CROSS

Codes and Restricted Objects Signature Scheme

Submission to the NIST Post-Quantum Cryptography
Standardization Process

Algorithm Specifications and Supporting Documentation

Version 2 - January 31, 2025

⊗ Marco Baldi, Polytechnic University of Marche, Department of Information Engineering

⊗ Alessandro Barenghi, Politecnico di Milano, Department of Electronics, Information and Bioengineering

⊗ Michele Battagliola, Università degli Studi di Trento, Department of Mathematics

⊗ Sebastian Bitzer, Technical University of Munich, Institute for Communications Engineering

⊗ Marco Gianvecchio, Politecnico di Milano, Department of Electronics, Information and Bioengineering

⊗ Patrick Karl, Technical University of Munich, Chair of Security in Information Technology

⊗ Felice Manganiello, Clemson University, School of Mathematical and Statistical Sciences

⊗ Alessio Pavoni, Polytechnic University of Marche, Department of Information Engineering

⊗ Gerardo Pelosi, Politecnico di Milano, Department of Electronics, Information and Bioengineering

⊗ Paolo Santini, Polytechnic University of Marche, Department of Information Engineering

⊗ Jonas Schupp, Technical University of Munich, Chair of Security in Information Technology

⊗ Edoardo Signorini, Telsy S.p.A.

⊗ Freeman Slaughter, Clemson University, School of Mathematical and Statistical Sciences

⊗ Antonia Wachter-Zeh, Technical University of Munich, Institute for Communications Engineering

⊗ Violetta Weger, Technical University of Munich, Department of Mathematics

Submitters: The team above, with names listed alphabetically, is the principal submitter. There are no auxiliary submitters.

Inventors/Developers: Same as the principal submitter.

Implementation Owners: The submitters.

Email Address (preferred): info@cross-crypto.com

Postal Address and Telephone:
Paolo Santini
Polytechnic University of Marche
Department for Communications Engineering
Brecce Bianche 12
60131 Ancona
Italy
Tel: +39 071 2204128

Backup Contact Telephone and Address:
Sebastian Bitzer, Violetta Weger
Technical University of Munich
Institute for Communications Engineering
Theresienstraße 90
80333 Munich
Germany
Tel: +498928929051

Signature: (See “Statement by Each Submitter” or “Cover Sheet”)

Contents

1 Design Rationale and Notation **5**

1.1 CROSS in a Nutshell . . . 5

1.2 Notation . . . 5

1.3 Basics . . . 6

2 Procedural Description of CROSS-ID and CROSS **10**

2.1 CROSS-ID . . . 10

2.2 CROSS Protocol . . . 13

2.2.1 Key Generation . . . 13

2.2.2 Signature Generation . . . 14

2.2.3 Verification . . . 18

2.3 Auxiliary Primitives . . . 20

3 Security **22**

3.1 Hardness of Restricted Decoding . . . 22

3.1.1 Underlying Hardness Assumptions . . . 22

3.1.2 Combinatorial Solvers for R-SDP . . . 23

3.1.3 Algebraic Solvers for R-SDP . . . 24

3.1.4 Solvers for R-SDP(G) . . . 24

3.2 Security of the Protocol . . . 25

3.2.1 Forgery Attacks . . . 25

3.2.2 Security Proof . . . 26

4 Parameters and Expected Security Strength **26**

5 Implementation Techniques **27**

5.1 Symmetric Primitives . . . 27

5.2 Seed- and Merkle Tree . . . 30

5.2.1 Tree Structures . . . 30

5.2.2 Tree Algorithms . . . 32

5.3 Parallelization of SHAKE . . . 39

5.4 Packing and Unpacking: . . . 40

5.5 Efficient arithmetic for $\mathbb{F}_7$, $\mathbb{F}_{127}$, and $\mathbb{F}_{509}$ . . . 42

5.6 Implementation Attacks . . . 43

6 Detailed Performance Analysis **43**

7 Known Answer Tests **44**

8 Advantages and Limitations **45**

9 Bibliography **46**

Change Log

This section summarizes the changes corresponding to different CROSS specification documents.

Version 2: We have changed the specification document to provide an operative summary of the CROSS features, together with operative descriptions and benchmarks, with the intent of improving readability. To this end, we have moved detailed explanations on

- combinatorial and algebraic attacks from Section 3.1,
- EUF-CMA security proof and forgery attacks from Sections 3.2, 3.2.1

to a separate security guide, available at `https://cross-crypto.com/`. Additionally, we reduced the amount of mathematical and cryptographic background given in Section 1.

We unified the notation across the entire document, both in sequence diagrams and the procedural descriptions of key generation, signing, and verification.

With respect to version 1, version 2 of the specification contains the following changes:

Protocol and parameters

1. We present a security proof for the ZK protocol and the proof of EUF-CMA security of the CROSS signature scheme in [39].
2. We include a novel forgery attack in Section 3.2.1 derived from [9]. This attack does not depend on R-SDP, nor on R-SDP(G), but only on the non-interactivity of the transformed ZK protocol. This version of the attack improves upon the forgery presented in the previous versions of this specification by exploiting, in a better way, the use of fixed-weight challenges.
3. Due to the novel forgery attack, we present novel parameters in Section 4. This results in an increase in signature size for the *small* parameter sets of 12% to 24%. The signature sizes for the *balanced* parameter increase by up to 7% depending on the parameter set while decreasing up to 7% for some other parameter sets due to a more precise bound for the attack cost. This bound also results in a decrease in signature size for the *fast* parameter sets of up to 3.3%.
4. An overview of combinatorial solvers and the algebraic solver of Beullens, Briaud, and Øygarden [15] is given in Section 3; for more details, see [39].
5. We are using a new bound for the size of the seed path and Merkle proof, see Section 2.2.2. The new bound now involves also the Hamming weight of the binary representation of the number of rounds t and is proven to be tight.

6. In determining our parameters, we now take into account, as a lower bound to a single forgery attempt, the cost, in Boolean operations, of a single SHAKE computation. This allows us to match the security requirements of category 1, 3, and 5 considering cheating probabilities higher than $2^{-128}, 2^{-192}$, and 2^{-256}, respectively, by an factor equal to the ratio between the Boolean operation cost of an AES computation, and the one of a SHAKE computation.

Implementation

1. We moved onto a homogeneous strategy to perform domain separation across the SHAKE calls which CROSS employs as both CSPRNG and Hash. We fixed issues in the implementation with respect to domain separation and improved portability among platforms with different endianess. We furthermore revised the rejection sampling strategy and fixed a problem where the required randomness was underestimated.

2. We present a novel section, Section 5.6, discussing side-channel attacks and countermeasures.

3. We revised the truncation structure for the seed tree and now employ the same structure for the seed- and Merkle trees as discussed in Section 5.2.1.

4. We slightly changed the order of elements in `resp` and sampling of $\overline{\mathbf{M}}$ and $\mathbf{H}$ for R-SDP(G) to benefit implementation optimizations.

5. A SIMD implementation of Keccak is used to speed up in-round commitment hashing, Merkle tree hashing, and seed tree computations (see Section 5.3).

6. Reductions modulo $p = 509$ are now performed using Barrett's method (see Section 5.5).

Version 1.2 Version 1.2 includes a set of minor updates with respect to Version 1.1:

1. The lengths of the required amount of randomness to be drawn from the CSPRNGs has been corrected in the codebase.
2. Two additional domain separation constants (c and dsc) are employed in computing $\mathtt{cmt}_0$, $\mathtt{cmt}_1$ and in the CSPRNG for transformation sampling.
3. The size of the signatures has been updated in Table 4 (λ bits were missing).
4. A new NP-hardness proof of R-SDP is included.

Version 1.1 With regards to Version 1.0, the following changes have been made to this second version.

1. Improved security analysis for R-SDP(G): In Section 3, we consider an improved solver for the R-SDP(G), and we updated the parameters for CROSS R-SDP(G) accordingly. The parameters for the CROSS R-SDP instances are unchanged.
2. To prevent collision attacks on CSPRNG seeds, we include salting and a unique index per CSPRNG instance in each round of the signature. We detail these tweaks in the procedural description of CROSS.
3. To add hedging against multikey attacks we raise the length of the seeds for keypair generation to 2λ: this allows to prevent collision attacks relying on the collection of $2^{\frac{\lambda}{2}}$ keypairs. We updated Algorithm 1, Algorithm 2, and Algorithm 3 accordingly.
4. We propose parameters for an additional optimization corner that aims for even lower latency than the previous *fast* optimization corner (at the cost of larger signatures). While previous parameter sets featured a *small* and *fast* optimization targets, the new security categories provide a *small*, *balanced* (formerly *fast*), and *fast* version.
5. We report updated versions of Algorithm 1, Algorithm 2, and Algorithm 3, where we revised the generation of the $\mathbf{V}, \overline{\mathbf{W}}$, values. Doing so saves a CSPRNG call during key generation, $2t$ CSPRNG during signature, and $2w$ CSPRNG calls during verification, at no security margin loss.
6. We revised the CSPRNG implementation strategy, extracting always a constant amount of pseudorandom bits from each CSPRNG call. We make this possible, in the rejection sampling scenarios, considering the amount of required bits so that the CSPRNG extraction fails with probability $\frac{1}{2^\lambda}$. We detail this CSPRNG strategy in Section 5.1. This approach makes constant time implementations easier.
7. We now consider the objects in their bit-packed representation when they are employed as the inputs of cryptographic hashed, reducing the amount of required computation.

8. We switched from SHA-3 as a cryptographic hash to SHAKE with a 2λ bit extracted string. This improves on the overall speed, while keeping the same security margin (as the bottleneck for attacks was SHA-3 collision resistance, which matches the one of the appropriate SHAKE with a 2λ bit output). We provide details in Section 5.1.

