Optimizing the Post Quantum Signature Scheme CROSS for Resource Constrained Devices

Jonas Schupp[1], Marco Gianvecchio[2], Alessandro Barenghi[2], Patrick Karl[1], Gerardo Pelosi[2] and Georg Sigl[1,3]

[1] TUM School of Computation, Information and Technology, Technical University of Munich, Munich, Germany, {jonas.schupp,patrick.karl,sigl}@tum.de
[2] Politecnico di Milano, Department of Electronics, Information and Bioengineering (DEIB), Milan, Italy, {marco.gianvecchio,alessandro.barenghi,gerardo.pelosi}@polimi.it
[3] Fraunhofer Institute for Applied and Integrated Security, Garching, Germany, georg.sigl@aisec.fraunhofer.de

Abstract. Post-quantum cryptosystems are currently attracting significant research efforts due to the continuous improvements in quantum computing technologies, which led the National Institute of Standards and Technology (NIST) to open standardization competitions to foster proposals and public scrutiny of cryptosystems and digital signatures. Whilst NIST has chosen, after four selection rounds, three digital signature algorithms, it also has opened a new selection process as the chosen candidates were either relying only on lattice-based computationally hard problems, or had unsatisfactory performance figures. In this work, we propose two optimized implementations of the Codes and Restricted Objects Signature Scheme (CROSS) targeting the Cortex-M4 platform. One implementation targets the minimal possible stack size while the other trades some memory space for performance optimization using DSP instructions for some performance critical arithmetic operations. We show that all parameter sets fit within at maximum 24 kB of stack which corresponds to a reduction by a factor of 15 to 45 with respect to the reference implementation. The memory footprint of our implementation, taking the size of the signature also into account, is less than 128 kB. We additionally outline different stack reduction options which allow for a fine grained trade-off between memory footprint and performance of the algorithm. Notably, we also show that our memory optimizations alone have no significant performance impact on the signature verification of CROSS while we even achieve a speed-up factor of up to 1.7 when taking the stack and speed optimizations into account.

Keywords: Post-quantum Signature · Code-based signature · Software optimized implementation

1 Introduction

While the first Post-Quantum Cryptographic (PQC) standards based on NIST's competition [NIS25b] are now consolidated, there is still interest in more diversity in the underlying mathematical problems for signature schemes as all currently chosen schemes are either based on structured lattices such as ML-DSA [Nat24b] (former Crystals-Dilithium) and FN-DSA [Nat24a] (former Falcon) or on cryptographic hashes such as SLH-DSA [Nat24c] (former SPHINCS+). This interest was concretized in NIST's *Call for Additional Signature Schemes*, which is currently in its second round, with 14 schemes still being under evaluation [NIS25a]. These additional schemes rely for their security on the computational hardness of problems coming from different domains such as error correcting codes or

multivariate quadratic equations over finite fields. Two of these schemes were classified as code-based by NIST and one is the Codes and Restricted Objects Signature Scheme (CROSS) and the other is LESS [BBB+25b]. CROSS offers compact public key sizes and signature sizes in the range of the ones of SLH-DSA, while being computationally less expensive than SLH-DSA itself: CROSS is therefore a promising candidate in the competition for general purpose applications.

CROSS is built employing a Fiat-Shamir transform on a set of repetitions of a Zero Knowledge IDentification (ZKID) protocol. This approach provides strong security guarantees, and allows to employ trapdoor-less instances of hard problems, but it is characterized by a significant memory footprint. While CROSS has shown good computational performance on non-constrained devices, such as x86_64 CPUs in its AVX2 optimized implementation [BGP26], its usability on resource constrained devices is less explored, despite being of significant practical interest for a general purpose signature scheme. We therefore explore in this work an optimized implementation of CROSS targeting memory constrained devices. In doing so, we consider two figures of merit, namely *stack size*, defined as the amount of runtime memory required to execute the implementation and *memory footprint*, which, in addition to stack size, takes into account the size of the signature and the key material.

Related Work. In [HMK+24][1], Hart et al. propose two runtime memory optimized variants of CROSS which result in stack sizes between 14.8 kB and 523.9 kB. We discuss their optimizations in Section 3 and provide a comparison with their results in Section 5. Version 1.2 of the CROSS specification also reports some figures on the memory consumption of the first round submission of CROSS but does not go into details on techniques employed. With respect to other post-quantum signature algorithms, there is a work on Crystals-Dilithium [BRS22] by Bos et al. reporting stack sizes of less than 8 kB for most parameter sets. Regarding the on-ramp signature schemes, Beullens et al. investigated the optimization of Mayo in [BCC+23], while Bettaieb et al. optimized the digital signature scheme based on the PERmuted Kernel problem, PERK, and other multi-party computation (MPC) based schemes in [BBB+24]. We provide a comprehensive comparison of CROSS with other on-ramp signature schemes in our experimental validation.

Contributions. Our contributions can be summarized as follows:

- *A memory footprint optimized CROSS.* Our proposed optimized implementation reduces the stack size to between 5.1 kB and 23.9 kB, depending on the security level, while trading off only a moderate penalty on the runtime of the scheme. The computational overhead to achieve such a footprint reduction is about 2.18× with respect to a non-memory optimized implementation, a figure we validated comparing with the reference implementation on all parameter sets for which it can be run on the same platform. For signature verification, the memory optimized implementation achieves an improvement in terms of clock cycles of up to 19%, besides the improvement on memory footprint. This places CROSS on par with other post quantum signature schemes in terms of implementability on constrained devices as shown in Section 5.4.

- *An efficient implementation of the arithmetic of CROSS.* On top of the memory footprint optimizations, we describe optimized scalar arithmetics for the finite fields employed in CROSS, fully exploiting the design choices on the module values to speed up the computations. We complement this description reporting how to exploit the fused multiply-and-accumulate operations in Cortex-M4 in CROSS, obtaining a

[1] Concurrent to this work, [HMK+24] received numerous updates throughout the last year, heavily changing the employed optimizations and results. Throughout this work, we compare ourselves against their latest version available which is of October, 23rd 2025 and available here: https://eprint.iacr.org/archive/2024/1929/20251023:081431

speed-up between 12.1% and 40.7% for signature generation at the cost of a slightly increased stack size with respect to the stack optimized implementation.

The implementations of this work are publicly available at https://github.com/mupq/pqm4.[2]

Organization. The remainder of this work is structured as follows: Within Section 2 we provide a high-level overview of the CROSS signature scheme outlining its key operations relevant for the subsequent optimizations. This is followed by an introduction to our memory footprint optimizations in Section 3 and our speed optimizations in Section 4. The experimental results of these optimizations are discussed in Section 5 while Section 6 concludes this work.

2 The Codes and Restricted Objects Signature Scheme

Our notation throughout this paper follows the notation introduced in the CROSS specification [BBB+25a]. The descriptions of the Algorithms 1, 2 and 3 are furthermore taken directly from the specification.

CROSS is based on an interactive zero-knowledge protocol turned non-interactive using the Fiat-Shamir transform. Its hardness is based on two variants of the syndrome decoding problem, namely the *Restricted Syndrome Decoding Problem (RSDP)* and the *Restricted Syndrome Decoding Problem with Subgroup (RSDP(G))*. CROSS has parameter sets based on either of these two variants. As usual for a zero-knowledge protocol, the underlying identification scheme has a certain cheating probability requiring the protocol to be executed several times to reduce the overall cheating probability below an acceptable threshold. In the following, we briefly outline the Key Generation, Signature Generation and Verification procedures of the scheme. Throughout all algorithms, steps exclusive to the RSDP variants are shown in teal, while steps specific for RSDP(G) are shown in orange.

CROSS.KeyGen. During key generation, as shown in Algorithm 1, CROSS expands a random seed (line 1) into a public and private seed using a CSPRNG (line 2). These seeds are then expanded into a secret error vector and a public parity check matrix using rejection sampling (line 3). The public syndrome is the result of the multiplication of the secret error vector with the public parity check matrix (line 7). The secret key only consists of this private seed while the public key contains the public seed as well as the public syndrome.

CROSS.Sign. As shown in Algorithm 2, the signature generation starts by expanding the secret key in the same manner as outlined in the Key Generation above (line 1). In the next step (line 2-3), the seed for each of the t rounds of the protocol is expanded using a tree structure called Seed Tree allowing to publish a subset of the tree in case multiple leaves in a region of the tree are to be published. Based on these seeds the commitments are computed for all rounds (line 4-13). This is followed by the computation of the first challenge (line 14-16) based on the commitments, the first response (line 17-20) based on the commitments and the first challenge, the second challenge (line 21-22) based on the first response and the commitments and finally the second response (line 23-27).

CROSS.Verify. For verification, as shown in Algorithm 3, we first expand the public key (line 1-3), followed by the extraction of the first and second challenge from the signature (line 4-7). Based on the second challenge the missing values from the identification protocol executed in signature generation are computed (line 9-21). This includes retrieving the corresponding seed from the published subset and subsequent computation of the

[2]The implementations are either already merged or can be found as open pull request.

Algorithm 1: KeyGen()

Input: None
Output: $\mathtt{sk} : \mathtt{Seed}_{\mathtt{sk}}$: secret key seed; $\mathtt{pk} : (\mathtt{Seed}_{\mathtt{pk}}, \mathbf{s})$ public key;
Data: λ: security parameter; $g \in \mathbb{F}_p^*$: generator of $\mathbb{E}$;

// Sampling seeds
1 $\mathtt{Seed}_{\mathtt{sk}} \xleftarrow{\$} \{0,1\}^{2\lambda}$
2 $(\mathtt{Seed}_{\mathtt{e}}, \mathtt{Seed}_{\mathtt{pk}}) \leftarrow \mathsf{CSPRNG}_{\{0,1\}^{2\lambda} \times \{0,1\}^{2\lambda}}(\mathtt{Seed}_{\mathtt{sk}} \mid 3t+1)$

// Sampling random matrices $\mathbf{H}$ and $\overline{\mathbf{M}}$
3 $(\overline{\mathbf{W}}, \mathbf{V}) \leftarrow \mathsf{CSPRNG}_{\mathbb{F}_z^{m \times (n-m)} \times \mathbb{F}_p^{(n-k) \times k}}(\mathtt{Seed}_{\mathtt{pk}} \mid 3t+2)$ | $\mathbf{V} \leftarrow \mathsf{CSPRNG}_{\mathbb{F}_p^{(n-k) \times k}}(\mathtt{Seed}_{\mathtt{pk}} \mid 3t+2)$
4 $\mathbf{H} \leftarrow [\mathbf{V} \mid \mathrm{Id}_{n-k}]$

// Computing $\mathbf{e}$
$\overline{\mathbf{M}} \leftarrow [\overline{\mathbf{W}} \mid \mathrm{Id}_m]$
5 $\overline{\mathbf{e}}_G \leftarrow \mathsf{CSPRNG}_{\mathbb{F}_z^m}(\mathtt{Seed}_{\mathtt{e}} \mid 3t+3)$ | $\overline{\mathbf{e}} \leftarrow \mathsf{CSPRNG}_{\mathbb{F}_z^n}(\mathtt{Seed}_{\mathtt{e}} \mid 3t+3)$
$\overline{\mathbf{e}} \leftarrow \overline{\mathbf{e}}_G \overline{\mathbf{M}}$
for j **from** 1 **to** n **do**
6 $\quad \mathbf{e}_j \leftarrow g^{\overline{\mathbf{e}}_j}$

// Computing the syndrome s
7 $\mathbf{s} \leftarrow \mathbf{e}\mathbf{H}^\top$

// Return secret key sk and public key pk
8 $\mathtt{sk} \leftarrow \mathtt{Seed}_{\mathtt{sk}}$
9 $\mathtt{pk} \leftarrow (\mathtt{Seed}_{\mathtt{pk}}, \mathbf{s})$
10 **return** $(\mathtt{sk}, \mathtt{pk})$;

corresponding values in the first response or the usage of some published intermediate values to recompute the two digests which need to match the values in the signature for a successful verification (line 22-28).

