LightCROSS: A Secure and Memory Optimized Post-Quantum Digital Signature CROSS

-Authors' Version-

Puja Mondal[1], Suparna Kundu[2], Supriya Adhikary[1] and Angshuman Karmakar[1]

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

Abstract. CROSS is a code-based post-quantum digital signature scheme based on a zero-knowledge (ZK) framework. It is a second-round candidate of the National Institute of Standards and Technology's additional call for standardizing post-quantum digital signatures. The memory footprint of this scheme is prohibitively large, especially for small embedded devices. In this work, we propose various techniques to reduce the memory footprint of the key generation, signature generation, and verification by as much as 50%, 52%, and 74%, respectively, on an ARM Cortex-M4 device. Moreover, our memory-optimized implementations adapt the countermeasure against the recently proposed (ASIACRYPT-24) fault attacks against the ZK-based signature schemes.

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

1 Introduction

In August 2022, 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 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 SPHIINCS+ [BHK+19] have very large signatures. This is very problematic for many real-world scenarios, e.g. chain-of-trust-based authentication mechanisms. NIST has also acknowledged these issues and initiated another standardization procedure for additional post-quantum DS schemes [NIS23]. It aims to diversify the portfolio of digital signature schemes and to develop digital signatures with small signature sizes and fast verification.

CROSS [BBB+24] is a promising new code-based scheme currently in the second round of the NIST's additional DS standardization procedure [NIS24d]. It is based on the hard restricted syndrome decoding problem (RSDP) [BBC+21] in an interactive zero-knowledge framework. One major drawback of this scheme is the huge memory footprint. It is prohibitively large for small microcontrollers, and some of its parameters do not even fit on these platforms. Therefore, reducing the memory footprint is a crucial step for widespread real-world deployment. It is also essential for further progress in the standardization

procedure, as evident from the previous NIST standardization procedures. Therefore, addressing this issue is our major focus in this work. In particular, our contributions are as follows.

- **Memory optimized key generation:** We have analyzed the key generation routine and observed that two big matrices ($\boldsymbol{V}$ and $\boldsymbol{W}$) are the most significant contributors to the large memory footprint consuming almost $2118 - 15150$ bytes[1] of memory to generate a vector $\boldsymbol{\eta}$. We have shown that storing the full matrices at a time is unnecessary. Instead, $\boldsymbol{\eta}$ can be generated on the fly. With this modification, we have reduced memory requirement $43\% - 50\%$ on an ARM Cortex-M4 microcontroller.

- **Memory efficient signature generation and verification:** In several applications of DS, the key generation is usually performed once and, if required, can be done outside the device. On the other hand, signature generation and verification are performed more frequently and must be performed on-device. Therefore, memory-efficient implementations of these two procedures are even more important. First, in CROSS, these procedures use a large Merkle tree of size $2t - 1$ ($9760 - 127424$ bytes) and an array of *commitments* ($4896 - 63744$ bytes) to generate the Merkle root and `MerkleProof` consuming significant memory. We show a memory-efficient technique of Merkle root generation without storing the commitments. Second, in some cases, the input of `HASH` is stored in a big array before applying the `HASH` (e.g., the digests $\boldsymbol{d}_1$ and d_b generation). We have exploited the sponge construction of `SHAKE` [KjCP16]. We have streamlined the input generation and `xof_shake_update` function to reduce the memory requirement of the input array to a fraction of the original requirement. Third, we removed many redundant memory usages through a thorough execution flow analysis.

- **Countermeasure against ZKFault:** Recently, a fault attack ZKFault [MAKK24], has been shown on zero-knowledge based signature schemes. The authors have shown that their attack can recover the full signing key of CROSS from a single faulted signature. We have observed that this attack is also valid for the latest version of CROSS-v1.2. The paper [MAKK24] also proposed a secure implementation of LESS [BMPS20] against the fault attack. In this work, we have adapted their countermeasure technique for CROSS-v1.2. Further, we have also integrated our memory reduction techniques into this countermeasure. Finally, our implementations of the signature generation and verification algorithm reduce the memory consumption $40\% - 52\%$ and $52\% - 74\%$ compared to the original implementations with minuscule performance overhead even after the addition of the countermeasure.

We want to note that all our implementations[2] are constant-time, and we do not introduce any secret dependent branches or any other new side-channel vulnerability. Also, our implementations are fully compatible with the original implementation of CROSS.

2 CROSS and its memory footprint

In this section, we briefly describe the CROSS digital signature. We also describe the primary factors responsible for its large memory requirements. We begin with the notions used in the text. $\mathbb{F}_p$ and $\mathbb{F}_p^*$ represent the finite field of prime order p and the corresponding multiplicative group. The set of all size k vectors over the field $\mathbb{F}_p$ is denoted by $\mathbb{F}_p^k$ and all $k \times n$ matrices over the field $\mathbb{F}_p$ is indicated by $\mathbb{F}_p^{k \times n}$. We denote elements of $\mathbb{F}_p$ or

[1] Since the CROSS scheme has several parameter sets, we present the smallest and largest values of the corresponding parameter sets. We follow this notation throughout the text.

[2] Our code is available at https://github.com/s-adhikary/pqm4-CROSS

scalars with lowercase letters, elements of $\mathbb{F}_p^k$ and $\mathbb{F}_p^{k\times n}$ with bold lowercase letters and bold uppercase letters. $\mathscr{I}_m$ and $\boldsymbol{I}_m$ denote the set $\{0,\ 1,\ \ldots,\ m-1\}$ and identity matrix of dimension m, respectively. We denote $\boldsymbol{a}[i]$ and $\boldsymbol{A}[i,\ j]$ to represent the i-th element of the vector $\boldsymbol{a}$ and $(i,\ j)$-th element of the matrix $\boldsymbol{A}$ respectively. Let $\mathscr{J} \subset \mathscr{I}_n$ be an ordered set, then the notation $\boldsymbol{a}_{(\mathscr{J})}$ and $\boldsymbol{A}_{(*,\ \mathscr{J})}$ (corresponding $\boldsymbol{A}_{(\mathscr{J},\ *)}$) represents the subvector of $\boldsymbol{a}$ and submatrix of $\boldsymbol{A}$ formed by selecting the entries of the vector $\boldsymbol{a}$ and columns (rows) of $\boldsymbol{A}$ with indices specified in the set $\mathscr{J}$.

2.1 CROSS signature scheme

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

In this section, we discuss the key generation and signature generation algorithms and explain some portions of these algorithms that are required for our work. However, the verification algorithm is quite similar to the signature generation algorithm. Due to the page restriction, we do not describe the verification algorithm in our paper. For more details, we refer to the CROSS documentation [BBB+24].

