\mathsf{sd}_0, \ldots, \mathsf{sd}_{N-1}$ would require λN memory to be stored. Futhermore, $\overline{\mathsf{com}}_i$ for $i \in [0...N-1]$ must be stored alongside to compute the commitment $\mathsf{com} = \mathsf{H}(\overline{\mathsf{com}}_0, \ldots, \overline{\mathsf{com}}_{N-1})$, requiring $2\lambda N$ additional memory. FAEST additionally stores all internal nodes of the GGM tree, to save re-computing the tree during opening, contributing an additional $\lambda(2N-1)$ bits of memory. For the $\lambda = 128$ variants of FAEST v1.1, doing this naively for all τ trees at once requires between 260 kB and 2.9 MB of memory for the fast and small variant, respectively.

To reduce this memory cost, we use a tree-walk strategy to incrementally compute the outputs of each vector commitment. By computing a single leaf message, sd_i, at a time, we can improve the memory requirements to $O(\lambda \log N)$, since computing sd_i only requires computing a path from the root of the GGM tree to the i-th leaf. This path is of length $d = \log N$, thus requiring $\lambda \log N$ memory. The leaf hash $\overline{\mathsf{com}}_i$ can be absorbed into com on-the-fly, removing the need to store all the leaf values.

Doing a pre-order tree walk over one GGM tree, the seeds will be output in order, from sd_0 to sd_{N-1}. If the algorithm following the vector commitments consumes these seeds at once, only one seed is required in memory at any given point. If both children of a node are stored when computed, i.e, two nodes in each level, no child will ever be recomputed;

hence, the tree walk requires the optimal number of PRG calls. This is illustrated in Figure 2, where the preorder tree walk is shown while computing the neighbouring nodes in each level. This reduces the memory consumption from $O(N)$ to $O(1)$ seeds, with no increase in computation (except that the tree may need to be recomputed later). By computing each tree of the τ trees separately, only one tree walk will be in memory at any given point, further decreasing the memory consumption by a factor τ. This independent computation of trees approach closely resembles the streamlining of MPCitH schemes presented in [BBB+24], which ensures only one tree is present in memory at once.

At the end of the signing algorithm, the signer needs to compute the openings of the vector commitments. Recall that this is done by revealing one node on each level in the GGM tree except the root, allowing the verifier to recover all but one leaf. Since we no longer store the GGM trees in memory, the opening now needs to be recomputed. Note that we do not need to recompute all nodes of the tree, as only the path of the revealed nodes is required, increasing the number of PRG evaluations by $\tau \log N$. In total, the tree-walk algorithms for vector commitments slightly increase the number of PRG evaluations by $\tau \log N$, while decreasing the memory consumption from $O(\tau\lambda N)$ to $O(\lambda \log N)$ bits.

3.1.2 Memory-Optimized VC2VOLE

To convert the vector commitments into VOLE (as in Equation (2) for the signer), FAEST uses an optimized divide-and-conquer algorithm from [Roy22] (see also [BBdSG+23, Fig. 5.2]). This requires the storage of all expanded seeds, $\mathbf{r}_i$, giving a memory footprint of $N\hat{\ell}$ for each tree. Our approach instead consumes one sd_i from the tree walk and expands it to $\mathbf{r}_i$ immediately. Then, instead of storing all $\mathbf{r}_i$, we use it to incrementally update the VOLE correlation and then discard $\mathbf{r}_i$. Since the VOLE correlation is a sum of all the expanded seeds, they can all be applied independently of each other, so each column of the signer's VOLE can be computed using:

$$\mathbf{V}^j = \bigoplus_{i=0}^{N-1} \mathbf{r}_i(\lfloor i/2^j \rfloor \bmod 2), \quad \mathbf{u} = \bigoplus_{i=0}^{N-1} \mathbf{r}_i.$$

Here, each expanded seed is added to the j-th column if the j-th bit of the index i is set. Since more than one bit can be set in the index i, each $\mathbf{r}_i$ is applied $d/2$ times on average. This increases the number of XOR operations needed, compared with [Roy22], by a factor $O(\log N)$, but the memory consumption decreases to $\hat{\ell}$. Since the depth, d, of each GGM tree is limited to 12, this performance penalty is minor, because XOR operations are very cheap in practice.