CROSS Parameter Choices. CROSS uses the same small fixed primes throughout all security levels, namely $p = 127$ and $z = 7$ for RSDP and $p = 509$, $z = 127$ for RSDP(G). Different security levels are obtained by changing the code parameters of the underlying random linear code. Furthermore, different design trade-offs are proposed, targeting either a fast implementation, small signatures or a trade-off between these two.

3 Stack Optimizations

Previous work already presented memory optimizations for CROSS, reducing the memory consumption by up to 92% and 85% for signature generation and verification, respectively, as shown in Table 4 of [HMK+24]. While this work poses a first step towards applicability on embedded devices, the memory consumption can be further reduced. In the following, we will briefly revisit the concepts of [HMK+24] before we explain our performance and memory optimizations.

3.1 Optimizations of [HMK+24].

This work essentially focuses on stack optimization to enable CROSS on embedded devices. The first observation made by the authors was that the computed vectors for the commitment $\mathtt{cmt}_1[i]$ and response $\mathbf{y}[i]$ for each round i can be processed on-the-fly by using an additional Keccak state. Instead of computing all t vectors of these objects at once, the authors suggested to compute only as many vectors as required to process one block of the Keccak permutation. Another observation is that the vectors $\mathbf{y}[i]$ and $\mathbf{u}'[i]$ can share a

Algorithm 2: Sign($\mathtt{sk}, \mathtt{Msg}$)

Input: $\mathtt{sk}$: secret key $\mathtt{Seed}_{\mathtt{sk}} \in \{0,1\}^{2\lambda}$; $\mathtt{Msg} \in \{0,1\}^*$: message;
Output: $\mathtt{Sgn}$: signature;
Data: λ: security parameter; c: constant defined as $c = 2t - 1$; $g \in \mathbb{F}_p^*$: generator of $\mathbb{E}$; t: number of rounds; w: weight of the second challenge;

// Expanding secret key
$\overline{\mathbf{e}}, \overline{\mathbf{e}}_G, \mathbf{H}, \overline{\mathbf{M}} \leftarrow \mathsf{ExpandSK}(\mathtt{Seed}_{\mathtt{sk}})$ | $\overline{\mathbf{e}}, \mathbf{H} \leftarrow \mathsf{ExpandSK}(\mathtt{Seed}_{\mathtt{sk}})$
// Computing the commitments
$\mathtt{Seed} \xleftarrow{\$} \{0,1\}^{\lambda}, \quad \mathtt{Salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$
$(\mathtt{Seed}[1], \ldots, \mathtt{Seed}[t]) \leftarrow \mathsf{SeedLeaves}(\mathtt{Seed}, \mathtt{Salt})$
// Compute $\mathbf{v}[i]$ such that $\mathbf{v}[i] \star \mathbf{e}'[i] = \mathbf{e}$
for i **from** 1 **to** t **do**
$\overline{\mathbf{e}}'_G[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG}_{-\mathbb{F}_z^m \times \mathbb{F}_p^n}(\mathtt{Seed}[i] \mid \mathtt{Salt} \mid i + c)$ | $\overline{\mathbf{e}}'[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG}_{-\mathbb{F}_z^n \times \mathbb{F}_p^n}(\mathtt{Seed}[i] \mid \mathtt{Salt} \mid i + c)$
$\overline{\mathbf{v}}_G[i] \leftarrow \overline{\mathbf{e}}_G - \overline{\mathbf{e}}'_G[i]$
$\overline{\mathbf{e}}'[i] \leftarrow \overline{\mathbf{e}}'_G[i]\overline{\mathbf{M}}$
$\overline{\mathbf{v}}[i] \leftarrow \overline{\mathbf{e}} - \overline{\mathbf{e}}'[i]$
for j **from** 1 **to** n **do**
$\mathbf{v}[i]_j \leftarrow g^{\overline{\mathbf{v}}[i]_j}$
$\mathbf{u}[i] \leftarrow \mathbf{v}[i] \star \mathbf{u}'[i]$
$\mathbf{s}'[i] \leftarrow \mathbf{u}[i]\mathbf{H}^\top$
$\mathtt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}_G[i] \mid \mathtt{Salt} \mid i + c)$ | $\mathtt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}[i] \mid \mathtt{Salt} \mid i + c)$
$\mathtt{cmt}_1[i] \leftarrow \mathsf{Hash}(\mathtt{Seed}[i] \mid \mathtt{Salt} \mid i + c)$
$\mathtt{digest}_{\mathtt{cmt}_0} \leftarrow \mathsf{TreeRoot}(\mathtt{cmt}_0[1], \cdots, \mathtt{cmt}_0[t])$
$\mathtt{digest}_{\mathtt{cmt}_1} \leftarrow \mathsf{Hash}(\mathtt{cmt}_1[1] \mid \cdots \mid \mathtt{cmt}_1[t])$
$\mathtt{digest}_{\mathtt{cmt}} \leftarrow \mathsf{Hash}(\mathtt{digest}_{\mathtt{cmt}_0} \mid \mathtt{digest}_{\mathtt{cmt}_1})$
// Computing first challenge
$\mathtt{digest}_{\mathtt{Msg}} \leftarrow \mathsf{Hash}(\mathtt{Msg})$
$\mathtt{digest}_{\mathtt{chall}_1} \leftarrow \mathsf{Hash}(\mathtt{digest}_{\mathtt{Msg}} \mid \mathtt{digest}_{\mathtt{cmt}} \mid \mathtt{Salt})$
$\mathtt{chall}_1 \leftarrow \mathsf{CSPRNG}_{-(\mathbb{F}_p^*)^t}\left(\mathtt{digest}_{\mathtt{chall}_1} \mid t + c\right)$
// Computing first response
for i **from** 1 **to** t **do**
for j **from** 1 **to** n **do**
$\mathbf{e}'[i]_j \leftarrow g^{\overline{\mathbf{e}}'[i]_j}$
$\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \mathtt{chall}_1[i]\mathbf{e}'[i]$
// Computing second challenge
$\mathtt{digest}_{\mathtt{chall}_2} \leftarrow \mathsf{Hash}(\mathbf{y}[1] \mid \cdots \mid \mathbf{y}[t] \mid \mathtt{digest}_{\mathtt{chall}_1})$
$\mathtt{chall}_2 \leftarrow \mathsf{CSPRNG}_{-\mathcal{B}_{(t,w)}}\left(\mathtt{digest}_{\mathtt{chall}_2} \mid t + c + 1\right)$
// Computing second response
$\mathtt{Proof} \leftarrow \mathsf{TreeProof}(\mathtt{cmt}_0[1], \cdots, \mathtt{cmt}_0[t], \mathtt{chall}_2)$
$\mathtt{Path} \leftarrow \mathsf{SeedPath}(\mathtt{Seed}, \mathtt{Salt}, \mathtt{chall}_2)$
for i **from** 1 **to** t **do**
if $\mathtt{chall}_2[i] = 0$ **then**
$\mathtt{resp}[i]_0 \leftarrow (\mathbf{y}[i], \overline{\mathbf{v}}_G[i])$ | $\mathtt{resp}[i]_0 \leftarrow (\mathbf{y}[i], \overline{\mathbf{v}}[i])$
$\mathtt{resp}[i]_1 \leftarrow \mathtt{cmt}_1[i]$
// Assembling signature
$\mathtt{Sgn} \leftarrow (\mathtt{Salt}, \mathtt{digest}_{\mathtt{cmt}}, \mathtt{digest}_{\mathtt{chall}_2}, \mathtt{Path}, \mathtt{Proof}, \mathtt{resp})$
return $\mathtt{Sgn}$

single memory location, i.e. $\mathbf{y}[i]$ is computed in-place from $\mathbf{u}'[i]$. Given that the reference implementation computes t of these vectors using unique memory locations, halving the cost here denotes significant savings. The third optimization is that the transformation $\overline{\mathbf{v}}[i]$ (formerly σ) does not need to be stored for all t rounds but can be re-computed when assembling the second response. Finally, the last optimization is storing the values of $\mathtt{cmt}_0[i]$ directly on the Merkle tree instead of using a separate array in an intermediate

Algorithm 3: Verify($\mathtt{pk}, \mathtt{Msg}, \mathtt{Sgn}$)

Input: $\mathtt{pk}$: $(\mathtt{Seed}_{\mathtt{pk}}, \mathbf{s})$ public key; $\mathtt{Msg} \in \{0,1\}^*$: message; $\mathtt{Sgn}$: $(\mathtt{Salt}, \mathtt{digest}_{\mathtt{cmt}}, \mathtt{digest}_{\mathtt{chall}_2}, \mathtt{Path}, \mathtt{Proof}, \mathtt{resp})$ signature;
Output: {True, False};
Data: λ: security parameter; g: generator of $\mathbb{E}$; t: number of rounds; w: weight of second challenge; c: constant defined as $2t-1$;

// Recovering public key

$(\overline{\mathbf{W}}, \mathbf{V}) \leftarrow \mathsf{CSPRNG}_{-\mathbb{F}_z^{m\times(n-m)}\times\mathbb{F}_p^{(n-k)\times k}}(\mathtt{Seed}_{\mathtt{pk}} \mid 3t+2)$ | $\mathbf{V} \leftarrow \mathsf{CSPRNG}_{-\mathbb{F}_p^{(n-k)\times k}}(\mathtt{Seed}_{\mathtt{pk}} \mid 3t+2)$

