Lightening CROSS: Memory Optimized Implementations of CROSS

Harry Hart*[1], Puja Mondal*[2], Suparna Kundu[3], Supriya Adhikary[2], Angshuman Karmakar[2] and Chaoyun Li[1]

[1] University of Surrey, UK
{h.hart,c.li}@surrey.ac.uk
[2] Department of Computer Science and Engineering, IIT Kanpur, India
{pujamondal,adhikarys,angshuman}@cse.iitk.ac.in
[3] COSIC, KU Leuven, Kasteelpark Arenberg 10, Bus 2452, B-3001 Leuven-Heverlee, Belgium
suparna.kundu@esat.kuleuven.be

Abstract. Digital signature schemes derived from non-interactive zero-knowledge (NIZK) proofs are rapidly gaining prominence within post-quantum cryptography. CROSS is a promising new code-based post-quantum digital signature scheme based on the NIZK framework. It is currently in the second round of the NIST's additional call for standardization for post-quantum digital signatures. However, CROSS's reference implementation has a substantially large memory footprint. This makes its deployment on resource-constrained platforms prohibitively difficult.

In particular, we identified several mechanisms, such as Merkle tree and GGM tree structures, commitment generation process, which are part of the zero-knowledge proof generation, are one of the most memory-intensive operations. We propose several novel algorithms and implementation strategies to reduce the memory requirement of these components. Apart from these, we also propose several memory optimization techniques, such as just-in-time hashing and execution flow analysis. As a result, our implementation reduces the memory footprint of Key Generation, Signature Generation, and Verification of the CROSS reference code by as much as 95%, 92%, and 85%, respectively. This results in a suite of implementations in which all variants are under 128kB (for all security levels of KeyGen/Sign/Verify) and six variants under 32kB. Our memory optimization techniques are not specific to CROSS, but can be applied to other NIZK-based signature schemes.

Regarding efficiency, matrix multiplications are crucial to the performance of CROSS. We show how the Digital Signal Processing (DSP) instructions on ARM Cortex-M4, specifically packing and multiplying, can be utilized to efficiently implement matrix operations over finite fields. The DSP optimizations combined with the memory reductions improve the efficiency of CROSS by up to 32% and 33% in Signature Generation and Verification respectively.

Keywords: Post-quantum cryptography · Digital signatures · Code-based digital signatures · CROSS · Software implementation · Memory footprint · ARM Cortex-M4

1 Introduction

Transition to Post-quantum cryptography. In August 2022, the global push to transition from classical to post-quantum cryptography (PQC) reached a significant milestone when the National Institute of Standards and Technology (NIST) released its first set of standards (FIPS 203-5) [NIS24a, NIS24b, NIS24c] for post-quantum (PQ) key-encapsulation mechanisms and

*Starred authors contributed equally to this research.

digital signature (DS) schemes. One observation that immediately stands out is that these standard schemes are heavily dependent on hard lattice problems. Therefore, one major cryptanalysis breakthrough may jeopardize the whole migration plan from classical to PQC. Further, the current standard DS schemes, *i.e.* Dilithium [DLL+17], Falcon [FHK+20], and SPHINCS+ [BHK+19] have very large signatures. This is problematic for many real-world scenarios, e.g. chain-of-trust-based authentication mechanism in Transport Layer Security. NIST has also acknowledged these issues and initiated another standardization procedure for additional post-quantum DS schemes [NIS23]. This process aims to diversify the portfolio of post-quantum DS schemes and develop DS with small signature sizes and fast verification.

CROSS digital signature scheme. CROSS [BBB+25a] is a promising code-based scheme currently in the second round of NIST's additional DS standardization procedure [NIS24d]. It is based on the hard Restricted Syndrome Decoding Problem (`RSDP`) [BBC+21] or its variant `RSDPG` problem in a non-interactive zero-knowledge (NIZK) framework. One major limitation of this scheme is the prohibitively large memory footprint of the reference implementation. Incidentally, almost all DS schemes based on the NIZK framework suffer from such a large memory requirement as they need to store large data structures to generate the various components, such as proofs and commitments, which are operational requirements of this framework. We present algorithmic details of CROSS in Sec. 2.

CROSS on Cortex-M4. Incorporating PQC into small IoT devices, sensor nodes, and other resource-constrained devices is a crucial milestone in the transition to PQC. During the first NIST PQC standardization process, NIST designated Cortex-M4 microcontrollers as a benchmark platform to demonstrate and evaluate the performance of different schemes on small resource-constrained devices[1]. Nevertheless, due to a large memory footprint, many variants of CROSS do not even fit into Cortex-M4. This is also true for many other schemes based on the NIZK framework that are currently in the second round of the NIST's additional call for standardization. While improving the speed of a scheme in these devices is both important and desirable, reducing the memory footprint to a level where these schemes can fit into the device is indispensable. A scheme with high clock cycles will run on a device with inconvenience to the end user, but a scheme that does not even fit into a device induces a complete cessation of operation. Further, a larger memory also leads to a larger power requirement as it needs larger memory banks in the host device.

In our investigation, we find many subroutines of CROSS that contribute significantly to the memory footprint, such as the generation and traversal of trees, and large scale hashing, could be substantially optimized. Therefore, the primary motivation of this work is to design memory-efficient algorithms for these components.

1.1 Related work

The Merkle tree, a binary hash tree, was introduced in [Mer87] to aggregate many one-time signatures, which has become a standard building block in hash-based signature schemes. Many subsequent works have improved the Merkle tree root and path computation, and [BDS09] provides an extensive overview. While the existing algorithms focus on hash-based signatures, applying them to the ZK proof of CROSS is not straightforward due to the unbalanced tree structure and order-sensitive tree proof. They are not fully optimized for memory minimization and use in constrained environments. This paper addresses the gap by extending the memory-efficient Merkle tree root and path algorithms to CROSS, with algorithms explicitly designed to leverage the tree structure.

The method of just-in-time public matrix generation exploiting the sponge mechanism of Keccak and on-the-fly matrix multiplication was first proposed by Karmakar *et*

[1] https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/cJxMq0_90gU/m/qbGEs3TXGwAJ

al. [KMRV18] to reduce the memory footprint of Saber post-quantum key-encapsulation mechanism (KEM) [DKRV18]. In the same work, it was also shown how DSP instructions such as `SMLADX/SMLABT/PKHBT` can be used to speed up the school-book multiplication used in Saber. Later, just-in-time public matrix generation exploiting the sponge mechanism of Keccak and on-the-fly matrix multiplication technique was also used to develop memory-efficient implementations of Kyber [BKS19]. In addition, the acceleration in matrix-vector multiplications with DSP instructions strategy is adopted in other lattice-based schemes to design high-speed implementations, e.g., in Frodo [BFM+18], in Kyber and Dilithium [AHKS22a]. However, applying these techniques in the implementations of cryptographic algorithms does not inherently improve memory or performance. Therefore, the application of these techniques to the complex, dynamically generated, and structurally unbalanced tree-based memory requirements of code-based NIZK schemes like CROSS presents a unique and unexplored challenge. An investigation needs to be conducted to examine whether those techniques can be utilized to benefit the implementations of CROSS's three main algorithms: Key Generation, Signature Generation, and Verification.

Our work is the first to systematically address this challenge through tailored, structure-aware algorithms.

1.2 Contribution

We can divide our optimization methods into memory optimization techniques and speed optimization techniques. Note that, although we have demonstrated our memory optimization techniques on a Cortex-M4 platform, most of the improvements do not use any platform-specific features. Notably, since some building blocks are shared by many NIZK-based DS, our approach can also be adapted to optimize the memory footprint of other NIZK-based signature schemes. Also we would like to point out that we are the first to do the memory reduction so that all the parameters of CROSS can fit in the STM32 Nucleo-144 development board featuring an ARM Cortex-M4 chip. We describe our contribution below.

- **Memory-efficient Merkle Tree Algorithms.** We observe that it is not necessary to hold the full Merkle tree to perform operations on it. This allows reducing memory significantly as the tree contributes substantially to the stack usage of both the Signature Generation and Verification algorithms. We present an algorithm that computes the root node with memory complexity of $\mathcal{O}(\log_2(t))$ compared to the $\mathcal{O}(t)$-memory reference implementation, where t is the number of rounds of the ZK protocol. We can then subsequently improve the generation of `Proof` in the signature using a similarly optimized process. Using these techniques, we reduce memory by ~10%/~31%[2] in Signature Generation/Verification with a small impact on speed (~±2%).
- **Memory-efficient GGM Tree Algorithms.** In this work, we have proposed a new algorithm for on-the-fly Goldreich-Goldwasser-Micali (GGM) [GGM86] tree generation, which allows us to reduce memory usage by not fully storing the GGM tree. It is achieved by leveraging the structure of the GGM tree. This allows us to save ~8% in Signature Generation with a speed cost of ~5%.
- **Memory Optimized Key Generation.** Our analysis of the Key Generation routine indicates that two big matrices are the most significant contributors to the large memory footprint, consuming around $2118 - 15150$[3] bytes of memory to perform vector-matrix multiplication. We have shown that storing the full matrices is unnecessary. Instead, the multiplications can be evaluated on the fly. With this modification, we have

[2]We give approximations for many of the optimisation contributions to the improvement because many of them interact with each other resulting in greater or smaller gains. This makes it nontrivial to define an exact value. This is compounded by the different `RSDP/RSDPG`, 1/3/5, fast/balanced/small variants.

[3]Since the CROSS scheme has several parameter sets, we present the smallest (`RSDPG`-1-fast) and largest (`RSDP`-5-small) values of the corresponding parameter sets. We follow this notation throughout the text.

reduced the memory requirement of Key Generation by 61-95%[4].

- **Memory Optimized `Hash` Evaluation.** We note that some inputs to the `Hash` function are stored in a big array before applying the hash (e.g., the commitment $\texttt{cmt}_1$). We have exploited an incremental implementation of the `SHAKE` [KjCP16] hash function to absorb partial input as computed instead of buffering the full input. This results in a ~34%/~63% decrease in Signature Generation/Verification memory usage with ~4%/~0.2% speed impact.
- **Improving Efficiency Using DSP Instructions.** In the past, Digital Signal Processing (DSP) instructions have been used to speed up polynomial multiplication of lattice-based PQC [KMRV18, AHKS22b]. We noticed that the CROSS scheme performs lots of matrix-vector multiplications, which is the performance bottleneck of this scheme. Given that our target platform Cortex-M4 supports DSP instructions, we take advantage of parallel load, pack, and multiply operations, achieving a speed up of ~32%/~33% in Signature Generation/Verification.
- **Ultralight Memory Variation.** In pursuit of an extreme optimization corner for environments where memory is constrained to 128kB, 32kB, or even 16kB. We provide an "ultra" variant which achieves this goal. This variant implements the minimal variable usage optimization outlined in Section 4.3.2. Using this we achieve all variants under 128kB (maximum 110kB), 6 variants under 32kB (4 in security level 1, 2 in security level 3) and 2 variants under 16kB. See Tables 4 and 5 for memory and speed results respectively.

Consequently, our implementation reduces the memory consumption of the Key Generation, Signature Generation, and Verification by 61-95%, 48-92%, and 71-85% respectively, compared to the reference implementations. Furthermore, due to the DSP based speed optimizations, our implementation maintains the same performance in Signature Generation and Verification, and even gains considerable speed-ups for some variants. Therefore, our implementation outperforms the reference implementation in almost all use cases.

2 Background

In this section, we briefly introduce the CROSS digital signature. We also describe the primary factors responsible for its large memory requirements. We begin with the notation used in the text. Let $\mathbb{F}_p$ and $\mathbb{F}_p^*$ represent the finite field of prime order p and the corresponding multiplicative group respectively. A vector of length k and a matrix of size $k \times n$ with elements from $\mathbb{F}_p$ are denoted as a member of $\mathbb{F}_p^k$ and $\mathbb{F}_p^{k \times n}$, respectively. We denote $\boldsymbol{a}[i]$ and $\boldsymbol{A}[i, j]$ to represent the i-th element of the vector $\mathbf{a}$ and (i, j)-th element of the matrix $\boldsymbol{A}$ respectively. We use $\boldsymbol{a}[i,...,j]$ to denote the slice of vector $\mathbf{a}$ from i (inclusive) to j (exclusive). The symbols $\wedge$ and $\vee$ represent bitwise AND and OR, respectively. We use $a \,|\, b$ to denote the concatenation of two entities a,b (potentially vectors or bitstrings). Let $\mathscr{J} \subset \{0,...,n\}$ be an ordered set, then the notation $\mathbf{a}_{(\mathscr{J})}$ and $\boldsymbol{A}_{(*,\ \mathscr{J})}$ (corresponding $\boldsymbol{A}_{(\mathscr{J},\ *)}$) represents the subvector of $\mathbf{a}$ and submatrix of $\boldsymbol{A}$ formed by selecting the entries of the vector $\mathbf{a}$ and columns (rows) of $\boldsymbol{A}$ with indices specified in the set $\mathscr{J}$.

We use two cryptographic primitives: (i) Hash: a cryptographic hash function with codomain $\{0,1\}^{2\lambda}$ and (ii) $\mathsf{CSPRNG}(\cdot, S)$: a cryptographically secure pseudo-random number generator with output in the set S e.g. $\mathsf{CSPRNG}(s, \mathbb{F}_p)$ uses the seed s to generate a sequence of elements in $\mathbb{F}_p$. Both of these are implemented with `SHAKE`-128[NIS15] for NIST security category 1, and `SHAKE`-256 for categories 3 and 5. The Hash output length 2λ is set to 256, 384, and 512 bits for categories 1, 3, and 5, respectively. In the CROSS algorithms, there are variations in some lines depending on the `RSDP` or `RSDPG` variant. We include both versions

[4]We give an exact value here because it is the only optimization done to Key Generation, so we can attribute all the improvements to this technique.

in the same algorithm, but where they differ, the left column in orange is the RSDPG version while the right column in teal is the RSDP version.