3.1.3 VOLE Black Box Interface

Although the two memory-optimized algorithms for computing the VOLE relations significantly reduce memory consumption, the VOLE correlation still has a memory footprint of $\lambda\hat{\ell}$ for the tag and key during signing and verification, respectively. This may be prohibitive for the high-security variants of FAEST on resource-constrained devices, as they require more than 130 kB for $\lambda = 256$. By encapsulating the VOLE correlation behind an interface, we can allow the VBB to store only a subset of the VOLE correlation and recompute different subsets as needed. Using the techniques above, we can now define how the VBB interface operates. The interface provides three methods: `init(budget)`, `get_vole(i)`, and `open()`. The `init` function takes a *budget* parameter specifying the maximum memory consumption of the VBB; this determines the subset of the VOLE correlation the VBB can store, which we refer to as its cache. The `get_vole(i)` function returns the i-th OLE of the VOLE and triggers recomputation when necessary. Lastly, `open()` computes and returns the nodes from each GGM tree that must be revealed to the verifier during the opening.

In the $\hat{\ell} \times \lambda$ VOLE matrices $\mathbf{V}$ and $\mathbf{Q}$, each row v_i, q_i corresponds to a single OLE, where $v_i, q_i \in \mathbb{F}_{2^\lambda}$. If we wish to store only a range of n of these OLEs, say, from row j to row $j + n - 1$, we can modify the conversion to VOLE step so that, instead of computing all $\hat{\ell}$ bits of the expanded seed $\mathbf{r}_i$, we expand only the portion corresponding to that range, starting from the j-th bit. This allows the VBB to store a consecutive subset of the VOLE correlation. This approach can be generalized to arbitrary subsets by computing only the entries of $\mathbf{r}_i$ that fall within the subset. We omit this generalization here, as it is not necessary for our usage of the VBB.

This functionality is hidden inside the VBB function `get_vole(i)`, as it simply returns the i-th OLE when requested.[5] Each time a VOLE is requested that lies outside the current cache, the VBB has to recompute the vector commitment messages using the tree-walk algorithm, to compute the new subset of VOLE that should be present in the cache. Therefore, the memory budget provides a tradeoff between computation and storage costs. Using this interface ensures that the signing and verification algorithms are independent of the VBB and its configuration, so our approach generalizes to any VOLEitH-based signature scheme. Furthermore, the memory budget of the VBB can be configured independently for the signer and verifier, without any coordination.

Since the VOLE correlation is hidden within the VBB, the `open()` function enables the opening of the vector commitments. As this function returns the nodes of the tree that are required to reconstruct the verifier's part of the VOLE correlation. These nodes are computed as described at the end of Section 3.1.1. Since the VBB ensures the underlying algorithms are hidden to the protocol using the VOLE, the algorithms can efficiently be replaced by any further improvement to the VOLEitH construction. This is without any changes to the user of the VOLE, further increasing the modularity of the FAEST signature scheme.

3.2 FAEST-Specific Optimizations

In FAEST, most of the arithmetic is performed on elements of the binary field $\mathbb{F}_{2^\lambda}$. Since $\lambda \in \{128, 192, 256\}$, these arithmetic operations can become a performance bottleneck on constrained devices due to the limited word size and instruction set.

The FAEST reference implementation uses a simple implementation for binary field multiplication that multiplies one polynomial by the other bit by bit and reduces the result each time. Consequently, binary field multiplication constitutes a significant portion of the total runtime on the embedded device. Thus, we implement an optimized algorithm for binary field multiplication, combined with fast polynomial-specific implementations for modular reduction.

Furthermore, when using the VBB with a reduced storage size, part of the VOLE output has to be recomputed each time a requested OLE is not present in the cache. To avoid recomputing the same OLE multiple times, we modified the ZK proof implementation, resulting in an (almost) linear access pattern, reducing the amount of recomputations to a minimum as each OLE is only computed once.