9. We report the performance figures from an optimized implementation for the Intel AVX2 instruction set in Section 6.

10. We report the memory footprints of a stack-size optimized portable implementation, fitting all our parameter sets on a Cortex-M4-based microcontroller, namely the STM32F407VG present on the STM32F4 Discovery board by STMicroelectronics employed by the `pqm4` benchmarking project in Section 6.

1 Design Rationale and Notation

1.1 CROSS in a Nutshell

CROSS is a signature scheme based on the hardness of decoding restricted vectors [4, 5]. CROSS is obtained by transforming an interactive zero-knowledge protocol (CROSS-ID) into a signature scheme via the Fiat-Shamir transform.

Restricted vector decoding: The computationally hard problem underlying CROSS consists in decoding a given syndrome into a *restricted vector*. CROSS instances use one of the following two types of restrictions.

– Vectors having only entries in $\mathbb{E}$, a cyclic subgroup of the multiplicative group $\mathbb{F}_p^*$. The associated problem is called Restricted Syndrome Decoding Problem (R-SDP).

– Vectors with entries in G, a subgroup of $\mathbb{E}^n$. The associated problem is called Restricted Syndrome Decoding Problem with Subgroup (R-SDP(G)).

The security of these problems has been studied in [4, 5, 15, 16]; notably, the decisional versions of R-SDP and R-SDP(G) were proven to be NP-complete.

Zero-knowledge and Fiat-Shamir transform: CROSS is obtained by applying the Fiat-Shamir transform to an interactive Zero-Knowledge (ZK) proof of knowledge. The used ZK protocol, called CROSS-ID, is an adaption of the 5-pass protocol CVE proposed in [19].

Fast and simple: CROSS has been designed striving for simplicity and computational efficiency. The underlying finite fields are chosen such that efficient modular arithmetic for Mersenne primes can be largely employed. Furthermore, choosing the same finite fields for all security categories allows the reuse of a single set of hardware units to accelerate the underlying operations. The proposed parameters target three applicative scenarios having a fast, a balanced, and a small-signature variant. CROSS relies only on well-studied signature size optimization techniques, such as Puncturable Pseudo-Random Functions (PRFs) based on GGM trees [27] (to which we will also refer as seed trees) and Merkle trees.

1.2 Notation

The mathematical symbols employed in this specification are listed in Table 1.

We adopt the following mathematical conventions:

- vectors over $\mathbb{F}_p$ are in bold letters, e.g., $\mathbf{e} \in \mathbb{E}^n \subset \mathbb{F}_p^n$;
- matrices over $\mathbb{F}_p$ are in bold capital letters, e.g., $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$;
- vectors over $\mathbb{F}_z$ are in bold overlined letters, e.g., $\overline{\mathbf{e}} \in \mathbb{F}_z^n$ and $\overline{\mathbf{e}}_G \in \mathbb{F}_z^m$;
- matrices over $\mathbb{F}_z$ are in bold overlined capital letters, e.g., $\overline{\mathbf{M}} \in \mathbb{F}_z^{m\times n}$;
- concatenation between x, y is denoted as $x \mid y$;

Table 1: Mathematical symbols

Symbol	**Meaning**
p, z	Prime numbers, $z < p$
$\mathbb{F}_p$	Finite field with p elements
$\mathbb{F}_p^*$	Multiplicative group $\mathbb{F}_p \setminus \{0\}$
$\mathbb{F}_z$	Finite field with z elements
$\mathbb{E}$	Cyclic subgroup of $(\mathbb{F}_p^*, \cdot)$, with generator g of order z
$\star$	Component-wise multiplication
G	Subgroup of $(\mathbb{E}^n, \star)$ of size z^m
Id_ℓ	Identity matrix of size $\ell \times \ell$
n	Code length and length of restricted vectors
m	Size of the subgroup G is z^m, $m < n$
k	Code dimension, with $k < n$
λ	Security parameter
t	Number of rounds
w	Weight of the second challenge
$\mathcal{B}_{(t,w)}$	Hamming sphere of vectors in $\mathbb{F}_2^t$ with radius w
$\mathsf{HW}(t)$	Hamming weight of the binary representation of t

- in algorithms, we write $a \leftarrow b$ to denote that a is assigned the value b and $a \overset{\$}{\leftarrow} A$ denotes that a is drawn uniformly at random from A.

The main cryptographic notation is reported in Table 2.

1.3 Basics

Restricted vectors: CROSS is based on the so-called Restricted Syndrome Decoding Problem (R-SDP), an NP-complete problem that can be seen as a variant of the classical Syndrome Decoding Problem (SDP).

Let $\mathbb{F}_p$ be the finite field with p elements and let $g \in \mathbb{F}_p^\star$ of prime order z. A restricted vector has all entries in $\mathbb{E} = \langle g \rangle = \{g^i \mid i \in \{1, \dots, z\}\} \subseteq \mathbb{F}_p^\star$.

The R-SDP is then defined as: Given a parity-check matrix $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$, a syndrome $\mathbf{s} \in \mathbb{F}_p^{n-k}$ and a restricted set $\mathbb{E}$, find a vector $\mathbf{e} \in \mathbb{E}^n$ such that $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$.

The rationale behind the restriction is to make use of the fact that restricted vectors together with componentwise multiplication are isomorphic to vectors in $\mathbb{F}_z^n$ with addition, i.e., $(\mathbb{E}^n, \star) \cong (\mathbb{F}_z^n, +)$, where $\star$ denotes componentwise multiplication. We often write $g^{\overline{\mathbf{e}}}$, where $\overline{\mathbf{e}} \in \mathbb{F}_z^n$, to denote $(g^{\overline{\mathbf{e}}_1}, \dots, g^{\overline{\mathbf{e}}_n})$. Additionally, for $\mathbf{v} \in \mathbb{E}^n$ with exponent $\overline{\mathbf{v}} \in \mathbb{F}_z^n$, we denote by $\mathbf{v}^{-1}$ the restricted vector with exponent $-\overline{\mathbf{v}}$.

We also consider a specialized version of R-SDP, called R-SDP(G), in which solutions are required to live in a subgroup $(G, \star) \leq (\mathbb{E}^n, \star)$ where

$$G = \langle \mathbf{a}_1, \dots, \mathbf{a}_m \rangle = \left\{ \star_{i=1}^m \mathbf{a}_i^{\overline{u}_i} \mid \overline{u}_i \in \mathbb{F}_z \right\},$$

Table 2: Notation employed for inputs, outputs and cryptographic components.

Symbol	**Meaning**
$\texttt{Msg}$	Message to be signed
$\texttt{sk}$	Secret key
$\texttt{pk}$	Public key
$\texttt{Sgn}$	Signature
Hash	A cryptographic hash function with codomain $\{0,1\}^{2\lambda}$
$\texttt{Salt}$	binary string of length 2λ randomly drawn for each signature generation
$\texttt{Seed}_x$	Seed used to draw x
$\texttt{digest}_x$	Digest of a cryptographic hash function on x
$\texttt{cmt}_0[i], \texttt{cmt}_1[i]$	Commitments for round i
$\texttt{chall}_1$	First challenge vector in $(\mathbb{F}_p^\star)^t$
$\texttt{chall}_2$	Second challenge vector in $\mathcal{B}_{(t,w)}$
$\texttt{resp}[i]$	Response to the second challenge for round i
$\mathcal{T}$	A tree structure where each node consists of a λ- or 2λ bit string depending on its context.
$\mathcal{T}'$	A reference tree structure where each node consists of a single bit.
$\mathsf{CSPRNG}\text{--}_S(\cdot)$	A cryptographically secure pseudo-random number generator with output in the set S.

with $|G| < |\mathbb{E}^n| = z^n$.

The subgroup G can be represented in a compact way by collecting the exponents $\overline{\mathbf{a}}_i \in \mathbb{F}_z^n$ of the generators $\mathbf{a}_i$ into a matrix. That is, we define the matrix $\overline{\mathbf{M}} \in \mathbb{F}_z^{m\times n}$ as

$$\overline{\mathbf{M}} = \begin{pmatrix} (\overline{\mathbf{a}}_1)_1 & \cdots & (\overline{\mathbf{a}}_1)_n \\ \vdots & & \vdots \\ (\overline{\mathbf{a}}_m)_1 & \cdots & (\overline{\mathbf{a}}_m)_n \end{pmatrix} = \begin{pmatrix} \overline{\mathbf{a}}_1 \\ \vdots \\ \overline{\mathbf{a}}_m \end{pmatrix}.$$

Hence, contrary to vectors $g^{\overline{\mathbf{e}}} \in \mathbb{E}^n$, which allow any exponent $\overline{\mathbf{e}} \in \mathbb{F}_z^n$, we now only allow exponents $\overline{\mathbf{e}} \in \langle \overline{\mathbf{M}} \rangle$, a code of dimension m in $\mathbb{F}_z^n$.
Similarly to the restriction $\mathbb{E}$, we want to make use of the isomorphism $G \cong \mathbb{F}_z^m$. Instead of sending $\mathbf{e} \in G$, or $\overline{\mathbf{e}} \in \langle \overline{\mathbf{M}} \rangle$, we can send the information vector $\overline{\mathbf{e}}_G \in \mathbb{F}_z^m$. That is $\overline{\mathbf{e}} = \overline{\mathbf{e}}_G \overline{\mathbf{M}}$ and $\mathbf{e} = g^{\overline{\mathbf{e}}} \in G$.

The reasoning for this restriction is that restricted vectors $\mathbf{e} \in \mathbb{E}^n$, respectively $\mathbf{e} \in G$, have very compact size, namely $n \log_2(z)$, respectively $m \log_2(z)$, bits.