2.1 Overview of the CROSS Scheme

Let g be an element of the multiplicative group $\mathbb{F}_p^*$ of prime order z. Given the parity check matrix $\boldsymbol{H} \in \mathbb{F}_p^{(n-k)\times n}$ and $\mathbf{s} \in \mathbb{F}_p^{n-k}$, the RSDP problem is to decide if there exists a vector $\mathbf{e} \in \mathbb{E}^n = \{g^i \,|\, i \in \{1,\ldots,z\}\}^n$ such that it satisfies $\mathbf{s} = \mathbf{e}\boldsymbol{H}^\top$. CROSS has another compact variant, called RSDPG [BBP+24]. Here, the problem is to decide if there exists a vector $\mathbf{e} \in G$ such that the equality $\mathbf{s} = \mathbf{e}\boldsymbol{H}^\top$ holds where G is a subgroup of $\mathbb{E}^n$ with cardinality $z^m < z^n$. CROSS provides three primary parameters that satisfy three NIST security levels of 1, 3, and 5 for each of the following two variants, RSDP and RSDPG. Furthermore, each of these primary parameters has three optimization corner variants, namely (i) `fast` version aims to attain speed, (ii) `small` version targets to reduce the signature size, and (iii) `balanced` version strikes a trade-off between them.

In the following sections, we discuss the Key Generation and Signature Generation algorithms and explain some portions of these algorithms that are required for this work. The Verification algorithm is quite similar to the Signature Generation algorithm. Therefore, we do not describe the Verification algorithm due to page restrictions. We present parameter sets for all the variants of CROSS in Table 1. We also provide a complete breakdown of the memory usage by variable for both the reference implementation and our optimized implementation in Fig. 3. For further details, we refer to the CROSS documentation [BBB+25a].

2.1.1 Key Generation Algorithm (Alg. 1)

Algorithm 1 CROSS_KeyGen() [BBB+25a]

Input: Security parameter λ
Output: Public key: $(\mathsf{Seed}_{\mathsf{pk}}, \mathbf{s})$, Private key: $\mathsf{Seed}_{\mathsf{sk}}$

1: // Sampling seeds
2: $\mathsf{Seed}_{\mathsf{sk}} \xleftarrow{\$} \{0,1\}^\lambda$
3: $(\mathsf{Seed}_{\mathsf{e}}, \mathsf{Seed}_{\mathsf{pk}}) \leftarrow \mathsf{CSPRNG}\left(\mathsf{Seed}_{\mathsf{sk}} \,|\, 3t+1, \{0,1\}^{2\lambda} \times \{0,1\}^{2\lambda}\right)$
4: // Sampling random matrices $\boldsymbol{H}$ and $\overline{\boldsymbol{M}}$
5: $(\overline{\boldsymbol{W}}, \boldsymbol{V}) \leftarrow \mathsf{CSPRNG}\left(\mathsf{Seed}_{\mathsf{pk}} \,|\, 3t+2, \mathbb{F}_z^{m\times(n-m)} \times \mathbb{F}_p^{(n-k)\times k}\right)$ | $\boldsymbol{V} \leftarrow \mathsf{CSPRNG}\left(\mathsf{Seed}_{\mathsf{pk}} \,|\, 3t+2, \mathbb{F}_p^{(n-k)\times k}\right)$
6: $\boldsymbol{H} \leftarrow [\boldsymbol{V} \,|\, \mathrm{Id}_{n-k}]$
7: // Computing $\mathbf{e}$
$\overline{\boldsymbol{M}} \leftarrow [\overline{\boldsymbol{W}} \,|\, \mathrm{Id}_m]$
8: $\overline{\mathbf{e}}_G \leftarrow \mathsf{CSPRNG}(\mathsf{Seed}_{\mathsf{e}} \,|\, 3t+3, \mathbb{F}_z^m)$
$\overline{\mathbf{e}} \leftarrow \overline{\mathbf{e}}_G \overline{\boldsymbol{M}}$ | $\overline{\mathbf{e}} \leftarrow \mathsf{CSPRNG}(\mathsf{Seed}_{\mathsf{e}} \,|\, 3t+3, \mathbb{F}_z^n)$
9: **for** $j = 1,\ldots,n$ **do**
10: $\quad \mathbf{e}_j \leftarrow g^{\overline{\mathbf{e}}_j}$
11: // Computing the syndrome s
12: $\mathbf{s} \leftarrow \mathbf{e}\boldsymbol{H}^\top$
13: // Return the keys
14: **return** Public key: $(\mathsf{Seed}_{\mathsf{pk}}, \mathbf{s})$, Private key: $\mathsf{Seed}_{\mathsf{sk}}$

Starting with our secret $\mathsf{Seed}_{\mathsf{sk}}$, we first generate two seeds $\mathsf{Seed}_{\mathsf{pk}}$ and $\mathsf{Seed}_{\mathsf{e}}$ using a CSPRNG. Then, using these seeds and CSPRNG, we generate the parity check matrix $\boldsymbol{H} = [\boldsymbol{V} | \mathrm{Id}_{n-k}]$ from $\mathsf{Seed}_{\mathsf{pk}}$, and the restricted vector $\mathbf{e}$ from $\mathsf{Seed}_{\mathsf{e}}$. For the RSDPG variant, instead of directly generating $\mathbf{e}$ from $\mathsf{Seed}_{\mathsf{e}}$, an additional matrix $\overline{\boldsymbol{W}}$ is generated from $\mathsf{Seed}_{\mathsf{pk}}$. Using $\mathsf{Seed}_{\mathsf{e}}$, we can produce the intermediate vector $\mathbf{e}_G$, which is then used to compute $\mathbf{e}$. The $\mathsf{Seed}_{\mathsf{pk}}$ and $\mathbf{s} = \mathbf{e}\boldsymbol{H}^\top$ are set as public key, and $\mathsf{Seed}_{\mathsf{sk}}$ as secret key.

Algorithm 2 CROSS_Sign($\mathtt{Seed}_{\mathtt{sk}}$,Msg) [BBB+25a]

Input: Message: Msg and secret key: $\mathtt{Seed}_{\mathtt{sk}}$.
Output: Signature: signature

1: $c \leftarrow 2t-1$
2: // Expand the secret key
3: $\left(\bar{\mathbf{e}}, \bar{\mathbf{e}}_G, \boldsymbol{H}, \bar{\boldsymbol{M}}\right) \leftarrow \mathsf{ExpandSK}(\mathtt{Seed}_{\mathtt{pk}})$ | $(\bar{\mathbf{e}}, \boldsymbol{H}) \leftarrow \mathsf{ExpandSK}(\mathtt{Seed}_{\mathtt{sk}})$
4: // Compute the commitments
5: $\mathtt{Seed} \xleftarrow{\$} \{0,1\}^{\lambda}$, $\mathtt{Salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$
6: $(\mathtt{Seed}[1], \cdots, \mathtt{Seed}[t]) \leftarrow \mathsf{SeedLeaves}(\mathtt{Seed}|\mathtt{Salt})$
7: // Compute $\mathbf{v}[i]$ such that $\mathbf{v}[i] \star \mathbf{e}'[i] = \mathbf{e}$
8: **for** $i = 1,\ldots,t$ **do**
$\bar{\mathbf{e}}'_G[i], \mathbf{u}'_G[i] \leftarrow \mathsf{CSPRNG}\left(\mathtt{Seed}[i] \,|\, \mathtt{Salt} \,|\, i+c, \mathbb{F}_z^m \times \mathbb{F}_p^n\right)$
9: $\bar{\mathbf{v}}_G[i] \leftarrow \bar{\mathbf{e}}_G - \bar{\mathbf{e}}'_G[i]$
$\bar{\mathbf{e}}'[i] \leftarrow \bar{\mathbf{e}}'_G[i]\bar{\mathbf{M}}$ | $\bar{\mathbf{e}}'[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG}\left(\mathtt{Seed}[i] \,|\, \mathtt{Salt} \,|\, i+c, \mathbb{F}_z^n \times \mathbb{F}_p^n\right)$
10: $\bar{\mathbf{v}}[i] \leftarrow \bar{\mathbf{e}} - \bar{\mathbf{e}}'[i]$
11: **for** $j = 1,\ldots,n$ **do**
12: $\mathbf{v}[i]_j \leftarrow g^{\bar{\mathbf{v}}[i]_j}$
13: $\mathbf{u}[i] \leftarrow \mathbf{v}[i] \star \mathbf{u}'[i]$
14: $\mathbf{s}' \leftarrow \mathbf{u}[i]\boldsymbol{H}^{\top}$
15: $\mathtt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \,|\, \bar{\mathbf{v}}_G[i] \,|\, \mathtt{Salt} \,|\, i+c)$ | $\mathtt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \,|\, \bar{\mathbf{v}}[i] \,|\, \mathtt{Salt} \,|\, i+c)$
16: $\mathtt{cmt}_1[i] \leftarrow \mathsf{Hash}(\mathtt{Seed}[i] \,|\, \mathtt{Salt} \,|\, i+c)$
17: $\mathtt{digest}_{\mathtt{cmt}_0} \leftarrow \mathsf{TreeRoot}(\mathtt{cmt}_0[1],\ldots,\mathtt{cmt}_0[t])$
18: $\mathtt{digest}_{\mathtt{cmt}_1} \leftarrow \mathsf{Hash}(\mathtt{cmt}_1[1] \,|\, \cdots \,|\, \mathtt{cmt}_1[t])$
19: $\mathtt{digest}_{\mathtt{cmt}} \leftarrow \mathsf{Hash}(\mathtt{digest}_{cmt_0} \,|\, \mathtt{digest}_{\mathtt{cmt}_1})$
20: // Compute first challenge
21: $\mathtt{digest}_{\mathtt{Msg}} \leftarrow \mathsf{Hash}(\mathtt{Msg})$
22: $\mathtt{digest}_{\mathtt{chall}_1} \leftarrow \mathsf{Hash}(\mathtt{digest}_{\mathtt{Msg}} \,|\, \mathtt{digest}_{\mathtt{cmt}} \,|\, \mathtt{Salt})$
23: $\mathtt{chall}_1 \leftarrow \mathsf{CSPRNG}\left(\mathtt{digest}_{\mathtt{chall}_1} \,|\, t+c, (\mathbb{F}_p^*)^t\right)$
24: // Compute first response
25: **for** $i = 1,\ldots,t$ **do**
26: **for** $j = 1,\ldots,n$ **do**
27: $\mathbf{e}'[i]_j \leftarrow g^{\bar{\mathbf{e}}'[i]_j}$
28: $\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \mathtt{chall}_1[i]\mathbf{e}'[i]$
29: // Compute second challenge
30: $\mathtt{digest}_{\mathtt{chall}_2} \leftarrow \mathsf{Hash}(\mathbf{y}[1] \,|\, \cdots \,|\, \mathbf{y}[t] \,|\, \mathtt{digest}_{\mathtt{chall}_1})$
31: $\mathtt{chall}_2 \leftarrow \mathsf{CSPRNG}\left(\mathtt{digest}_{\mathtt{chall}_2} \,|\, t+c+1, \mathcal{B}_{(t,w)}\right)$
32: // Compute second response
33: $\mathtt{Proof} \leftarrow \mathsf{TreeProof}(\mathtt{cmt}_0[1],\ldots,\mathtt{cmt}_0[t],\mathtt{chall}_2)$
34: $\mathtt{Path} \leftarrow \mathsf{SeedPath}(\mathtt{Seed},\mathtt{Salt},\mathtt{chall}_2)$
35: **for** $i = 1,\ldots,t$ **do**
36: **if** $\mathtt{chall}_2[i] = 0$ **then**
37: $\mathtt{resp}[i]_0 \leftarrow (\mathbf{y}[i], \bar{\mathbf{v}}_G[i])$ | $\mathtt{resp}[i]_0 \leftarrow (\mathbf{y}[i], \bar{\mathbf{v}}[i])$
38: $\mathtt{resp}[i]_1 \leftarrow \mathtt{cmt}_1[i]$
39: // Combine signature
40: $\mathtt{signature} \leftarrow (\mathtt{Salt}, \mathtt{digest}_{\mathtt{cmt}}, \mathtt{digest}_{\mathtt{chall}_2}, \mathtt{Path}, \mathtt{Proof}, \mathtt{resp})$
41: **return** signature

2.1.2 Signature Generation Algorithm (Alg. 2)

The signature generation algorithm CROSS_Sign first constructs t many ephemeral seeds (Seed[1], ..., Seed[t]) from the root seed Seed and a random Salt using the function SeedLeaves (line 6). The seeds are generated in different ways depending on the variant and parameter sets. The fast variant does not use the GGM tree or Merkle tree construction. In this case, all Seeds are generated using $\mathtt{Seed} \in \{0,1\}^{\lambda}$, $\mathtt{Salt} \in \{0,1\}^{\lambda}$ as input seeds of the function CSPRNG. The remaining sets of parameters of the CROSS scheme use a GGM tree called SeedTree with t leaves i random ephemeral seeds. These seeds and Salt (with domain separation) are utilized to compute $\mathbf{u}'$ and $\bar{\mathbf{e}}'$. Using these vectors, we compute new syndromes and commit them with two sets of commitments $\mathtt{cmt}_0[1]$, ..., $\mathtt{cmt}_0[t]$ and $\mathtt{cmt}_1[1]$, ..., $\mathtt{cmt}_1[t]$. These commitments are compressed to digests, with $\mathtt{digest}_{\mathtt{cmt}_0}$ using the Merkle tree.

Next, the first challenge $\texttt{chall}_1$ is generated using the message and commitments. After that, $\bar{\mathbf{e}}'$, $\mathbf{u}'$ and $\texttt{chall}_1$ are used to generate $\mathbf{y}$, which is then utilized to sample the second challenge by the CSPRNG. CROSS_Sign responds to this challenge by revealing $\texttt{Seed}[i]$, $i \in \{1,\ldots,t\}$ if $\texttt{chall}_2[i] = 1$ using SeedPath; and releasing $(\mathbf{y}[i], \bar{\mathbf{v}}[i]/\bar{\mathbf{v}}_G[i])$ if $\texttt{chall}_2[i] = 0$. Additionally, TreeProof produces Proof to finalize the zero-knowledge proof of $\texttt{digest}_{\texttt{cmt}_0}$.

Table 1: Parameter set of all variants of CROSS [BBB+25a]. **opt.** refers to optimization.