3.2.1 Binary Field Arithmetic

We replaced the existing implementation of binary field arithmetic with the left-to-right López-Dahab comb method as proposed in [LD00]. It precomputes a table of size 2^w with the products of small polynomials and the first input polynomial. Splitting the second input polynomial into chunks of size w, each chunk is interpreted as a small polynomial, and the resulting polynomial is retrieved by a lookup in the table. The retrieved values are shifted according to the corresponding chunks' offset in the second polynomial and

[5] The VBB returns the OLE in column-major order, so when accessing an OLE in the ZK proof phase, the output must be transposed before use.

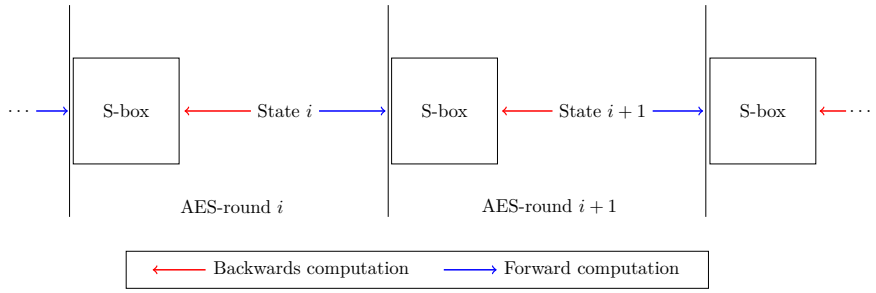


Figure 3: Diagram highlighting the direction of computation during the ZK proof. The AES state i stored in the witness is both computed in the forward direction of the AES round (blue arrow) and the backward direction of the AES round (red arrow). The reference implementation first computes the forward direction (blue) of all states, and then the backwards direction (red), requiring access to each state twice. The interleaved implementation computes both directions (red and blue) when a given state is accessed, requiring only a single access.

added together. The result is a polynomial of degree at most double any of the input polynomials.

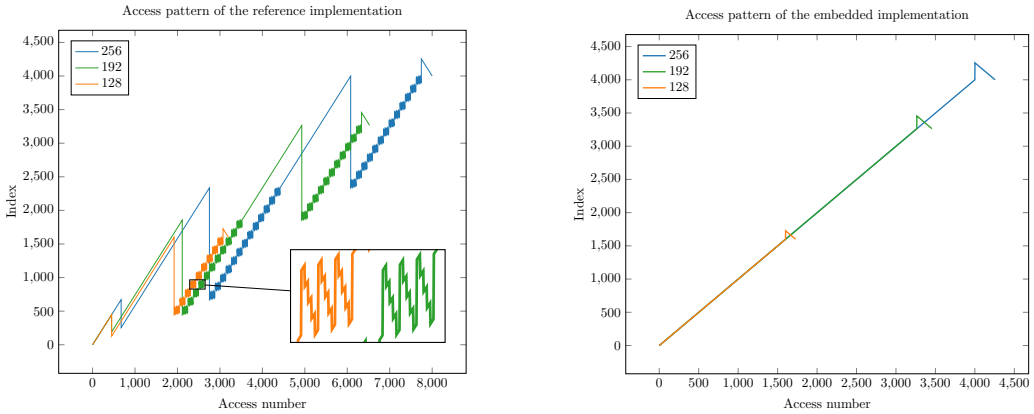
To reduce the resulting polynomial, we implemented highly optimized polynomial-specific modular reductions for each of the polynomials, exploiting their sparsity to perform just a few shifts and additions. This classical technique allows the reduction to be performed efficiently by rewriting the higher degree coefficients into lower degree coefficients multiplied by the low-degree terms of the irreducible polynomial. This approach is further simplified by the fact that the degrees of all irreducible polynomials are powers of two, making it possible to fit both the unreduced and reduced polynomials into a whole number of machine words.

3.2.2 Linear VOLE Access

Due to the ability of the VOLE Black Box to store only a subset of the VOLE, and the fact that each recomputation comes with a cost, the *access pattern* with which the VOLE is used now has a significant impact on performance. By access pattern, we refer to the sequence of indices requested from the VOLE. Clearly, random access to the VOLE will become very costly since the VBB only stores a consecutive subsection of the VOLE, hence can introduce a large number of recomputations. We have therefore modified the AESProve algorithm, computing the ZK-proof, to have an almost linear access pattern of the VOLE to minimise the number of recomputations.