2.1.1 Key generation algorithm (Alg. 1)

The parity check matrix in `CROSS_KeyGen`() is $\boldsymbol{H} = \begin{bmatrix} \boldsymbol{V} \\ \boldsymbol{I}_{n-k} \end{bmatrix}$, where the matrix $\boldsymbol{V}$ can be generated from the $\texttt{Seed}_{\texttt{pk}}$ using `CSPRNG` function. On the other hand, the restricted vector $\boldsymbol{e}$ is generated from `Seede` using `CSPRNG`. However, for the `RSDPG` variant, a vector $\boldsymbol{\zeta}$ and a matrix $\boldsymbol{G}$ are generated from `Seede` and $\texttt{Seed}_{\texttt{pk}}$ respectively. Then the matrix $\boldsymbol{M}_G = [\boldsymbol{W}|\boldsymbol{I}_m]$ and the vector $\boldsymbol{\zeta}$ are used to compute the restricted vector $\boldsymbol{e}$. The $\texttt{Seed}_{\texttt{pk}}$ and $\boldsymbol{s} = \boldsymbol{eH}$ is set as public key and $\texttt{Seed}_{\texttt{sk}}$ is our secret key.

Memory footprint of `CROSS_KeyGen`(): Here, we have observed that the matrices $\boldsymbol{V}$ of order $k \times (n-k)$ and $\boldsymbol{W}$ of order $m \times (n-m)$ occupy $1368 - 15150$ bytes and $750 - 2784$ bytes of memory, respectively. Therefore, for `RSDP` variant $1368 - 15150$ bytes (for matrix $\boldsymbol{V}$) and for `RSDPG`, $2118 - 15150$ bytes (for $\boldsymbol{V}$ and $\boldsymbol{W}$) are required in `CROSS_KenGen`() Alg. 1.

This algorithm mostly uses `CSPRNG` function. The above two matrices are generated using `CSPRNG` function, which will take *seed* as input and return a random element from the domain $\mathcal{D}$ as outputs. Fig. 1 shows the procedure of generating the matrices $\boldsymbol{V}$ and $\boldsymbol{W}$ for both $RSDP$ and $RSDPG$. The algorithm `initialized_csprng` first initializes a state. Then, `csprng_randombytes` generates a buffer say `buf` of length r from the state, where r is a fixed value depending on the domain $\mathcal{D}$ and the parameter sets. Next, the `CSPRNG_mat` function generates each element of the domain using the buffer. All of these functions are explained in the documentation of CROSS [BBB+24].

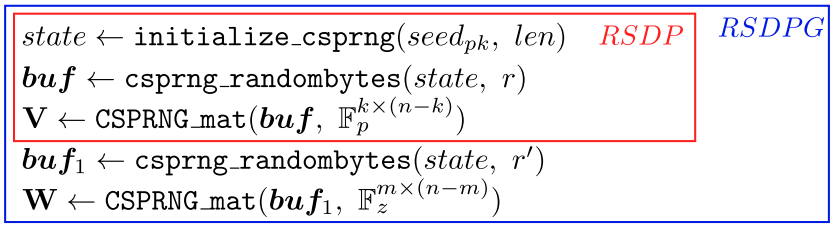

Figure 1: Generation of $\boldsymbol{V}$ and $\boldsymbol{W}$ from the seed $seed_{pk}$

Algorithm 1 CROSS_KeyGen() [BBB+24]

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

$\mathtt{Seed}_{\mathtt{sk}} \xleftarrow{\$} \{0,\ 1\}^{\lambda}$
$(\mathtt{Seede},\ \mathtt{Seed}_{\mathtt{pk}}) \leftarrow \mathtt{CSPRNG}\left(\mathtt{Seed}_{\mathtt{sk}},\ \{0,\ 1\}^{\lambda} \times \{0,\ 1\}^{\lambda}\right)$
if RSDP **then**
    $\boldsymbol{V} \leftarrow \mathtt{CSPRNG}\left(\mathtt{Seed}_{\mathtt{pk}},\ \mathbb{F}_p^{k\times(n-k)}\right)$
    $\boldsymbol{\eta} \leftarrow \mathtt{CSPRNG}(\mathtt{Seede},\ \mathbb{F}_z^{n})$
if RSDPG **then**
    $(\boldsymbol{V},\ \boldsymbol{W}) \leftarrow \mathtt{CSPRNG}\left(\mathtt{Seed}_{\mathtt{pk}},\ \mathbb{F}_p^{k\times(n-k)} \times \mathbb{F}_z^{m\times(n-m)}\right)$
    $\boldsymbol{M}_G \leftarrow [\boldsymbol{W} \,|\, \boldsymbol{I}_m]$
    $\boldsymbol{\zeta} \leftarrow \mathtt{CSPRNG}(\mathtt{Seede},\ \mathbb{F}_z^{m})$
    $\boldsymbol{\eta} \leftarrow \boldsymbol{\zeta}\boldsymbol{M}_G$
$\boldsymbol{e}_{(\mathscr{I}_n)} \leftarrow g^{\boldsymbol{\eta}_{(\mathscr{I}_n)}}$
$\boldsymbol{H} \leftarrow \begin{bmatrix} \boldsymbol{V} \\ \boldsymbol{I}_{n-k} \end{bmatrix}$
$\boldsymbol{s} \leftarrow \boldsymbol{e}\boldsymbol{H}$
return Public key: $(\mathtt{Seed}_{\mathtt{pk}},\ \boldsymbol{s})$, Private key: $\mathtt{Seed}_{\mathtt{sk}}$

2.1.2 Signature generation algorithm (Alg. 2)

The signature generation algorithm first constructs t many ephemeral seeds $\mathtt{Seed}[0],\ \ldots,\ \mathtt{Seed}[t-1]$ from the master seed MSeed and a random salt Salt. The seeds are generated in different ways depending on the parameter sets of the CROSS signature. For "fast" parameter sets of CROSS, the method does not use the Goldreich-Goldwasser-Micali (GGM) [GGM86] tree construction or Merkle tree construction. In this case, using $\mathtt{MSeed} \in \{0,\ 1\}^{\lambda}$, $\mathtt{Salt} \in \{0,\ 1\}^{\lambda}$ as input seeds we generate all the Seeds using CSPRNG function. The remaining parameter sets of CROSS construct a GGM tree called SeedTree to generate t many random ephemeral seeds. These seeds and the Salt are utilized to compute $\boldsymbol{u}'$ and $\boldsymbol{\eta}'$, which are also used to generate two sets of commitments $\mathtt{cmt}_0[0],\ \ldots,\ \mathtt{cmt}_0[t-1]$ $(5216 - 63744$ bytes) and $\mathtt{cmt}_1[0],\ \ldots,\ \mathtt{cmt}_1[t-1]$ $(5216 - 63744$ bytes) utilizing the SeedTree, and has a good contribution in memory-footprint.