$\mathbf{H} \leftarrow [\mathbf{V} \mid \mathrm{Id}_{n-k}]$

$\overline{\mathbf{M}} \leftarrow [\overline{\mathbf{W}} \mid \mathrm{Id}_m]$

// Computing challenges

$\mathtt{digest}_{\mathtt{Msg}} \leftarrow \mathsf{Hash}(\mathtt{Msg})$

$\mathtt{digest}_{\mathtt{chall}_1} \leftarrow \mathsf{Hash}(\mathtt{digest}_{\mathtt{Msg}} \mid \mathtt{digest}_{\mathtt{cmt}} \mid \mathtt{Salt})$

$\mathtt{chall}_1 \leftarrow \mathsf{CSPRNG}_{-(\mathbb{F}_p^*)^t}\left(\mathtt{digest}_{\mathtt{chall}_1} \mid t+c\right)$

$\mathtt{chall}_2 \leftarrow \mathsf{CSPRNG}_{-\mathcal{B}_{(t,w)}}\left(\mathtt{digest}_{\mathtt{chall}_2} \mid t+c+1\right)$

// Computing commitments

$(\mathtt{Seed}[i])_{i:\mathtt{chall}_2[i]=1} \leftarrow \mathsf{RebuildLeaves}(\mathtt{Path}, \mathtt{chall}_2, \mathtt{Salt})$

for i **from** 1 **to** t **do**

  if $\mathtt{chall}_2[i] = 1$ **then**

    $\mathtt{cmt}_1[i] \leftarrow \mathsf{Hash}(\mathtt{Seed}[i] \mid \mathtt{Salt} \mid i+c)$

    $\overline{\mathbf{e}}'_G[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG}_{-\mathbb{F}_z^m\times\mathbb{F}_p^n}(\mathtt{Seed}[i] \mid \mathtt{Salt} \mid i+c)$ | $\overline{\mathbf{e}}'[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG}_{-\mathbb{F}_z^n\times\mathbb{F}_p^n}(\mathtt{Seed}[i] \mid \mathtt{Salt} \mid i+c)$

    $\overline{\mathbf{e}}'[i] \leftarrow \overline{\mathbf{e}}'_G[i]\overline{\mathbf{M}}$

    for j **from** 1 **to** n **do**

      $\mathbf{e}'[i]_j \leftarrow g^{\overline{\mathbf{e}}[i]_j}$

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

  if $\mathtt{chall}_2[i] = 0$ **then**

    $\mathtt{cmt}_1[i] \leftarrow \mathtt{resp}[i]_1$

    $(\mathbf{y}[i], \overline{\mathbf{v}}_G[i]) \leftarrow \mathtt{resp}[i]_0$ | $(\mathbf{y}[i], \overline{\mathbf{v}}[i]) \leftarrow \mathtt{resp}[i]_0$

    Check if $\overline{\mathbf{v}}_G[i] \in \mathbb{F}_z^m$ | Check if $\overline{\mathbf{v}}[i] \in \mathbb{F}_z^n$

    $\overline{\mathbf{v}}[i] \leftarrow \overline{\mathbf{v}}_G[i]\overline{\mathbf{M}}$

    for j **from** 1 **to** n **do**

      $\mathbf{v}[i]_j \leftarrow g^{\overline{\mathbf{v}}[i]_j}$

    $\mathbf{y}'[i] \leftarrow \mathbf{v}[\mathbf{i}] \star \mathbf{y}[i]$

    $\mathbf{s}'[i] \leftarrow \mathbf{y}'[i]\mathbf{H}^\top - \mathtt{chall}_1[i]\mathbf{s}$

    $\mathtt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}_G[i] \mid \mathtt{Salt} \mid i+c)$ | $\mathtt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}[i] \mid \mathtt{Salt} \mid i+c)$

// Checking digests

$\mathtt{digest}_{\mathtt{cmt}_0} \leftarrow \mathsf{RecomputeRoot}(\mathtt{cmt}_0, \mathsf{Proof}, \mathtt{chall}_2)$

$\mathtt{digest}_{\mathtt{cmt}_1} \leftarrow \mathsf{Hash}(\mathtt{cmt}_1[1] \mid \cdots \mid \mathtt{cmt}_1[t])$

$\mathtt{digest}'_{\mathtt{cmt}} \leftarrow \mathsf{Hash}(\mathtt{digest}_{\mathtt{cmt}_0} \mid \mathtt{digest}_{\mathtt{cmt}_1})$

$\mathtt{digest}'_{\mathtt{chall}_2} \leftarrow \mathsf{Hash}(\mathbf{y}[1] \mid \cdots \mid \mathbf{y}[t] \mid \mathtt{digest}_{\mathtt{chall}_1})$

if $\mathtt{digest}_{\mathtt{cmt}} = \mathtt{digest}'_{\mathtt{cmt}}$ **and** $\mathtt{digest}_{\mathtt{chall}_2} = \mathtt{digest}'_{\mathtt{chall}_2}$ **then**

  return True

return False

step. Concurrently to this work, the authors of [HMK+24] updated their work several times such that their latest version now also includes a memory optimized strategy for the tree computations in CROSS though they aim at a different target storing a subset of the tree which avoids recomputations but still results in a larger memory footprint. We compare our results with theirs in Section 5.3.

3.2 Optimizations employed in this work

From a memory footprint viewpoint, the algorithm is dominated by the large number of intermediate results from the commitment loop required later in the algorithm as well as by the Seed and Merkle tree used to expand the seeds for each round or to hash the results to a single digest. All elements are furthermore obtained using rejection sampling on pre-filled random buffers. To this end, Key Generation, Signature Generation and Signature Verification of CROSS all rely on the same functions, though recomputations are only necessary during Signing and can be avoided in Key Generation and Signature Verification. The optimizations outlined above are therefore equally applicable to all three operations with their different impact on the runtime being discussed in Section 5. In general, the optimizations employed throughout this work can therefore be categorized into:

- **Memory-location based optimizations:** These include optimizing structures from the reference implementation which were mainly there for a better understanding of the implementation, such as storing the same value at two different locations, not reusing the storage space of intermediate variables to avoid confusion and keeping sampling buffers separated from the sampling results. The optimizations outlined here have no impact on the overall program flow of the scheme. This includes sharing the memory location of $\mathbf{y}[i]$ and $\mathbf{u}'[i]$ as outlined in [HMK+24] as well as storing the values of $\mathtt{cmt}_0[i]$ directly on the Merkle tree.
- **Incremental computations, possibly requiring recomputations**: Here, we reduce the storage requirements by discarding some intermediate results which are e.g. used to compute the initial commitments. As these variables are possibly also needed later in the scheme, this often requires to recompute them on demand which changes the program flow as operations originally only executed once are now executed each time a certain value is required. A similar idea is outlined in [BBB+24] with the notable difference that the MPCitH schemes outlined in this work typically require the recomputation of one commitment per round while we need to recompute at least some intermediate variables for all commitments throughout the protocol. This includes processing the vectors for the commitment $\mathtt{cmt}_1[i]$ and the response $\mathbf{y}[i]$ on-the-fly as well as recomputing the transformation $\overline{\mathbf{v}}[i]$ when required as shown in [HMK+24].

3.3 Memory-location based optimizations

In addition to sharing the memory location of $\mathbf{y}[i]$ and $\mathbf{u}'[i]$ and storing the values of $\mathtt{cmt}_0[i]$ directly on the Merkle tree as shown in [HMK+24], we employ the following optimizations:

In-place rejection sampling Throughout all three functions of CROSS pseudo random vectors and elements are sampled by expanding a seed to a sufficiently large random buffer which is subsequently used for rejection sampling. Especially for sampling the public matrices $\mathbf{V}$ and $\overline{\mathbf{W}}$ as well as for the challenges, these buffers exceed 10 kB depending on the parameter set. While this is a smaller part of the overall memory footprint for the non optimized implementation, it can be optimized at no cost simply by either sampling vectors and matrices in-place, i.e. by storing part of the required randomness in the same memory location as the resulting elements or by reusing the storage space of other elements which are either recomputed anyways or not yet in use. It is necessary to bound the amount of randomness stored in-place by the minimum amount of randomness required for the respective number of elements to avoid accidentally overwriting not yet used randomness in the buffer. This allows to avoid any additional memory allocation for the random buffers throughout the entire implementation. This approach is different from the one used

in [HMK+24] as we still generate the entire random buffer required at once thus saving the cost of subsequent calls to an already existing state of used CSPRNG function.

Direct retrieval of seeds from the seed tree In the reference implementation, the seed for each round i is first stored into a flattened array allowing for direct access to the seed for each round. Instead it is also possible to compute the location of the seed in the seed tree on-the-fly and retrieve the seed directly from there. This saves the storage space for t seeds (2.3 kB to 53 kB depending on the parameter set) in case one wants to avoid the overhead incurred by incrementally computing the Seed tree multiple times.

3.4 Incremental computations, possibly requiring recomputations

Incremental hashing. As noted in [HMK+24], the vectors $\texttt{cmt}_1[i]$ as well as $\mathbf{y}[i]$ can be processed on-the-fly using incremental hashing, i.e. directly absorbing the vectors of one of the t round iterations before computing the next one. This essentially reduces the memory consumption down to 1 instead of t variables at the low cost of 200B for the Keccak state thus saving between 20.8 kB and 524 kB. However, a subset of all $\texttt{cmt}_1[i]$ and $\mathbf{y}[i]$ selected by the challenge bits $\texttt{chall}_2[i]$ is required for signature composition and hence, we recompute only this particular subset after generating $\texttt{chall}_2$.

Vertical tree computations. In the balanced and small versions, the Seed- and Merkle trees are (truncated) binary trees consisting of $2t - 1$ nodes each, where each node has a size of λ or 2λ bits, respectively. This quickly accumulates to several 10s of kilobytes just for storing the tree structures. Therefore, we decided to compute both trees in a vertical manner and only compute the parts that are required by a corresponding iteration of the ID protocol. By doing so we make sure not to process the same node multiple times:

When computing the seed tree depth-first, we always store both children in each level of the tree to not being forced to expand an inner node separately for each child. Thus, we reduce the required memory from $2t - 1$ nodes to $2\log(t)$ nodes plus some pointer information requiring $2\log(t)$ bytes. Unfortunately, we need to re-compute the seed tree three times during signature generation: once for the commitment computations, once for computing the response $\mathbf{y}[i]$ as we need the round seeds to re-sample the vectors $\bar{\mathbf{e}}'[i]$ and $\mathbf{u}'[i]$, and a final time to compute the `path` variable in the second response phase. During signature verification, however, this recomputation is not necessary and thus, does not penalize performance. A similar idea is employed in [BBB+24] though in there, it seems that the entire path is recalculated for each seed. We avoid this by always keeping both children for a path allowing us to compute each node without recomputations.

The Merkle tree is computed with the same philosophy, i.e. bottom-up storing only a single node per tree level and, whenever the second node for each level is available, both are directly processed to compute the ancestor node. This essentially reduces the memory cost down to $\log(t)$ nodes plus $2\log(t)$ bytes of helper data. Again, the Merkle tree must be re-computed to extract the `proof` variable in the second response phase during signature generation. For verification, this vertical computation is again not penalizing performance.