As described in Section 2.2.3, the VOLE is used to create linear homomorphic commitments to the extended witness, which is based on the internal state of the AES key schedule and encryption algorithms. Thus, the order of how these commitments are made dictates how the VOLE is accessed. The reference implementation, which follows the specification, commits to all forward S-box computations, and then all backward computations. However, since the forward and backward computation originates from the same state, the VOLE elements needed to compute the commitment are based on the same OLEs, hence each OLE is accessed multiple times. This is illustrated in Figure 3, clearly showing that each state must be accessed twice. The same applies to the OLEs needed to commit to the round keys, as they are used in both the key expansion and during the encryption. This non-linear access pattern is shown in Fig. 4a.

We solve this problem with two ideas. First, we interleave the forward and backward computations such that they are computed simultaneously, with each state only being



(a) FAEST reference implementation

(b) Our implementation

Figure 4: Comparing access patterns of the VOLE in the reference implementation of the ZK proof with our implementation.

committed to once. This is done by committing to state i, using the respective OLEs needed, and then from state i, the backward computation computes the output of the previous S-box, and simultaneously, the forward computation computes the input to the next S-box. State i will now never be accessed again, likewise for the corresponding OLEs. As illustrated in Figure 3, both directions from state i will be computed when state i is being accessed. This is a more natural approach since the computation proceeds round by round of AES. As a bonus, by computing the input and output of the S-boxes state-by-state, we eliminate the need to store all the inputs and outputs at the same time. As soon as one state has been computed, we can also compute the relevant ZK constraints and absorb them into the linear combination to be checked. This not only creates a linear access pattern of the VOLE, but also decreases the memory usage of the ZK proof by a factor R, where R is the number of rounds in the AES encryption.

Secondly, for the key expansion phase, we allocate dedicated memory to a cache for the OLEs used to commit to the round keys. This avoids recomputations each time a round key is used during the encryption. This cache will contain ℓ_{ke} OLEs, which is the number of round key bits that are outputs from an S-box. By caching these OLEs, we further reduce the memory consumption of the ZK proof by not storing commitments to all round keys, but computing the commitment when needed from the cache. This has a small performance penalty, but reduces the memory consumption by $O(\lambda R)$.

Our final access pattern to the VBB is shown in Fig. 4b. Note that it is completely linear, save for the final λ OLEs used to mask the ZK constraint, which are accessed in reverse order. We can easily handle these λ OLEs without any recomputation, by treating these indices as a special case in the VBB implementation.

4 Experiments

In this section, we present the experimental results of our FAEST implementation.[6]

It is important to note that the implementation both for desktop and embedded, does not contain any optimizations that are architecture specific. The benchmarks serve to highlight the performance difference due to the changes, reducing the memory consumption, and give an indication of the performance on embedded devices.

[6] https://github.com/johandegn/pqm4-faest/

Table 1: Performance and memory benchmarks for 100 runs on a consumer-grade laptop. The memory consumption for both signing and verification includes the signature.