After the SeedTree generation, the Merkle tree T is constructed with commitments $\mathtt{cmt}_1[i], i \in \mathscr{I}_t$. Then Hash is applied on the commitments $\mathtt{cmt}_0$ and $\mathtt{cmt}_1$. The output is stored in d_{01} and sent as part of the signature. Next, the first challenge beta is generated using the message and commitments. After that, $\boldsymbol{\eta}'$, $\boldsymbol{u}'$ and beta are used to generate $\boldsymbol{y}$, which is utilized to construct the second challenge $\boldsymbol{b} = (\boldsymbol{b}[0],\ \ldots,\ \boldsymbol{b}[t-1])$. Here, $\boldsymbol{b}[i] \in \{0,\ 1\},\ \forall i \in \mathscr{I}_t$ and let us assume $J = \{i \in \mathscr{I}_t : \boldsymbol{b}[i] = 1\}$. Then, to ensure that all the seed tree leaves $\mathtt{Seed}[i],\ i \in J$ can be recomputed, SeedPath is calculated by

assembling intermediate seeds from GGM tree. Here, another complete binary tree $\boldsymbol{x}$, called `ReferenceTree` is generated from the digest $\boldsymbol{b}$, which decides which seed node will be published. Then, the intermediate seed $\texttt{SeedTree}[c - \texttt{missing\_nodes}[h']]$ will be published if all the leaf nodes corresponding to the subtree rooted as $\boldsymbol{x}[c]$ have to be published. `SeedPath` is also added as part of the signature together with $(\boldsymbol{y}_i,\ \boldsymbol{\sigma}_i)$ (or $(\boldsymbol{y}_i,\ \boldsymbol{\delta}_i)$), $\texttt{cmt}_1[i],\ i \in J$. Lastly, for the remaining leaves of the seed tree $i \notin J$, `MerkleProofs` is constructed with $\texttt{cmt}_0[i],\ i \notin J$ and sent as part of the signature.

Memory footprint of `CROSS_Sign`(): In the signature generation algorithm, when hash functions are computed, the hash inputs are large and are loaded into the main memory. We have shown that we do not need to load the complete input for hash computation. Rather, we can compute small portions of hash in part. Also, The variables in the signature algorithm are allocated memory when computations are needed, but once the computations are complete, the memory is not reclaimed, even if the data is no longer required for subsequent processes. Finally, in the Merkle proof generation process, the $\texttt{cmt}_0$ array is copied to the leaf nodes of the Merkle tree. After the tree generation, we compute the Merkle proof; thus, storing $\texttt{cmt}_0$ array requires considerably large memory ($5216 - 63744$ bytes). We have discussed our technique to avoid this extra memory consumption.

3 Our memory optimization of CROSS

This section describes our techniques to reduce the memory footprint of CROSS key generation, signature generation, and verification algorithms.

3.1 Our memory optimization of CROSS key generation

As we described earlier, the method `CSPRNG_mat` generate each of the matrix elements of the matrices $\boldsymbol{V} \in \mathbb{F}_p^{k\times(n-k)}$, $\boldsymbol{W} \in \mathbb{F}_z^{m\times(n-m)}$ sequentially and stores it. The vectors $\boldsymbol{\eta}$, $\boldsymbol{e}(= g^{\boldsymbol{\eta}})$ are also computed and stored. The public vector $\boldsymbol{s}$ is computed by multiplying the stored matrix $\boldsymbol{V}$ and vector $\boldsymbol{e}$. Similarly, for the `RSDPG` variant, the private vector $\boldsymbol{\zeta}$ is stored and multiplied with $\boldsymbol{W}$. The results of these multiplications are as follows:

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

$$\boldsymbol{\eta}[j] = \boldsymbol{\zeta}[m+j] + \sum_{i=0}^{m} \boldsymbol{\zeta}[i]\boldsymbol{W}[i,\ j], \ \text{where}\ 0 \le j \le n-m-1\,.$$

Observe that, each entry $\boldsymbol{V}[i,\ j]$ and $\boldsymbol{W}[i,\ j]$ is required only once to generate the vectors $\boldsymbol{s}$ and $\boldsymbol{\eta}$ respectively. Therefore, we compute $\boldsymbol{\zeta}$ before the computation of $\boldsymbol{W}$ and initialized each $\boldsymbol{\eta}[j]$ by $\boldsymbol{\zeta}[m+j]$. After that we generate (i,j)-th element of $\boldsymbol{W}$ say $w := \boldsymbol{W}[i,\ j]$, we can update $\boldsymbol{\eta}[j]$ as $\boldsymbol{\eta}[j] \leftarrow \boldsymbol{\eta}[j] + \boldsymbol{\zeta}[i] \cdot w$. Therefore, we do not need to store the complete matrix $\boldsymbol{W}$ at all. The final vector $\boldsymbol{\eta}$ is still the multiplication of matrix $\boldsymbol{W}$ and vector $\boldsymbol{\zeta}$. Similarly, multiplication of $\boldsymbol{V}$ and $\boldsymbol{e}$ is possible without storing $\boldsymbol{V}$. In Alg. 3, we have shown the matrix generation technique the authors of CROSS have used. We have modified the method to directly compute multiplication at the time of matrix generation. We have struck through the parts that we have omitted in our updated method, and the parts that we have included instead are coloured blue. As we discussed, to merge the generation process of matrix $\boldsymbol{V}$ and the computation of $\boldsymbol{s}$, only modification of the function `CSPRNG_mat` is not enough as we need to initialize $\boldsymbol{s}$ to some values of $\boldsymbol{\eta}$. We need to generate the vector $\boldsymbol{s}$ before the generation of $\boldsymbol{V}$. This step will not hamper the security, as the generation of $\boldsymbol{\eta}$ and $\boldsymbol{V}$ are independent of each other. Similarly, we also need to re-organize the part for the `RSDPG` variant. So, we have modified the steps of