The employed ZK protocols also involve linear transitive maps $\overline{\mathbf{v}} : \mathbb{E} \to \mathbb{E}$, respectively $\overline{\mathbf{v}}_G : G \to G$. As $\mathbb{E}$ and G act transitively on themselves. That is, $\overline{\mathbf{v}}(\mathbf{e}) = \mathbf{e} \star \mathbf{e}'$, for some $\mathbf{e}' \in \mathbb{E}^n$. Hence, in order to send $\overline{\mathbf{v}}$ it is enough to send $\overline{\mathbf{e}}'$, which is such that $\mathbf{e}' = g^{\overline{\mathbf{e}}'}$. Thus, also $\overline{\mathbf{v}}, \overline{\mathbf{v}}_G$ have size $n \log_2(z)$, respectively $m \log_2(z)$, bits.

The hardness of solving R-SDP and R-SDP(G) relates directly to that of SDP. The most efficient solvers are Information Set Decoding (ISD) algorithms. We discuss the security of R-SDP, respectively R-SDP(G), in Section 3 and provide the details in the security guide [39]. In particular, we provide an analysis specifically tailored to the recommended choices for p and z.

Due to the isomorphism $(\mathbb{E}^n, \star) \cong (\mathbb{F}_z^n, +)$, CROSS also profits in terms of performance, as most computations can be performed over $\mathbb{F}_z$.

The subgroup $\mathbb{E} \leq \mathbb{F}_p^*$ is generated by the public parameter $g \in \mathbb{F}_p^*$ and is constant: $g = 2$ for R-SDP, or $g = 16$ for R-SDP(G).

Zero-knowledge and Fiat-Shamir transform: Using ZK protocols and the Fiat-Shamir transform to create a signature scheme comes with a long history and strong security aspects. In addition, this approach typically leads to small public key sizes.

The ZK protocol CROSS-ID is an adaption of the classical CVE protocol [19]. CROSS-ID is a 5-pass protocol, which can be classified as a $q2$-Identification scheme.

The CROSS-ID protocol follows the same rationale as CVE:

- The public key consists of a syndrome $\mathbf{s} \in \mathbb{F}_p^{n-k}$, a seed $\texttt{Seed}_{\texttt{pk}}$ to compute a random parity-check matrix $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$, as well as a random matrix $\overline{\mathbf{M}} \in \mathbb{F}_z^{m\times n}$ which is used to generate the exponents of the vectors in G.
- The secret is given by a restricted vector $\mathbf{e} \in \mathbb{E}^n$, respectively by $\mathbf{e} \in G$.
- Within one round of the protocol, the signer either proves the syndrome equation $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$ or the restriction $\mathbf{e} \in \mathbb{E}^n$, respectively $\mathbf{e} \in G$.
- This invokes two commitments, one to prove the syndrome equation, $\texttt{cmt}_0$, and a second to prove the restriction, $\texttt{cmt}_1$.
- The protocol also requires two challenges, the first being $\texttt{chall}_1 \in \mathbb{F}_p^\star$ and the second $\texttt{chall}_2 \in \{0, 1\}$.

The protocol is repeated for t rounds and made non-interactive using the Fiat-Shamir transform. To do so, the signer generates the first challenge as the hash of the t commitments $\texttt{cmt}_0, \texttt{cmt}_1$ and the message $\texttt{Msg}$, that is $\texttt{chall}_1 = \mathsf{Hash}(\texttt{Msg}, \texttt{cmt}_0, \texttt{cmt}_1)$. The second challenge is similarly generated using also $\texttt{chall}_1$ and its responses $\mathbf{y}[i] \in \mathbb{F}_p^n$. The rationale of the Fiat-Shamir transform is also depicted in Figure 1 and the protocol is described in full details in Section 2.1. The signature consists of the transcripts of each of the rounds.

As the responses have different sizes, we reduce the signature sizes using weighted challenges. In the balanced and small versions, we also use a Merkle tree to reduce the cost of sending the commitments $\texttt{cmt}_0$.

We discuss the security of the protocol and the scheme resulting after the Fiat-Shamir transform, including a novel forgery attack, in Section 3.2.1 and provide the proofs in the security guide

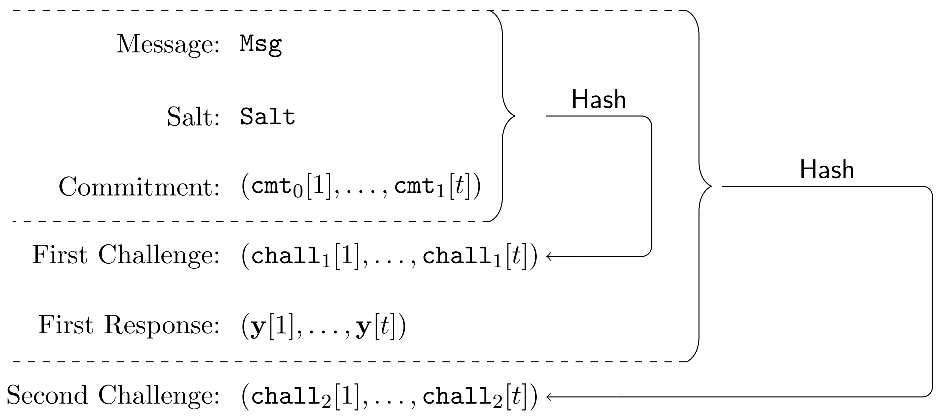
Message: Msg
Salt: Salt
Commitment: $(\mathtt{cmt}_0[1], \ldots, \mathtt{cmt}_1[t])$
First Challenge: $(\mathtt{chall}_1[1], \ldots, \mathtt{chall}_1[t])$
First Response: $(\mathbf{y}[1], \ldots, \mathbf{y}[t])$
Second Challenge: $(\mathtt{chall}_2[1], \ldots, \mathtt{chall}_2[t])$
Hash
Hash

Figure 1: Flowchart representation of challenges generation in the Fiat-Shamir transformation

[39]. In particular, the resulting signature scheme is EUF-CMA secure (see Theorem 8, as well as the security guide [39]).

Fast and simple: The compact sizes of the restricted vectors reduce the amount of computational effort compared to the traditional SDP in the Hamming metric.

In addition, the arithmetic becomes much simpler: roughly half of the arithmetic operations in CROSS are performed over a smaller field $\mathbb{F}_z$, where we can substitute modular multiplications with less expensive additions.

We choose p and z to be implementation-friendly values, namely Mersenne primes or close to Mersenne primes. We keep p and z fixed for all security categories. This allows for the implementation of only two sets of arithmetic primitives, which reduces the code size (in software implementations, where it is critical in Flash-memory-constrained microcontrollers) or the required silicon area (in hardware implementations).

All primitives in CROSS require only consolidated symmetric primitives (such as CSPRNGs and cryptographic hashes) and vector/matrix operations among small elements. This allows us to reduce the amount of implementation footguns [35], i.e., potential points for implementation errors that lead to vulnerabilities, either directly or through the exploitation of side-channel information leakage.

The structural simplicity also allows for a straightforward, constant-time implementation of the scheme, as all operations are natively performed in a memory-access-pattern oblivious way, while CSPRNGs and hashes are available as consolidated and tested constant-time implementations.

We provide three variants of CROSS to achieve heterogeneous trade-offs:

- CROSS-fast: small values of t (the number of repetitions for the ZK protocol). This variant aims at fast signature generation and verification and uses squashed tree structures for performance improvements.
- CROSS-small: large values of t. This variant aims at achieving short signatures and uses a classical seed- and Merkle tree for signature compression.

- CROSS-balanced: moderate values of t. This variant comes as a trade-off between the other two variants and also uses a classical seed- and Merkle tree for signature compression.

CROSS has very small keys: the private key is reduced to its optimal size, i.e., a single random seed. All the elements in the public key, apart from a short vector over $\mathbb{F}_p$, can also be regenerated from a seed with acceptable computational overhead. This results in a public key of less than 153 B for R-SDP and less than 106 B for R-SDP(G) - for all NIST security categories. These reduced key sizes allow CROSS keypairs to fit even on constrained embedded devices where persistent (flash) memory may be scarce, such as in low-end microcontrollers.

2 Procedural Description of CROSS-ID and CROSS

2.1 CROSS-ID

The CROSS-ID is a ZK protocol, which is 5-pass protocol and can be characterized as a $q2$-identification scheme, as the first challenge $\texttt{chall}_1 \in \mathbb{F}_p^*$ and the second challenge $\texttt{chall}_2 \in \{0, 1\}$.

The protocol is an adaption of CVE [19] and follows the same principle. The secret is given by the restricted vector $\mathbf{e} \in \mathbb{E}^n$, respectively $\mathbf{e} \in G$, which satisfies the syndrome equation $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$, and the verifier will either challenge $\texttt{chall}_2 = 0$ asking for a proof that the syndrome equation is satisfied or $\texttt{chall}_2 = 1$, asking for a proof that the secret vector is restricted. In order to provide such proof, the prover uses a linear transitive map $\mathbf{v} : \mathbb{E}^n \to \mathbb{E}^n$, respectively $\mathbf{v} : G \to G$.

We present the protocol using R-SDP(G) formulation, as this includes also the R-SDP, by setting $G = \mathbb{E}^n$.

Commitments: The prover computes a random $\mathbf{e}' \in G$ and $\mathbf{u}' \in \mathbb{F}_p^n$ from CSPRNG using a seed. The prover can also compute $\mathbf{v} \in G$ which is such that $\mathbf{v} \star \mathbf{e}' = \mathbf{e}$, i.e., $\mathbf{v}$ acts as transformation on G. The prover then commits to $\texttt{cmt}_0 = \mathsf{Hash}((\mathbf{v} \star \mathbf{u}')\mathbf{H}^\top \mid \mathbf{v})$, the hash of the syndrome of the transformed $\mathbf{u}'$ and the transformation $\mathbf{v}$, and to $\texttt{cmt}_1 = \mathsf{Hash}(\mathbf{u}' \mid \mathbf{e}')$, the two vectors computed through CSPRNG from a seed.