In difference to the concurrent work of [HMK+24], we do not keep $\texttt{cmt}_0$ around and exploit its memory space to incrementally compute the tree but only keep the above mentioned single node per layer and recompute the entire tree including the leaves containing $\texttt{cmt}_0$ on-the-fly when necessary.

In the fast version, we apply a similar strategy where we expand the four nodes of the intermediate layer for the seed tree first and then only keep one initialized Keccak state from which we squeeze one node per round incrementally. For the Merkle tree, we apply the inverse approach absorbing one $\texttt{cmt}_0[i]$ per round incrementally into the four

intermediate states which are then hashed to the root node. Depending on the parameter set, this saves between 7 kB and 638 kB of stack.

Recomputing intermediates When executing the t iterations of the commitment loop to compute $\texttt{cmt}_0[i]$ and $\texttt{cmt}_1[i]$, the vectors $\overline{\mathbf{e}}'[i]$ and $\mathbf{u}'[i]$ are randomly sampled and $\overline{\mathbf{v}}[i]$ is computed for each single iteration. In total, this accumulates to $3t$ vectors that need to be stored, since they are also required to compute the first response $\mathbf{y}[i]$ as well as to compute the second response. Given the large size of these accumulated vectors, we decided to only allocate memory for one of the t rounds of the commitment computation and re-use this memory location for all rounds when computing the commitment. When computing the responses $\mathbf{y}[i]$, the vectors $\overline{\mathbf{e}}'[i]$ and $\mathbf{u}'[i]$ are generated again from the corresponding seed. Finally, when assembling the signature, all the variables required to compute $\texttt{cmt}_1[i]$ as well as $\mathbf{y}[i]$ and $\overline{\mathbf{v}}[i]$ (resp. $\overline{\mathbf{v}}_G[i]$) are re-generated. Note, that this re-computation is only required for the cases where $\texttt{chall}_2[i]$ equals 1. Out of all memory optimization steps, this poses the largest savings but also the biggest performance penalty due to the excessive re-computation. The concurrent work of [HMK+24] also proposes to recompute $\overline{\mathbf{e}}'[i]$, $\mathbf{u}'[i]$ and $\overline{\mathbf{v}}[i]$. In summary, recomputing these variables saves between 32 kB and 626 kB of stack.

3.5 Possible further optimizations

In addition to the optimizations we explored in this work and outlined above, there are a few further options which we describe below.

Streaming the Signature After employing the optimizations mentioned above, the largest contributor to the total memory footprint of the implementation remaining is the size of the signature itself. Apart from specific use-cases where a signature is e.g. permanently stored in flash memory, we expect almost every application to either verify signatures received or to generate signatures which are then sent out. In both cases, one either needs to store the signature temporarily in the device's SRAM or needs to stream it in or out and compute it on-the-fly. Streaming the signature out comes with a substantial penalty in clock cycles as one needs to recompute the same values even more often. This is necessary as we can currently compute the nodes in the Merkle and Seed-Tree that are part of the signature as well as the corresponding elements of response on a per-round basis. Given that one wants to stream the signature this would have to be done sequentially to assemble the respective elements before sending them. Alternatively, one has to send the elements of the parts of a signature in an unordered manner and leave the task of packing them into a signature to a host device receiving them. As solutions here are application specific we leave this out of scope and just want to emphasize that while streaming is in principle possible, this has a significant impact on the implementation's performance and is not directly applicable without significant modifications.

On-the-fly-sampling of the public matrices As outlined in [HMK+24] it is possible to sample the public matrices on-the-fly without keeping them completely in memory. But, as the matrices are required t times per signature generation and t respectively $t - w$ times for signature verification, without taking recomputations into account which results in the matrices being used multiple times per round, we consider this an inefficient trade-off in comparison to the already outlined options. It should be noted though, that these matrices are by far the largest contributor to the stack of the optimized implementations though their contribution to the stack of the reference is limited in comparison to the optimizations outlined above. More specifically, for RSDP(G)-1, the parameter sets with the smallest stack between 5.1 kB and 7.7 kB, the matrices occupy 2.118 kB. For RSDP-5,

the parameter sets with the largest stack between 20.3 kB and 23.9 kB, the matrix occupies 15.15 kB. On the other hand, sampling the matrix in RSDP-5 takes roughly 1.5 million clock cycles, thus recomputing the matrix t times would for e.g. RSDP-5-fast increase the runtime for the stack and speed optimized variant from 124 million clock cycles to approximately 600 million clock cycles. Still, if speed is of no concern, and one needs to get as small as possible, sampling the matrix on-the-fly is feasible.

Bit-Packed representation of all used vectors Finally, it is also possible to store all vectors and matrices as bit-packed arrays. The savings from this optimization vary, depending on how close the number of bits required for an element is to the number of bits for the datatype used. For the RSDP parameter sets, the primes used are $p = 127$ and $z = 7$, thus resulting in one bit being saved for elements in $\mathbb{F}_p$ and five bits being saved per element in $\mathbb{F}_z$. For the RSDP(G) parameter sets on the other hand, the primes used are $p = 509$ and $z = 127$, thus resulting in seven and one bit of savings respectively. Storing vectors and matrices bit-packed results in increased access latency to these elements as one first needs to compute the respective offset and might also need to rebuild an element from two different words stored in memory. One can furthermore store $\texttt{chall}_2$ bit-packed as it consists of single bits thus reducing the storage requirement for it from t bytes to t bits. In summary, one can for e.g. RSDP-5-fast save around 3 kB here of which around 1.8 kB stem from storing the matrix in bit-packed form which is irrelevant in case one opts to compute the matrix on-the-fly as outlined in the previous paragraph. For RSDP(G)-1-fast which is the parameter set with the smallest memory footprint, one can save around 1 kB of which again around 740 B stem from bit-packing the matrices.

Maximum achievable savings It should be noted that directly employing the additional optimizations outlined in the last two paragraphs results in less savings than their independent results suggest as we used these available spaces when they are not occupied to hide e.g. the sizes of Keccak states required at the beginning of a signature generation. However as lower bound, though challenging to achieve in practice, we expect a stack of around 4 kB for RSDP-5-fast excluding the signature and a stack of around 2.8 kB for RSDP(G)-1-fast. These results would come at an increase in runtime of a factor between 4 and 10 leading us to not include these optimizations as their size impact is limited compared to the optimizations outlined above while being far more expensive in terms of clock cycles.

4 Speed Optimization

In this section, we introduce optimization techniques designed to reduce the signing and verification time of CROSS on the Cortex-M4. Specifically, we detail the use of precomputed exponentiation and a SIMD-oriented vector-matrix multiplication. The performance improvements resulting from these optimizations are reported in Section 5. Concurrently to this work, the authors of [HMK+24] updated their work such that their latest version also includes some optimizations exploiting the DSP instruction set on the Cortex-M4. We compare our results against theirs in Section 5.3.

Precomputed constant-time exponentiation Performing modular exponentiation operations ($g^x \bmod p$) is generally computationally intensive, as it relies on the *square and multiply* (S&M) method. This method requires a sequence of modular multiplications and squares that scale linearly with the bit-length of the exponent. Even for relatively small exponents, such as those used in CROSS, this approach necessitates 5 modular squares and up to 5 modular multiplications (for RDSP), or 6 modular squares and at most 6 modular multiplications (for RSDP(G)).

Given that one is targeting a board without SRAM caches, can ensure that all precomputed values for the exponentiation reside within one memory bank and that also no other system component introduces variable access times depending on the accessed location, one can also opt for using a lookup table as outlined below. This is possible because in CROSS, the base g and modulus p are fixed, while the exponent x is known to lie within a specific range $x \in [0, z)$.

As a result, modular exponentiation can be implemented efficiently by fully precomputing the values $(g^0 \bmod p, \ldots, g^{z-1} \bmod p)$ and storing them in a lookup table. At runtime, the only variable is the exponent x, which allows us to translate an element from its restricted representation in $\mathbb{F}_z$ to its corresponding value in $\mathbb{F}_p$ via a single table lookup at index x.

While we do not include this optimization by default as it is not generally applicable, we want to point out here that up to 10.5% of the clock cycles can be saved for the stack and speed optimized implementation for the RSDP(G) parameter sets. For the RSDP parameter sets, this optimizations yields not noticeable advantage.

Vector-matrix multiplication The fundamental arithmetic operation in CROSS is the syndrome computation: an error vector $\mathbf{e} \in \mathbb{F}_p^n$ is multiplied by the transposed parity-check matrix $\mathbf{H}^\top \in \mathbb{F}_p^{n \times (n-k)}$ to produce the syndrome $\mathbf{s}$. Optimizing this computation on the Cortex-M4 requires careful selection of efficient data-parallel operations, along with an appropriate memory layout.

Note that the parity-check matrix has the form $\mathbf{H} = [\mathbf{V} \mid \mathbf{I}]$, consisting of an arbitrary $(n-k) \times k$ block $\mathbf{V}$ on the left and an $(n-k)$-dimensional identity matrix $\mathbf{I}$ on the right. This structure provides two advantages: first, only the non-identity part ($\mathbf{V}^\top$) needs to be stored in memory; second, the last $n-k$ components of $\mathbf{e}$ are directly multiplied by the identity submatrix, yielding the same portion of $\mathbf{e}$. These values can therefore be used to initialize $\mathbf{s}$, after which the contributions from the remaining operations are accumulated by addition.

A vector-matrix multiplication can be expressed as a double nested loop, iterating over the rows and columns of $\mathbf{V}^\top$ and accumulating the result of scalar multiplications into the corresponding entries of $\mathbf{s}$. Organizing the computation with the inner loop traversing the rows of $\mathbf{V}^\top$ and the outer loop traversing its columns amounts to evaluating $n-k$ inner products between (the first k components of) $\mathbf{e}$ and a column of $\mathbf{V}^\top$. Because of commutativity, parallelism can, in principle, be introduced in either the inner or the outer loop.

On the Cortex-M4, we can adopt a technique previously employed in FrodoKEM [BFM+18] to parallelize the inner loop when vector and matrix elements are stored as 16-bit integers. This method, illustrated in Fig. 1, leverages two key instructions: `smlad` and `ldmia`.

- `smlad` (**Signed Multiply Accumulate Dual**): This instruction takes three inputs: (1) a 32-bit word containing two 16-bit operands (a_0 and a_1), (2) a 32-bit word containing another two 16-bit operands (b_0 and b_1), and (3) a 32-bit accumulator c. It returns a 32-bit word equal to the pairwise products added together and accumulated ($a_0 b_0 + a_1 b_1 + c$). On the Cortex-M4, `smlad` executes in a single clock cycle and is provided by the DSP extension of the ARMv7-M instruction set.