	Scheme	Memory Consumption			CPU Cycles		
		KeyGen	Sign	Verify	KeyGen	Sign	Verify
This work	128f	7.74 KB	52.1 KB	51.9 KB	790 K	60 M	59 M
	128s	7.74 KB	50.9 KB	50.8 KB	692 K	321 M	322 M
	192f	7.74 KB	127.7 KB	124.5 KB	845 K	113 M	98 M
	192s	7.74 KB	123.7 KB	120.8 KB	912 K	664 M	647 M
	256f	7.74 KB	208.8 KB	204.3 KB	964 K	143 M	133 M
	256s	7.74 KB	202.5 KB	198.4 KB	984 K	834 M	811 M
Ref.	128f	7.74 KB	514.0 KB	195.4 KB	835 K	66 M	60 M
	128s	7.74 KB	4.17 MB	2.30 MB	819 K	337 M	347 M
	192f	7.74 KB	1.09 MB	391.3 KB	1049 K	498 M	461 M
	192s	7.74 KB	11.5 MB	4.35 MB	1062 K	1026 M	1015 M
	256f	7.74 KB	1.78 MB	528.0 KB	1325 K	732 M	684 M
	256s	7.74 KB	14.0 MB	5.47 MB	1266 K	1401 M	1362 M
This work	EM-128f	7.74 KB	38.8 KB	39.3 KB	740 K	55 M	53 M
	EM-128s	7.74 KB	37.7 KB	38.1 KB	746 K	314 M	309 M
	EM-192f	7.74 KB	88.7 KB	91.9 KB	698 K	87 M	89 M
	EM-192s	7.74 KB	85.6 KB	88.8 KB	777 K	615 M	597 M
	EM-256f	7.74 KB	171.9 KB	177.0 KB	928 K	138 M	132 M
	EM-256s	7.74 KB	166.2 KB	171.3 KB	784 K	804 M	784 M
Ref.	EM-128f	7.74 KB	474.6 KB	169.2 KB	749 K	59 M	54 M
	EM-128s	7.74 KB	3.76 MB	1.98 MB	763 K	329 M	323 M
	EM-192f	7.74 KB	1.01 MB	303.9 KB	923 K	327 M	355 M
	EM-192s	7.74 KB	10.5 MB	3.37 MB	937 K	823 M	859 M
	EM-256f	7.74 KB	1.74 MB	486.4 KB	1135 K	602 M	632 M
	EM-256s	7.74 KB	13.8 MB	5.04 MB	1233 K	1245 M	1291 M

4.1 Embedded Compared to Reference

We first compare the new embedded implementation to the reference implementation on a consumer-grade laptop. This benchmark is used to highlight how the decrease in memory and the algorithmic changes from the VOLE Black Box affect the performance. This platform is chosen for comparison since not all variants for the reference implementation fit on the embedded platform. The laptop has an x86-64 AMD Ryzen 7 4750U processor, with 64 KB of L1 cache and 512 KB of L2 cache per core and a shared 8 MB L3 cache. The implementations were built using the XKCP [Tea21] implementation of SHA3, and AES from the OpenSSL library, which utilizes hardware instructions for better performance. They were compiled with GCC 14.2.1 using the `-O3` optimization flag. Each variant of the two implementations was executed one at a time and repeated 100 times to minimize noise. The results are shown in Table 1. The memory consumption figures include the size of the signature.

Across all variants and security levels, the memory requirements of the embedded implementation are significantly lower than the reference implementation, ranging from one to two orders of magnitude. Additionally, the embedded implementation is faster than the reference implementation for all variants and security levels. We conjecture the speedup is due to the much lower memory footprint, allowing computations to fit within a lower level of cache, which greatly decreases the time spent on memory access. The optimization of interleaving and linearization of the access pattern discussed in Section 3.2.2 further contribute to better cache utilization. The optimized binary field multiplication also contributes to the overall performance, especially for the high-security variants, but most of the speedup is most likely due to lower memory footprints. Due to the overall better performance of the embedded implementation of the signature scheme, our new

Table 2: Memory usage (excluding signature) and average CPU cycles based on 100 executions for fast variants and 25 executions for small variants. The "Random bytes" column measures the amount of randomness needed for computing a signature.

Scheme	Stack Consumption			CPU Cycles			Random bytes
	KeyGen	Sign	Verify	KeyGen	Sign	Verify	
128f	808 B	42.7 KB	41.4 KB	82.5 K	187 M	179 M	16 B
128s	808 B	42.7 KB	41.7 KB	82.5 K	1392 M	1353 M	16 B
192f	808 B	108.3 KB	105.6 KB	385 K	538 M	516 M	24 B
192s	808 B	108.3 KB	105.6 KB	446 K	4060 M	3931 M	24 B
256f	808 B	178.0 KB	174.8 KB	614 K	800 M	762 M	32 B
256s	808 B	178.0 KB	174.5 KB	606 K	5756 M	5576 M	32 B
EM-128f	808 B	31.3 KB	31.1 KB	75.3 K	158 M	154 M	16 B
EM-128s	808 B	31.3 KB	31.2 KB	80 K	1288 M	1256 M	16 B
EM-192f	904 B	73.1 KB	75.7 KB	225 K	363 M	354 M	24 B
EM-192s	904 B	73.2 KB	75.7 KB	164 K	3372 M	3282 M	24 B
EM-256f	1024 B	143.8 KB	148.3 KB	692 K	712 M	694 M	32 B
EM-256s	1024 B	143.8 KB	148.3 KB	672 K	5400 M	5252 M	32 B

implementation effectively replaces the reference implementation of FAEST.