Algorithm and security category	Dependent on opt. corner					Independent of opt. corner	
	p	z	n	k	m	t (fast/balanced/small)	w (fast/balanced/small)
RSDP 1	127	7	127	76	-	157/256/520	82/215/488
RSDP 3			187	111		239/384/580	125/321/527
RSDP 5			251	150		321/512/832	167/427/762
RSDPG 1	509	127	55	36	25	147/256/512	76/220/484
RSDPG 3			79	48	40	224/268/512	119/196/463
RSDPG 5			106	69	48	300/356/642	153/258/575

2.2 Tree Algorithms for `balanced` and `small` Variants

Merkle and GGM (seed) trees in CROSS share the same structure which are full binary tree. The tree is not balanced and is constructed as the concatenation, from right to left, of full binary sub-trees. The principal parameter is the number of leaves t. The t-leaf binary tree has $2t-1$ nodes in total [BBB+25a]. For example, the tree in Fig. 1 has $t=11$, which has three subtrees rooted at $\mathcal{T}[1]$,$\mathcal{T}[5]$,$\mathcal{T}[6]$ and are then combined from the smallest (right) subtree to the largest (left). The actual t values used in CROSS instances are shown in Table 1.

A binary tree is stored in memory as an array, flattened from root to leaves, with each level comprising contiguous array elements. The example tree is stored like $\mathcal{T} = \{\mathcal{T}[0],\mathcal{T}[1],\mathcal{T}[2],\ldots,\mathcal{T}[20]\}$. For ease of accessing different elements of the array as required for traversing the tree, the specification defines precomputed constant arrays [BBB+25a, Sec.5.2.1]. We list some of those arrays that have been used in this paper.

- `npl`: number of nodes per level;
- `lpl`: number of leaves per level;
- `off` : offsets required to move between two levels; defined as $\texttt{off}[0] := 0, \texttt{off}[i] := 2\cdot\texttt{npl}[i-1] - \texttt{npl}[i] + \texttt{off}[i-1]$ for $i \geq 1$;

For instance, the tree in Fig. 1 has levels from 0 (root) to 4 (leaves) and the parameters $\texttt{npl} = [1,2,4,6,8], \texttt{lpl} = [0,0,1,2,8], \texttt{off} = [0,0,0,2,6]$.

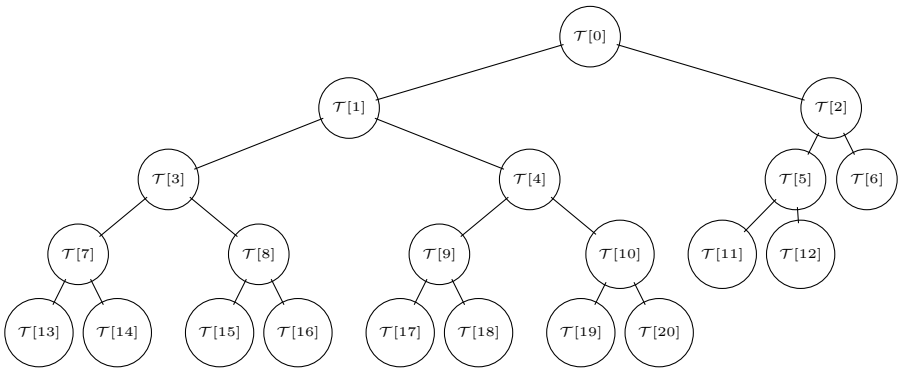

Figure 1: Example tree strucutre for $t=11$ (t is the number of leaves).

An Overview of Tree Algorithms. The `balanced` and `small` CROSS variants employ the Merkle tree to compress the commitment $\texttt{cmt}_0$ using TreeRoot, which is passed to TreeProof to produce a Proof of $\texttt{cmt}_0$. A GGM tree is used to derive round seeds by SeedLeaves. Using

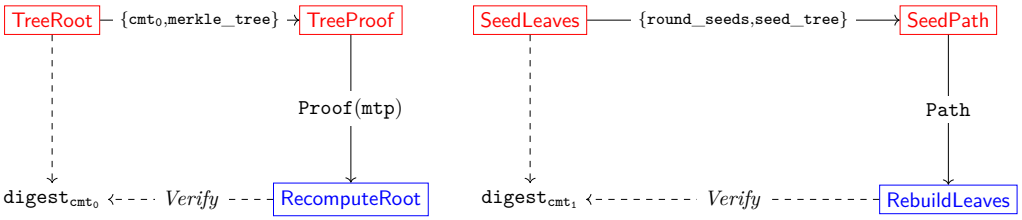

Figure 2: A flowchart of the tree algorithms. The functions in red are used in CROSS_Sign while those in blue are part of verification.

round seeds, CROSS_Sign computes the commitments $\texttt{cmt}_1$. On the other hand, SeedPath compresses the seeds to a `Path` utilizing $\texttt{chall}_2$. The verifier can then use the `Proof` and `Path` to recover certain leaves of the GGM tree by RebuildLeaves and reconstruct the root of the Merkle tree by RecomputeRoot. Thus, the verifier has enough information to verify the signature. The general architecture of the tree algorithms is illustrated in Fig. 2, where the solid lines indicate data being passed between algorithms.

3 Optimizing Tree Algorithms for Zero-Knowledge Proofs

This section focuses on the Merkle and GGM tree algorithms for the ZK protocol of CROSS. While the trees are implemented with efficiency in mind, their enormous memory consumption turns out to be a bottleneck of the overall memory complexity of CROSS. We present memory optimized on-the-fly implementations of four tree algorithms `TreeRoot`, `TreeProof`, `RecomputeRoot`, and `SeedPath` for `balanced` and `small` variants while they do not apply to the `fast` variant which uses a different tree construction.

3.1 On-the-fly Merkle Tree Algorithms

In this section we optimize the Merkle tree algorithms which are used for efficient computing of commitments in the ZK proofs. We will first recall how Merkle trees are generated in CROSS reference codes and then propose optimized implementations.

3.1.1 Optimizing TreeRoot

TreeRoot. This function computes the root node of the Merkle tree from leaf nodes. To be specific, the Merkle tree is initialized with the components of $\texttt{cmt}_0$ on its leaves which are hashed to compute the root of the tree. The value of a parent node is then calculated using a hash function on the child nodes. Computing the parent nodes from bottom to top, consequently, the root node is generated as the hash digest of $\texttt{cmt}_0$. For instance, in Fig. 1, we have

$$\mathcal{T}[7] = \mathsf{Hash}(\mathcal{T}[13] \,|\, \mathcal{T}[14]), \mathcal{T}[8] = \mathsf{Hash}(\mathcal{T}[15] \,|\, \mathcal{T}[16]), \cdots, \mathcal{T}[0] = \mathsf{Hash}(\mathcal{T}[1] \,|\, \mathcal{T}[2]).$$

Memory Footprint of the Reference Implementation. TreeRoot, as implemented on the left side of Alg. 3, computes the root $\mathcal{T}[0]$ in a straightforward way. Starting at the bottom level, right-most pair,[5] we iterate through pairs of `leftChild` and `rightChild`, hashing together and placing the result in their parent. Consequently, Alg. 3 returns the root node $\mathcal{T}[0]$, i.e., $\texttt{digest}_{\texttt{cmt}_0}$.

[5]Since we always calculate the parent as $H(l \mid r)$, the order of pairs visited doesn't really matter mathematically. We specify this here because in the CROSS specification it is explicit.

The memory footprint of the reference implementation is dominated by holding $\mathcal{T}$. The tree $\mathcal{T}$ occupies $(2t-1)\times 2\lambda$ (80,128 - 851,456 bits) memory. That is, the total memory complexity of TreeRoot is $(4t-2)\lambda$.

Algorithm 3 TreeRoot($\texttt{cmt}_0$) – `balanced` and `small`

Input: The commitment $\texttt{cmt}_0 \in \{0,1\}^{2\lambda \cdot t}$
Output: A 2λ-bit root of the Merkle tree

```
1:  // Reference version
2:  Create and store the tree T of size (2t−1)×2λ by placing the components of cmt_0 on the leaves.
3:  startNode ← lsi[0]
4:  // For each level
5:  for level = ⌈log_2(t)⌉,...,1 do
6:      // For each node on the level
7:      for i = npl[level]−2,...,0 step −2 do
8:          leftChild ← startNode + i
9:          rightChild ← leftChild + 1
10:         parent ← ⌊(leftChild − 1)/2⌋ + off[level − 1]/2
11:         T[parent] ← Hash(T[leftChild] | T[rightChild])
12:     startNode ← startNode − npl[level − 1]
13: return T[0]
```

```
1:  // Optimized version
2:  Create the buffer buf to hold each level's hash of size ⌈log_2(t)⌉ × 2λ
3:  leaf_level ← ⌈log_2(t)⌉ − 1, seen ← 0
4:  flag ← ∅
5:  // For each leaf
6:  for i = 1,...,t do
7:      // Find leaf level
8:      while seen = lpl[leaf_level] do
9:          leaf_level ← leaf_level − 1
10:         seen ← 0
11:     hash ← cmt_0[i]
12:     level ← leaf_level
13:     while flag[level] = 1 do
14:         // While the spot for this level in the buffer already has a hash stored
15:         hash ← Hash(buf[level] | hash)
16:         flag[level] ← 0
17:         if level = 0 then
18:             return hash
19:         level ← level − 1
20:     buf[level] ← hash
21:     flag[level] ← 1
22:     seen ← seen + 1
```

Memory Optimized TreeRoot. We notice that to compute the tree root it is not necessary to hold the full tree $\mathcal{T}$. If we visit the leaves in order (either right-to-left or left-to-right) any time we are holding two nodes n_1, n_2 from the same level, we can calculate their parents by $\mathsf{Hash}(n_1 \,|\, n_2)$, and discard the children. Then the intermediate nodes can be stored in a static array of size $T = \lceil \log_2(t) \rceil$ which is also the number of levels (i.e. depth) of the Merkle tree.

We propose a memory-efficient implementation of TreeRoot as on the right side of Alg. 3. This optimized version aims to compute the Merkle tree root $\mathcal{T}[0]$ from the commitments $\texttt{cmt}_0[1], \cdots, \texttt{cmt}_0[t]$ using buffer array of length T. Without loss of generality, let the $\mathcal{T}[j_T]\mathcal{T}[j_{T-1}], \cdots, \mathcal{T}[j_0]$ be the path to the root $\mathcal{T}[0]$ from the leftmost leaf node $\texttt{cmt}_0[1]$, that is, $\mathcal{T}[j_T] = \texttt{cmt}_0[1]$ and $\mathcal{T}[j_0] = \mathcal{T}[0]$. Here each node $\mathcal{T}[j_i]$ is the parent node of $\mathcal{T}[j_{i+1}]$. In our method, we first compute the node $\mathcal{T}[j_i]$ and then generate the parent node $\mathcal{T}[j_{i-1}]$ iteratively, where $1 \le i \le T$. To compute the parent node $\mathcal{T}[j_{i-1}]$, it is enough to compute the sibling of $\mathcal{T}[j_i]$.

Initially $\mathcal{T}[j_T] = \texttt{cmt}_0[1]$ is being stored in $\texttt{buf}[\texttt{leaf_level}]$ (in line 11 and line 20 of optimized version of Alg. 3, as initially $\texttt{flag}[\texttt{T}] = 0$). Then count the number of visited leaf nodes as $\texttt{seen} = 1$ and update the flag at this level as $\texttt{flag}[\texttt{T}] = 1$, which indicates that there is already one node in the buffer at the level T. Next, it uses an extra variable `hash` to store the sibling of $\mathcal{T}[j_T]$ which is $\texttt{cmt}_0[2]$. After computing the sibling of $\mathcal{T}[j_T]$ (which is stored in `hash` variable) we can compute the parent $\mathcal{T}[j_{T-1}] = \mathsf{Hash}(\texttt{buf}[T] \mid \texttt{hash})$ of $\mathcal{T}[j_T]$ (line 15 of Alg. 3) and stored in $\texttt{buf}[T-1]$ at level $T-1$. This increases the counter as $\texttt{seen} = 2$ and mark the flag as 1 at the level $T-1$. Similarly, if we have the node $\mathcal{T}[j_{T-1}]$, then we can compute the sibling of $\mathcal{T}[j_{T-1}]$ by the stored nodes $\texttt{cmt}_0[3]$ (stored in $\texttt{buf}[T]$) and $\texttt{cmt}_0[4]$ (stored in the variable `hash`). We will continue this process of finding the sibling of a node $\mathcal{T}[j_i]$ until

the count of visited leaf nodes i.e., `seen` is less than `lpl[T]` (this case will ignore the while loop from line 8 to 10). If all the leaf nodes of the level T have been visited, then it will go to the unseen leaf nodes at the previous level (will follow the instruction of lines 9-10). With similar way we can compute the parent of any node $\mathcal{T}[j_i]$. Finally, we can determine $\mathcal{T}[j_0] = \mathcal{T}[0]$.

The memory footprint of the optimized implementation is dominated by holding `buf` and `flag`. Thus, the total memory complexity is $2\log_2(t)\lambda + \log_2(t)$, which demonstrates an exponential improvement in t compared to the reference implementation.

Comparison with previous work. While memory-efficient Merkle tree root algorithms have been proposed in previous work such as in [BDS09], our goal is to minimize memory in our extremely constrained environment. The reference implementation holds the entire Merkle tree, which leaves open the question of adapting a memory-efficient Merkle tree root algorithm to TreeRoot. In [BDS09], two high-level data structures, *Node* and *Stack*, are used. Our optimized implementation improves on this by introducing more compact data structures and leveraging existing helper arrays. We devise a static array that encodes the depth of a node structurally instead of storing it, which benefits memory and speed efficiency. To the best of our knowledge, we are the first to apply it to CROSS.

3.1.2 Optimizing TreeProof

TreeProof. The Merkle tree construction is used by the signer to prove the knowledge of all commitments of $\texttt{cmt}_0$. The signer employs TreeProof to reveal a subset of nodes of the Merkle tree which consists of the `Proof` in the signature. To enable verification of the ZK proof, the signature contains some responses as per zero bits of $\texttt{chall}_2$, which can be used to generate some Merkle tree leaves and then recompute the Merkle tree root.

Memory Footprint of the Reference Implementation. The TreeProof reference implementation in Alg. 4 is designed to walk the Merkle tree in the same order as TreeRoot: from right to left, bottom to top. It computes the minimum set of nodes required in conjunction with the revealed leaves to recover the root. Hence the goal is to find the highest (closest to root) nodes such that all it's children cannot be *computed* from the given leaves. Here a node being *computed* means that all the descendant leaves of that node have had their seeds revealed to the verifier in the signature according to $\texttt{chall}_2$. In the flag tree $\mathcal{T}'$ the flag value 1 indicates that a node is **computed** and 0 indicates that it is **not computed**.

In terms of memory, the reference implementation uses two tree structures, the full Merkle tree, and a flag tree. Thus, the total memory footprint is $(2t-1)\times(2\lambda+8)$ (82,472 - 864,760 bits).