- `ldmia` (**Load Multiple Increment After**): This instruction takes as input a memory address and a list of 32-bit registers. It sequentially loads each register with a 32-bit word from memory, starting at the given address. The latency is one plus the number of registers loaded, measured in clock cycles. With respect to using multiple `ldr` instructions, `ldmia` brings no speed advantage, it does however reduce code size. Optionally, appending a `!` to the base address causes the memory pointer to be incremented after the load.

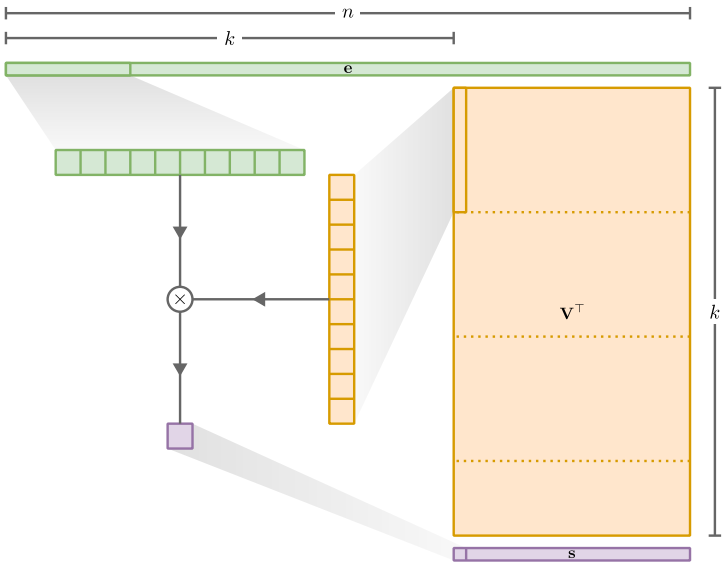

Figure 1: Fast vector by matrix multiplication on the M4. The syndrome $\mathbf{s}$ is the product of an error vector $\mathbf{e}$ and a transposed parity check matrix $\mathbf{H}^T$, of which only the non-identity portion ($\mathbf{V}^T$) is materialized. A sequence of 2 `ldmia` and 5 `smlad` instructions effectively processes 10 elements of the matrix.

We use `ldmia` to load multiple operands from $\mathbf{e}$ and $\mathbf{V}^\top$, and `smlad` to perform pairwise multiplications followed by accumulation. Specifically, a single `ldmia` loads five 32-bit words, which correspond to ten 16-bit elements of $\mathbf{e}$. A second `ldmia` loads ten elements from the matrix in the same fashion. With these 20 operands available, we perform five successive `smlad` calls: each invocation multiplies two elements of $\mathbf{e}$ with two elements of $\mathbf{V}^\top$, sums the results, and stores the outcome in a 32-bit register.

This procedure leverages all 13 general-purpose registers of the Cortex-M4: ten for operands, one for the accumulator, and two for the memory addresses used during loading. In total, the sequence of two `ldmia` instructions followed by five `smlad` instructions computes 10 multiplications and 9 additions, producing the result in a single 32-bit register. The process is repeated $\lfloor k/10 \rfloor$ times, and if k is not a multiple of 10 (as illustrated by the dotted lines in Fig. 1), the remaining k mod 10 elements are handled with additional `smlad` operations. Please note that the concurrent work of [HMK+24] also uses a DSP-based approach but they limit their optimizations to only four operands at a time and also keep the modulo reduction the same as the one employed in the reference implementation.

After fully parallelizing the inner loop of the vectors matrix multiplication, each entry of $\mathbf{s}$ resides in a 32-bit register representing the inner product between $\mathbf{e}$ and a column of $\mathbf{V}^\top$. This value must then be reduced modulo p. We employ a constant-time modular reduction routine described in [War13], which is based on Barrett's original idea [Bar86] of approximating the division of an integer x by a fixed integer p using two precomputed constants μ and β. These constants are chosen so that $\frac{\mu}{2^\beta}$ approximates $\frac{1}{p}$. Instead of dividing, one multiplies by $\frac{\mu}{2^\beta}$ to obtain the quotient q, then computes the remainder as $r = x - qp$. Crucially, the division by 2^β is implemented as a bit-shift right, giving:

$$x \bmod p = x - ((x \cdot \mu) \gg \beta) \cdot p$$

As shown in Section 10-9 of [War13], given the maximum value to be reduced, $x_{\max}$, the

optimal β is the smallest integer that satisfies:

$$2^\beta > \left(\left\lfloor \frac{x_{\max} + 1}{p} \right\rfloor p - 1 \right) (p - 1 - (2^\beta - 1) \bmod p)$$

In our case, we can perform a simple range analysis to find $x_{\max}$: each element of $\mathbf{s}$ is the sum of $k + 1$ terms, consisting of one initial value taken directly from the last portion of $\mathbf{e}$ and k products of elements from $\mathbb{F}_p$. Therefore, the reduction function must handle values up to $x_{\max} = (p - 1) + k(p - 1)^2$. This bound has to be adjusted slightly for the RSDP variants, where elements of $\mathbb{F}_p$ use a double-zero notation (the 0 element can be stored as either 0 or 127 [BGP26]), the result is $x_{\max} = p + kp^2$. Given $x_{\max}$, we determine β as shown above, and subsequently compute μ as:

$$\mu = \frac{2^\beta + p - 1 - (2^\beta - 1) \bmod p}{p}$$

With this choice of μ and β, we achieve an optimal modular reduction for the vector-matrix product, requiring only a single reduction per element of $\mathbf{s}$. Notice that the classical Barrett technique performs a conditional subtraction when the result is $\geq p$, which must be implemented in constant-time, often using masking. Warren's method on the other hand, finds the correct result directly, without the need for this subtraction. The resulting formulas used to reduce elements of the syndrome $\mathbf{s}$ are:

$$x \bmod 127 = x - ((x \cdot 4227331) \gg 29) \cdot 127$$

$$x \bmod 509 = x - ((x \cdot 4219025) \gg 31) \cdot 509$$

A final consideration concerns the memory layout. We store the matrix $\mathbf{V}^\top$ in column-major order, allowing `ldmia` to load pairs of elements from a column directly, as opposed to accessing two 16-bit elements from separate 32-bit registers and then combining them into a single one.

The technique we described integrates an efficient memory layout, data parallelization, and high register utilization to compute the syndrome resulting from the multiplication of an error vector with a parity-check matrix. It can be applied directly to the RSDP(G) variants of CROSS, which operate over $\mathbb{F}_{509}$, where elements are naturally represented as 16-bit unsigned integers. The RSDP variants, however, use $\mathbb{F}_{127}$, whose elements normally fit within 8-bit integers. To make the approach compatible, we extend the inputs from 8 to 16 bits in RSDP, incurring the non-negligible cost of doubling their memory footprint.

5 Experimental Results

We report the results of our benchmarks on the *Nucleo-L4R5ZI* board endowed with a STM32L4R5ZI Cortex-M4 microcontroller clocked at 20 MHz, 640kiB SRAM and 2 MiB Flash, using arm-gcc version 13.2 with optimization flag *-O3*. This board is the default evaluation board of the pqm4 project for NIST's Additional Signature Call as the former default evaluation platform based on an STM32F407G provided insufficient SRAM for most submissions. We also employ the benchmarking infrastructure from pqm4. While all parameter sets of our implementation with all size-preserving-optimizations enabled also fit on the smaller *STM32F4Discovery* board from the pqm4 project, we report the benchmarks on the *Nucleo-L4R5ZI* as this board allows us to perform a full speed-size trade-off exploration and indicate the memory overhead for each optimization individually. The board is also used in the concurrent work of [HMK+24], thus using the same platform allows for a direct comparison. We refer to the supplementary material in Section 6 for results on the *STM32F4Discovery* and only point out here that the same

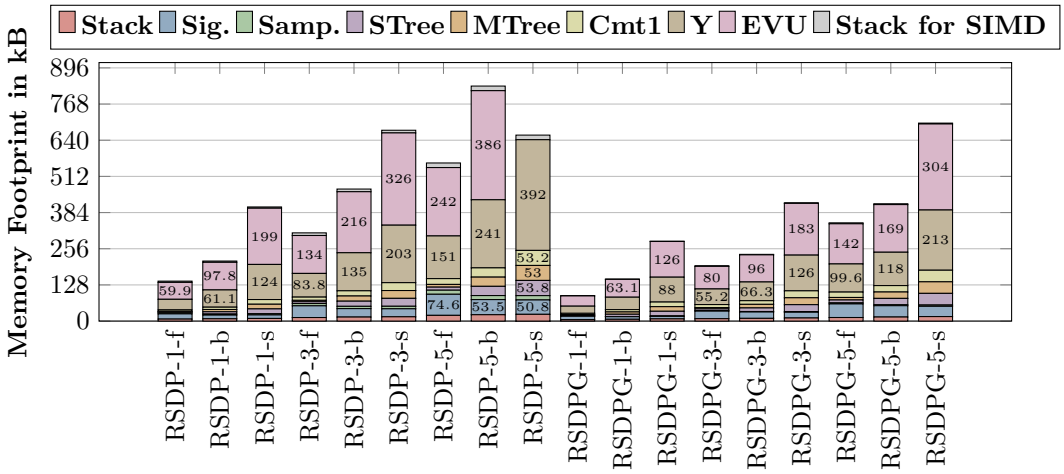

Figure 2: Impact of the different optimizations on the memory footprint of the CROSS signature generation, EVU missing for RSDP-5-s due to size constraints of the benchmarking platform

implementation takes roughly $5-7\%$ more clock cycles on the *Nucleo-L4R5ZI* while the memory consumption is almost identical. All results in this paper were averaged over 50 iterations. Results exceeding 640 kB like those in Figure 2 are obtained by running each optimization separately and stacking the results.

Taking into account the optimizations outlined above, we need to distinguish between signature generation and verification. In signature generation, we can trade performance in terms of clock cycles versus stack size depending on the needs of a certain application depending on the enabled optimizations. For this work, we consider two different memory optimization corners which are either based on what is at maximum achievable or on benchmarking devices used in the pqm4 project[3]. Namely we first target the minimal memory footprint possible and then outline which optimizations can be discarded in case one targets the *STM32F4Discovery* which offers 128 kB of normal SRAM.

The reference implementation of CROSS requires more than one MB for the largest parameter sets, which is infeasible for microcontrollers like the ones used in the pqm4 project. This significant footprint can partly be attributed to the fact that the reference implementation is focused on comprehensibility and not on stack optimization such that straightforward approaches like the memory reuse outlined in [HMK+24] are not applied here by default. We therefore start our evaluation at the point where we already applied the memory reuse for $\mathbf{y}$ and $\mathbf{u}'$ in the signature generation and $\mathbf{u}'$ and $\mathbf{y}'$ in the signature verification as these optimizations come at no cost in runtime and therefore don't impose a trade-off between runtime and stack size. The impact of all further optimizations on the overall memory footprint is depicted in Figure 2 for signature generation and in Figure 3 for signature verification.