Algorithm 2 $\texttt{CROSS\_Sign}(\texttt{Msg}, \texttt{Seed}_{\texttt{sk}})$

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

if RSDP **then** $(\boldsymbol{\eta},\ \boldsymbol{\zeta},\ \boldsymbol{H},\ \boldsymbol{M}_G) \leftarrow \texttt{ExpandPrivateSeed}\,(\texttt{Seed}_{\texttt{sk}})$
else $(\boldsymbol{\eta},\ \boldsymbol{\zeta},\ \boldsymbol{H},\ \boldsymbol{M}_G) \leftarrow \texttt{ExpandPrivateSeed}\,(\texttt{Seed}_{\texttt{sk}})$
$\texttt{MSeed} \xleftarrow{\$} \{0,\ 1\}^{\lambda}$, $\texttt{Salt} \xleftarrow{\$} \{0,\ 1\}^{2\lambda}$
if No_Tree **then**
    $(\texttt{Seed}[0],\ \ldots,\ \texttt{Seed}[t-1]) \leftarrow \texttt{ComputeRoundSeed}\,(\texttt{Mseed},\ \texttt{Salt})$
else
    $\texttt{Seed\_Tree} \leftarrow \texttt{SeedTreeGen}(\texttt{Mseed},\ \texttt{Salt})$
    $(\texttt{Seed}[0],\ \ldots,\ \texttt{Seed}[t-1]) \leftarrow \texttt{SeedTreeLeaves}\,(\texttt{Seed\_Tree})$
for $i = 0$ to $t-1$ **do**
    $(\texttt{Seed}_{\texttt{u}'},\ \texttt{Seed}_{\texttt{e}'}) \leftarrow \texttt{CSPRNG}\left(\texttt{Seed}[i],\ \{0,\ 1\}^{\lambda} \times \{0,\ 1\}^{\lambda}\right)$
    if RSDP **then** $\boldsymbol{\eta}'_i,\ \boldsymbol{u}'_i \leftarrow \texttt{CSPRNG}(\texttt{Seed}[i]||\texttt{Salt}||i+c,\ \mathbb{F}_z^n \times \mathbb{F}_p^n)$
    else
        $\boldsymbol{\zeta}',\ \boldsymbol{u}'_i \leftarrow \texttt{CSPRNG}(\texttt{Seed}[\texttt{i}]||\texttt{Salt}||i+c,\ \mathbb{F}_z^m \times \mathbb{F}_p^n)$
        $\boldsymbol{\delta}_i \leftarrow \boldsymbol{\zeta} - \boldsymbol{\zeta}',\ \ \boldsymbol{\eta}'_i \leftarrow \boldsymbol{\zeta}'\boldsymbol{M}_G$
    $\boldsymbol{\sigma}_i \leftarrow \boldsymbol{\eta} - \boldsymbol{\eta}'_i$
    $\boldsymbol{v}_{(\mathscr{I}_n)} = g^{\boldsymbol{\sigma}_{i(\mathscr{I}_n)}}$
    $\boldsymbol{u} = \boldsymbol{v} * \boldsymbol{u}'_i$ // $*$ represents component-wise product
    $\tilde{\boldsymbol{s}} \leftarrow \boldsymbol{u}\boldsymbol{H}$
    if RSDP **then** $\texttt{cmt}_0[i] \leftarrow \texttt{Hash}\,(\tilde{\boldsymbol{s}}||\boldsymbol{\sigma}_i||\texttt{Salt}||i+c+\texttt{dsc})$
    else $\texttt{cmt}_0[i] \leftarrow \texttt{Hash}\,(\tilde{\boldsymbol{s}}||\boldsymbol{\delta}_i||\texttt{Salt}||i+c+\texttt{dsc})$
    $\texttt{cmt}_1[i] \leftarrow \texttt{Hash}\,(\texttt{Seed}[i]||\texttt{Salt}||i+c+\texttt{dsc})$
$d_0,\ \texttt{Merkle\_Tree} \leftarrow \texttt{MerkleRoot}(\texttt{cmt}_0[0],\ \ldots,\ \texttt{cmt}_0[t-1])$
$d_1 \leftarrow \texttt{Hash}(\texttt{cmt}_1[0]||\ldots||\texttt{cmt}_1[t-1])$
$d_{01} \leftarrow \texttt{Hash}(d_0||d_1),\ \ d_m \leftarrow \texttt{Hash}(\texttt{Msg})$
$d_\beta \leftarrow \texttt{Hash}(d_m||d_{01}||\texttt{Salt})$
$\texttt{beta} \leftarrow \texttt{CSPRNG}\left(d_\beta,\ (\mathbb{F}_p^*)^t\right)$
for $i = 0$ to $t-1$ **do**
    $\boldsymbol{e}'_{(\mathscr{I}_n)} \leftarrow g^{\boldsymbol{\eta}_{i'(\mathscr{I}_n)}}$
    $\boldsymbol{y}_i \leftarrow \boldsymbol{u}'_i + \texttt{beta}[i]\boldsymbol{e}'_i$
$d_b \leftarrow \texttt{Hash}(\boldsymbol{y}_0||\ldots||\boldsymbol{y}_{t-1}||d_\beta)$
$\boldsymbol{b} \leftarrow \texttt{CSPRNG}\,(d_b,\ \mathcal{B}_w^t)$
if No_Tree **then**
    $\texttt{MerkleProofs} \leftarrow \texttt{MerkleProof}(\texttt{cmt}_0[0],\ \ldots,\ \texttt{cmt}_0[t-1],\ b)$
    $\texttt{SeedPath} \leftarrow \texttt{SeedTreePaths}(\texttt{Seed}, \boldsymbol{b})$
else
    $\texttt{MerkleProofs} \leftarrow \texttt{MerkleProof}(\texttt{Merkle\_Tree},\ b)$
    $\texttt{SeedPath} \leftarrow \texttt{SeedTreePaths}(\texttt{Seed\_Tree}, \boldsymbol{b})$
$j \leftarrow 0$
for $i = 0$ to $t-1$ **do**
    if $\boldsymbol{b}[i] = 0$ **then**
        if RSDP **then** $\texttt{rsp}_0[j] \leftarrow (\boldsymbol{y}_i,\ \boldsymbol{\sigma}_i)$
        else $\texttt{rsp}_0[j] \leftarrow (\boldsymbol{y}_i,\ \boldsymbol{\delta}_i)$
        $\texttt{rsp}_1[j] \leftarrow \texttt{cmt}_1[i]$
        $j \leftarrow j+1$
return $\texttt{signature} \leftarrow \texttt{Salt}||d_{01}||d_b||\texttt{MerkleProofs}||\texttt{SeedPath}||\texttt{rsp}_0||\texttt{rsp}_1$

Algorithm 3 `CSPRNG_mat_update`(`buf`, $\mathcal{D}(=\mathbb{F}_q^{k\times n})$, $\boldsymbol{a}$)

Input: A buffer `buf`, a domain $\mathcal{D} = \mathbb{F}_q^{k\times n}$ of the matrix that is generated from the buffer, a vector $\boldsymbol{a}$.

Output: ~~The matrix A is generated from `buf`~~ The resultant multiplication $\boldsymbol{b} = \boldsymbol{aA}$

1: `mask` $= 2^{\log q} - 1$
2: `placed` $= 0$, $i = 8$
3: $j = 64$, $i' = 0$, $j' = 0$
4: `s_buf` $=$ `buf`$[0]$
5: **while** `placed` $< kn$ **do**
6: **if** $j \leq 32$ **then**
7: `r_buf` $=$ `buf`$[i]$
8: $i = i + 4$
9: `s_buf` $= ($`s_buf` $\vee$ `r_buf`$) \times 2^j$
10: $j = j + 32$
11: ~~$\boldsymbol{v}$[placed, :]s_buf $\wedge$ mask~~ ▷ This value is equivalent to $\boldsymbol{A}[i',\ j']$
12: **if** ~~$\boldsymbol{v}$[placed] $< q$~~ (`s_buf` $\wedge$ `mask`) $< q$ **then**
13: $\boldsymbol{s}[j'] = \boldsymbol{s}[j'] + (\boldsymbol{\eta}[i'] \times$ (`s_buf` $\wedge$ `mask`))
14: $j' = j' + 1$
15: **if** $j' = n - k$ **then** $j' = 0$, $i' = i' + 1$
16: `placed` $=$ `placed` $+ 1$
17: `s_buf` $=$ `s_buf`$/2^{\log q}$
18: $j = j - \log q$
19: **else**
20: `s_buf` $=$ `s_buf`$/2$, $j = j - 1$

`CROSS_KeyGen` in Alg. 1 and provided the modified version as `Updated_CROSS_KeyGen` in Alg. 4.

3.2 Optimizing signature generation and verification

This section explains our techniques for memory-efficient signature generation and verification algorithms. Although this work explains the strategy for memory optimization of the signature generation procedure, we have used similar techniques in the verification.

3.2.1 Just-in-time strategy for `HASH` function

The latest CROSS (v1.2) [BBB+24] uses `SHAKE` as the `HASH` function. It takes an input buffer (message), say `buf`, of arbitrary length, say `len`, and returns a fixed length string as output. However, in the CROSS signature, the hash input is first stored separately e.g., the arrays of $\texttt{cmt}_1$ ($5216 - 63744$ bytes) and $\boldsymbol{y}$ ($16830 - 242968$ bytes). However, we do not have to generate the complete hash input at once. Instead, we can only generate the first few commitments and proceed with hash generation. After that, the next batch of commitments is generated and fed to the hash generation procedure. This process is repeated until all commitments are passed.