Memory Optimized TreeProof. In the reference implementation the flag tree $\mathcal{T}'$ of $(2t-1)$ nodes and the stored Merkle tree $\mathcal{T}$ with $(2t-1)$ nodes are used to visit the tree nodes as per the values of $\mathcal{T}'$, then publish or skip nodes in $\mathcal{T}$. Whereas, our optimization technique reduces memory consumption by employing smaller data strucures: a flag array `flags` of size t and the commitments $\texttt{cmt}_0$ of length t. Our method is described below.

The flag array `flags` use the same flag values as in flag tree $\mathcal{T}'$ to represent if a node is computed. Given a pair, a parent is not computable when both of its children are set to 0. So, if either one of the flag value of the pair is 1 (indicating it is computable), the flag for their parent node in the previous level must be 1, otherwise it will be 0 (line 20).

At each level, while updating the flag values we publish the eligible node to the `Proof` (lines 14 and 17) or update the storage array $\texttt{cmt}_0$ (line 24). Recall that, we first consider a pair of consecutive flag values which can take either of the four values (0, 0), (0, 1), (1, 0), (1, 1).

- (0,1) or (1,0): append `Proof` with the values in $\texttt{cmt}_0$ corresponding to the flag location with value 0; set the flag value of parent node to 1.

Algorithm 4 TreeProof($\texttt{cmt}_0$,$\texttt{chall}_2$)– `balanced` and `small`

Input: The challenge $\texttt{chall}_2$ indicating leaves are published. In the reference version, the tree $\mathcal{T}$ is required, in the optimized version only $\texttt{cmt}_0$ instead.
Output: `Proof`, an array of tree node values.

```
// Reference Version
Create flag tree T' to pre-compute which nodes to reveal
Proof ← ∅
startNode ← lsi[0], published ← 0
for level = ⌈log2(t)⌉...1 do
    for i = npl[level] − 2...0 step −2 do
        node ← startNode + i
        parent ← ⌊(node − 1)/2⌋ + off[level − 1]/2
        // add left sibling only if right one was computed but left was not
        if T'[node] = 0 and T'[node + 1] = 1 then
            Proof[pubNodes] ← T[node]
            pubNodes ← pubNodes + 1
        // add right sibling only if left one was computed but right was not
        if T'[node] = 1 and T'[node + 1] = 0 then
            Proof[pubNodes] ← T[node + 1]
            pubNodes ← pubNodes + 1
    startNode ← startNode − npl[level − 1]
return Proof
```

```
// Optimized Version
Proof ← ∅, flags ← chall2
leaves_left ← t, published ← 0
for level = ⌈log2(t)⌉..1 do
    // Offset resolves overlap of flags
    poffset ← 0
    leaves_left ← leaves_left − lpl[level]
    end_index ← t − leaves_left − 1
    start_index ← end_index − npl[level]
    // Iterate over all nodes at a level
    for i = end_index..start_index step −2 do
        // If siblings don't match, add the (so-far) uncomputed one
        if flags[i] = 1 and flags[i − 1] = 0 then
            Proof[published] ← cmt0[i − 1]
            published ← published + 1
        if flags[i] = 0 and flags[i − 1] = 1 then
            Proof[published] ← cmt0[i]
            published ← published + 1
        // Set the parent flag if it is computed
        if flags[i] = 0 or flags[i − 1] = 0 then
            flags[i + poffset] ← 0
        else
            flags[i + poffset] ← 1
            cmt0[i + poffset] ← Hash(cmt0[i − 1] | cmt0[i])
        poffset ← poffset + 1
return Proof
```

- (0,0): set the flag value of parent node to 0.
- (1,1): set the flag of the parent to 1 as it is computed; then compute and store the hash of the parent node in $\texttt{cmt}_0$. Note that the flag and hash value of the parent node should be updated at the same location of `flags` and $\texttt{cmt}_0$ to maintain the consistency.

Our optimized implementation does not use the full Merkle and flag trees to save memory. This means it must calculate the hashes as it traverses by using a hash buffer with a spot for each level and calculating hashes as needed. Since we store the hashes in the $\texttt{cmt}_0$ array, overwriting the values already in there, we avoid storing the tree with no extra cost to memory. The above optimized `TreeProof` together with optimized `TreeRoot` results in a memory usage decrease of 7.6-10.7% in the Signature Generation algorithm.

Since we recalculate the hashes, the natural question is how this affects speed. But our key advantage is that we need not re-calculate the whole tree, but rather a subset. We only need to hash the paths to the required nodes to publish in the proof. In our experimental testing this memory optimization only results in approximately a 5.5% performance decrease of CROSS_Sign.

Comparison with previous work. CROSS Verify is required to prove the Merkle root from *multiple* leaf nodes revelations as opposed to hash-based signature schemes which mainly focus on the proof of a *single* leaf node. The more relevant so-called *Merkle multiproof* is only explicitly covered in the context of sparse data structure [RA20]. But the algorithm in [RA20] cannot be directly used in TreeProof as the proof must be of a certain order, namely from bottom to top, right to left, as required by CROSS_Sign. The reference implementation

holds the Merkle tree and a flag tree, which is not optimal in terms of memory consumption. In contrast, our optimized TreeProof avoids storing the Merkle tree by reusing and overwriting the input $\texttt{cmt}_0$. Another advantage of reusing $\texttt{cmt}_0$ is that it is straightforward to align with the flag array to maintain the proof order.

3.1.3 Optimizing RecomputeRoot

RecomputeRoot. The verification algorithm recomputes the root node of the Merkle tree from nodes in `Proof` and the revealed leaf nodes corresponding to zero bits of $\texttt{chall}_2$. Due to the similarity between sign and verification algorithms, RecomputeRoot essentially combines the operations of TreeRoot and TreeProof in CROSS_Sign. Thus, this algorithm is optimized by combining ideas from both previous optimizations. To avoid repetition, we will only briefly sketch the reference implementation and our optimized one. For further details on verification and RecomputeRoot, refer to [BBB+25a]

Reference Implementation. The reference implementation first calculates values of $\texttt{cmt}_0$ from the revealed leaves. Then after filling the available values of $\texttt{cmt}_0$, it walks on the Merkle tree in the same way as TreeProof since there is no ordering information given in the signature for the `Proof`. Finally, the whole Merkle tree is reconstructed and the root is computed similar to the reference TreeRoot. The implementation is costly in memory due to the storage of $\texttt{cmt}_0$, the Merkle tree, and a flag tree.

Memory Optimized RecomputeRoot. Our optimizations are implemented sequentially, first to retrieve the proof order, then to calculate the root. First, we use the strategy of optimized TreeProof to disambiguate the proof node order. We then integrate the same algorithmic idea as TreeRoot. At each leaf, we move further up the Merkle tree. We fetch the required hash values from the proof, using the ordering information stored in the previously calculated proof node order. This enables us to not hold the tree at any point and also avoid holding $\texttt{cmt}_0$, resulting in a reduction of memory complexity from $(3t-1)\times 2\lambda$ to $\log_2(t)\times 2\lambda + 2t$.

3.2 On-the-fly GGM Tree Algorithms

We first recall the construction of the GGM tree in CROSS. Then we introduce the SeedPath function and our optimizations.

3.2.1 SeedPath and Its Memory Footprint

A GGM tree uses a CSPRNG function to generate nodes from top to bottom and left to right. The root is initialized with the λ-bit `Seed`. Then, each node is expanded into two children by appending the `Salt` and the 16-bit index of the node to the seed of the node and feeding it into the CSPRNG that produces two seeds of λ bits. The leaves are then used to construct commitments.

The function SeedPath selects a subset of GGM tree nodes that enable the verifier to reconstruct the leaves corresponding to indexes $\{1 \le i \le t \mid \texttt{chall}_2[\texttt{i}] = 1\}$. A trivial way to implement it is to reveal all the $\texttt{Seed}[\texttt{i}]_{i \mid \texttt{chall}_2[\texttt{i}]=1}$, but this would result in a large signature size. SeedPath aims to determine a path with minimum number of nodes by leveraging the GGM tree structure. As a result, it can give a strictly smaller subset of (potentially) intermediate nodes such that all the required leaf nodes can be calculated.

The reference implementation of SeedPath traverses the GGM tree from root to leaves. A flag tree is exploited to reveal the nodes closest to the root such that the union of their descendant leaves is $\{\texttt{Seed}[\texttt{i}] \mid \texttt{chall}_2[\texttt{i}] = 1, 1 \le i \le t\}$. As a result, the major memory consumption of SeedPath comes from holding the GGM tree $\mathcal{T}$. The tree $\mathcal{T}$ is generated in

SeedLeaves and then passed to this algorithm, which uses $(2t-1)\times\lambda$ (37,504-425,728 bits) memory, then an additional flag tree of size $(2t-1)\times 8$ (2,344-13,304 bit).

3.2.2 Memory Optimized SeedPath

The main goal of our memory optimization is to calculate the node that we need to reveal in the `Path` without holding the whole GGM tree.

There were two main insights which allowed us to save considerable memory when running SeedPath. Firstly, we noticed that as we traversed the GGM tree from root to leaves, due to the structure of the constructed binary tree, we can determine the number of descendant leaves. Given a node with d total descendant leaves the left subtree always had a number of leaf descendants d_l, such that d_l was the highest power of 2 strictly less than d. Thus, ensuring the right subtree to have $d-d_l$ descendent leaves.

Furthermore, given the number of leaves descendant from each subtree, we are able to determine on the fly which nodes to hide, and which to reveal. This allowed us to, instead of holding all of the GGM tree hashes, recalculate the hashes as needed, more than halving the memory needed to store nodes.

Consequently, by adapting the SeedPath algorithm with these insights, we were able to achieve a benchmarked memory reduction of approximately 8% in the Signature Generation algorithm.

4 Scheme Specific Memory Optimizations

This section describes different techniques to reduce the memory footprint of various components used in different CROSS algorithms. Unlike the previous optimizations, these techniques have been developed specifically for CROSS by close inspection of various algorithms of CROSS. The main techniques include

- Streaming evaluations of matrix multiplications,
- Just-in-time evaluation of `Hash` functions,
- Reuse of variables based on program flow analysis.

4.1 Memory-efficient Matrix Multiplications

In the Key Generation algorithm (`CROSS_KeyGen` Alg. 1), the matrices $\boldsymbol{V}\in\mathbb{F}_p^{k\times(n-k)}$ and $\overline{\boldsymbol{W}}\in\mathbb{F}_z^{m\times(n-m)}$ have a large memory footprint. The matrices $\boldsymbol{V}$ and $\overline{\boldsymbol{W}}$ occupy $1368-15150$ bytes and $750-2784$ bytes of memory, respectively. The `RSDP` variant of CROSS uses $1368-15150$ bytes for matrix $\boldsymbol{V}$ and `RSDPG` variant uses $2118-15150$ bytes for $\boldsymbol{V}$ and $\overline{\boldsymbol{W}}$. Naturally, reducing the memory footprint of these matrices is important milestone to achieve our goal of memory efficient CROSS.

In Alg. 1, CSPRNG first generates these matrices fully and then stores them. After that, vectors $\overline{\mathbf{e}}_G$, $\overline{\mathbf{e}}$ and $\mathbf{e}$ are also computed and stored. Now, the stored vectors $\mathbf{e}$, $\overline{\mathbf{e}}$ and matrices $\boldsymbol{V}$, $\overline{\boldsymbol{W}}$ are multiplied to generate the public vector $\mathbf{s}$ as shown below.

$$\boldsymbol{s}[j]=\boldsymbol{e}[k+j]+\sum_{i=0}^{k}\boldsymbol{e}[i]\cdot\boldsymbol{V}[i,j],\ \text{for each}\ \ 0\le j\le n-k-1.$$

$$\overline{\boldsymbol{e}}[j]=\overline{\boldsymbol{e}}_{\boldsymbol{G}}[m+j]+\sum_{i=0}^{m}\overline{\boldsymbol{e}}_{\boldsymbol{G}}[i]\cdot\overline{\boldsymbol{W}}[i,j],\ \text{where}\ 0\le j\le n-m-1.$$

Notice that each entry $\boldsymbol{V}[i,j]$ and $\overline{\boldsymbol{W}}[i,j]$ is required only once to generate the vectors $\mathbf{s}$ and $\overline{\mathbf{e}}$ respectively. Therefore, we first compute $\overline{\mathbf{e}}_{\mathbf{G}}$ before computing $\boldsymbol{W}$. We initialize each $\overline{\boldsymbol{e}}[j]$ with $\overline{\boldsymbol{e}}_{\boldsymbol{G}}[m+j]$. Then, as we generate each (i,j)-th element of $\overline{\boldsymbol{W}}$ say $w:=\overline{\boldsymbol{W}}[i,j]$, we update

$\overline{\boldsymbol{e}}[j]$ as $\overline{\boldsymbol{e}}[j]+\overline{\boldsymbol{e}}_{\boldsymbol{G}}[i]\cdot w$. This saves us from storing the entire matrix $\overline{\boldsymbol{W}}$ without any effect on the final values of the vector $\overline{\mathbf{e}}$. Similarly, the multiplication of $\boldsymbol{V}$ and $\mathbf{e}$ is possible without storing $\boldsymbol{V}$. However, this solution is not this straightforward as the elements of the matrices $\boldsymbol{V}$ and $\overline{\boldsymbol{W}}$ are populated using rejection sampling. The reference code generates one large fixed buffer and then uses rejection sampling to populate these matrices. However, this approach is clearly counterproductive for us as we need a large buffer memory. Instead, we generate a small buffer window and refill it using the CSPRNG as needed. We sample the exact number of bits required, buffer window, and right-shift it into the matrix element. Additionally, in the RSDPG variant, we must permute the hash state to the constant sample bits at the end for correctness.

So, for the RSDPG variant, the computation flow should be like: 1) $\overline{\mathbf{e}}_G$ computation; 2) $\overline{\mathbf{e}}$ computation without storing the matrix $\overline{\boldsymbol{W}}$; and 3) $\mathbf{s}$ computation without storing the matrix $\boldsymbol{V}$. Contrast it with CROSS_KeyGen Alg. 1, where the execution sequence was: 1) compute and store the matrix $\boldsymbol{V}$, $\overline{\boldsymbol{W}}$; 2) compute and store the vector $\overline{\mathbf{e}}_G$; 3) compute the vector $\overline{\mathbf{e}}$; and 4) compute the vector $\mathbf{s}$. We explain our method in Updated_CROSS_KeyGen() Alg. 5, where we call the compute_syndrome line 7 to generate the public vector $\mathbf{s}$ using the secret vector $\overline{\mathbf{e}}$ and public seed $\mathtt{seed}_{\mathtt{pk}}$.