First challenge and response: The verifier chooses a challenge $\texttt{chall}_1 \in \mathbb{F}_p^*$ and the prover computes

$$\mathbf{y} = \mathbf{u}' + \texttt{chall}_1\mathbf{e}' = \mathbf{v}^{-1} \star (\mathbf{u} + \texttt{chall}_1\mathbf{e}) = \mathbf{v}^{-1} \star \mathbf{u} + \texttt{chall}_1\mathbf{v}^{-1} \star \mathbf{e}.$$

To reduce communication cost, the prover only sends $\mathsf{Hash}(\mathbf{y})$.

Second challenge and response: The verifier chooses the second challenge $\texttt{chall}_2 \in \{0, 1\}$; The response is then either formed to verify $\texttt{cmt}_0$, by sending $\texttt{resp} = (\mathbf{y}, \mathbf{v})$ or to verify $\texttt{cmt}_1$ by sending $\texttt{resp} = \texttt{Seed}$ which is used to compute $\mathbf{e}', \mathbf{u}'$.

Verification: To recover $\texttt{cmt}_0$ from $\mathbf{H}, \mathbf{s}$ the prover needs to send $\mathbf{y}, \mathbf{v}$. Note that this step differs from the original CVE, where one directly sends $\mathbf{y}$ instead of the digest. The verifier first

```
Private Key $\mathbf{e} \in G$
Public Key  $G \subseteq \mathbb{E}^n$, $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$, $\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_p^{n-k}$

PROVER                                                                VERIFIER

// Sampling Seed to compute $\mathbf{e}', \mathbf{u}'$
Seed $\xleftarrow{\$} \{0,1\}^\lambda$
$(\mathbf{e}', \mathbf{u}')$  $\leftarrow$  CSPRNG(Seed)
// with co − domain $G \times \mathbb{F}_p^n$

// Computing $\mathbf{v}, \mathbf{u}, \mathbf{s}'$
$\mathbf{v} \leftarrow \mathbf{e} \star (\mathbf{e}')^{-1}$
$\mathbf{u} \leftarrow \mathbf{v} \star \mathbf{u}'$
$\mathbf{s}' \leftarrow \mathbf{u}\mathbf{H}^\top$

// Computing commitments
$\mathtt{cmt}_0 \leftarrow \mathsf{Hash}(\mathbf{s}' \mid \mathbf{v})$
$\mathtt{cmt}_1 \leftarrow \mathsf{Hash}(\mathbf{u}' \mid \mathbf{e}')$
                         $\xrightarrow{\mathtt{cmt}_0, \mathtt{cmt}_1}$
                              // Sampling first challenge
                         $\xleftarrow{\mathtt{chall}_1}$          $\mathtt{chall}_1 \xleftarrow{\$} \mathbb{F}_p^*$
// Computing first response
$\mathbf{y} \leftarrow \mathbf{u}' + \mathtt{chall}_1\mathbf{e}'$
$\mathtt{digest}_\mathbf{y} \leftarrow \mathsf{Hash}(\mathbf{y})$
                         $\xrightarrow{\mathtt{digest}_\mathbf{y}}$
                              // Sampling second challenge
                         $\xleftarrow{\mathtt{chall}_2}$          $\mathtt{chall}_2 \xleftarrow{\$} \{0,1\}$
// Computing second response
If $\mathtt{chall}_2 = 0$, $\mathtt{resp} \leftarrow (\mathbf{y}, \mathbf{v})$
If $\mathtt{chall}_2 = 1$, $\mathtt{resp} \leftarrow$ Seed
                         $\xrightarrow{\mathtt{resp}}$
                              // Verification
                              If $\mathtt{chall}_2 = 0$:
                                $\mathbf{y}' \leftarrow \mathbf{v} \star \mathbf{y}$
                                $\mathbf{s}' \leftarrow \mathbf{y}'\mathbf{H}^\top - \mathtt{chall}_1\mathbf{s}$
                                Accept if:
                                  1) $\mathsf{Hash}(\mathbf{y}) = \mathtt{digest}_\mathbf{y}$
                                  2) $\mathsf{Hash}(\mathbf{s}' \mid \mathbf{v}) = \mathtt{cmt}_0$
                                  3) $\mathbf{v} \in G$
                              If $\mathtt{chall}_2 = 1$:
                                $(\mathbf{e}', \mathbf{u}') \leftarrow$ CSPRNG(Seed) // with co − domain $G \times \mathbb{F}_p^n$
                                $\mathbf{y} \leftarrow \mathbf{u}' + \mathtt{chall}_1\mathbf{e}'$

                                Accept if:
                                  1) $\mathsf{Hash}(\mathbf{y}) = \mathtt{digest}_\mathbf{y}$
                                  2) $\mathsf{Hash}(\mathbf{u}' \mid \mathbf{e}') = \mathtt{cmt}_1$
```

Figure 2: CROSS-ID

checks the validity of the response, i.e., $\mathbf{v} \in G, \mathtt{digest}_\mathbf{y} = \mathsf{Hash}(\mathbf{y})$ and then recovers $\mathtt{cmt}_0$ as

$$\mathtt{cmt}_0 = \mathsf{Hash}(\mathbf{v} \star \mathbf{y}\mathbf{H}^\top - \mathtt{chall}_1\mathbf{s} \mid \mathbf{v}),$$

as $\mathbf{v} \star \mathbf{y} = \mathbf{u} + \mathtt{chall}_1\mathbf{e}$.

To recover $\mathtt{cmt}_1$ the prover only needs to send the seed from which $\mathbf{e}', \mathbf{u}'$ were computed. The verifier can then check if $\mathtt{digest}_\mathbf{y} = \mathsf{Hash}(\mathbf{u}' + \mathtt{chall}_1\mathbf{e}')$ and recovers $\mathtt{cmt}_1 = \mathsf{Hash}(\mathbf{u}' \mid \mathbf{e}')$. In both cases, the verifier checks that $\mathbf{y}$ has been formed correctly.

Communication cost: The transcript for one round consists of $\texttt{cmt}_0, \texttt{cmt}_1$, both digests of length 2λ, $\texttt{chall}_1$ of bit size $\log_2(p-1)$, $\texttt{digest}_\mathbf{y}$ of length 2λ, $\texttt{chall}_2$ which is one bit and finally the response. If $\texttt{chall}_2 = 0$, the response $(\mathbf{y}, \mathbf{v})$ is of size $n\log_2(p) + m\log_2(z)$ bits. If $\texttt{chall}_2 = 1$, the response consists only of $\texttt{Seed}$ of length λ.

Security: The protocol enjoys the same security as the original CVE, namely

Proposition 1. The CROSS-ID protocol in Figure 2 is complete, achieves zero-knowledge and is sound, with soundness error $\frac{p}{2(p-1)}$.

The proof can be found in [39].

Weighted challenges: Since the two possible responses have different bit sizes, we use weighted second challenges in order to reduce the final signature size. The response for $\texttt{chall}_2 = 1$ is much smaller than the response for $\texttt{chall}_2 = 0$. Thus, we force the second challenge vector $\texttt{chall}_2 = (\texttt{chall}_2[1], \ldots, \texttt{chall}_2[t]) \in \{0,1\}^t$ to be of weight w, i.e., w many rounds i are such that $\texttt{chall}_2[i] = 1$. The weighted challenges also make the signature size constant and simplifies constant-time implementations.

Fixing the weight w of the second challenge brings several consequences:

- When w is large, the majority of the rounds have a small response and we can apply further optimizations, such as a Merkle tree for the t commitments $\texttt{cmt}_0[1], \ldots, \texttt{cmt}_0[t]$ and a seed tree for the t seeds $\texttt{Seed}[1], \ldots, \texttt{Seed}[t]$, which allows to compress the signature (see Section 2.2.2 and Section 5.2.1). This choice leads to the two variants CROSS-small and CROSS-balanced.

- When $w \sim \frac{t}{2}$, the protocol is closer to actual t parallel repetitions and using classical trees will not yield any compression benefits. Instead, we use squashed tree structures that allow for performance optimizations instead of compression (details in Section 5.3). This results in the third variant CROSS-fast.

- The constant weight w can be used for forgery attacks, as discussed in Section 3.

Fiat-Shamir transform: We use t parallel executions of the CROSS-ID, where we employ weighted challenges and apply the Fiat-Shamir transform. To prevent from attacks based on commitment collisions, we use $\texttt{Salt}$ of length 2λ, to form the commitments as suggested in [20]. Figure 1 summarizes the Fiat-Shamir transform.

The resulting signature scheme is EUF-CMA secure and even enjoys beyond unforgeability features [3].

Theorem 2. CROSS is EUF-CMA secure.

More details, can be found in Section 3.2.2.

2.2 CROSS Protocol

CROSS consists of three algorithms: the key generation, KeyGen, in Algorithm 1, the signature generation, Sign, in Algorithm 2, and the verification, Verify, in Algorithm 3.

2.2.1 Key Generation

The algorithm KeyGen takes as inputs the public data, i.e., the security parameter λ and the restriction $\mathbb{E}$, parametrized through its generator g.

The algorithm outputs the secret key sk, given by a 2λ bits long seed and the public key, pk given by a 2λ bit long seed, which is used to derive the random matrices $\mathbf{H}$ and $\overline{\mathbf{M}}$, and the syndrome $\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_p^{n-k}$.

We distinguish between the R-SDP and R-SDP(G) variant by colors, that is: steps which are only required for R-SDP(G) are in orange and on the left of the algorithm, and R-SDP steps are in teal on the right side.