`SHAKE` has a parameter r, called the rate of the hash. `SHAKE` takes an input message `buf` of length `len`. Then, it divides the whole buffer into blocks of r bits and updates the state. It works as follows:

- First, `state` is initialized using the function `xof_shake_init`. After initialization, the function `xof_shake_update` takes the buffer `buf` and the length of the buffer

Algorithm 4 `Updated_CROSS_KeyGen()`

Input: None
Output: Public key: $(\texttt{Seed}_{\texttt{pk}},\ \texttt{s})$, Private key: $\texttt{Seed}_{\texttt{sk}}$

$\texttt{Seed}_{\texttt{sk}} \xleftarrow{\$} \{0,\ 1\}^{\lambda}$
$(\texttt{Seede},\ \texttt{Seed}_{\texttt{pk}}) \leftarrow \texttt{CSPRNG}\Big(\texttt{Seed}_{\texttt{sk}},\ \{0,\ 1\}^{\lambda} \times \{0,\ 1\}^{\lambda}\Big)$
if `RSDP` **then**
    $\eta \leftarrow \texttt{CSPRNG}(\texttt{Seede},\ \mathbb{F}_z^n)$
    $\texttt{state} \leftarrow \texttt{initialized\_csprng}(\texttt{seed}_{\texttt{pk}},\ \texttt{len})$
    $\texttt{buffer} \leftarrow \texttt{csprng\_randombytes}(\texttt{state},\ r)$
if `RSDPG` **then**
    $\boldsymbol{\zeta} \leftarrow \texttt{CSPRNG}(\texttt{Seede},\ \mathbb{F}_z^m)$
    $\texttt{state} \leftarrow \texttt{initialized\_csprng}(\texttt{seed}_{\texttt{pk}},\ \texttt{len})$
    $\texttt{buffer} \leftarrow \texttt{csprng\_randombytes}(\texttt{state},\ r)$
    $\texttt{buffer1} \leftarrow \texttt{csprng\_randombytes}(\texttt{state},\ r')$
    $\boldsymbol{\eta}_{(\mathscr{I}_{n-m})} = 0$
    $\boldsymbol{\eta}_{(\{n-m,\cdots,n-1\})} = \boldsymbol{\zeta}_{(\mathscr{I}_m)}$
    $\boldsymbol{\eta} \leftarrow \texttt{CSPRNG\_mat\_update}(\texttt{buffer1},\ \mathbb{F}_z^n,\ \boldsymbol{\zeta})$
$\boldsymbol{e}_{(\mathscr{I}_n)} = g^{\boldsymbol{\eta}_{(\mathscr{I}_n)}}$
$\boldsymbol{s}_{(\mathscr{I}_{n-k})} = \boldsymbol{e}_{(\{k,\cdots,\ n-1\})}$
$\boldsymbol{s} \leftarrow \texttt{CSPRNG\_mat\_update}(\texttt{buffer},\ \mathbb{F}_p^{k\times(n-k)},\ \boldsymbol{\eta})$
return Public key: $(\texttt{Seed}_{\texttt{pk}},\ \boldsymbol{s})$, Private key: $\texttt{Seed}_{\texttt{sk}}$

`len` as inputs and modifies the `state`. For each message, this function takes the block of r bits and XORs the first r bits of `state` with it. Then, it permutes the updated state to introduce a pseudorandomness. We continue this process until the remaining bits in the buffer are less than r bits, after which the remaining buffer is only XOR-ed with the `state`.

- Next, the function `xof_shake_final` function finalize the value of `state`. Finally, the function `xof_shake_extract` is used to extract the fixed length digest value from the state `state`.

We have observed that the functions `xof_shake_init`, `xof_shake_final` and `xof_shake_extract` take equal memory for different sizes of input buffers. Therefore, we keep these functions unchanged. However, the function `xof_shake_update` takes the complete buffer as input, but what can be observed is that only a small part of the buffer is used at each step of the hash computation. Therefore, we do not need to generate the complete buffer at once. We have used a technique that only generates a small amount of the buffer and operates on it before generating the next part of the buffer. This approach helps us reduce the memory required to store the complete buffer. We have explained the technique in detail.

- Step 1: We initialize `state`.
- Step 2: We generate the first few commitments and store them until the total length is $\geq r$. A small part of the generated commitments may not fit in the buffer. Therefore, we fix the buffer size to $\ell = r +$ *digest length* bits to store the excess bits.
- Step 3: If the loaded elements in buffer ℓ_1 bits, and $\ell_1 \geq r$ in the previous step, then we set our flag "Flag" to 0. If $\ell_1 = 0$ then we go to Step 6. Otherwise, we set "Flag" to 1.

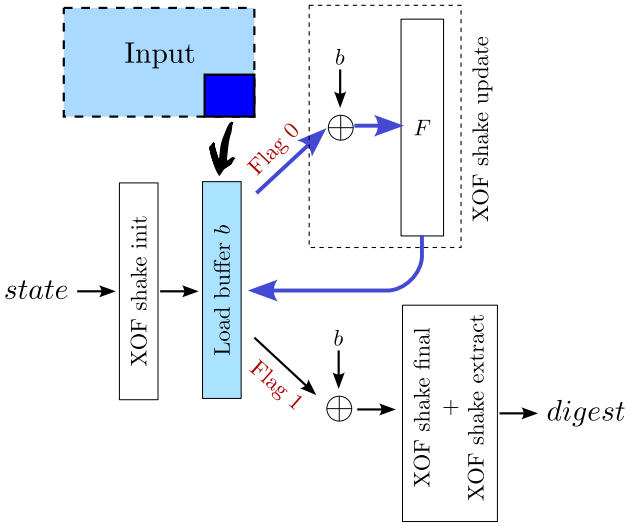

Figure 2: Our streamlined integration of Hash function with input array. Here F denotes the `KeccakF1600_StatePermute` function.

- Step 4: If the Flag is 0, then we use the method `xof_shake_update` and update the `state`. Next put the stored excess part from the previous step at the beginning of the buffer and generate the next few commitments to fill the rest of the buffer and repeat the process from step 2.
- Step 5: If the Flag is 1 then the input of the buffer is XORed with the `state`.
- Step 6: After all the commitments are generated and went through the above process we apply the final `xof_shake_final` and `xof_shake_extract` function to get the digest.

Fig. 2 illustrates the method we have implemented. Our technique does not generate a new hash digest but generates the same digest as the original method.