Both vectors $\mathbf{e}$ and $\overline{\mathbf{e}}_G$ corresponding to RSDP and RSDPG variants, respectively, depend only on the secret seed $\mathtt{seed}_{\mathtt{e}}$. Similarly, the matrices $\boldsymbol{V}$ and $\overline{\boldsymbol{W}}$ are dependent on another seed $\mathtt{seed}_{\mathtt{pk}}$. If we generate the vector $\mathbf{e}$ (or $\overline{\mathbf{e}}_G$) before generating matrices $\boldsymbol{V}$ and $\overline{\boldsymbol{W}}$ (change the generation order but use the same seed), then the vectors $\mathbf{e}$, $\mathbf{e}_G$ and the matrices $\boldsymbol{V}$ and $\overline{\boldsymbol{W}}$ will be the same for fixed seeds $\mathtt{seed}_{\mathtt{e}}$ and $\mathtt{seed}_{\mathtt{pk}}$. Also, as the matrices $\boldsymbol{V}$, $\overline{\boldsymbol{W}}$ and the vectors $\mathbf{e}$ and $\overline{\mathbf{e}}_G$ are the same for both the algorithms, our computed $\mathbf{s}$ will be the same as the one computed by the reference implementation. In conclusion, the input-output behavior of both algorithms are the same *i.e.* for fixed $\mathtt{seed}_{\mathtt{sk}}$, the public key ($\mathtt{seed}_{\mathtt{pk}}$, $\mathbf{s}$) and the private key $\mathtt{seed}_{\mathtt{sk}}$ generated in our optimized Alg. 5 will be identical to the public key and private key generated in Alg. 1.

4.2 Just-in-time Strategy for Hash Function

In the CROSS_Sign (Alg. 2), we have observed that two large arrays have been stored. One is the commitment array $\mathtt{cmt}_1$, and the other is the response $\mathbf{y}$. The arrays $\mathtt{cmt}_1$ and $\mathbf{y}$ have been used as input to the Hash function (Line 18 and Line 30), which takes $5216-63744$ bytes and $16830-242968$ bytes of memory, respectively. In this section, we describe our *Just-in-time* computation method of the hash value of these commitments and responses without storing these arrays. This strategy has been used in several previous works to reduce stack memory requirements [MKV20, BKS19]. However, this strategy often affects the performance of the scheme to keep the functionality. In this work, our goal is to study whether the *Just-in-time* strategy can be used in the implementation of CROSS to reduce memory without affecting its speed significantly.

Most common fips202 libraries [KSSW22, KRSS] include an **incremental** SHAKE variation (xof_shake_(init|update|final|extract)) which we can utilize to apply *Just-in-time* strategy. The basic SHAKE uses a state comprising of 25 64-bit words defining the 5×5 matrix for XOR and permutation. However, the incremental variant uses a state of 26 64-bit words, with the additional word holding the not-yet-permuted/not-yet-squeezed bytes. This allows performing both absorb and squeeze operations on inputs that are not multiples of r without explicit bookkeeping. So in the absorb step, if we don't pass a multiple of r, it sets state[25] to buffer len mod r *i.e.,* not-yet-permuted bytes. Then we have to use a finalize function to add the domain separation bytes after absorbing all the desired bytes. Finally, the bytes are squeezed out while tracking the not-yet-squeezed bytes in state[25]. We have explained the technique to generate the $\mathsf{Hash}(\mathtt{cmt}_1[1], \cdots, \mathtt{cmt}_1[t])$ below.

Step 1. Initialize state before the generation of the first value ($\mathtt{cmt}_1[1]$) of the commitment $\mathtt{cmt}_1$. We have to initialize state before the first for loop (between lines 4 and 6) in CROSS_Sign Alg . 2.

Algorithm 5 Updated_CROSS_KeyGen()

Input: Security parameter λ
Output: Public key: $(\mathtt{Seed}_{\mathtt{pk}}, \mathbf{s})$, Private key: $\mathtt{Seed}_{\mathtt{sk}}$

1: // Sampling seeds
2: $\mathtt{Seed}_{\mathtt{sk}} \xleftarrow{\$} \{0,1\}^{\lambda}$
3: $(\mathtt{Seed}_{\mathtt{e}}, \mathtt{Seed}_{\mathtt{pk}}) \leftarrow \mathsf{CSPRNG}\left(\mathtt{Seed}_{\mathtt{sk}} \,|\, 3t+1, \{0,1\}^{2\lambda} \times \{0,1\}^{2\lambda}\right)$
4: // Computing $\vec{e}$
5: $\overline{\mathbf{e}}_G \leftarrow \mathsf{CSPRNG}(\mathtt{Seed}_{\mathtt{e}} \,|\, 3t+3, \mathbb{F}_z^m)$; $\overline{\mathbf{e}}[n-m,\ldots,n] \leftarrow \overline{\mathbf{e}}_G$; $\overline{\mathbf{e}} \leftarrow \mathsf{CSPRNG}(\mathtt{Seed}_{\mathtt{e}} \,|\, 3t+3, \mathbb{F}_z^n)$
6: // Computing the syndrome s
7: $\mathbf{s} \leftarrow$ compute_syndrome($\overline{\mathbf{e}}$,$\mathtt{Seed}_{\mathtt{pk}}$)
8: // Return the keys
9: **return** Public key: $(\mathtt{Seed}_{\mathtt{pk}}, \mathbf{s})$, Private key: $\mathtt{Seed}_{\mathtt{sk}}$

Function: compute_syndrome($\overline{\mathbf{e}}$, $\mathtt{Seed}_{\mathtt{pk}}$)
10: $\mathbf{w} \leftarrow \emptyset, r_w \leftarrow 0$
11: $\mathbf{b} \leftarrow \emptyset, r_b \leftarrow 0, i_b \leftarrow 0, l_b \leftarrow \mathtt{SHAKE_RATE}+4$
12: $p_b \leftarrow \textsc{BitsToRepresent}(p)$
$\overline{\boldsymbol{W}} \leftarrow \mathsf{CSPRNG}\left(\mathtt{Seed}_{\mathtt{pk}} \,|\, 3t+2, \mathbb{F}_z^{m \times (n-m)}\right)$
13: $\overline{\boldsymbol{M}} \leftarrow [\overline{\boldsymbol{W}} \,|\, \mathrm{Id}_m]$
$\overline{\mathbf{e}} \leftarrow \overline{\mathbf{e}}[n-m,\ldots,n]\overline{\boldsymbol{M}}$ ▷ Recall $\overline{\boldsymbol{e}}[n-m,\ldots,n] = \overline{\mathbf{e}}_G$
14: **for** $j = 0,\ldots,n-1$ **do**
15: $\quad \boldsymbol{e}[j] \leftarrow g^{\overline{\mathbf{e}}[j]}$
16: $\mathbf{s} \leftarrow \boldsymbol{e}[k,\cdots,n]$
17: **for** $i \in \{0,\ldots,k-1\}$ **do**
18: $\quad$ **for** $j \in \{0,\ldots,n-k-1\}$ **do**
19: $\quad\quad$ **loop**
20: $\quad\quad\quad$ **if** $r_w < 32$ **then**
21: $\quad\quad\quad\quad$ **if** $\mathtt{to_squeeze} - \mathtt{sampled} + r_b < 4$ **then**
22: $\quad\quad\quad\quad\quad \mathtt{sample_size} \leftarrow \mathtt{to_squeeze} - \mathtt{sampled} + r_b$
23: $\quad\quad\quad\quad$ **else**
24: $\quad\quad\quad\quad\quad \mathtt{sample_size} \leftarrow 4$
25: $\quad\quad\quad\quad$ **if** $r_b \leq \mathtt{sample_size} \wedge \mathtt{sampled} < \mathtt{to_squeeze}$ **then**
26: $\quad\quad\quad\quad\quad s_b \leftarrow \min(\mathtt{rem_to_sample}, \mathtt{SHAKE_RATE})$
27: $\quad\quad\quad\quad\quad \mathbf{b}[0 \ldots r_b] \leftarrow \mathbf{b}[i_b \ldots i_b + r_b]$
28: $\quad\quad\quad\quad\quad i_b \leftarrow 0$
29: $\quad\quad\quad\quad\quad \mathbf{b}[r_b+1 \ldots r_b+1+s_b] \leftarrow \mathsf{CSPRNG}\left(\mathtt{Seed}_{\mathtt{pk}} \,|\, 3t+2, \mathbb{F}_p^{s_b}\right)$
30: $\quad\quad\quad\quad\quad \mathtt{sampled} \leftarrow \mathtt{sampled} + s_b$
31: $\quad\quad\quad\quad\quad r_b \leftarrow r_b + s_b$
32: $\quad\quad\quad\quad \mathbf{w} \leftarrow \mathbf{b}[i_b, \ldots, i_b+4]$
33: $\quad\quad\quad\quad i_b \leftarrow i_b + 4$
34: $\quad\quad\quad\quad \mathbf{v}_w \leftarrow \textsc{ShiftLeft}(\mathbf{w}, r_w)$
35: $\quad\quad\quad\quad r_w \leftarrow r_w + \mathtt{sample_size} * 8$
36: $\quad\quad\quad\quad r_b \leftarrow r_b - \mathtt{sample_size}$
37: $\quad\quad\quad v \leftarrow \mathbf{w} \wedge \{1\}^{p_b}$
38: $\quad\quad\quad \mathbf{w} \leftarrow \textsc{ShiftRight}(\mathbf{w}, p_b)$
39: $\quad\quad\quad r_w \leftarrow r_w - p_b$
40: $\quad\quad\quad$ **if** $v < p$ **then**
41: $\quad\quad\quad\quad$ **break**
42: $\quad\quad s[j] \leftarrow \boldsymbol{s}[j] + \overline{\boldsymbol{e}}[i] \cdot v$
43: **return** $\mathbf{s}$

Step 2. Generate the commitment $\mathtt{cmt}_1[1]$ (for $i = 1$ in the for loop) and store it as $\mathtt{cmt}' = \mathtt{cmt}_1[1]$. Pass the commitment $\mathtt{cmt}'$ to `xof_shake_update` that will update the `state` inside the for loop with the input $\mathtt{cmt}'$ after $\mathtt{cmt}'$ generation (after line 14). Repeat thie above steps process sequentially (except the first step) for $i = 2, \cdots, t$, *i.e.*, store each commitment $\mathtt{cmt}_1[i]$ as $\mathtt{cmt}'$ and update `state`.

Step 3. After all commitments have been generated, pass domain separation to `xof_shake_update`. Call `xof_shake_final` and `xof_shake_extract` functions to generate the final hash (immediately after the first for loop).

Using the above method, the same hash value $\mathtt{digest}_{\mathtt{cmt}_1}$ (line 16, Alg. 2) can be calculated without storing the full commitment array $\mathtt{cmt}_1$. However, we need the values $\mathtt{cmt}_1[i]$ to assign the signature component `resp` where $1 \leq i \leq t$ satisfies $\mathtt{chall}_2[i] = 0$. Since we do

not store the full arrays $\texttt{cmt}_1$ we have to recompute $\texttt{cmt}_1[i]$ for i which satisfies $\texttt{chall}_2[i]=0$. Since the number of such i is much less (for example, for the `RSDP 1 small` parameter set, this number of values of such i is 32, where $t=520$), it will require very little extra computation. With this method, we significantly reduce the amount of stack memory at the cost of a few extra computations. Similarly, we can pass the $\mathbf{y}$ values to compute $\texttt{digest}_{\texttt{chall}_2}$ (line 28, Alg. 2) without storing the response array $\mathbf{y}$.

We explain the optimization techniques of the just-in-time strategy for the Hash function in the signature generation; we use similar techniques on the counterpart functions in the verification algorithm. According to benchmarking, the hashing optimizations result in a memory improvement of approximately 34% in Signature Generation and 63% in Verification.

4.3 Variable Usage Optimization

Another approach to optimize the memory footprint of an operation is to reuse variables based on mathematical equivalences and execution flow analysis. This straightforward idea turns out to be very useful. This section shows some examples of optimizing certain operations of CROSS_Sign.

4.3.1 Light CROSS

Optimized Exponents Generation. Consider line 10 of CROSS_Sign. We can see that each column of $\overline{\mathbf{v}}$ and $\overline{\mathbf{e}}'$ (both vectors in $\mathbb{F}_z^n$) encode the same information simply offset by the error vector exponent $\overline{\mathbf{e}}\in\mathbb{F}_z^n$, where $\overline{\mathbf{e}}$ was generated and stored as shown in line 3. That is to say:

$$\overline{\mathbf{v}}[i]=\overline{\mathbf{e}}-\overline{\mathbf{e}}'[i].$$

To reduce memory usage, we hold only one of $\overline{\mathbf{v}}$ and $\overline{\mathbf{e}}'$. When the alternative one is needed, we compute that using the above equation and the stored $\overline{\mathbf{e}}$. Since each of $\overline{\mathbf{v}}$ and $\overline{\mathbf{e}}'$ will only be used once in CROSS_Sign, the computational cost is small. Suppose that we store only the variable $\overline{\mathbf{e}}'$ and compute $\overline{\mathbf{v}}$ whenever required. We are computing the exact value $\overline{\mathbf{v}}$ and using it in the sign function, but not storing it. So, all the $\overline{\mathbf{v}}$-dependent component will not be changed. This gives us a memory improvement in Signature Generation of 8.79-17.3%.

Optimization by Execution Flow Analysis. By utilizing execution flow analysis, we gain some speed and memory benefits in computing the first response and the second challenge. Observe that in line 28 of CROSS_Sign, we calculate $\mathbf{y}$ as:

$$\mathbf{y}[i]\leftarrow\mathbf{u}'[i]+\texttt{chall}_1[i]\mathbf{e}'[i].$$

Please note that we never use the $\mathbf{u}'$ vector again in the signature generation algorithm. Therefore, we can discard the value of $\mathbf{u}'$ and use it to store the computed value of $\mathbf{y}'$ to save space. For line 30, we can immediately feed $\mathbf{y}[i]$ directly to the incremental `Hash` just after generation, following the strategy of Sect. 4.2. Here, we are computing the value $\mathbf{y}'$ and storing it in $\mathbf{u}'$, which is eventually passed to `resp`. This current `resp` is the same as the value of `resp` of the original CROSS.