Algorithm 1: KeyGen()

```
Input: None
Output: sk : Seed_sk: secret key seed;
        pk : (Seed_pk, s) public key;
Data: $\lambda$: security parameter;
      $g \in \mathbb{F}_p^*$: generator of $\mathbb{E}$;
   // Sampling seeds
1  $\mathsf{Seed}_{\mathsf{sk}} \xleftarrow{\$} \{0,1\}^{2\lambda}$
2  $(\mathsf{Seed}_{\mathsf{e}}, \mathsf{Seed}_{\mathsf{pk}}) \leftarrow \mathsf{CSPRNG}\text{—}_{\{0,1\}^{2\lambda}\times\{0,1\}^{2\lambda}}(\mathsf{Seed}_{\mathsf{sk}} \mid 3t+1)$
   // Sampling random matrices H and $\overline{\mathbf{M}}$
3  $(\overline{\mathbf{W}}, \mathbf{V}) \leftarrow \mathsf{CSPRNG}\text{—}_{\mathbb{F}_z^{m\times(n-m)}\times\mathbb{F}_p^{(n-k)\times k}}(\mathsf{Seed}_{\mathsf{pk}} \mid 3t+2)$  |  $\mathbf{V} \leftarrow \mathsf{CSPRNG}\text{—}_{\mathbb{F}_p^{(n-k)\times k}}(\mathsf{Seed}_{\mathsf{pk}} \mid 3t+2)$
4  $\mathbf{H} \leftarrow [\mathbf{V} \mid \mathrm{Id}_{n-k}]$
   // Computing e
   $\overline{\mathbf{M}} \leftarrow [\overline{\mathbf{W}} \mid \mathrm{Id}_m]$
5  $\overline{\mathbf{e}}_G \leftarrow \mathsf{CSPRNG}\text{—}_{\mathbb{F}_z^m}(\mathsf{Seed}_{\mathsf{e}} \mid 3t+3)$  |  $\overline{\mathbf{e}} \leftarrow \mathsf{CSPRNG}\text{—}_{\mathbb{F}_z^n}(\mathsf{Seed}_{\mathsf{e}} \mid 3t+3)$
   $\overline{\mathbf{e}} \leftarrow \overline{\mathbf{e}}_G\overline{\mathbf{M}}$
   for $j$ from 1 to $n$ do
6      $\mathbf{e}_j \leftarrow g^{\overline{\mathbf{e}}_j}$
   // Computing the syndrome s
7  $\mathbf{s} \leftarrow \mathbf{e}\mathbf{H}^\top$
   // Return secret key sk and public key pk
8  sk ← Seed_sk
9  pk ← (Seed_pk, s)
10 return (sk, pk);
```

Sampling random full-rank matrices: the random matrix $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$ of rank $n-k$ is constructed as $\mathbf{H} = [\mathbf{V} \mid \mathrm{Id}_{n-k}]$, where $\mathbf{V} \in \mathbb{F}_p^{(n-k)\times k}$ is sampled by calling CSPRNG on $\mathsf{Seed}_{\mathsf{pk}}$. Similarly, $\overline{\mathbf{M}} \in \mathbb{F}_z^{m\times n}$ of rank m is constructed as $\overline{\mathbf{M}} = [\overline{\mathbf{W}} \mid \mathrm{Id}_m]$, where $\overline{\mathbf{W}} \in \mathbb{F}_z^{m\times(n-m)}$ is sampled by calling CSPRNG on $\mathsf{Seed}_{\mathsf{pk}}$.

Constructing restricted errors: In the R-SDP variant, we sample $\overline{\mathbf{e}} \in \mathbb{F}_z^n$ from CSPRNG, while in the R-SDP(G) variant, we first sample $\overline{\mathbf{e}}_G \in \mathbb{F}_z^m$ from CSPRNG, and then compute $\overline{\mathbf{e}} = \overline{\mathbf{e}}_G \overline{\mathbf{M}} \in \mathbb{F}_z^n$. In both cases, we then compute the restricted vector $\mathbf{e} = g^{\overline{\mathbf{e}}} \in \mathbb{E}^n \subset \mathbb{F}_p^n$.

Key sizes: The secret key `sk` has size 2λ, while the public key has size $2\lambda + (n-k)\log_2(p)$ bits, padded to the next byte multiple as described in Section 5.4.

2.2.2 Signature Generation

Algorithm Sign is given the public data, i.e., the security parameter λ, the restriction $\mathbb{E}$, parametrized through its generator g, the number of rounds t, the weight of the second challenge w and c a constant defined as $2t-1$. Sign takes as input the secret key seed `sk` and the message `Msg` to be signed.

The algorithm outputs the signature `Sgn` consisting of $\texttt{Salt}, \texttt{digest}_{\texttt{cmt}}, \texttt{digest}_{\texttt{chall}_2}, \texttt{Path}, \texttt{Proof}$, and the second response `resp`.

Small and balanced version vs. fast version: In the small and balanced version of CROSS we use a large weight $w > t/2$, while in the fast version we set $w \sim t/2$. The large weight w allows us to make use of the seed- and Merkle tree to reduce the signature sizes by adding only those tree nodes to the signature (i.e., the seed `Path` and Merkle `Proof`), that the verifier requires to compute the remaining nodes for signature verification. In the fast version, using these classical tree structures yields no benefit since $w \sim t/2$. Nevertheless, we use trees consisting of only three levels for implementation efficiency and the `Path` and `Proof` nodes only consist of selected leaf nodes of these trees, slightly abusing the terminology of `Path` and `Proof`. For a detailed explanation of this difference, we refer to Section 5.2.1.

Seed path: For proper signature verification, the signer has to reveal w round seeds indicated by the second challenge $\texttt{chall}_2[i] = 1$. Since $\texttt{Seed}[i]$ are leaves of a seed tree, the number of sent seeds can be compressed by sending a path, which allows to recover all $\texttt{Seed}[i]$, where $\texttt{chall}_2[i] = 1$. The maximum number of tree nodes to be sent can be computed according to [17] as

$$|\texttt{Path}| = \left\lfloor (t-w)\log_2\left(\frac{t}{t-w}\right) + \mathsf{HW}(t) - 1 \right\rfloor,$$

where $\mathsf{HW}(t)$ being the Hamming weight of the binary representation of t.

For the fast version, the signature includes exactly w round seeds from the leaves of the tree indicated by $\texttt{chall}_2[i] = 1$, as a compression is not efficient.

Merkle proof: In the balanced and small versions, with large w, we have to send $\texttt{cmt}_0[i]$ w times as part of the signature. Since w is rather close to t, we can compute the root of a Merkle tree with its leaves being $\texttt{cmt}_0[1], \ldots, \texttt{cmt}_0[t]$. Instead of sending all w required commitments, we can send a Merkle proof such that the verifier can re-compute the Merkle root using the proof nodes and the recomputed $\texttt{cmt}_0[i]$ for $\texttt{chall}_2[i] = 0$. As for the seed path, the maximum number of nodes to include in the signature is thus

$$|\texttt{Proof}| = \left\lfloor (t-w)\log_2\left(\frac{t}{t-w}\right) + \mathsf{HW}(t) - 1 \right\rfloor.$$

Again, for the fast version, the signature includes exactly w commitments $\texttt{cmt}_0[i]$ for $\texttt{chall}_2[i] = 1$, as a compression is not efficient.

We distinguish between the R-SDP and R-SDP(G) variant by colors, that is: steps which are only required for R-SDP(G) are in orange and on the left of the algorithm, and R-SDP steps are in teal on the right side.