4.2 Performance and Memory Usage on Embedded

We now present the performance and memory usage of the embedded implementation running on an embedded device. The benchmark has been performed using the `pqm4` framework on the Nucleo-L4R5ZI board, which has 2 MB flash and 640 KB RAM and is based on the 32-bit Arm Cortex-M4 architecture. All benchmarks have been performed with full cache size for the VBB, which implies no recomputation of the VOLEitH construction.

The memory consumption and average latency in CPU cycles for the embedded implementation on the Nucleo-L4R5ZI board are shown in Table 2. Note that the memory usage excludes the signature size, as this is reported by `pqm4`; this is different from Table 1. The reference implementation has not been benchmarked on the embedded device, as its memory consumption is too large for most variants. All variants of the embedded implementation are running in constant-time with respect to the secret key, although not all operations are implemented in constant time. Incrementing the iv depends on the number of machine words that need to be changed and opening the GGM trees depends on the bits of Δ. There is no security issue regarding the timing of the computation, since both values are public.

There is a direct correlation between the security level and both the memory consumption and the running time, as increasing the security level also increases the computational cost. The least expensive variant is the EM-128f, running in 31.3 KB of memory and 158 M cycles for signing and 31.1 KB and 154 M cycles for verification. This highlights the small memory footprint in relation to the overall memory available on the benchmarked device, showing that the implementation can also be run on more resource restricted devices. The memory consumption without the signature is a meaningful parameter, since the signature can be streamed out as soon as it is computed. The FAEST signature cannot enter a fail state, invalidating part of the signature; hence, if the device fails in some way, only a subset of a valid signature can be revealed, which will not degrade security. The memory usage is almost equal between the fast and small variants, which argues for the fast variant to be used on resource-constrained devices due to the significantly better performance. If the device has access to a hardware implementation of AES, the VOLEitH computation will be significantly faster for all variants, reducing the cycle count for the 128f variant by approximately 40 million cycles.

A comparison between the embedded implementation of FAEST and other post-

Table 3: Memory usage and CPU cycles on the target platform (Cortex M4). All signature schemes are the fastest level 1 security variant presented, where the citation is the source of the benchmark.

Scheme		**Stack Consumption**			**CPU Cycles**		
		KeyGen	Sign	Verify	KeyGen	Sign	Verify
128f	[This work]	808 B	42.7 KB	41.4 KB	82.5 K	187 M	179 M
Dilithium	[KRSS19]	38.3 KB	49.4 KB	36.1 KB	1.87 M	7.28 M	2.06 M
SPHINCS-α	[KRSS19]	281 KB	281 KB	281 KB	30.2 M	382 M	35.7 M
Falcon-512	[Por25]	14.3 KB	42.0 KB	2.95 KB	71.9 M	22.0 M	255 K
Picnic3	[ABE+21]	–	32.5 KB	–	–	302 M	–
MQOM	[KRSS19]	184 KB	399 KB	253 KB	7.79 M	149 M	137 M
PERK	[BBB+24]	9.03 KB	25.1 KB	21.8 KB	0.82 M	240 M	78.1 M
UOV-Ip	[KNO+24]	4.48 KB	5.23 KB	–	390 M	7.13 M	–

quantum signature schemes can be found in Table 3. The embedded implementation of FAEST does not have a memory consumption that is significant worse than other implementations of the signature scheme for both signing and verification. When directly comparing to the NIST standardized schemes (Dilithium, SPHINCS, and Falcon), embedded FAEST is in the mix with memory consumption. Focusing on speed of the signature scheme, FAEST has by far the fastest key generation of all, and similar speed during signing and verification with other non-lattice-based schemes. The lattice based schemes do have performance improvement in 2 orders of magnitudes compared to embedded FAEST.