Although we explain the strategy for memory optimization techniques just-in-time strategy for Hash function and variable usage optimization for signature algorithm, these similar techniques can be applied in the verification algorithm as well. Through our benchmarking this results in an 8.8-17.4% improvement in memory usage in the Signature Generation, and a 7.2-8.8% improvement in the Verification.

4.3.2 Ultra CROSS: Allowing Recalculation

For the maximally stack-optimized implementation, we go a step further than paring down variables that encode overlapping information as above. We allow for the complete recalculation of variables from their respective seeds. From the memory profiling of reference

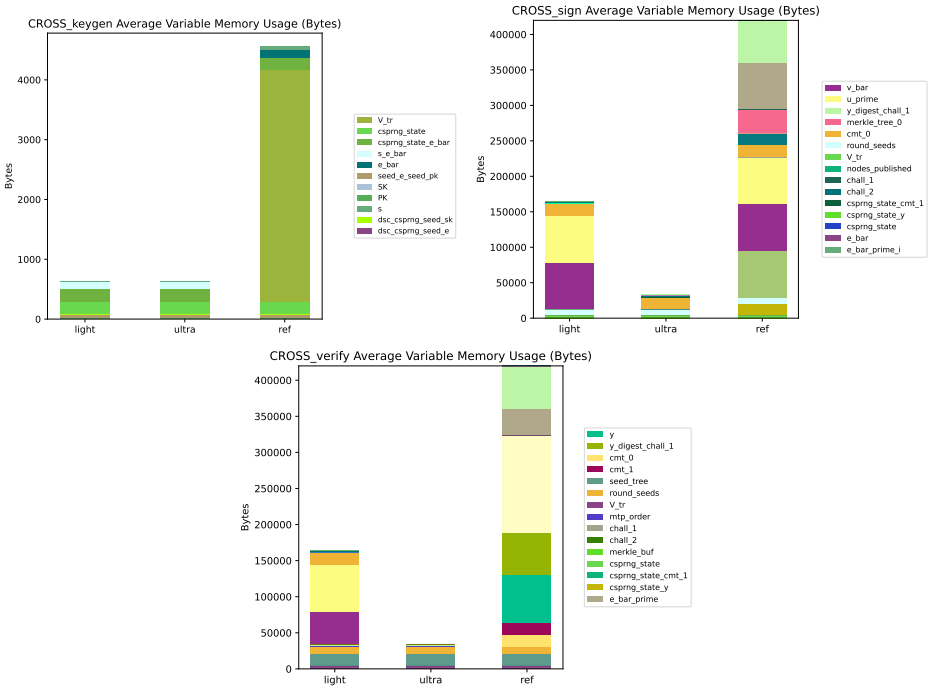

Figure 3: The comparison between the different implementations of each of the main algorithms. The average breakdown of variable memory usage (over the lifetime of the algorithm) is shown. The legend shows the top 15 memory consumers, which is given for the RSDP-1-small variant.

implementation (`ref`) shown in Figure 3, we observe the three matrices `u_prime`, `e_bar` and `v_bar`, each of order $t \times n$, consume the majority of the memory in the implementation.

If we use these variables without storing them *i.e.*, we generate them on-the-fly every time we need them, then we can save a significant amount of memory. However, these variables are used multiple times in the implementation. For example, the matrix `u_prime` is used to generate another matrix `u` (line 13, Alg. 2) as well as `y` (line 28, Alg. 2). Similarly, `e_bar` and `v_bar` are also used in multiple locations (lines 12, 37, and 10, Alg. 2). Therefore, in this case, we must regenerate them multiple times. This regeneration process requires a significant amount of extra computation, as the variables are calculated from the CSPRNG as shown in Alg. 2, lines 9 and 10, which substantially reduces the speed. Therefore, in some memory-constrained use-cases where we can sacrifice some efficiency, we can use this extreme memory reduction method.

The comparison of variable memory usage of different implementations (reference, light, ultra) is shown in Figure 3. We present the benchmark results of all implementations of CROSS in Table 3.

5 Speed Optimizations

In this section, we focus on improving the performance of CROSS's algorithms. Speed is another optimization dimension for implementing a cryptographic scheme on embedded devices such as the ARM Cortex-M4. We observed that the vector-matrix multiplication is the most expensive operation after hash functions, e.g., in RSDP version of the CROSS Signature Generation algorithm with security level 1, optimization variant of 'small', it takes 22.7% of the whole sign algorithm, while the hashing takes 32%. Since this subroutine is invoked

several times in all three algorithms *i.e.* key generation, signature generation, and verification of CROSS, even a small improvement in this subroutine will make a significant impact on the overall runtime of all three algorithms. Therefore, we choose to optimize this subroutine.

As our target platform is the Cortex-M4, our main recourse for optimized assembly instructions is the Digital Signal Processing (DSP) instruction set. The DSP instruction set is designed to accelerate real-time, signal-processing-intensive applications. However, one can use these instructions to their advantage for faster arithmetic operations. The 32-bit registers allow us to store two 16-bit operands for parallel operations, thus allowing us to do calculations on 8-bit values without losing precision from overflow. The only implementations of the CROSS scheme currently available are the reference implementation and the optimized AVX2 implementation. Thus, this would be the first work to apply DSP instructions to optimize the CROSS algorithm. Currently, we use this for two main purposes: when doing vector-matrix multiplication in $\mathbb{F}_p$, and when doing it in $\mathbb{F}_z$. We found the following two DSP instructions for this purpose:

- `UXTB16`: Dual extract 8-bits and zero-extend to 16-bits
- `SMLALD`: Dual 16-bit signed multiply with single 64-bit accumulator

More specifically, given a matrix $\boldsymbol{M} \in \mathbb{F}_p^{k \times n}$ and some vector $\mathbf{v} \in \mathbb{F}_p^k$, the vector-matrix multiplication $\mathbf{p} = \mathbf{v}\boldsymbol{M}$ can be written as $\mathbf{p}[i] = \sum_{j=0}^{k-1} \mathbf{v}[j] \cdot \boldsymbol{M}[j][i], 0 \leq i \leq n-1$.

In the reference implementation, the assembly of the above computation first multiplies $\mathbf{v}[j]$ with $\mathbf{M}[i][j]$, and then the result of the multiplication is added to 64-bit 'row accumulators' `r7` and `r8`. This requires three instructions per value of $\mathbf{p}[i]$. We used DSP instructions, which are able to multiply and accumulate elements 2 at a time and thus multiply full blocks of 4 elements in 8 instructions.

Table 2: Comparing a naive vector-matrix multiplication of 4 elements, to one that leverages DSP instructions. Both routines multiply four bytes along some column i from row j to $j+3$ and accumulate the final 64-bit sum in `r7` and `r8`. All cycle counts obtained from ARM Technical Reference Manual[ARM10]

Naive multiplication with accumulation	Cycles	Parallel multiplication using DSP	Cycles
`LDRB r3, <v[j]>;`	2	`LDR r3, <v[j]>;`	2
`LDRB r4, <M[j][i]>;`	1[1]	`LDR r4, <M[j][i]>;`	1[1]
`SMLABB r7, r8, r3, r4;`	1	`UXTB16 r5, r3;`	1
`LDRB r3, <v[j+1]>;`	2	`UXTB16 r6, r4;`	1
`LDRB r4, <M[j+1][i]>;`	1[1]	`SMLALD r7, r8, r5, r6;`	1
`SMLABB r7, r8, r3, r4;`	1	`UXTB16 r5, r3, ROR #8;`	1
`LDRB r3, <v[j+2]>;`	2	`UXTB16 r6, r4, ROR #8;`	1
`LDRB r4, <M[j+2][i]>;`	1[1]	`SMLALD r7, r8, r5, r6;`	1
`SMLABB r7, r8, r3, r4;`	1		
`LDRB r3, <v[j+3]>;`	2		
`LDRB r4, <M[j+3][i]>;`	1[1]		
`SMLABB r7, r8, r3, r4;`	1		
	16		**9**

[1] This second load is 1 cycle instead of two because of memory pipelining.

There are three parts to this optimized algorithm: First, we need to load four values each from the vector and matrix for parallel multiplication. This is straightforward for the vector values as they are stored in contiguous memory locations. Although, storing matrix $\boldsymbol{M}$ in row-major order is standard, we store $\boldsymbol{M}$ in column-major order in our implementation.

This does not affect any computation, but it does require us to make small changes to our matrix access pattern. To store it in a column-major way, we do all the generation the same, and simply in the final assignment, do $\boldsymbol{M}[j][i] = m_{ij}$. Similarly, when accessing an element in the i-th and j-th column, we use $\boldsymbol{M}[j][i]$ instead of $\boldsymbol{M}[i][j]$. This resulted in the values in the same column being adjacent in memory, allowing us to load multiple values from the same column as one block into a register.

With four values loaded per register, we can now pack them for parallel multiplications. The `UXTB16` will allow us to pack two alternating (index 1 and 3) 8-bit values into one 32-bit value (zero extending to 16 bits each). It also takes a third argument allowing us to rotate the input register before loading it, allowing us to pack the other two alternating 8-bit values (index 0 and 2) for the second round of multiplication. Using this instruction, we can pack four registers:

$r5 = \{\mathbf{v}[j+1], \mathbf{v}[j+3]\}, r6 = \{\boldsymbol{M}[j+1], \boldsymbol{M}[j+3]\}$For the first multiplication
$r5 = \{\mathbf{v}[j+0], \mathbf{v}[j+2]\}, r6 = \{\boldsymbol{M}[j+0], \boldsymbol{M}[j+2]\}$For the second multiplication (after rotation)

The final part of the algorithm is the parallel multiply-and-accumulate operation. This takes the two registers packed with two 16-bit values respectively and then accumulate them in a 64-bit accumulator. An implementation note is that the `SMLALD` operation used here requires a 64-bit operator, so we must use two registers. From Table 2, we can see that this implementation saves us 50% of the cycles from the naive implementation. Note that if the number of elements to be multiplied is not a perfect multiple of 4, we must use the naive implementation for the remaining values. We must take care to account for the fact that all of these operations are done as group arithmetic in either $\mathbb{F}_z$ or $\mathbb{F}_p$. For this purpose, we use the same modulo reduction used in the reference implementation.

The full function is given here where `p` holds the end result, `v` and `M` are the vector and matrix, respectively. Here, `P` is the prime modulus of $\mathbb{F}_p$. Note that we have changed some argument names, function styling, and macro usage from the actual code for ease of reading:

```
static void fp_vec_by_fp_matrix(uint8_t p[N - K], uint8_t v[N], uint8_t M[N - K][K]) {
  memcpy(p, v + K, (N - K));
  // Loop over every column
  for (int j = 0; j < N - K; j++) {
    uint64_t col_accum = 0; int i = 0;
    // Loop over every block of 4 rows
    for (; i < K; i += 4) {
      uint32_t v_block = *((uint32_t *)&v[i]);
      uint32_t M_block = *((uint32_t *)&M[j][i]);
      // Extract value v[i+1], v[i+3], M[i+1], M[i+3]
      uint32_t bottom_v = UXTB16(v_block);
      uint32_t bottom_M = UXTB16(M_block);
      // Extract value v[i], v[i+2], M[i], M[i+2]
      uint32_t top_v = UXTB16(ROR(v_block, 8));
      uint32_t top_M = UXTB16(ROR(M_block, 8));
      // Calculate
      col_accum = SMLALD(bottom_v, bottom_M, col_accum);
      col_accum = SMLALD(top_v, top_M, col_accum);
      col_accum = MOD(col_accum, P);
    }
    // Finish remaining rows if it is not a multiple of 4
    for (; i < K; i++)
      col_accum += MOD(((uint16_t)v[i] * (uint16_t)M[j][i]), P);
    // Store and reduce modulo group modulo P
    p[j] = MOD(((uint64_t)p[j] + col_accum), P);
  }
}
```

This optimisation can be used for the 8-bit values in $\mathbb{F}_p$ and $\mathbb{F}_z$ arithmetic in the `RSDP` variant, and additionally for $\mathbb{F}_z$ arithmetic in the `RSDPG` variant. But due to the larger p value for `RSDPG`, we take a slightly different approach where we do not pack the values using `UXTB16`. The two 16-bit values can be directly loaded from the arrays with `LDR`. Following the load, the parallel multiplication and accumulation operations remain the same.

This optimization greatly improves the speed of Signature Generation and Verification where we have the matrices stored as the matrices are used more than once for computation. However, for key generation, we did implement a modified version which stores 4 rows at a

time to use in conjunction with the matrix optimizations in section 4.1. The overhead from temporarily storing the values and then returning to multiply outweighed the performance gain from the improved multiplication. This was supported by testing where we saw a 4% slowdown with a 17% increase in memory usage. So we decided not to include this variation in our Key Generation implementation.

6 Implementation Results

Table 3: The improvements each of our optimization steps provides on the implementation of the CROSS RSDP-1 small variant on ARM CortexM4.