Throughout the results section, the abbreviations in the Figures are defined as follows. **Stack** corresponds to the minimal stack size with all optimizations enabled, **Sig.** is the size of the corresponding signature, **Samp.** refers to the overhead induced by the sampling implementation from the reference implementation with respect to our optimized version, **STree** and **MTree** correspond to the stack requirements by the references tree structures, **Cmt1** is the additional space necessary if *cmt1* is not absorbed on-the-fly, **Y** refers to the amount of memory necessary to store the vector $\mathbf{y}[i]$ for all rounds, **EVU** corresponds to the space necessary if the vectors e, v, and u are not recomputed and **Stack for SIMD**

[3] https://github.com/mupq/pqm4

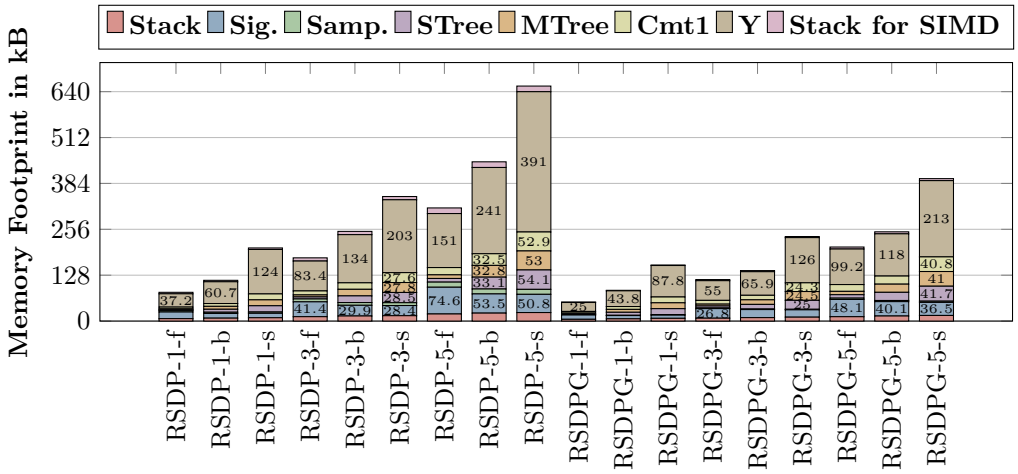

Figure 3: Depiction of stack consumption for the verify operation for CROSS

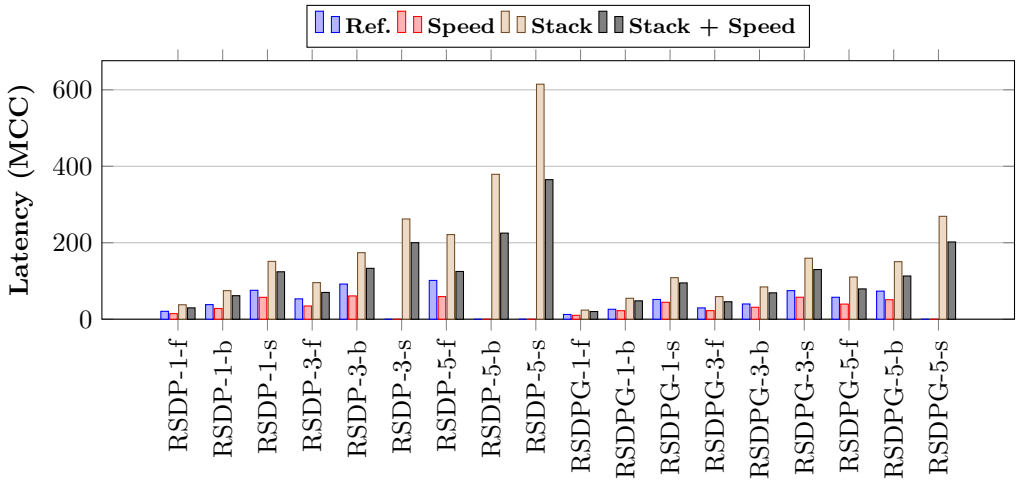

Figure 4: Mega Clock Cycles for Signature Generation with all stack optimizations enabled and SIMD optional, reference values for RSDP-5-b, RSDP-5-s and RSDPG-5-s not available due to memory size constraints of the benchmarking platform

finally is the memory overhead induced by our DSP instruction based optimizations. In difference to all other optimizations which reduce the stack, the DSP instruction based optimizations included in **Stack for SIMD** slightly increase the minimal possible stack and the overhead indicated by **Stack for SIMD** therefore needs to be added to the **Stack** value in case one opts to use them. We refer to the end of Section 4 where we explain this overhead. The different parameter sets outlined correspond to the different security levels (1, 3, 5) as well as to the different optimization corners available for CROSS, namely f stands for *fast* which refers to the parameter sets optimized for runtime, b stands for *balanced* referring to the parameter sets yielding a trade-off between runtime and signature size while s stands for *small* referring to the parameter sets optimized for signature size. In addition, the overall overhead of all optimizations as well as the performance improvement gained from using the above mentioned arithmetic optimizations on Cortex-M4 in terms of clock cycles are shown in Figure 4 and Figure 5.

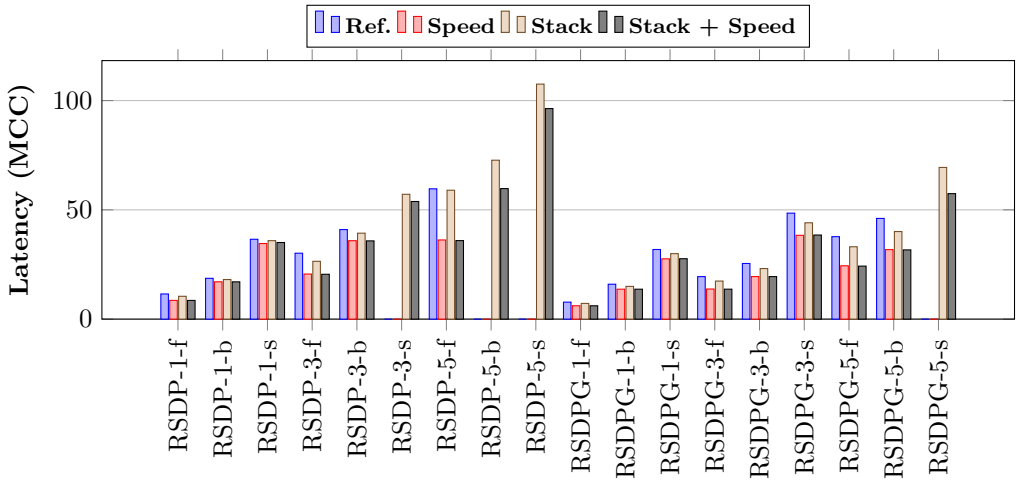

Figure 5: Mega Clock Cycles for Signature Verification with all stack optimizations enabled and SIMD optional, reference values for RSDP-5-b, RSDP-5-s and RSDPG-5-s not available due to memory size constraints of the benchmarking platform

5.1 Minimal memory footprint and speed optimization

We achieve a minimal stack of between 5.1 kB and 23.9 kB depending on the parameter set as shown in Figure 6 for the signature generation of CROSS. When taking the size of the signature into account, we end up with a total memory footprint between 15.5 kB and 94.9 kB. The size of the corresponding binaries is between 12 kB and 25 kB depending on the parameter set. These binaries contain the key and signature generation as well as the verification function. In total all parameter sets except RSDP on security level 5 and RSDPG-fast on security level 3 fit on a microcontroller with 64 kB even with speed optimizations enabled. All parameter sets for security level 1 furthermore fit on a microcontroller with 32 kB of memory. We want to emphasize here that, in difference to [HMK+24], we consider the stack and the size of the signature when determining on which devices a certain parameter set of our implementation fits. In terms of memory footprint the results for the signature generation are directly transferable to the signature verification as both algorithms exhibit nearly identical footprints after optimization.

On the performance side, the optimizations come with a significant impact as shown in Figure 4 for the signature generation. While some optimizations, especially those only requiring the incremental hashing of some values generated in each round of the protocol, e.g. the incremental hashing of cmt_0 and cmt_1 are almost for free, the others require recomputations of large parts of the algorithm to compute either the first or the second response and therefore significantly influence the runtime. Taking the SIMD optimizations above into account, we can reduce the total number of clock cycles by between 40.7% and 13.2% for signature generation with respect to the memory optimized implementation depending on the parameter set as shown in Figure 4.

For signature verification, CROSS requires no recomputation of any variables. Any overhead caused here stems from either storing and loading an incremental Keccak state multiple times or from the slightly more complicated on the fly computation of the incremental tree indexes. For all parameter sets, this overhead is completely compensated by improvements like less memory movements on elements to and from the tree structures due to the above mentioned removal of intermediate storage locations. The stack optimized implementation thus reduces the number of clock cycles required by up to 18% for the

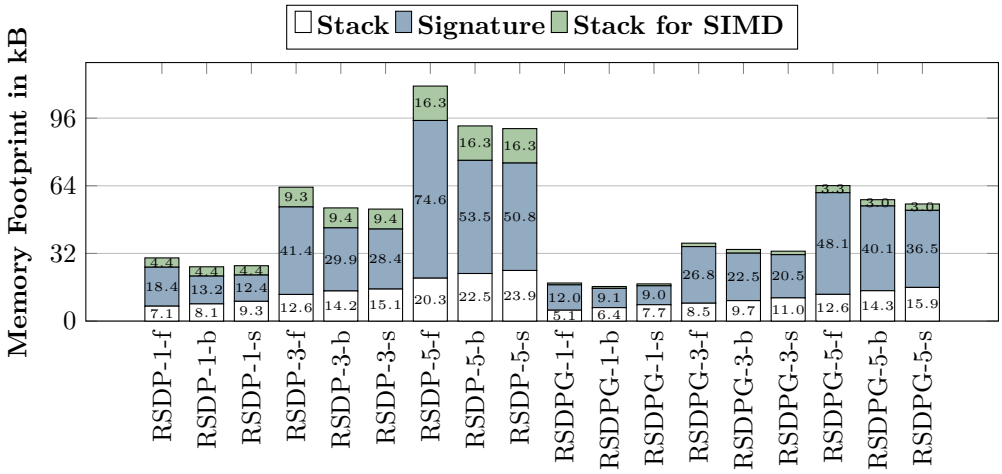

Figure 6: Depiction of stack consumption for the sign operation for CROSS with all optimizations enabled

parameter sets where we were able to obtain results from the reference implementation as visible in Figure 5. In comparison to the reference implementation the speed optimizations result in up to 36% of savings.