4.3 Memory-Speed Trade-off on Embedded

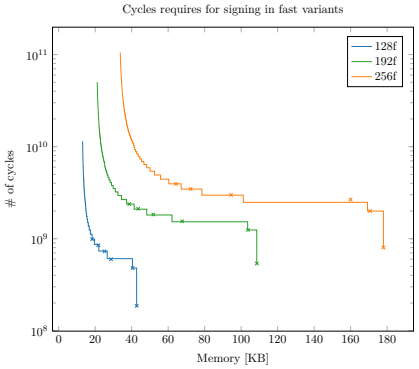
The embedded implementation uses the VBB, which enables it to decrease the memory consumption by partially recomputing the VOLE. This gives an interesting relation between the decreased memory consumption and the increased computation time.

The graph in Figure 5 shows the relationship for all available memory sizes. The line resembles the extrapolated data for all memory sizes, where the marks are experimentally observed data points. Due to the linearization of the access pattern of the VOLE, the performance resembles an exponential decay. Each jump in the cycle count is due to an increase in the needed number of recomputations of the VOLE, where each recomputation has an average cost of 127 M cycles for the 128f variant. Reducing the cache to the minimal size, a signature can be computed using only 13 KB of memory, but with an impractical cost of 11.4 billion cycles. The data shows that a significant additional decrease in memory is possible by paying the cost of a few recomputations, which gives an additional flexibility for the embedded implementation, as the memory requirement can be tweaked to fit the specific use case.

5 Conclusion and Future Perspectives

In this paper, we have presented a memory optimized algorithm for VOLEitH, which is applicable for all VOLEitH based signature schemes. We achieve a memory decrease by a factor $O(\tau n/\log n)$ from the reference algorithm, with only an increase of $O(\log n)$ computation complexity. In practice, this reduction ranges from 10 to 100 times less memory usage across all variants, and furthermore increases performance on a modern laptop CPU due to better placement in the memory hierarchy.

The reduced memory consumption allows for the implementation of the VOLEitH based signature scheme FAEST on resource-constrained devices, such as the Cortex-M4 microprocessor. This is the first embedded implementation of a VOLEitH-based signature scheme, showcasing techniques to reduce the memory consumption, and in return improving



Scheme	Memory		Cycles	
	Sign	Verify	Sign	Verify
Fastest	43 KB	41 KB	187 M	179 M
Smallest	13 KB	13 KB	11.4 B	13.8 B

Figure 5: Memory vs. speed tradeoff for the embedded implementation. The graph shows the cycle counts (in millions or billions) for the signing algorithm as a function of the memory consumption excluding the signature. Marks represent experimentally observed data points, and lines represent the extrapolated data. The table shows the optimal points minimizing memory consumption and running time. The optimal setting with respect to both memory and cycle count is the fastest variant.

the performance on modern CPUs and allowing for computation on restricted hardware. This work can be used as a starting point for future research on the VOLEitH construction both in terms of performance, masking, and portability to different architectures.

References

[ABE+21] Diego F. Aranha, Sebastian Berndt, Thomas Eisenbarth, Okan Seker, Akira Takahashi, Luca Wilke, and Greg Zaverucha. Side-channel protections for picnic signatures. *IACR TCHES*, 2021(4):239–282, 2021. URL: https://tches.iacr.org/index.php/TCHES/article/view/9066, doi:10.46586/tches.v2021.i4.239-282.

[ADE+25] Diego F. Aranha, Johan Degn, Jonathan Eilath, Kent Nielsen, and Peter Scholl. FAEST for memory-constrained devices with side-channel protections. Cryptology ePrint Archive, Paper 2025/1261, 2025. URL: https://eprint.iacr.org/archive/2025/1261/20250713:223639.

[BBB+24] Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Marco Palumbi, and Lucas Pandolfo Perin. Enabling perk and other mpc-in-the-head signatures on resource-constrained devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):84–109, Sep. 2024. URL: https://tches.iacr.org/index.php/TCHES/article/view/11784, doi:10.46586/tches.v2024.i4.84-109.