CROSS-RSDP-I small variant	Steps	Memory (bytes)			Performance (kcycles)		
		KG (diff [%])	Sign (diff [%])	Verify (diff [%])	KG (diff [%])	Sign (diff [%])	Verify (diff [%])
Ref. [BBB+25a]	-	8,112	421,392	223,832	400	76,710	36,495
Our optimizations	OPT_KEYGEN 4.1	1,120 (86.19)	421,392 (0)	223,896 (-0.03)	518 (-29.5)	78,764 (-2.68)	36,242 (0.69)
	OPT_MERKLE 3.1	1,120 (0)	404,744 (3.95)	207,384 (7.37)	518 (0)	76,299 (3.13)	36,413 (-0.47)
	OPT_HASH_CMT1 4.2	1,120 (0)	388,584 (3.99)	190,984 (7.91)	518 (0)	76,862 (-0.74)	36,505 (-0.25)
	OPT_HASH_Y 4.2	1,120 (0)	264,608 (31.9)	67,048 (64.89)	518 (0)	79,447 (-3.36)	36,326 (0.49)
	OPT_V_BAR 4.3	1,120 (0)	198,696 (24.91)	67,048 (0)	518 (0)	79,353 (0.12)	36,325 (0)
	OPT_E_BAR_PRIME 4.3	1,120 (0)	198,696 (0)	67,048 (0)	518 (0)	77,142 (2.79)	36,327 (-0.01)
	OPT_OTF_MERKLE 3.1	1,120 (0)	183,080 (7.86)	50,424 (24.79)	518 (0)	81,752 (-5.98)	36,699 (-1.02)
	OPT_GGM 3.2	1,120 (0)	168,352 (8.04)	50,424 (0)	518 (0)	86,206 (-5.45)	36,700 (0)
	OPT_RECOMPUTE_ROOT 3.1	1,120 (0)	168,352 (0)	35,340 (29.91)	518 (0)	86,229 (-0.03)	36,422 (0.76)
	OPT_DSP 5	1,120 (0)	168,360 (0)	35,252 (0.25)	518 (0)	68,236 (20.87)	35,514 (2.49)
	OPT_Y_U_OVERLAP 4.3	1,120 (0)	168,216 (0.09)	35,252 (0)	518 (0)	68,117 (0.17)	35,511 (0.01)
	OPT_KEYGEN_BLOCKS 4.1	1,296 (-15.71)	168,216 (0)	35,252 (0)	438 (15.44)	68,069 (0.07)	35,512 (0)
	OPT_U_PRIME_EPH 4.3	1,296 (0)	36,752 (78.15)	35,252 (0)	438 (0)	85,421 (-25.49)	35,511 (0)
	OPT_U_V_VERIFY 4.3	1,296 (0)	36,752 (0)	35,132 (0.34)	438 (0)	85,438 (-0.02)	35,514 (-0.01)

In this section, we present the results of our implementation of CROSS on the STM32 Nucleo-144 development board with the STM32L4R5ZI MCU, featuring an ARM Cortex-M4 chip. For our implementation, we have used the popular post-quantum cryptographic framework PQM4 [KRSS], along with the arm-none-eabi-gcc compiler, version 14.2.1.

We apply all the optimization procedures described in Sections 3, 4, and 5 to our CROSS implementations. In Table 3, we present the effect of each of our optimization steps on the implementation of CROSS-RSDP-1 small variant. We have implemented two versions of each of the variants of CROSS. One is with low memory, without any significant effect on the performance, we call it **light**. Another is with ultra-light memory, which we call **ultra**. The goal for the ultra version is to reduce the memory requirement for all the variants of CROSS below 128kB for all the security levels. In Fig. 3, we present the memory profile of the top 15 variables by average size over the course of Key Generation, Signnature Generation, and Verification algorithms for three versions of CROSS implementations: reference, light, and ultra.

The major speed bottlenecks were the field arithmetic operations after the Hash or CSPRNG functions. We conduct a bottleneck analysis and observe that Key Generation and Verifying algorithms spend 90%+ cycles on the Hash or CSPRNG. Whereas in the Signature Generation algorithm, the Hash operation takes $\sim$40%, the CSPRNG operation takes $\sim$25%, the field arithmetic takes $\sim$ 22%, and the remaining operations take $\sim$ 13% of the total clock cycles. Therefore, we focus on speeding up the field arithmetic, as improving hashing libraries is of independent interest and beyond the scope of this work.

In Table 4, we present the memory consumption of our implementations of CROSS for all variants. In Table 5, we present the speed performance of our implementations of CROSS for all variants. Our efficient implementations of all the variants of CROSS are presented in the table under the "our work" column. To compare them with the reference code of CROSS-v2.1 [BBB+25a], we run the reference on the ARM Cortex-M4 board and add the result obtained in the column "ref". In Table 4 and 5, we also compare our implementation results with the implementations of the NIST current standard DS schemes: ML-DSA [KRSS], FN-DSA [KRSS], SPHINCS$^+$-shake [KRSS], Korean PQC [Qua25] standard DS schemes: HAETAE [KRSS], AIMer-speed [KRSS], and some Round-two candidates of the NIST additional DS standardization procedure: FAEST [ADE+25], Mayo [KRSS].

Table 4: Memory benchmarking results of the reference CROSSv2.1 against our implementation and other digital signature algorithms on ARM Cortex-M4. b represents `balanced`, f represents `fast`, and s represents `small`. $*$ values are calculated using QEMU memory emulation as they do not run on the board. The implementations of CROSS variants that require less than 32 KB are in bold.

Scheme	Sec.	Opt.	Key Gen. [bytes]		Sign [bytes]		Verify [bytes]	
			ref	our work (diff [%])	ref	our work (diff [%])	ref	our work (diff [%])
CROSS-RSDP	1	b	8,112	light: 1,296 (-84) **ultra: 1,296 (-84)**	210,016	light: 86,144 (-59) **ultra: 21,736 (-90)**	113,184	light: 20,600 (-82) **ultra: 20,708 (-82)**
		f	8,112	light: 1,296 (-84) **ultra: 1,296 (-84)**	116,224	light: 54,480 (-53) **ultra: 14,848 (-87)**	56,720	light: 14,672 (-74) **ultra: 14,672 (-74)**
		s	8,112	light: 1,296 (-84) ultra: 1,296 (-84)	421,392	light: 168,216 (-60) ultra: 36,752 (-91)	223,832	light: 35,132 (-84) ultra: 35,132 (-84)
	3	b	16,832	light: 1,400 (-92) ultra: 1,400 (-92)	463,484	light: 186,460 (-60) ultra: 43,576 (-91)	248,804	light: 42,248 (-83) ultra: 42,248 (-83)
		f	16,832	light: 1,400 (-92) **ultra: 1,400 (-92)**	257,976	light: 118,840 (-54) **ultra: 29,840 (-89)**	124,648	light: 29,832 (-76) **ultra: 29,832 (-76)**
		s	16,908*	light: 1,400 (-92) ultra: 1,400 (-92)	694,816*	light: 275,528 (-60) ultra: 59,532 (-91)	370,056*	light: 57,744 (-84) ultra: 57,744 (-84)
	5	b	29,700*	light: 1,544 (-95) ultra: 1,544 (-95)	826,208*	light: 329,720 (-60) ultra: 73,900 (-91)	441,672*	light: 72,016 (-84) ultra: 72,016 (-84)
		f	29,624	light: 1,544 (-95) ultra: 1,544 (-95)	463,008	light: 211,704 (-54) ultra: 51,152 (-89)	222,424	light: 51,312 (-77) ultra: 51,312 (-77)
		s	29,700*	light: 1,544 (-95) ultra: 1,544 (-95)	1,332,096*	light: 523,980 (-61) ultra: 107,460 (-92)	706,352*	light: 105,012 (-85) ultra: 105,012 (-85)
CROSS-RSDPG	1	b	3,984	light: 1,544 (-61) **ultra: 1,544 (-61)**	156,712	light: 68,032 (-57) **ultra: 26,420 (-83)**	94,372	light: 19,004 (-80) **ultra: 18,896 (-80)**
		f	3,984	light: 1,544 (-61) **ultra: 1,544 (-61)**	77,768	light: 40,280 (-48) **ultra: 16,208 (-79)**	42,168	light: 12,312 (-71) **ultra: 12,312 (-71)**
		s	3,984	light: 1,544 (-61) ultra: 1,544 (-61)	309,608	light: 131,488 (-58) ultra: 47,636 (-85)	184,632	light: 33,188 (-82) ultra: 33,188 (-82)
	3	b	7,440	light: 1,608 (-78) ultra: 1,608 (-78)	240,896	light: 103,680 (-57) ultra: 40,900 (-83)	145,156	light: 29,240 (-80) ultra: 29,240 (-80)
		f	7,440	light: 1,608 (-78) **ultra: 1,608 (-78)**	170,424	light: 86,512 (-49) **ultra: 33,688 (-80)**	91,312	light: 24,680 (-73) **ultra: 24,680 (-73)**
		s	7,440	light: 1,608 (-78) ultra: 1,608 (-78)	453,168	light: 191,184 (-58) ultra: 70,580 (-84)	270,912	light: 48,572 (-82) ultra: 48,524 (-82)
	5	b	11,776	light: 1,744 (-85) ultra: 1,744 (-85)	423,232	light: 179,328 (-58) ultra: 66,996 (-84)	255,760	light: 48,824 (-81) ultra: 48,824 (-81)
		f	11,776	light: 1,744 (-85) ultra: 1,744 (-85)	300,704	light: 151,032 (-50) ultra: 55,992 (-81)	159,872	light: 41,712 (-74) ultra: 41,712 (-74)
		s	11,852*	light: 1,744 (-85) ultra: 1,744 (-85)	755,496*	light: 314,300 (-58) ultra: 110,924 (-85)	452,840*	light: 78,560 (-83) ultra: 78,560 (-83)
ML-DSA-stack [KRSS]	1	-	4,408		5,080		2,712	
ML-DSA-speed [KRSS]	1	-	38,296		44,816		8,912	
FN-DSA [KRSS]	1	-	14,348		41,952		2,976	
SPHINCS$^+$-shake [KRSS]	1	f	2,760		2,372		2,568	
		s	2,988		2,460		1,872	
HAETAE [KRSS]	1	-	19,756		55,568		23,296	
AIMer [KRSS]	1	f	8,720		13,864		15,440	
		s	8,720		27,324		34,624	
FAEST [ADE+25]	1	f	808		43,725		42,394	
		s	808		43,725		42,701	
Mayo [KRSS]	1	-	179,276		251,204		253,484	

Thanks to our memory optimizations, we could run all parameter sets on the board, whereas only 14 out of 18 parameter sets of the CROSS-v2.1 reference implementation fit, as shown in Table 4. This was achieved through a 61-95% decrease in memory requirement for Key Generation, 48-92% for the Signature Generation, and 71-84% for Verification compared to the original reference implementation of CROSS. On top of that, the memory requirement for the "ultra" version of `CROSS-RSDP`, NIST security level 1 `balanced` and `fast` variants, `CROSS-RSDP`, NIST security level 3 `fast` variant, `CROSS-RSDPG`, NIST security level 1 `balanced` and `fast` variants, and `CROSS-RSDPG`, NIST security level 3 `fast` variant are under 32kB. The Key Generation algorithm of CROSS uses less stack memory compared to ML-DSA, FN-DSA, SPHINCS$^+$-shake, HAETAE, AIMer, and Mayo. The signature generation algorithm of CROSS (f) requires less stack memory than HAETAE and Mayo. The CROSS (f) verification algorithm needs less memory than HAETAE, AIMer (f), FAEST, and Mayo.

Our optimized implementations of CROSS's signature generation and verification algorithms are at least as fast as the reference implementations of CROSS. In some cases, our

Table 5: Speed benchmarking results of the reference CROSSv2.1 against our implementation and other digital signature algorithms on ARM Cortex-M4. Values do not run on the board and percentages that cannot be calculated are marked "-". b represents `balanced`, f represents `fast`, and s represents `small`.

Scheme	Sec.	Opt.	Key Gen. [kcycles] (mean)		Sign [kcycles] (mean)		Verify [kcycles](mean)	
			ref	our work (diff [%])	ref	our work (diff [%])	ref	our work (diff [%])
CROSS-RSDP	1	b	400	light: 438 (9) ultra: 438 (9)	38,880	light: 34,160 (-12) ultra: 43,303 (11)	18,643	light: 17,457 (-6) ultra: 17,456 (-6)
		f	400	light: 438 (9) ultra: 438 (9)	20,403	light: 17,002 (-17) ultra: 24,504 (20)	11,766	light: 9,117 (-23) ultra: 9,117 (-23)
		s	400	light: 438 (9) ultra: 438 (9)	76,710	light: 68,489 (-11) ultra: 85,421 (11)	36,495	light: 35,517 (-3) ultra: 35,516 (-3)
	3	b	982	light: 1,022 (4) ultra: 1,022 (4)	92,880	light: 75,336 (-19) ultra: 95,833 (3)	41,773	light: 37,226 (-11) ultra: 37,226 (-11)
		f	982	light: 1,022 (4) ultra: 1,022 (4)	53,985	light: 41,923 (-22) ultra: 58,501 (8)	31,198	light: 22,778(-27) ultra: 22,778 (-27)
		s	-	light: 1,022 (4) ultra: 1,022 (4)	-	light: 113,512 ultra: 141,538	-	light: 55,593 ultra: 55,591
	5	b	-	light: 1,824 ultra: 1,824	-	light: 130,171 ultra: 166,434	-	light: 63,723 ultra: 63,724
		f	1,681	light: 1,824 (9) ultra: 1,824 (9)	108,896	light: 75,305 (-31) ultra: 105,414 (-3)	58,706	light: 41,250 (-30) ultra: 41,250 (-30)
		s	-	light: 1,824 ultra: 1,824	-	light: 199,545 ultra: 263,838	-	light: 99,579 ultra: 99,577
CROSS-RSDPG	1	b	192	light: 200 (4) ultra: 200 (4)	26,038	light: 25,994 (-.2) ultra: 31,890 (22)	15,931	light: 13,772 (-13) ultra: 13,771 (-14)
		f	192	light: 200 (4) ultra: 200 (4)	12,416	light: 11,283 (-9) ultra: 15,931 (28)	7,770	light: 6,190 (-20) ultra: 6,190 (-20)
		s	192	light: 200 (4) ultra: 200 (4)	51,829	light: 51,547 (-.5) ultra: 62,218 (20)	31,859	light: 27,742 (-13) ultra: 27,743 (-13)
	3	b	413	light: 445 (8) ultra: 445 (8)	39,839	light: 36,276 (-9) ultra: 48,490 (22)	25,539	light: 19,962 (-22) ultra: 19,963 (-22)
		f	414	light: 445 (8) ultra: 445 (8)	29,651	light: 24,835 (-17) ultra: 36,876 (24)	19,490	light: 14,264 (-27) ultra: 14,265 (-27)
		s	414	light: 445 (8) ultra: 445 (8)	75,747	light: 68,554 (-9) ultra: 87,887 (16)	48,560	light: 39,120 (-19) ultra: 39,119 (-19)
	5	b	707	light: 744 (5) ultra: 744 (5)	73,735	light: 60,457 (-18) ultra: 80,384 (9)	45,888	light: 32,941 (-28) ultra: 32,941 (-28)
		f	707	light: 744 (5) ultra: 744 (5)	57,001	light: 44,142 (-23) ultra: 64,293 (13)	38,195	light: 25,592 (-33) ultra: 25,593 (-33)
		s	-	light: 744 ultra: 744	-	light: 107,790 ultra: 138,268	-	light: 58,983 ultra: 58,979
ML-DSA-stack [KRSS]	1	-	1,799		12,134		3,242	
ML-DSA-speed [KRSS]	1	-	1,426		3,943		1,421	
FN-DSA [KRSS]	1	-	67,693		22,469		396	
SPHINCS$^+$-shake [KRSS]	1	f	50,505		1,182,422		70,501	
		s	3,231,401		24,553,696		24,366	
HAETAE [KRSS]	1	-	6,743		21,993		918	
AIMer [KRSS]	1	f	490		28,590		26,750	
		s	490		277,353		225,136	
FAEST [ADE+25]	1	f	790		60,000		59,000	
		s	692		321,000		322,000	
Mayo [KRSS]	1	-	9,572		16,860		11,031	

implementations are even faster: signature generation is up to 31% faster, and verification is up to 30% faster than the reference implementations of CROSS. Key Generation is consistently slower or equal to the reference implementation; however, it is often a one-time cost compared to ongoing signing and verifying with the same keys. The key generation algorithm of CROSS is faster compared to ML-DSA, FN-DSA, SPHINCS$^+$-shake, HAETAE, FAEST, and Mayo. The signature generation algorithm of CROSS (f) has to spend fewer clock cycles than FN-DSA, HAETAE, SPHINCS$^+$-shake, AIMer, and FAEST. The CROSS verification algorithm is faster than SPHINCS$^+$-shake, AIMer, FAEST, and Mayo.