5.2 Constraining to 128kB

While all memory optimizations outlined above can be switched off individually, we outline here the effect of switching off the incremental computation of the Merkle tree in the signature generation as would be possible for most parameter sets on a device with 128 kB of memory available. Please note that in difference to [HMK+24] we consider stack and signature size here, thus we only have part of the 128 kB available for the stack, the remainder is needed to store the signature. While there might be scenarios where storing the signature in volatile memory is not necessary, e.g. because it is stored in Flash memory, it is certainly the most common case that a signature generated or to be verified is at least temporarily stored in the device's SRAM. This results in a reduction in runtime of up to 14% while increasing the stack size by a factor of roughly three compared to an implementation with all optimizations switched on. All RSDP-5 parameter sets as well as RSDP(G)-5-small do not fit into 128 kB anymore with this optimization switched off.

5.3 Comparison with the results from [HMK+24]

Both implementations of CROSS proposed in [HMK+24] are included in Figure 7 and 8 where we compare our results against other signature schemes. In comparison to the concurrent work in [HMK+24], we achieve significantly smaller stack sizes, their results range between 16 kB and 110.9 kB for signature generation for their "ultra" variant while we achieve between 5.1 kB and 24 kB for all parameter sets when considering only stack optimizations. When considering speed and stack optimizations, we achieve between 6 kB and 36.6 kB while their "light" variant achieves between 40.2 kB and 523.98 kB. For signature verification, we achieve between 4.9 kB and 23.7 kB which is also significantly smaller than their 12.3 kB to 105.012 kB depending on the parameter set. [HMK+24] achieves here almost exactly the same results for their "light" and their "ultra" variant.

On the performance side, both of their implementations are faster in signature generation. Concretely their "light" version is between 39% and 47% faster compared with

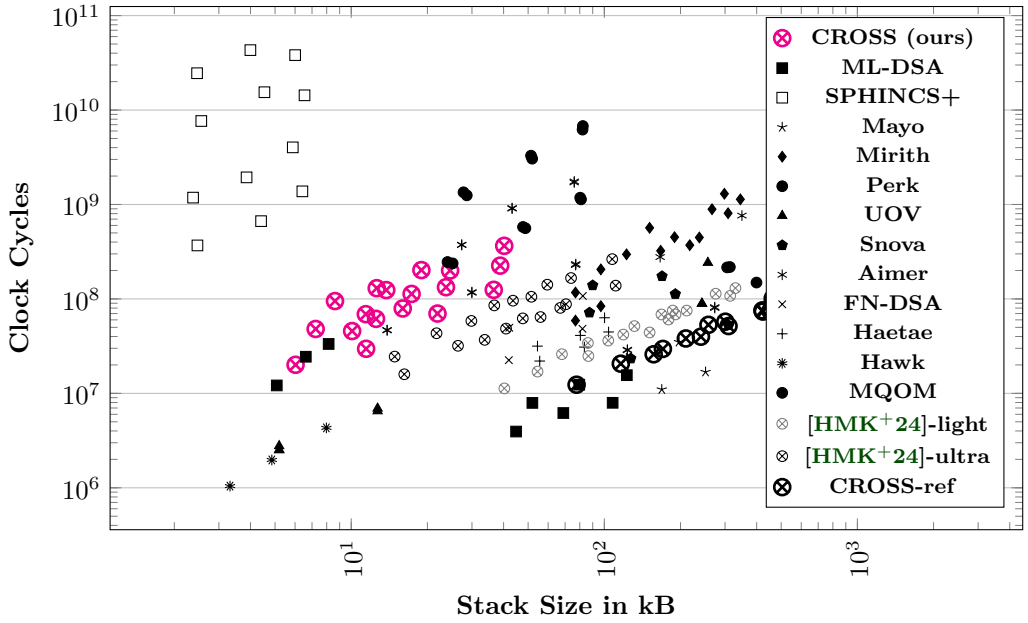

Figure 7: Comparison of signature schemes from pqm4 with this work, signature generation, please note that both axes are logarithmic and that the y-axes does not start at zero

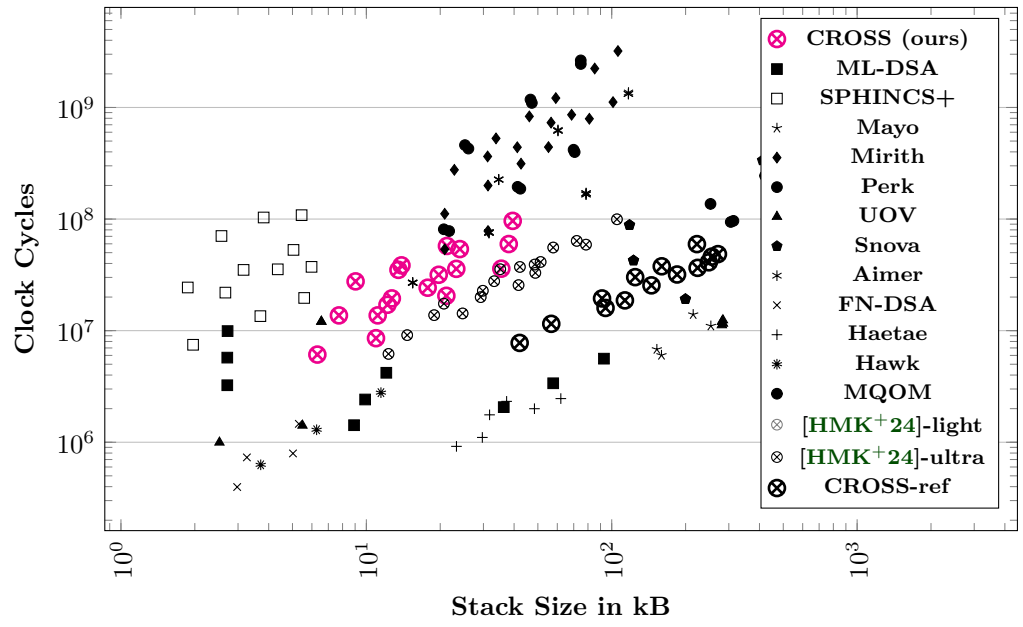

Figure 8: Comparison of signature schemes from pqm4 with this work, signature verification, please note that both axes are logarithmic, that the y-axes does not start at zero and that the results for the light and ultra variants of [HMK+24] are almost identical here

our size + speed optimized version, but it is also between a factor of 4.7 to 16.6 larger. Their "ultra" version is between 35% and 57% faster compared with our stack optimized version, but it is also between a factor of 2.1 and 7.0 larger. For signature verification on the other hand, we outperform their results by between 0.3% and 14.8% when comparing their "light" against our stack + speed optimized version depending on the parameter set while the stack is between a factor of 1.3 and 3.7 smaller. Our purely stack optimized version is slower than their versions here but achieves even smaller stack sizes than our stack + speed optimized version. In addition to the comparison in Figure 7 and 8, we provide all performance and stack results of their and our work in Table 2.

5.4 Comparison with other schemes

We compare the results of our implementation to the other signature schemes integrated into pqm4. This includes the standardized signature scheme ML-DSA, FN-DSA as well as SPHINCS+, all Round 2 candidates from NIST's additional signatures call given that they fit on the benchmarking platform and the schemes HAETAE and AIMer from the KpqC competition. Please note that though the schemes included in pqm4 are in Round 2 of NIST's competition, the implementations included are not necessarily the round 2 versions of these schemes. Especially all MPCitH schemes currently included in pqm4 are still the Round 1 versions of their implementations. The comparison with the other schemes is shown in Figure 7 and Figure 8 for signature generation and verification respectively. Please note the memory footprint includes only the stack memory required at runtime. Within these plots, we have all stack and speed optimizations discussed throughout this paper enabled. As can be seen, our implementation exhibits among the lowest memory footprints of all available schemes with only SPHINCS+, Hawk, UOV and ML-DSA in signature generation and verification and FN-DSA in signature verification being smaller. On the performance side, CROSS is outperformed by ML-DSA, Hawk and UOV in signature generation while being considerably faster than SPHINCS+ and on par with most of the other Round 2 submissions. For signature verification, out implementation is again outperformed by the lattice-based schemes ML-DSA, Hawk and FN-DSA as well as Haetae while being on par with SPHINCS+ and some of the multivariate schemes but faster than the MPCitH schemes PERK and Mirith.

For the sake of completeness, we have also added benchmarking results of the CROSS reference implementation to Figure 7 and 8 for all parameter sets where the implementation fitted on the benchmarking platform.

6 Conclusion

In this work, we proposed a runtime memory optimized implementation of CROSS tailored for resource constrained devices as well as additional speed optimizations targeting the Cortex-M4 platform. While our implementation achieves stack sizes between 5.1kB and 24kB we also reach a maximum performance gain for signature verification of 39% with respect to the reference implementation. Furthermore, our implementation does not introduce any secret dependent runtime to the best of our knowledge but we leave an in-depth security analysis as well as protections against other side-channel attacks up for future work. Finally, we note that, while it is possible to further reduce the memory footprint of our implementation, through optimizations such as storing all elements in a packed form, this approach offers a worse trade-off between speed and footprint, increasing runtime by a factor between 4 and 10, while saving only a few kB. We consider such a set of optimizations as an interesting target for future investigation on the viability of CROSS on ultra-low memory devices.

Acknowledgments. This work was supported in part by project SERICS (PE00000014)

under the Italian NRRP MUR program funded by the EU - NGEU. The authors furthermore acknowledge the financial support by the Federal Ministry of Research, Technology and Space of Germany in the project “Quoryptan”, project identification number 16KIS2030K.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Appendix - Performance results on Cortex-M4 devices

The following tables provide full results for the stack and speed optimized versions of CROSS on the two benchmarking platforms employed throughout this work.

STM32F4 results

As the reference implementation fits only for few parameter sets on the *STM32F4-Discovery* with 128kB of readily usable SRAM, we only provide results for the stack and speed optimized implementations for this platform.