[BBD+23] Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 581–615. Springer, Cham, August 2023. doi:10.1007/978-3-031-38554-4_19.

[BBdSG+23] Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. *FAEST: Algorithm Specifications*, version 1.1 edition, July 2023. URL: https://faest.info/faest-spec-v1.1.pdf.

[BBGK24] Slim Bettaieb, Loïc Bidoux, Philippe Gaborit, and Mukul Kulkarni. Modelings for generic PoK and applications: Shorter SD and PKP based signatures. Cryptology ePrint Archive, Report 2024/1668, 2024. URL: https://eprint.iacr.org/2024/1668.

[BBM+24] Carsten Baum, Ward Beullens, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. One tree to rule them all: Optimizing GGM trees and OWFs for post-quantum signatures. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part I*, volume 15484 of *LNCS*, pages 463–493. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0875-1_15.

[BF26] Ryad Benadjila and Thibauld Feneuil. Breaking the myth of MPCitH inefficiency: Optimizing MQOM for embedded platforms. Cryptology ePrint Archive, Paper 2026/078, 2026. URL: https://eprint.iacr.org/2026/078.

[BFR23] Ryad Benadjila, Thibauld Feneuil, and Matthieu Rivain. Mq on my mind: Post-quantum signatures from the non-structured multivariate quadratic problem. Cryptology ePrint Archive, Paper 2023/1719, 2023. https://eprint.iacr.org/2023/1719. URL: https://eprint.iacr.org/2023/1719.

[Bui24] Dung Bui. Shorter VOLEitH signature from multivariate quadratic. Cryptology ePrint Archive, Report 2024/465, 2024. URL: https://eprint.iacr.org/2024/465.

[CDG+20] Melissa Chase, David Derler, Steven Goldfeder, Daniel Kales, Jonathan Katz, Vladimir Kolesnikov, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Xiao Wang, and Greg Zaverucha. The picnic signature scheme, version 3.0. 2020. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Berlin, Heidelberg, August 1987. doi:10.1007/3-540-47721-7_12.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986. doi:10.1145/6490.6503.

[KNO+24] Suparna Kundu, Quinten Norga, Uttam Kumar Ojha, Anindya Ganguly, Angshuman Karmakar, and Ingrid Verbauwhede. mUOV: Masking the unbalanced oil and vinegar digital sigital signature scheme at first- and higher-order. Cryptology ePrint Archive, Report 2024/1875, 2024. URL: https://eprint.iacr.org/2024/1875.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019. URL: https://eprint.iacr.org/2019/844.

[LD00] Julio Cesar López-Hernández and Ricardo Dahab. High-speed software multiplication in $\mathbb{F}_{2^m}$. In Bimal K. Roy and Eiji Okamoto, editors, *INDOCRYPT 2000*, volume 1977 of *LNCS*, pages 203–212. Springer, Berlin, Heidelberg, December 2000. doi:10.1007/3-540-44495-5_18.

[LDK+22] Vadim Lyubashevsky, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehle, and Shi Bai. Crystals-dilithium. Technical report, National Institute of Standards and Technology, 2022. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[OTX24] Ying Ouyang, Deng Tang, and Yanhong Xu. Code-based zero-knowledge from VOLE-in-the-head and their applications: Simpler, faster, and smaller. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part V*, volume 15488 of *LNCS*, pages 436–470. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0935-2_14.

[Por25] Thomas Pornin. Falcon on ARM cortex-m4: an update. Cryptology ePrint Archive, Paper 2025/123, 2025. URL: https://eprint.iacr.org/2025/123.

[Roy22] Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 657–687. Springer, Cham, August 2022. doi:10.1007/978-3-031-15802-5_23.

[Tea21] Team Keccak. XKCP – extended Keccak code package. https://github.com/XKCP/XKCP, 2021.

[YCZ24] Yu Yu, Hongrui Cui, and Kaiyi Zhang. Sphincs-alpha. Technical report, National Institute of Standards and Technology, 2024. https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021. doi:10.1145/3460120.3484556.