Algorithm 2: Sign(sk, Msg)

```
   Input: sk: secret key $\texttt{Seed}_{\texttt{sk}} \in \{0,1\}^{2\lambda}$;
          $\texttt{Msg} \in \{0,1\}^*$: message;
   Output: Sgn: signature;
   Data: $\lambda$: 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
1  $\overline{\mathbf{e}}, \overline{\mathbf{e}}_G, \mathbf{H}, \overline{\mathbf{M}} \leftarrow \mathsf{ExpandSK}(\texttt{Seed}_{\texttt{sk}})$        $\overline{\mathbf{e}}, \mathbf{H} \leftarrow \mathsf{ExpandSK}(\texttt{Seed}_{\texttt{sk}})$
   // Computing the commitments
2  $\texttt{Seed} \xleftarrow{\$} \{0,1\}^{\lambda}$,   $\texttt{Salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$
3  $(\texttt{Seed}[1], \dots, \texttt{Seed}[t]) \leftarrow \mathsf{SeedLeaves}(\texttt{Seed} \mid \texttt{Salt})$
   // Compute $\mathbf{v}[i]$ such that $\mathbf{v}[i] \star \mathbf{e}'[i] = \mathbf{e}$
4  for $i$ from 1 to $t$ do
       $\overline{\mathbf{e}}'_G[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG}\text{--}_{\mathbb{F}_z^m \times \mathbb{F}_p^n}(\texttt{Seed}[i] \mid \texttt{Salt} \mid i{+}c)$        $\overline{\mathbf{e}}'[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG}\text{--}_{\mathbb{F}_z^n \times \mathbb{F}_p^n}(\texttt{Seed}[i] \mid \texttt{Salt} \mid i{+}c)$
5
       $\overline{\mathbf{v}}_G[i] \leftarrow \overline{\mathbf{e}}_G - \overline{\mathbf{e}}'_G[i]$
       $\overline{\mathbf{e}}'[i] \leftarrow \overline{\mathbf{e}}'_G[i]\overline{\mathbf{M}}$
       $\overline{\mathbf{v}}[i] \leftarrow \overline{\mathbf{e}} - \overline{\mathbf{e}}'[i]$
6      for $j$ from 1 to $n$ do
7          $\mathbf{v}[i]_j \leftarrow g^{\overline{\mathbf{v}}[i]_j}$
8      $\mathbf{u}[i] \leftarrow \mathbf{v}[i] \star \mathbf{u}'[i]$
9      $\mathbf{s}'[i] \leftarrow \mathbf{u}[i]\mathbf{H}^\top$
10     $\texttt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}_G[i] \mid \texttt{Salt} \mid i + c)$        $\texttt{cmt}_0[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}[i] \mid \texttt{Salt} \mid i + c)$
       $\texttt{cmt}_1[i] \leftarrow \mathsf{Hash}(\texttt{Seed}[i] \mid \texttt{Salt} \mid i + c)$
11 $\texttt{digest}_{\texttt{cmt}_0} \leftarrow \mathsf{TreeRoot}(\texttt{cmt}_0[1] \mid \cdots \mid \texttt{cmt}_0[t])$
12 $\texttt{digest}_{\texttt{cmt}_1} \leftarrow \mathsf{Hash}(\texttt{cmt}_1[1] \mid \cdots \mid \texttt{cmt}_1[t])$
13 $\texttt{digest}_{\texttt{cmt}} \leftarrow \mathsf{Hash}(\texttt{digest}_{\texttt{cmt}_0} \mid \texttt{digest}_{\texttt{cmt}_1})$
   // Computing first challenge
14 $\texttt{digest}_{\texttt{Msg}} \leftarrow \mathsf{Hash}(\texttt{Msg})$
15 $\texttt{digest}_{\texttt{chall}_1} \leftarrow \mathsf{Hash}(\texttt{digest}_{\texttt{Msg}} \mid \texttt{digest}_{\texttt{cmt}} \mid \texttt{Salt})$
16 $\texttt{chall}_1 \leftarrow \mathsf{CSPRNG}\text{--}_{(\mathbb{F}_p^*)^t}\left(\texttt{digest}_{\texttt{chall}_1} \mid t + c\right)$
   // Computing first response
17 for $i$ from 1 to $t$ do
18     for $j$ from 1 to $n$ do
19         $\mathbf{e}'[i]_j \leftarrow g^{\overline{\mathbf{e}}'[i]_j}$
20     $\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \texttt{chall}_1[i]\mathbf{e}'[i]$
   // Computing second challenge
21 $\texttt{digest}_{\texttt{chall}_2} \leftarrow \mathsf{Hash}(\mathbf{y}[1] \mid \cdots \mid \mathbf{y}[t] \mid \texttt{digest}_{\texttt{chall}_1})$
22 $\texttt{chall}_2 \leftarrow \mathsf{CSPRNG}\text{--}_{\mathcal{B}_{(t,w)}}\left(\texttt{digest}_{\texttt{chall}_2} \mid t + c + 1\right)$
   // Computing second response
23 $\texttt{Proof} \leftarrow \mathsf{TreeProof}(\texttt{cmt}_0[1] \mid \cdots \mid \texttt{cmt}_0[t] \mid \texttt{chall}_2)$
24 $\texttt{Path} \leftarrow \mathsf{SeedPath}(\texttt{Seed} \mid \texttt{Salt} \mid \texttt{chall}_2)$
25 for $i$ from 1 to $t$ do
26     if $\texttt{chall}_2[i] = 0$ then
27         $\texttt{resp}[i]_0 \leftarrow (\mathbf{y}[i], \overline{\mathbf{v}}_G[i])$        $\texttt{resp}[i]_0 \leftarrow (\mathbf{y}[i], \overline{\mathbf{v}}[i])$
           $\texttt{resp}[i]_1 \leftarrow \texttt{cmt}_1[i]$
   // Assembling signature
28 $\texttt{Sgn} \leftarrow (\texttt{Salt}, \texttt{digest}_{\texttt{cmt}}, \texttt{digest}_{\texttt{chall}_2}, \texttt{Path}, \texttt{Proof}, \texttt{resp})$
29 return Sgn
```

Expanding the secret key: Using $\texttt{Seed}_{\texttt{sk}}$, the function ExpandSK (details in Section 5, Algorithm 4) re-generates $\overline{\mathbf{e}}$ and $\mathbf{H}$ in the R-SDP version and $\overline{\mathbf{e}}, \overline{\mathbf{e}}_G, \overline{\mathbf{M}}, \mathbf{H}$ in the R-SDP(G) version. The function ExpandSK performs exactly the same computations as KeyGen, with the only difference that we do not need the syndrome $\mathbf{s}$ and do not need to compute $\mathbf{e} \in \mathbb{F}_p^n$.

Preparing the commitment phase: The signer samples an initial seed Seed of λ bits and a salt Salt of 2λ bits. SeedLeaves (details in Section 5, Algorithm 5 and 6) then takes the Seed and Salt and internally computes a seed tree with t leaves using CSPRNG with proper domain separation. Within the tree, each node consists of a λ-bit seed. The function returns the leaves which then serve as round seeds, $\texttt{Seed}[i]$, for the signature generation.

For each round i, the signer then samples an $\mathbf{u}'[i]$ and $\overline{\mathbf{e}}_G[i]$, respectively $\overline{\mathbf{e}}[i]$, using CSPRNG on the $\texttt{Seed}[i]$, Salt and a 2 byte constant $i + c$ in little endian byte order.

To compute $\mathbf{v}[i]$ such that $\mathbf{v}[i] \star \mathbf{e}'[i] = \mathbf{e}$, the signer computes $\mathbf{v}[i] = \mathbf{e} \star \mathbf{e}'[i]^{-1}$, which means computing the exponent $\overline{\mathbf{v}}[i] = \overline{\mathbf{e}} - \overline{\mathbf{e}}[i]'$. For the R-SDP$(G)$ version, the signer first computes $\overline{\mathbf{v}}_G[i] = \overline{\mathbf{e}}_G - \overline{\mathbf{e}}_G[i]'$ and then $\overline{\mathbf{e}}'[i] = \overline{\mathbf{e}}'_G \overline{\mathbf{M}}$.

The signer further computes the auxiliary vector $\mathbf{u}[i] = \mathbf{v}[i] \star \mathbf{u}'[i]$ and its syndrome $\mathbf{s}'[i] = \mathbf{u}[i]\mathbf{H}^\top$.

Computing the commitments: The commitments are then computed as

$$\begin{aligned}
\texttt{cmt}_1[i] &= \mathsf{Hash}(\texttt{Seed}[i] \mid i + c), \\
\texttt{cmt}_0[i] &= \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}[i] \mid \texttt{Salt} \mid i + c), \qquad \text{for R-SDP} \\
\texttt{cmt}_0[i] &= \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}_G[i] \mid \texttt{Salt} \mid i + c), \qquad \text{for R-SDP}(G).
\end{aligned}$$

Computing first challenge: The commitments are hashed together, first all commitments $\texttt{cmt}_0$ in $\texttt{digest}_{\texttt{cmt}_0}$ and secondly all $\texttt{cmt}_1$ into $\texttt{digest}_{\texttt{cmt}_1}$, to finally get

$$\texttt{digest}_{\texttt{cmt}} = \mathsf{Hash}(\texttt{digest}_{\texttt{cmt}_0} \mid \texttt{digest}_{\texttt{cmt}_1}).$$

Therefore, the function TreeRoot takes the commitments $\texttt{cmt}_0[i]$ as input, which constitute the leaves of an internally computed Merkle tree. The Merkle tree is computed from bottom to top where two nodes are hashed to compute a parent node. Consequently, each node consists of a hash of 2λ bits. The root of the tree is then returned and represents $\texttt{digest}_{\texttt{cmt}_0}$.

In the fast version of CROSS, the underlying tree is not a classical Merkle tree, in the sense that two children are hashed to a parent node. Instead, multiple nodes are hashed to four intermediate nodes which are finally hashed to a root node. For details, we refer to Section 5.2.1.

To compute the first challenge, the signer then hashes $\texttt{digest}_{\texttt{Msg}}$, $\texttt{digest}_{\texttt{cmt}}$ and Salt to obtain $\texttt{digest}_{\texttt{chall}_1}$. The first challenge is then sampled using CSPRNG on the input $\texttt{digest}_{\texttt{chall}_1}$.

Computing first response: The signer computes $\mathbf{y}[i] = \mathbf{u}'[i] + \texttt{chall}_1[i]\mathbf{e}'[i]$ and hashes all the $\mathbf{y}[i]$ as well as $\texttt{digest}_{\texttt{chall}_1}$ to obtain $\texttt{digest}_{\texttt{chall}_2}$.

Computing second challenge: The second challenge $\texttt{chall}_2$ is computed through CSPRNG on the input $\texttt{digest}_{\texttt{chall}_2}$.

Computing second response: If $\texttt{chall}_2[i] = 1$, the signer does not need to include any additional information in a response vector. The verifier can rebuild $\texttt{cmt}_1[i]$ for these rounds from Path and $\texttt{cmt}_0[i]$ from Proof.
In order to compute the Path, the function SeedPath takes the definition of the seed tree and the second challenge $\texttt{chall}_2$ as input and returns the subset of tree nodes, denoted as Path, that are required

to re-generate all round seeds $\texttt{Seed}[i]$, for which $\texttt{chall}_2[i] = 1$. Due to the tree construction in the balanced and small versions, we can include inner tree nodes in the $\texttt{Path}$ to some extend, which serves as a compression mechanism. In the fast version, the $\texttt{Path}$ consists of exactly w leaves selected by $\texttt{chall}_2[i] = 1$.