3.2.2 Efficient memory utilization through execution flow analysis

We performed a thorough analysis of the execution flow of the signature generation algorithm to remove the redundant memory usage leading to a compact implementation. Note that we need to keep a value stored in the memory only if it is going to be used by any operation in subsequent steps. Otherwise, we can overwrite it with any different value without affecting the correctness. We observed that in `CROSS_Sign` (Alg 2), the value $\boldsymbol{y}_i \in \mathbb{F}_p^n$ is computed as $\boldsymbol{y}_i = \boldsymbol{u}'_i + \texttt{beta}[i]\boldsymbol{e}'_i$ (line: 29 of Alg 2), where $\boldsymbol{u}'_i,\ \boldsymbol{e}'_i \in \mathbb{F}_p^n$, $\texttt{beta}[i] \in \mathbb{F}_p^*$, for each $i \in \mathscr{I}_n$. Also, the value $\boldsymbol{u}'_i$ is not used after the above computation. Therefore, we can overwrite $\boldsymbol{u}'_i$ with the value of $\boldsymbol{y}_i$ and the resultant computation would be the same. Therefore, replacing the variables $\boldsymbol{y}_i$ by $\boldsymbol{u}'_i$ saves us $16830 - 242968$ bytes (size $tn \log p$) memory as $\boldsymbol{y} = (\boldsymbol{y}_0, \cdots,\ \boldsymbol{y}_{t-1}) \in (\mathbb{F}_p^n)^t$.

In Alg 2, for each $i \in \mathscr{I}_t$ the vector $\boldsymbol{\sigma}_i$ is computed as $\boldsymbol{\sigma}_i = \boldsymbol{\eta} - \boldsymbol{\eta}_i$ (line: 15). The vector $\boldsymbol{\sigma}_i$ is used only to compute the commitment $\texttt{cmt}_0[i]$ for `RSDP` parameter set and to compute $\boldsymbol{v}$. Therefore, we do not need to store the vector $\boldsymbol{\sigma}_i$ once we have generated vector $\boldsymbol{v}$ and commitment $\texttt{cmt}_0[i]$. However, later we need to store the i-th value $\boldsymbol{\sigma}_i$ for the output response component $\texttt{rsp}_0[i]$ for `RSDP` parameter set which satisfies $\boldsymbol{b}[i] = 0$, where $\boldsymbol{b}$ is a fixed length random digest vector. For this reason, all the vectors $\boldsymbol{\sigma}_i,\ \forall i \in \mathscr{I}_t$ are

stored. However, we observe that the cardinality of the set $\{i \in \mathscr{I}_t : \boldsymbol{b}[i] = 0\}$ rather small (usually in the range $22 - 158$). So, if we re-compute the $\boldsymbol{\sigma}_i$ corresponding to $\boldsymbol{b}[i] = 0$, then some computations needed to be add-on in lieu of storing the vector $\boldsymbol{\sigma}$. This way, we can save $16720 - 242717$ bytes of (size $(t-1)n \log p$) memory by incurring small computation costs.

3.2.3 Re-formulate the Merkle root generation

We have found that two of the most memory-intensive functions in the `CROSS_Sign` algorithm are `Merkle_Root` and `Merkle_Proof` generation. However, this Merkle tree generation is not used for "`fast`" parameter sets of CROSS. Therefore, the techniques described in this section do not contribute towards memory reduction for the "`fast`" parameters.

The function `Merkle_Root` uses a `Merkle_Tree` and the commitment array $\texttt{cmt}_0$ to compute the root. Array $\texttt{cmt}_0$ and the Merkle tree occupy $t\times$ *digest length* ($5216 - 63744$ bytes) and $(2t-1)\times$ *digest length* ($10400 - 127424$ bytes) amount of memory, respectively. Also, the function `Merkle_Proof` uses the $\texttt{cmt}_0$ array and Merkle tree with another binary flag tree with $2t - 1$ nodes. These two functions (`Merkle_Root` and `Merkle_Proof`) use a huge amount of memory. Therefore, the memory reduction in this part will significantly reduce the scheme's memory consumption.

The `Merkle_Root` function copies each element of the vector $\texttt{cmt}_0$ one by one to the leaf nodes of the `Merkle_Tree` in an intermediate step. This is the only location where the stored commitment vector $\texttt{cmt}_0$ is used. We can do the same work by copying the i-th element $\texttt{cmt}_0[i]$ of $\texttt{cmt}_0$ to the i-th leaf node of `Merkle_tree` just after $\texttt{cmt}_0[i]$ generation. To maintain the execution flow of the algorithm, we have to re-organize the steps of the `Merkle_Root` function. We first allocate the required memory for the Merkle tree and then start generating the commitments $\texttt{cmt}_0$. After each element of $\texttt{cmt}_0$ is generated, it is stored in the corresponding leaf node of the Merkle tree. This way, we do not need to allocate memory to store $\texttt{cmt}_0$. Finally, we can generate the Merkle root using the remaining parts of the Merkle root generation after the storing process of the leaf nodes of `Merkle_Tree`.

3.3 Countermeasure

The paper [MAKK24] proposed a countermeasure strategy for LESS against ZKFault and claimed that it can be applied to the initial version of CROSS. However, recently, the CROSS signature scheme's algorithms and implementation have been modified significantly in the updated version. Unfortunately, the prior fault attack is still possible on the new version of CROSS. Therefore, we adapted the countermeasure strategy against the ZKFault attack for the current version of CROSS, along with the memory-optimized implementation in this work. We elaborate on this below.

3.3.1 ZKFault attack on CROSS

The binary vector $\boldsymbol{b}$ has been checked twice in Alg. 2 (lines: 34/37 depending on `RSDP`/`RSDPG` variant and 40). First, whether $\boldsymbol{b}[i] = 1$ is checked, and if satisfies then the information of $\texttt{Seed}[i]$ is published using the function `SeedTreePaths`. Second, whether $\boldsymbol{b}[i] = 0$ is checked, and if satisfies then $\texttt{rsp}_0$ is computed (lines:41-42 in Alg. 2). Let us assume that for some i-th location, $\boldsymbol{b}[i]$ is zero. Then, depending on the `RSDP` or `RSDPG` parameter set, the pair $(\boldsymbol{y}_i,\ \boldsymbol{\sigma}_i)$ or $(\boldsymbol{y}_i,\ \boldsymbol{\delta}_i)$ is stored in $\texttt{rsp}_0$. The $\texttt{Seed}[i]$ information will not be public in this case. However, if a fault is injected in `SeedTreePaths` algorithm and $\texttt{Seed}[i]$

information is retrieved then $\boldsymbol{\eta}'_i$ or $\boldsymbol{\zeta}'$ can be calculated from $\texttt{Seed}[i]$. These values help us to find the secret vector $\boldsymbol{\eta}$ or $\boldsymbol{\zeta}$ by computing $\boldsymbol{\eta} = \boldsymbol{\sigma}_i + \boldsymbol{\eta}'_i$ for RSDP version (or $\boldsymbol{\zeta} = \boldsymbol{\delta}_i + \boldsymbol{\zeta}'$ for RSDPG). The security of the scheme will be compromised as the attacker can generate a valid signature of any message from recovered $\boldsymbol{\eta}$ or $\boldsymbol{\zeta}$.

In the CROSS version v1.2 [BBB+24], SeedTreePaths is defined for two cases: one is for "fast" parameters, which does not use any tree construction, and the remaining parameter sets that use SeedTree. We explain the attack for both cases as follows:

Attack on "fast" parameters: For each $i \in \mathscr{I}_t$, this function SeedTreePaths checks whether $\boldsymbol{b}[i] = 1$. If $\boldsymbol{b}[i] = 1$ for some $i \in \mathscr{I}_t$, then it stores $\texttt{Seed}[i]$ in SeedPath, which is a component of signature. In this case, if we skip the checking condition for some i which satisfies $\boldsymbol{b}[i] = 0$, then we will get corresponding seed $\texttt{Seed}[i]$. The next checking condition, we will get $(\boldsymbol{y}_i,\ \boldsymbol{\sigma}'_i)$. As mentioned earlier if we can get both $\texttt{Seed}[i]$ and the corresponding pair $(\boldsymbol{y}_i,\ \boldsymbol{\sigma}'_i)$ (or $(\boldsymbol{y}_i,\ \boldsymbol{\delta}'_i)$), then the ZKFault attack will work.