Table 1: Performance results on the *STM32F4 Discovery* board, stack size in bytes

		Size Opt.		**Size + Speed**	
		Perf. kCC	**Stack kB**	**Perf. kCC**	**Stack kB**
RSDP-1-f	Sign	35 719	7.1	27 859	11.4
	Verify	9886	6.8	8051	10.9
RSDP-1-b	Sign	70 580	8.1	57 967	12.5
	Verify	17 169	8.1	16 168	12.2
RSDP-1-s	Sign	142 856	9.3	116 843	13.8
	Verify	34 065	9.3	33 283	13.4
RSDP-3-f	Sign	90 161	12.6	65 842	21.9
	Verify	24 966	12.2	19 325	21.2
RSDP-3-b	Sign	164 181	14.2	125 288	23.7
	Verify	37 254	14.1	33 933	23.3
RSDP-3-s	Sign	247 368	15.0	188 472	24.6
	Verify	54 206	15.0	51 087	24.0
RSDP-5-f	Sign	205 748	20.3	117 305	36.5
	Verify	54 974	19.8	33 783	35.5
RSDP-5-b	Sign	352 962	22.4	211 820	38.8
	Verify	68 495	22.2	56 550	38.1
RSDP-5-s	Sign	572 100	23.8	343 358	40.2
	Verify	101 700	23.7	91 391	39.6
RSDPG-1-f	Sign	22 546	5.1	18 994	6.0
	Verify	6808	4.9	5783	6.3
RSDPG-1-b	Sign	51 752	6.3	45 490	7.2
	Verify	14 227	6.3	12 986	7.7
RSDPG-1-s	Sign	102 758	7.7	89 809	8.6
	Verify	28 444	7.7	26 217	9.0
RSDPG-3-f	Sign	55 858	8.4	43 161	10.1
	Verify	16 496	8.0	12 973	11.1
RSDPG-3-b	Sign	79 844	9.7	65 163	11.4
	Verify	21 952	9.6	18 431	12.7
RSDPG-3-s	Sign	150 914	11.0	122 862	12.6
	Verify	41 896	10.8	36 494	13.9
RSDPG-5-f	Sign	104 140	12.6	74 650	16.0
	Verify	31 289	12.1	22 870	17.8
RSDPG-5-b	Sign	142 054	14.3	106 508	17.4
	Verify	37 941	14.1	29 939	19.7
RSDPG-5-s	Sign	254 050	15.9	190 389	18.9
	Verify	65 814	15.7	54 279	21.3

Nucleo-L4R5ZI results

Table 2: Performance values on the *Nucleo-L4R5ZI* board, results where the implementation did not fit in the available memory are replaced null values, stack size in bytes

		Ref. [BBB+25a]		**Ours**						[**HMK+24**]			
				Speed Opt.		**Size Opt.**		**Size + Speed**		**light**		**ultra**	
		Perf. (kCC)	**Stack (kB)**	**Perf. (kCC)**	**Stack (kB)**	**Perf. (kCC)**	**Stack (kB)**	**Perf. (kCC)**	**Stack (kB)**	**Perf. (kCC)**	**Stack (kB)**	**Perf. (kCC)**	**Stack (kB)**
RSDP-1-f	Sign	20 611	116.2	14 526	120.9	37 784	7.1	29 647	11.5	17 002	54.5	24 504	14.8
	Verify	11 511	56.7	8576	60.7	10 454	6.8	8560	10.9	9117	14.7	9117	14.7
RSDP-1-b	Sign	38 129	210.0	28 003	214.8	74 663	8.1	61 510	12.5	34 160	86.1	43 303	21.7
	Verify	18 676	113.2	17 068	117.1	18 106	8.2	17 056	12.2	17 457	20.7	17 456	20.7
RSDP-1-s	Sign	75 637	421.4	57 208	426.2	151 246	9.3	124 024	13.8	68 489	168.2	85 421	36.8
	Verify	36 541	223.8	34 588	227.8	35 903	9.3	35 054	13.5	35 517	35.1	35 516	35.1
RSDP-3-f	Sign	53 204	258.0	34 619	267.8	95 658	12.6	70 003	21.9	41 923	118.8	58 501	29.8
	Verify	30 165	124.6	20 624	133.3	26 453	12.3	20 526	21.2	22 778	29.8	22 778	29.8
RSDP-3-b	Sign	91 854	463.4	60 901	473.2	173 993	14.2	133 068	23.7	75 336	186.5	95 833	43.6
	Verify	40 970	248.7	35 891	257.4	39 329	14.2	35 803	23.3	37 226	42.2	37 226	42.2
RSDP-3-s	Sign	0	0.0	0	0.0	262 147	15.1	200 125	24.6	113 512	275.5	141 538	59.5
	Verify	0	0.0	0	0.0	57 127	15.0	53 827	24.0	55 593	57.7	55 591	57.7
RSDP-5-f	Sign	101 433	463.0	59 146	479.7	221 300	20.3	124 853	36.6	75 305	211.7	105 414	51.5
	Verify	59 625	222.4	36 224	237.8	58 979	19.8	35 931	35.5	41 250	51.3	41 250	51.3
RSDP-5-b	Sign	0	0.0	0	0.0	379 035	22.5	225 184	38.8	130 171	329.7	166 434	73.9
	Verify	0	0.0	0	0.0	72 723	22.2	59 724	38.1	63 723	72.0	63 724	72.0
RSDP-5-s	Sign	0	0.0	0	0.0	614 671	23.9	365 019	40.2	199 545	524.0	263 838	107.5
	Verify	0	0.0	0	0.0	107 544	23.8	96 338	39.5	99 579	105.0	99 577	105.0
RSDPG-1-f	Sign	12 344	77.7	10 180	82.6	23 840	5.1	20 078	6.0	11 283	40.3	15 931	16.2
	Verify	7765	42.2	6112	43.4	7187	4.9	6105	6.3	6190	12.3	6190	12.3
RSDPG-1-b	Sign	25 929	156.7	22 273	164.3	54 687	6.4	48 040	7.2	25 994	68.0	31 890	26.4
	Verify	15 977	94.4	13 690	95.5	14 988	6.3	13 700	7.7	13 772	18.9	13 771	18.9
RSDPG-1-s	Sign	51 610	309.6	44 253	323.6	108 634	7.7	94 923	8.6	51 547	131.5	62 218	47.6
	Verify	31 861	184.6	27 590	186.0	29 962	7.7	27 645	9.0	27 742	33.2	27 743	33.2
RSDPG-3-f	Sign	29 558	170.4	22 370	181.5	59 124	8.5	45 690	10.1	24 835	86.5	36 876	33.7
	Verify	19 438	91.3	13 778	94.2	17 415	8.0	13 704	11.1	14 264	24.7	14 265	24.7
RSDPG-3-b	Sign	39 817	240.9	31 225	253.7	84 412	9.7	68 960	11.4	36 276	103.7	48 490	40.9
	Verify	25 462	145.1	19 464	148.0	23 134	9.6	19 456	12.7	19 962	29.2	19 963	29.2
RSDPG-3-s	Sign	74 717	453.2	57 514	475.8	159 557	11.0	129 931	12.6	68 554	191.2	87 887	70.6
	Verify	48 502	270.9	38 353	273.8	44 082	10.9	38 486	13.9	39 120	48.5	39 119	48.5
RSDPG-5-f	Sign	57 393	300.8	39 540	318.9	110 353	12.6	79 247	16.0	44 142	151.0	64 293	56.0
	Verify	37 730	159.9	24 406	165.2	33 105	12.2	24 259	17.8	25 592	41.7	25 593	41.7
RSDPG-5-b	Sign	73 446	423.2	51 006	443.8	150 497	14.3	112 994	17.3	60 457	179.3	80 384	67.0
	Verify	46 080	255.7	31 791	261.1	40 067	14.1	31 677	19.7	32 941	48.8	32 941	48.8
RSDPG-5-s	Sign	0	0.0	0	0.0	269 069	15.9	202 045	18.9	107 790	314.3	138 268	110.9
	Verify	0	0.0	0	0.0	69 410	15.7	57 418	21.3	58 983	78.6	58 979	78.6

References

[Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[BBB+24] Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Marco Palumbi, and Lucas Pandolfo Perin. Enabling perk and other mpc-in-the-head signatures on resource-constrained devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):84–109, Sep. 2024.

[BBB+25a] Marco Baldi, Alessandro Barenghi, Michele Battagliola, Sebastian Bitzer, Marco Gianvecchio, Patrick Karl, Felice Manganiello, Alessio Pavoni, Gerardo Pelosi, Federico Pintore, Paolo Santini, Jonas Schupp, Edoardo Signorini, Freeman Slaughter, Antonia Wachter-Zeh, and Violetta Weger. CROSS: Codes and Restricted Objects Signature Scheme - Specification v2.2 — July

31, 2025. https://www.cross-crypto.com/CROSS_Specification_v2.2.pdf, 2025.

[BBB+25b] Marco Baldi, Alessandro Barenghi, Luke Beckwith, Tung Chou, Jean-François Biasse, Andre Esser, Kris Gaj, Patrick Karl, Kamyar Mohajerani, Gerardo Pelosi, Edoardo Persichetti, Markku-Juhani Saarinen, Paolo Santini, Robert Wallace, and Floyd Zweydinger. LESS: Linear Equivalence Signature Scheme - Specification v2.0 — Feb. 7, 2025. https://www.less-project.com/LESS-2025-02-07.pdf, 2025.

[BCC+23] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. Nibbling MAYO: Optimized implementations for AVX2 and cortex-m4. Cryptology ePrint Archive, Paper 2023/1683, 2023.

[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., Paper 2018/1116, https://eprint.iacr.org/2018/1116, 2018.

[BGP26] Alessandro Barenghi, Marco Gianvecchio, and Gerardo Pelosi. A high efficiency avx2-optimized engineering of the post-quantum digital signature cross. *Journal of Information Security and Applications*, 97:104331, 2026.

[BRS22] Joppe W. Bos, Joost Renes, and Amber Sprenkels. Dilithium for memory constrained devices. Cryptology ePrint Archive, Paper 2022/323, 2022. https://eprint.iacr.org/2022/323.

[HMK+24] Harry Hart, Puja Mondal, Suparna Kundu, Supriya Adhikary, Angshuman Karmakar, and Chaoyun Li. LightCROSS: A Secure and Memory Optimized Post-Quantum Digital Signature CROSS. IACR Cryptol. ePrint Arch., Paper 2024/1929, https://eprint.iacr.org/2024/1929, 2024.

[Nat24a] National Institute of Standards and Technology. FFT over NTRU-Lattice-Based Digital Signature Algorithm. https://csrc.nist.gov/csrc/media/Presentations/2024/falcon/images-media/prest-falcon-pqc2024.pdf, 2024.

[Nat24b] National Institute of Standards and Technology. FIPS 204 - Module-Lattice-Based Digital Signature Standard. https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.204.pdf, 2024.

[Nat24c] National Institute of Standards and Technology. FIPS 205 - Stateless Hash-Based Digital Signature Standard. https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.205.pdf, 2024.

[NIS25a] NIST Information Technology Laboratory - Computer Security Resource Center. Post-Quantum Cryptography: Additional Digital Signature Schemes. https://csrc.nist.gov/projects/pqc-dig-sig, 2025. [Online accessed on July 2025].

[NIS25b] NIST Information Technology Laboratory - Computer Security Resource Center. Post-Quantum Cryptography Standardization. https://csrc.nist.gov/pqc-standardization, 2025. [Online accessed on July 2025].

[War13] Henry S. Warren Jr. *Hacker's Delight, Second Edition*. Pearson Education, 2013.