To compute the $\texttt{Proof}$, the function TreeProof is used. It has as input the definition of the previously mentioned Merkle tree as well as $\texttt{chall}_2$. With those, the function computes a classical Merkle proof (for the balanced and small versions). That is: it returns the subset of nodes, $\texttt{Proof}$, in the tree that is required to re-compute the root given that a verifier has all $\texttt{cmt}_0[i]$ with $\texttt{chall}[i]_2 = 0$. In the fast version, we slightly abuse the term $\texttt{Proof}$, as it refers to w many leaves $\texttt{cmt}_0[i]$ with $\texttt{chall}_2[i] = 1$.

$\texttt{resp}$ consists of two parts $\texttt{resp}_0$ and $\texttt{resp}_1$. If $\texttt{chall}_2[i] = 0$, we set $\texttt{resp}[i]_0 = (\mathbf{y}[i], \overline{\mathbf{v}}[i])$ in the R-SDP version, respectively $\texttt{resp}[i]_0 = (\mathbf{y}[i], \overline{\mathbf{v}}_G[i])$ for the R-SDP(G) version, and $\texttt{resp}[i]_1 = \texttt{cmt}_1[i]$. The commitment $\texttt{cmt}_1[i]$ is added as the provided response $\mathbf{y}[i], \overline{\mathbf{v}}[i]$, respectively $\overline{\mathbf{v}}_G[i]$, is only able to recover $\texttt{cmt}_0[i]$.

Signatures size: The signature consists of $\texttt{Salt}, \texttt{digest}_{\texttt{cmt}}, \texttt{digest}_{\texttt{chall}_2}, \texttt{Path}, \texttt{Proof}$ and the second response $\texttt{resp}$.

Since $\texttt{chall}_2$ has weight w, the response (respectively signature) of the fast version consists of precisely w times of $(\texttt{Seed}[i], \texttt{cmt}_0[i])$ (in this case the $\texttt{Path}$ and $\texttt{Proof}$ are set exactly to be $\texttt{Seed}[i], \texttt{cmt}_0[i]$) and $(t-w)$ times of $(\mathbf{y}[i], \overline{\mathbf{v}}[i], \texttt{cmt}_1[i])$, respectively of $(\mathbf{y}[i], \overline{\mathbf{v}}_G[i], \texttt{cmt}_1[i])$.

Each vector in $(\mathbf{y}, \overline{\mathbf{v}})$, respectively $(\mathbf{y}, \overline{\mathbf{v}}_G)$, requires ideally $(n\lceil\log_2(p)\rceil, n\lceil\log_2(z)\rceil)$, respectively $(n\lceil\log_2(p)\rceil, m\lceil\log_2(z)\rceil)$ bits to store them. To increase usability of these packed vectors, we pack each vector separately in a byte string, resulting in a small increase in signature size. We denote this *Rounding to the next Byte* by $\mathsf{R2B}(x) = \lfloor (x+7)/8 \rfloor \cdot 8$ in the equations below, indicating the number of bits necessary for the byte string containing x. Further details on this padding and its implications are provided in Section 5.4.

For the fast versions, this results in

$$|\mathsf{Sgn}| = \underbrace{6\lambda}_{\texttt{Salt},\texttt{digest}_{\texttt{cmt}},\texttt{digest}_{\texttt{chall}_2}} + w \cdot \underbrace{3\lambda}_{\texttt{resp}[i],\texttt{chall}_2[i]=1} + (t-w) \cdot \underbrace{(2\lambda + \mathsf{R2B}(n\lceil\log_2(p)\rceil) + \mathsf{R2B}(m\lceil\log_2(z)\rceil))}_{\texttt{resp}[i],\texttt{chall}_2[i]=0}$$

for R-SDP(G) and

$$|\mathsf{Sgn}| = \underbrace{6\lambda}_{\texttt{Salt},\texttt{digest}_{\texttt{cmt}},\texttt{digest}_{\texttt{chall}_2}} + w \cdot \underbrace{3\lambda}_{\texttt{resp}[i],\texttt{chall}_2[i]=1} + (t-w) \cdot \underbrace{(2\lambda + \mathsf{R2B}(n\lceil\log_2(p)\rceil) + \mathsf{R2B}(n\lceil\log_2(z)\rceil))}_{\texttt{resp}[i],\texttt{chall}_2[i]=0}$$

for R-SDP.

In the balanced and small version, the $\texttt{Path}, \texttt{Proof}$ parts do not consist of exactly w entries, but can be compressed by sending the seed path and classical Merkle proof nodes as explained above. This results in

$$\begin{aligned} |\mathsf{Sgn}| = & \underbrace{6\lambda}_{\texttt{Salt},\texttt{digest}_{\texttt{cmt}},\texttt{digest}_{\texttt{chall}_2}} + (t-w)\underbrace{(2\lambda + \mathsf{R2B}(n\lceil\log_2(p)\rceil) + \mathsf{R2B}(m\lceil\log_2(z)\rceil))}_{\texttt{resp}[i],\texttt{chall}_2[i]=0} \\ & \underbrace{+ 3\lambda\left(\left\lfloor (t-w)\log_2\left(\frac{t}{t-w}\right) + \mathsf{HW}(t) - 1\right\rfloor\right)}_{\texttt{Path},\texttt{Proof}} \end{aligned} \qquad (1)$$

for R-SDP(G) and

$$
\begin{aligned}
|\mathtt{Sgn}| = \underbrace{6\lambda}_{\mathtt{Salt},\mathtt{digest}_{\mathtt{cmt}},\mathtt{digest}_{\mathtt{chall}_2}} &+ (t-w)\cdot\underbrace{\left(2\lambda + \mathsf{R2B}\left(n\lceil\log_2(p)\rceil\right) + \mathsf{R2B}\left(n\lceil\log_2(z)\rceil\right)\right)}_{\mathtt{resp}[i],\mathtt{chall}_2[i]=0} \\
&+ \underbrace{3\lambda\left(\left\lfloor (t-w)\log_2\left(\frac{t}{t-w}\right) + \mathsf{HW}(t) - 1\right\rfloor\right)}_{\mathtt{Path},\mathtt{Proof}} \qquad (2)
\end{aligned}
$$

for R-SDP.

Difference to implementation: Whenever we use a variable for counting or indexing, we start counting from 1 throughout Algorithm 1 to Algorithm 3. In the implementation itself we naturally start counting from 0. Whenever vectors are used as input for Hash, we use them in their bit-packed form.

2.2.3 Verification

The algorithm Verify is given the public data, i.e., the security parameter λ, the restriction $\mathbb{E}$, parametrized through its generator g, the number of rounds t, the weight of the second challenge w and c defined as $2t-1$. Verify takes as input the public key `pk`, the message `Msg` and the signature `Sgn`. The algorithm outputs the Boolean value `True`/ `False` depending whether the signature `Sgn` is valid or not.

We distinguish between the R-SDP and R-SDP(G) variant by colors, that is: steps which are only required for R-SDP(G) are in orange and on the left of the algorithm, and R-SDP steps are in teal on the right side.

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

```
Input: pk: $(\mathtt{Seed_{pk}}, \mathbf{s})$ public key;
       $\mathtt{Msg} \in \{0,1\}^*$: message;
       Sgn: $(\mathtt{Salt}, \mathtt{digest_{cmt}}, \mathtt{digest_{chall_2}}, \mathtt{Path}, \mathtt{Proof}, \mathtt{resp})$ signature;
Output: $\{\mathtt{True}, \mathtt{False}\}$;
Data: $\lambda$: 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
1   $(\overline{\mathbf{W}}, \mathbf{V}) \leftarrow \mathsf{CSPRNG}\text{—}_{\mathbb{F}_z^{m\times(n-m)}\times\mathbb{F}_p^{(n-k)\times k}}(\mathtt{Seed_{pk}} \mid 3t+2)$  |  $\mathbf{V} \leftarrow \mathsf{CSPRNG}\text{—}_{\mathbb{F}_p^{(n-k)\times k}}(\mathtt{Seed_{pk}} \mid 3t+2)$
2   $\mathbf{H} \leftarrow [\mathbf{V} \mid \mathrm{Id}_{n-k}]$
3   $\overline{\mathbf{M}} \leftarrow [\overline{\mathbf{W}} \mid \mathrm{Id}_m]$
 // Computing challenges
4   $\mathtt{digest_{Msg}} \leftarrow \mathsf{Hash}(\mathtt{Msg})$
5   $\mathtt{digest_{chall_1}} \leftarrow \mathsf{Hash}(\mathtt{digest_{Msg}} \mid \mathtt{digest_{cmt}} \mid \mathtt{Salt})$
6   $\mathtt{chall_1} \leftarrow \mathsf{CSPRNG}\text{—}_{(\mathbb{F}_p^*)^t}(\mathtt{digest_{chall_1}} \mid t+c)$
7   $\mathtt{chall_2} \leftarrow \mathsf{CSPRNG}\text{—}_{\mathcal{B}_{(t,w)}}(\mathtt{digest_{chall_2}} \mid t+c+1)$
 // Computing commitments
8   $(\mathtt{Seed}[i])_{i:\mathtt{chall_2}[i]=1} \leftarrow \mathsf{RebuildLeaves}(\mathtt{Path} \mid \mathtt{chall_2} \mid \mathtt{Salt})$
9   for $i$ from 1 to $t$ do
10      if $\mathtt{chall_2}[i] = 1$ then
11          $\mathtt{cmt_1}[i] \leftarrow \mathsf{Hash}(\mathtt{Seed}[i] \mid \mathtt{Salt} \mid i+c)$
12          $\overline{\mathbf{e}}'_G[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG}\text{—}_{\mathbb{F}_z^m\times\mathbb{F}_p^n}(\mathtt{Seed}[i] \mid \mathtt{Salt} \mid i+c)$  |  $\overline{\mathbf{e}}'[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG}\text{—}_{\mathbb{F}_z^n\times\mathbb{F}_p^n}(\mathtt{Seed}[i] \mid \mathtt{Salt} \mid i+c)$
            $\overline{\mathbf{e}}'[i] \leftarrow \overline{\mathbf{e}}'_G[i]\overline{\mathbf{M}}$
            for $j$ from 1 to $n$ do
13              $\mathbf{e}'[i]_j \leftarrow g^{\overline{\mathbf{e}}[i]_j}$
14          $\mathbf{y}[i] \leftarrow \mathbf{u}'[i] + \mathtt{chall_1}[i]\mathbf{e}'[i]$
15      if $\mathtt{chall_2}[i] = 0$ then
16          $\mathtt{cmt_1}[i] \leftarrow \mathtt{resp}[i]_1$
17          $(\mathbf{y}[i], \overline{\mathbf{v}}_G[i]) \leftarrow \mathtt{resp}[i]_0$  |  $(\mathbf{y}[i], \overline{\mathbf{v}}[i]) \leftarrow \mathtt{resp}[i]_0$
            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
18              $\mathbf{v}[i]_j \leftarrow g^{\overline{\mathbf{v}}[i]_j}$
19          $\mathbf{y}'[i] \leftarrow \mathbf{v}[\mathbf{i}] \star \mathbf{y}[i]$
20          $\mathbf{s}'[i] \leftarrow \mathbf{y}'[i]\mathbf{H}^\top - \mathtt{chall_1}[i]\mathbf{s}$
21          $\mathtt{cmt_0}[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}_G[i] \mid \mathtt{Salt} \mid i+c)$  |  $\mathtt{cmt_0}[i] \leftarrow \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}[i] \mid \mathtt{Salt} \mid i+c)$
 // Checking digests
22  $\mathtt{digest_{cmt_0}} \leftarrow \mathsf{RecomputeRoot}(\mathtt{cmt_0} \mid \mathtt{Proof} \mid \mathtt{chall_2})$
23  $\mathtt{digest_{cmt_1}} \leftarrow \mathsf{Hash}(\mathtt{cmt_1}[1] \mid \cdots \mid \mathtt{cmt_1}[t])$
24  $\mathtt{digest'_{cmt}} \leftarrow \mathsf{Hash}(\mathtt{digest_{cmt_0}} \mid \mathtt{digest_{cmt_1}})$
25  $\mathtt{digest'_{chall_2}} \leftarrow \mathsf{Hash}(\mathbf{y}[1] \mid \cdots \mid \mathbf{y}[t] \mid \mathtt{digest_{chall_1}})$
26  if $\mathtt{digest_{cmt}} = \mathtt{digest'_{cmt}}$ and $\mathtt{digest_{chall_2}} = \mathtt{digest'_{chall_2}}$ then
27      return True
28  return False
```