Attack on other parameters: These schemes use a technique that compactly store the Seed's in SeedPath. The function SeedTreePaths computes a complete binary tree called as Reference_Tree $\boldsymbol{x}$ using another function compute_seeds_to_publish. The leaf nodes of $\boldsymbol{x}$ are assigned by the digest vector $\boldsymbol{b}$ and the remaining nodes are computed as $\boldsymbol{x}[i] = \boldsymbol{x}[\texttt{Left\_child}(i)] \wedge \boldsymbol{x}[\texttt{Right\_child}(i)]$. After the computation of $\boldsymbol{x}$, for each node $\boldsymbol{x}[i]$ (from top to bottom), it checks whether $\boldsymbol{x}[i] = 1$ and $\boldsymbol{x}[\texttt{Parent}(i)] = 0$. If satisfies, then SeedPath stores $\texttt{Seed\_Tree}[i - \texttt{missing\_nodes}[h']]$, where h' is the height of the node $\boldsymbol{x}[i]$. Let $\boldsymbol{x}[i] = \boldsymbol{b}[i-l+1] = 0$ for some i-th leaf node of the of the tree $\boldsymbol{x}$. But, if we change the value of the node $\boldsymbol{x}[i]$ to 1 by injecting ZKFault, then the SeedTreePaths stores the information of the seed $\texttt{Seed}[i-l+1] = \texttt{Seed\_Tree}[i - \texttt{missing\_nodes}[\log t]]$ in SeedPath. Therefore, the attack will work.

3.3.2 Countermeasure against ZKFault attack

Now, we describe the countermeasure strategy against ZKFault for the fast and other parameter sets of CROSS.

Countermeasure for "fast" parameters: To counter this attack, we will check each value of $\boldsymbol{b}$ once. For each $i \in \mathscr{I}_t$, if we observe that $\boldsymbol{b}[i] = 0$, then we will store $(\boldsymbol{y}_i,\ \boldsymbol{\sigma}_i)$ or $(\boldsymbol{y}_i,\ \boldsymbol{\delta}_i)$ in $\texttt{rsp}_0$, otherwise we will store $\texttt{Seed}[i]$ in SeedPath. Since for each digest value $\boldsymbol{b}[i]$ either $(\boldsymbol{y}_i,\ \boldsymbol{\sigma}_i)$(or corresponding $(\boldsymbol{y}_i,\ \boldsymbol{\delta}_i)$) or $\texttt{Seed}[i]$ will be published; therefore the attacker can not get both values at a time. Therefore, this attack can be prevented.

Countermeasure for other parameters: To prevent the ZKFault attack in this scenario, we have to compute SeedPath and two responses by checking each digest element only once. We can observe that the path of each leaf node of $\boldsymbol{x}$ toward the root will be of the form 1^r0^s due to the computation process of the tree $\boldsymbol{x}$. We check this path of each leaf node starting from the leftmost node of $\boldsymbol{x}$ ($\boldsymbol{x}[l-1]$). Then, we must find the location of the last 1, which contains the index c' and the node height (h') that satisfies the storage condition. Suppose that there is no 1 in this path for node $\boldsymbol{x}[i]$. In that case, the corresponding element of the digest vector $\boldsymbol{d}[i-l+1]$ is zero. We store the corresponding $\texttt{rsp}_0$ and $\texttt{rsp}_1$. Otherwise, we will store $\texttt{Seed\_Tree}[c' - \texttt{missing\_nodes}[h']]$ in SeedPath and ignore the next $2^{\log t - h'}$ many nodes. The algorithm CROSS_Countermeasure_Others in Alg. 5 shows the exact process of the $\texttt{rsp}_0$, $\texttt{rsp}_1$ and SeedPath generation.

Let a value of a node $\boldsymbol{x}[i]$ be changed from 0 to 1 by the ZKFault attack. Let $\boldsymbol{x}[i'']$ be the leftmost leaf node corresponding to the subtree rooted as $\boldsymbol{x}[i]$ and $\boldsymbol{x}[i']$ be the highest ancestor of node $\boldsymbol{x}[i]$ (with height h) having the value 1. Therefore, SeedPath must store the seed $\texttt{SeedTree}[i' - \texttt{missing\_nodes}[h]]$ and the next $2^{\log t - h}$ leaf nodes will be skipped. Therefore, we cannot get both the values $(\boldsymbol{y}_i,\ \boldsymbol{\sigma}_i)$ and $\texttt{Seed}[i]$ from this fault preventing the attack.

Algorithm 5 `CROSS_Countermeasure_Others()`

Input: $\boldsymbol{b}$: the digest vector, For `RSDP` parameter set, $(\boldsymbol{y}_i,\ \boldsymbol{\sigma}_i)$, and $(\boldsymbol{y}_i,\ \boldsymbol{\sigma}_i)$ otherwise for all $0 \leq i \leq t-1$.
Output: Compute $\texttt{rsp}_0$, $\texttt{rsp}_1$, `SeedPath`

$\boldsymbol{x} \leftarrow$ `compute_seeds_to_publish`$(\boldsymbol{b})$
$i = 0,\ j = 0,\ j' = 0$
while $i < t$ **do**
  $c = l - 1 + i,\ c' = c,\ h' = \log t$
  `flag` $= 0,\ h = \log t$
  while $c \neq 0$ **do**
    if $\boldsymbol{x}[c] = 1$ **then**
      $c' = c,\ h' = h,$ `flag` $= 1$
    $c =$ `Parent`$(c),\ h = h - 1$
  if `flag` $= 0$ **then**
    if `RSDP` **then** $\texttt{rsp}_0[j] \leftarrow (\boldsymbol{y}_i,\ \boldsymbol{\sigma}_i)$
    else $\texttt{rsp}_0[j] \leftarrow (\boldsymbol{y}_i,\ \boldsymbol{\delta}_i)$
    $\texttt{rsp}_1[j] \leftarrow \texttt{cmt}_1[i]$
    $j \leftarrow j + 1,\ i \leftarrow i + 1$
  else
    `SeedPath`$[j'] =$ `Seed_Tree`$[c' -$ `missing_nodes_before`$[h']$
    $j' \leftarrow j' + 1,\ i \leftarrow i + 2^{\log t - h'}$
Return $\texttt{rsp}_0$, $\texttt{rsp}_1$, `SeedPath`

4 Results

In this section, we present the results of our implementation of CROSS on the STM32 Nucleo-144 development board with 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 10.3.1.