7 Security Analysis

In Sec. 3, we have explained the memory optimized techniques in the Merkle root, Merkle proof, and GGM path generation process. Our memory-optimization process computes the same Merkle root $\mathcal{T}[0]$ as the original CROSS version for the commitments $\texttt{cmt}_0[1], \cdots, \texttt{cmt}_0[t]$. Also, for a challenge $\texttt{chall}_2$, the output of our technique of Merkle proof and GGM path will be same as the original version. So, the optimization technique in this section does not provide any cryptanalytic advantage to an attacker. In Sec. 4, we have explained the optimization technique of matrix multiplications used in CROSS key generation, the optimized hash computation, and methods of reusing some variables for optimization in signature and verification. As we explained in Sec. 4.1, for fixed secret seed $\texttt{seed}_{\texttt{sk}}$, the public key ($\texttt{seed}_{\texttt{pk}}$, $\mathbf{s}$) and the private key $\texttt{seed}_{\texttt{sk}}$ generated in our optimized Alg. 5 will be the same as the public key and secret key generated in the original version of CROSS-KeyGen Alg. 1. In our just-in-time strategy for the hash function, we do not change the sequence of the hash operations; we only pass the input values one by one, just after the generation. So, our technique generates the same hash digest as the original, at a fraction of the memory cost, with a small dip in performance. Since it does not change the output for a fixed input, the signature components $\texttt{digest}_{\texttt{cmt}}$,$\texttt{digest}_{\texttt{chall}_2}$, which are the output of these HASH, will be the same as the corresponding components of the original CROSS signature. Finally, in the final speed optimization Sec . 5, we also compute the same multiplication as the original CROSS. Since the techniques used in Sec. 4 and 5 generate the same output as the original CROSS version, they will not hamper the mathematical security of CROSS.

Regarding the side channel attacks, our optimization do not introduce any timing side-channel attack. The timing side-channel attacks arise for the three main reasons: the usage of secret data-dependent conditional instructions [AKS07, GJN20, Ber24], the usage of variable-time instructions on secret dependent data [KPVV16, BBB+25b], or the usage of secret data-dependent table-based look-up tables [Koc96]. In our optimization techniques, we have refrained from using any such optimization techniques. Although the implementations of CROSS algorithms are not constant-time, due to the rejection sampling in CROSS_Keygen and tree traversal variability in TreeProof, TreeRoot, and SeedPath according to $\texttt{chall}_2$, these timing variations only leak information about the Merkle and GGM tree structure, $\texttt{chall}_2$ value, and public key; all of which can be recovered from the signature. None of the timing variations are due to the secret key or sensitive variables that can potentially help to recover the secret key. Therefore, we claim that our optimized implementations do not introduce any new timing side-channel attack vulnerabilities.

8 Discussions and Future Directions

We observe that the GGM tree is becoming more common across multiple post-quantum digital signature schemes such as LESS [BMPS20], FAEST [BBdSG+23], SDith [MBB+25], Mirath [AAB+25], etc. Like CROSS, each of the above-mentioned signature schemes used the stored GGM tree to generate a part of the signature (for CROSS, we called this part `Path`). For this reason, these schemes often have huge stack memory requirements to store the full GGM tree. For CROSS, we use GGM tree memory optimization methods to compute this `Path` without storing the entire GGM tree. Our GGM tree memory optimization methods are not limited to CROSS. We can apply the same GGM tree memory optimization methods to the schemes that follow a similar equivalent signature's part computation from the stored GGM tree and use a stack space to hold the GGM tree. LESS [BMPS20] is another second-round candidate in the NIST additional signature standardization competition that uses the stored GGM tree to compute the equivalent signature's `Path` using the similar function `GGMPath` of `SeedPath` of CROSS. Our GGM tree optimization technique can be adapted to LESS, which will potentially save $1500-5000$ bytes in LESS. Similarly, for FAEST signature scheme this GGM tree is gen-

erated using the function `vole_commit` and use similar function `bavc_open` to compute the equivalent signature's part `signature_decom_i` of CROSS. However, the optimal usage of our optimization method on LESS and FAEST needs to be explored further and left as future work.

During the course of this work, a work on implementations of CROSS algorithms targeting FPGA platforms appeared [KAB+25]. These implementations of CROSS require large memory resources (LUT/Flip-Flops/BRAM/DSP/Slice), and do not address the issue of large memory requirements. We believe that our optimizations can be applied directly to this work, resulting in a reduction in their resource usage. We left it as future work.

Most of the second-round candidates of the NIST additional digital signature standardization procedure, including CROSS, use relatively newer constructions compared to standard digital signature schemes. Many practical aspects of such designs must be properly investigated, such as efficient implementations, analysis of physical attacks (side-channel and fault-injection attacks), and countermeasures against physical attacks. These works will significantly impact the current NIST standardization procedure for selecting additional digital signature algorithms.

9 Acknowledgements

The work of Angshuman Karmakar is supported by the Research-I foundation from Infosys, the Initiation grant from IIT Kanpur, Prime Minister Early Career Research Grant from Anusandhan National Research Foundation, and the Google India research fellowship. Supriya Adhikary is supported by the Prime Minister's Research Fellowship. Chaoyun Li was partly funded by the European Unions Horizon research and innovation program under grant agreement number: 101167904 (CASTOR). This project is funded by the UK governments Horizon Europe guarantee and administered by UKRI. This work of Suparna Kundu was partially supported by Horizon 2020 ERC Advanced Grant (101020005 Belfort), Horizon Europe (101070008 ORSHIN), CyberSecurity Research Flanders with reference number VOEWICS02, BE QCI: Belgian-QCI (3E230370) (see beqci.eu), Research Council KU Leuven (C14/18/067), and Intel Corporation.

References

[AAB+25] Gora Adj, Nicolas Aragon, Stefano Barbero, Magali Bardet, Emanuele Bellini, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Andre Esser, Thibauld Feneuil, Philippe Gaborit, Romaric Neveu, Matthieu Rivain, Luis Rivera-Zamarripa, Carlo Sanna, Jean-Pierre Tillich, Javier Verbel, and Floyd Zweydinger. Mirath Signature Scheme, 2025. https://pqc-mirath.org/assets/downloads/mirath_v2.0.pdf.

[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.

[AHKS22a] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster kyber and dilithium on the cortex-m4. In Giuseppe Ateniese and Daniele Venturi, editors, Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings, volume 13269 of Lecture Notes in Computer Science, pages 853–871. Springer, 2022.

[AHKS22b] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, ACNS 22International Conference on

Applied Cryptography and Network Security, volume 13269 of LNCS, pages 853–871. Springer, Cham, June 2022.

[AKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In Masayuki Abe, editor, Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings, volume 4377 of Lecture Notes in Computer Science, pages 225–242. Springer, 2007.

[ARM10] ARM. ARM Cortex-M4 Technical Reference Manual - ARM DDI 0439B Errata 01. ARM, March 2010.

[BBB+25a] Marco Baldi, Alessandro Barenghi, Sebastian Bitzer, Patrick Karl, Felice Manganiello, Alessio Pavoni, Gerardo Pelosi, Paolo Santini, Jonas Schupp, Freeman Slaughter, Antonia Wachter-Zeh, and Violetta Weger. CROSS: Codes and Restricted Objects Signature Scheme - Specification Document, January 2025.

[BBB+25b] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales B. Paiva, Prasanna Ravi, and Goutam Tamvada. Kyberslash: Exploiting secret-dependent division timings in kyber implementations. IACR Trans. Cryptogr. Hardw. Embed. Syst., 2025(2):209–234, 2025.

[BBC+21] Marco Baldi, Massimo Battaglioni, Franco Chiaraluce, Anna-Lena Horlemann-Trautmann, Edoardo Persichetti, Paolo Santini, and Violetta Weger. A new path to code-based signatures via identification schemes with restricted errors, 2021.

[BBdSG+23] Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from vole-in-the-head. In Helena Handschuh and Anna Lysyanskaya, editors, Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V, volume 14085 of Lecture Notes in Computer Science, pages 581–615. Springer, 2023.

[BBP+24] Marco Baldi, Sebastian Bitzer, Alessio Pavoni, Paolo Santini, Antonia Wachter-Zeh, and Violetta Weger. Zero knowledge protocols and signatures from the restricted syndrome decoding problem. In Qiang Tang and Vanessa Teague, editors, Public-Key Cryptography - PKC 2024 - 27th IACR International Conference on Practice and Theory of Public-Key Cryptography, Sydney, NSW, Australia, April 15-17, 2024, Proceedings, Part II, volume 14602 of Lecture Notes in Computer Science, pages 243–274. Springer, 2024.

[BDS09] Johannes Buchmann, Erik Dahmen, and Michael Szydlo. Hash-based Digital Signature Schemes, pages 35–93. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Ber24] Daniel J. Bernstein. timing attack against SMAUG-T software, 2024 (accessed 21-December-2024). https://groups.google.com/d/msgid/kpqc-bulletin/20240511181246.1998064.qmail%40cr.yp.to.

[BFM+18] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Fly, you fool! faster frodo for the ARM cortex-m4. IACR Cryptol. ePrint Arch., page 1116, 2018.

[BHK+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS+ Signature Framework. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19, page 21292146, New York, NY, USA, 2019. Association for Computing Machinery.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of kyber on cortex-m4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje-eddine Rachidi, editors, Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings, volume 11627 of Lecture Notes in Computer Science, pages 209–228. Springer, 2019.

[BMPS20] Jean-François Biasse, Giacomo Micheli, Edoardo Persichetti, and Paolo Santini. LESS is More: Code-Based Signatures Without Syndromes. In Abderrahmane Nitaj and Amr Youssef, editors, Progress in Cryptology - AFRICACRYPT 2020, pages 45–65, Cham, 2020. Springer International Publishing.

[DKRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, AFRICACRYPT 18, volume 10831 of LNCS, pages 282–305. Springer, Cham, May 2018.

[DLL+17] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS – Dilithium: Digital Signatures from Module Lattices. Cryptology ePrint Archive, Paper 2017/633, 2017. https://eprint.iacr.org/2017/633.

[FHK+20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU, 2020. https://falcon-sign.info/falcon.pdf.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. J. ACM, 33(4):792807, August 1986.

[GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the fujisaki-okamoto transformation and its application on frodokem. In Daniele Micciancio and Thomas Ristenpart, editors, Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II, volume 12171 of Lecture Notes in Computer Science, pages 359–386. Springer, 2020.

[KAB+25] Patrick Karl, Francesco Antognazza, Alessandro Barenghi, Gerardo Pelosi, and Georg Sigl. High-performance FPGA accelerator for the post-quantum signature scheme CROSS. Cryptology ePrint Archive, Paper 2025/1161, 2025.

[KjCP16] John Kelsey, Shu jen Chang, and Ray Perlner. SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash, 2016-12-22 00:12:00 2016.

[KMRV18] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM cca-secure module lattice-based key encapsulation on ARM. IACR Trans. Cryptogr. Hardw. Embed. Syst., 2018(3):243–266, 2018.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings, volume 1109 of Lecture Notes in Computer Science, pages 104–113. Springer, 1996.

[KPVV16] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015. In Sara Foresti and Giuseppe Persiano, editors, Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings, volume 10052 of Lecture Notes in Computer Science, pages 573–582, 2016.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[KSSW22] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.

[MBB+25] Carlos Aguilar Melchor, Slim Bettaieb, Loïc Bidoux, Thibauld Feneuil, Philippe Gaborit, Nicolas Gama, Shay Gueron, James Howe, Andreas Hülsing, David Joseph, Antoine Joux, Mukul Kulkarni, Edoardo Persichetti, Tovohery H. Randrianarisoa, Matthieu Rivain, and Dongze Yue. The Syndrome Decoding in the Head (SD-in-the-Head) Signature Scheme, 2025. https://sdith.org/docs/sdith-v2.0.pdf.

[Mer87] Ralph C Merkle. A digital signature based on a conventional encryption function. In Advances in Cryptology — CRYPTO '87 Proceedings, pages 369–378. Springer, 1987.

[MKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in toom-cook multiplication: an application to module-lattice based cryptography. IACR Trans. Cryptogr. Hardw. Embed. Syst., 2020(2):222–244, 2020.

[NIS15] NIST. Sha-3 standard: Permutation-based hash and extendable-output functions, 08 2015.

[NIS23] NIST. NIST Announces Additional Digital Signature Candidates for the PQC Standardization Process. Online. Accessed 10th November, 2024, 2023.

[NIS24a] NIST. FIPS 203 Module-Lattice-Based Key-Encapsulation Mechanism Standard. Online. Accessed 10th November, 2024, 2024.

[NIS24b] NIST. FIPS 204 Module-Lattice-Based Digital Signature Standard. Online. Accessed 10th November, 2024, 2024.

[NIS24c] NIST. FIPS 205 Stateless Hash-Based Digital Signature Standard. Online. Accessed 10th November, 2024, 2024.

[NIS24d] NIST. PQC digital signature second round announcement. Online. Accessed 10th November, 2024, 2024.

[Qua25] Quantum-Resistant Cryptography Research Group. Korean post-quantum cryptography competition, 2025. Accessed: 2025-07-16.

[RA20] Lum Ramabaja and Arber Avdullahu. Compact merkle multiproofs. arXiv preprint arXiv:2002.07648, 2020.