We provide the improvement in memory consumption of CROSS `RSDPG`-1 after applying the optimizations described in this paper in Table 1. S-1 represents the key generation optimization steps described in Sec. 3.1 and the signature generation and verification optimization steps illustrated in Sec. 3.2.1. S-2 and S-3 denote the optimization strategy for the signature generation and verification explained in Sec. 3.2.2 and Sec. 3.2.3, respectively. CM represents the countermeasure of ZKFault described in Sec. 3.3. We also provide the implementation results of the original CROSS-v1.2, along with our improvement for comparison. We observe that S-1 provides 43% improvements in memory consumption for key generation in all the versions. Signature generation improves memory consumption by 27%/15%/16% for `fast`/`balanced`/`small` versions in S-1. Verification improves memory consumption by 35%/10%/11% for `fast`/`balanced`/`small` versions in S-1. Signature generation and verification improves memory consumption by 37%/32%/33% and 56%/41%/44%, respectively, for `fast`/`balanced`/`small` versions in S-2 compared to S-1. In S-3, signature generation and verification improves memory consumption by 9%/9% and 19%/21%, respectively, for `balanced`/`small` versions. Finally, after including the countermeasure, the overall memory requirements of our CROSS implementation with respect to the original one decreased by 43% for the key generation, 54%/47%/49% for signature generation in `fast`/`balanced`/`small` versions, and 71%/57%/60% for the verification in `fast`/`balanced`/`small` versions with negligible performance overhead.

In Table 2, we present the memory consumption of our secure implementations of all the versions of CROSS for all the security categories and compare them with the original

Table 1: Improvement steps of our implementations for CROSS RSDPG-1 on ARM Cortex-M4. bal. denotes the balanced version.

SDPG I	Opt.	Steps	Performance (kcycles)			Memory (bytes)		
			KG	Sign	Verify	KG	Sign	Verify
This work	fast	S-1	248	13,586	7,958	2,216	67,600	30,184
		S-2	248	13,544	7,961	2,216	42,408	13,344
		S-3	248	13,544	7,960	**2,216**	**42,408**	**13,344**
		CM	248	13,548	7,960	**2,216**	**42,408**	**13,344**
v1.2[BBB+24]			182	12,710	7,958	3,920	93,096	46,772
This work	bal.	S-1	248	25,021	14,972	2,216	123,792	64,996
		S-2	248	25,107	14,954	2,216	83,784	38,252
		S-3	248	25,069	14,954	**2,216**	**76,568**	**31,120**
		CM	248	25,101	14,961	**2,216**	**76,864**	**31,120**
v1.2[BBB+24]			182	24,624	15,030	3,920	145,484	71,860
This work	small	S-1	248	88,592	53,674	2,216	431,220	220,132
		S-2	248	87,905	53,543	2,216	287,668	124,316
		S-3	248	88,238	53,429	**2,216**	**261,620**	**98,660**
		CM	248	88,297	53,409	**2,216**	**263,356**	**98,668**
v1.2[BBB+24]			182	86,124	53,618	3,920	512,284	247,092

Table 2: Comparison of memory consumption (in bytes) between our implementations of different versions of CROSS with the state-of-the-art implementation on ARM Cortex-M4. KG denotes the key generation algorithm, bal. denotes the balanced version.

Algo. & sec.	Opt.	CROSS-v1.2[BBB+24]			This work		
		KG	Sign	Verify	KG	Sign	Verify
RSDP 1	fast	7,984	120,416	58,296	**4,240**	**59,136**	**15,220**
	bal.	7,984	203,436	92,524	**4,240**	**99,368**	**33,480**
	small	-	-	-	**4,240**	**352,004**	**108,772**
RSDP 3	fast	16,616	264,160	127,024	**8,352**	**158,024**	**61,100**
	bal.	16,616	471,268	211,444	**8,352**	**227,552**	**73,480**
	small	-	-	-	**8,352**	**514,144**	**157,796**
RSDP 5	fast	-	-	-	**14,368**	**227,864**	**51,648**
	bal.	-	-	-	**14,368**	**386,680**	**121,688**
	small	-	-	-	-	-	-
RSDPG 1	fast	3,920	80,664	43,568	**2,216**	**42,408**	**13,344**
	bal.	3,920	145,484	71,860	**2,216**	**76,864**	**31,120**
	small	3,920	512,284	247,092	**2,216**	**263,356**	**98,660**
RSDPG 3	fast	7,336	174,680	93,368	**3,936**	**90,144**	**26,528**
	bal.	7,336	223,656	110,604	**3,936**	**118,320**	**48,344**
	small	-	-	-	**3,936**	**417,164**	**155,284**
RSDPG 5	fast	11,312	306,368	162,140	**6,296**	**156,096**	**42,076**
	bal.	11,312	412,636	203,212	**6,296**	**215,384**	**84,352**
	small	-	-	-	**6,296**	**577,468**	**214,108**

CROSS-v1.2 [BBB+24] implementations on the PQM4 framework [KKPY24]. Thanks to our memory optimizations, we could run all parameter sets except one on the board, whereas only 11 out of 18 parameter sets of CROSS-v1.2 fit on the board. Finally, we show 43-50% decrease in memory requirement for the key generation, 40-52% for the signature generation, and 52-74% for the verification compared to the original CROSS.

5 Conclusion

CROSS uses a novel zero-knowledge framework with a quantum hard problem to design a PQ DS scheme. Compared to the other DS schemes such types of constructions are relatively newer has not been explored much. Many practical aspects of such designs, such as efficient and secure implementations, side-channel resistance, and countermeasures, etc., have not been investigated properly yet. Due to their structures, these schemes often have huge memory requirements. We would like to note that our optimization methods are not limited to CROSS and can be applied to other code-based schemes that use similar ZK frameworks, like LESS [BMPS20]. It is another second-round candidate in the NIST additional signature standardization competition. Due to its massive dynamic memory allocation of the commitment array, the implementation of LESS can not run on the board[3]. We believe that our memory optimization process will help resolve LESS's memory issues. As memory-optimized implementations are essential for real-world applications, we believe this work will benefit the overall advancement of PQC DS schemes.

Acknowledgements

This work was partially supported by Horizon 2020 ERC Advanced Grant (101020005 Belfort), CyberSecurity Research Flanders with reference number VR20192203, BE QCI: Belgian-QCI (3E230370) (see beqci.eu), Intel Corporation, Secure Implementation of Post-Quantum Cryptosystems (SECPQC) DST-India, BELSPO, Google India Research and IIT Kanpur initiation grant. Puja Mondal is supported by C3iHub, IIT Kanpur. Supriya Adhikary is supported by the Prime Minister's Research Fellowship (PMRF), India.

References

[BBB+24] 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, February 2024.

[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.

[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.

[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 2129–2146, New York, NY, USA, 2019. Association for Computing Machinery.

[3] https://github.com/mupq/pqm4/issues/278

[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.

[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):792–807, August 1986.

[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.

[KKPY24] Matthias J. Kannwischer, Markus Krausz, Richard Petri, and Shang-Yi Yang. pqm4: Benchmarking NIST additional post-quantum signature schemes on microcontrollers. Cryptology ePrint Archive, Paper 2024/112, 2024.

[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.

[MAKK24] Puja Mondal, Supriya Adhikary, Suparna Kundu, and Angshuman Karmakar. ZKFault: Fault attack analysis on zero-knowledge based post-quantum digital signature schemes. In Advances in Cryptology - ASIACRYPT, 2024. just accepted.

[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.