Recovering public key: The verifier can compute the public key, either consisting of $\mathbf{H}$ or $\overline{\mathbf{M}}, \mathbf{H}$ in the case of R-SDP(G), using CSPRNG on $\mathtt{Seed_{pk}}$

Computing challenges: The verifier computes $\texttt{digest}_{\texttt{chall}_1}$ by hashing $\texttt{digest}_{\texttt{Msg}}, \texttt{digest}_{\texttt{cmt}}, \texttt{Salt}$. The first challenge $\texttt{chall}_1$ is then computed by CSPRNG on the input $\texttt{digest}_{\texttt{chall}_1}$ and similarly, the second challenge $\texttt{chall}_2$ by CSPRNG on the input $\texttt{digest}_{\texttt{chall}_2}$.

Computing the commitments: The verifier can rebuild the leaves $\texttt{Seed}[i]$, where $\texttt{chall}_2[i] = 1$, using the function RebuildLeaves. RebuildLeaves uses the $\texttt{Path}$ and $\texttt{Salt}$ from the signature, as well as the re-computed challenge $\texttt{chall}_2$, and derives the round seeds $\texttt{Seed}[i]$, for which $\texttt{chall}_2[i] = 1$, from it. Internally it thus computes a subset of the seed tree used during signature generation.

If $\texttt{chall}_2[i] = 1$, the commitment $\texttt{cmt}_1[i]$ is computed by

$$\texttt{cmt}_1[i] = \mathsf{Hash}(\texttt{Seed}[i] \mid \texttt{Salt} \mid i + c).$$

Note that we do not need to recover $\texttt{cmt}_0[i]$, as we are provided with $\texttt{Proof}$, able to recover $\texttt{digest}_{\texttt{cmt}_0}$.

The verifier then reconstructs $\mathbf{y}[i]$, by computing $\mathbf{u}'[i]$ and either $\overline{\mathbf{e}}'[i]$ in the R-SDP version, or $\overline{\mathbf{e}}'_G[i]$ in the R-SDP(G) version, using CSPRNG on the input $\texttt{Seed}[i], \texttt{Salt}, i + c$. The verifier then computes $\mathbf{e}'[i]$ and

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

If $\texttt{chall}_2[i] = 0$, $\texttt{cmt}_1[i]$ is recovered from $\texttt{resp}[i]_1$. For the commitment $\texttt{cmt}_0[i]$, the verifier first has to compute $\mathbf{s}'[i]$. For this, the verifier recovers $(\mathbf{y}[i], \overline{\mathbf{v}}[i])$ from $\texttt{resp}[i]_0$ in the case of R-SDP and $(\mathbf{y}[i], \overline{\mathbf{v}}_G[i])$ from $\texttt{resp}[i]_0$ in the case of R-SDP(G). As a first step, the verifier checks if $\overline{\mathbf{v}}[i] \in \mathbb{F}_z^n$, respectively if $\overline{\mathbf{v}}_G[i] \in \mathbb{F}_z^m$.

The verifier can then construct $\mathbf{v}[i]$ and compute $\mathbf{y}'[i] = \mathbf{v}[i] \star \mathbf{y}[i]$ and

$$\mathbf{s}'[i] = \mathbf{y}'[i]\mathbf{H}^\top - \texttt{chall}_1\mathbf{s}.$$

The commitment is then computed as

$$\begin{aligned} \texttt{cmt}_0[i] &= \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}[i] \mid \texttt{Salt} \mid i + c), && \text{for R-SDP} \\ \texttt{cmt}_0[i] &= \mathsf{Hash}(\mathbf{s}'[i] \mid \overline{\mathbf{v}}_G[i] \mid \texttt{Salt} \mid i + c), && \text{for R-SDP}(G). \end{aligned}$$

Checking digests: Given $\mathbf{y}[1], \ldots, \mathbf{y}[t]$, the verifier can recompute all digests, $\texttt{digest}_{\texttt{cmt}_0}, \texttt{digest}_{\texttt{cmt}_1}$, and thus also the candidates for $\texttt{digest}'_{\texttt{cmt}}$ and $\texttt{digest}'_{\texttt{chall}_2}$.

To recompute $\texttt{digest}_{\texttt{cmt}_0}$, the function RecomputeRoot is used. It takes the $\texttt{Proof}$ from the signature, the $\texttt{cmt}_0[i]$ that the verifier re-computed (indicated by $\texttt{chall}_2[i] = 0$), as well as the second challenge $\texttt{chall}_2$. Using that information, RecomputeRoot internally re-computes the root of a Merkle tree. This root represents $\texttt{digest}_{\texttt{cmt}_0}$. Like in TreeRoot, the fast version uses a slightly different tree structure, but similarly to the balanced and fast version, generates a tree root through iterative hashing.

The verifier accepts the signature and outputs $\texttt{True}$, if the candidates $\texttt{digest}'_{\texttt{cmt}}$ and $\texttt{digest}'_{\texttt{chall}_2}$ coincide with the sent values for $\texttt{digest}_{\texttt{cmt}}$ and $\texttt{digest}_{\texttt{chall}_2}$.

2.3 Auxiliary Primitives

CROSS requires two auxiliary primitives: a cryptographically secure pseudo-random number generator (CSPRNG) and a cryptographic hash function (Hash). All CSPRNGs and Hashes variants employed in CROSS benefit from the cryptographic guarantees provided by the NIST standard extendable-output functions (XOFs) SHAKE128 and SHAKE256 (as specified in FIPS202), which in turn exhibit a collision

Table 3: SHAKE variants employed to realize the auxiliary primitives used in CROSS, for each NIST category

NIST category	**CSPRNG**	Hash
1	SHAKE128	SHAKE128 with 256-bit output
3	SHAKE256	SHAKE256 with 384-bit output
5	SHAKE256	SHAKE256 with 512-bit output

resistance equal to $\texttt{min}(2^{d/2}, 2^{128})$ and $\texttt{min}(2^{d/2}, 2^{256})$, respectively, where d is bit-length of their output result. Table 3 summarizes the choice we made for realizing each CSPRNG and each Hash function to ensure that CROSS exhibits the security level prescribed by the NIST categories.

Starting from one of the SHAKE variants as per Table 3, which acts as a function from arbitrary length binary strings to arbitrary length binary strings, we realize the CSPRNGs as follows.
$\mathsf{CSPRNG}\text{–}_{\{0,1\}^{a\lambda}\times\{0,1\}^{a\lambda}}(\cdot)$, $\mathsf{CSPRNG}\text{–}_{\{0,1\}^{a\lambda}}(\cdot)$ where a is a positive integer: The output of SHAKE is interpreted as either a single binary string or multiple binary strings, depending on how many of them are needed, by simply splitting a single output into appropriately sized parts. In case pair of binary strings is required as an output, its first (leftmost) element is sampled first.

$\mathsf{CSPRNG}\text{–}_{\mathbb{F}_z^m}(\cdot)$, $\mathsf{CSPRNG}\text{–}_{\mathbb{F}_z^n}(\cdot)$: A rejection sampling strategy is employed to turn the binary string output from SHAKE into sequences of numbers in $\mathbb{F}_z$. The approach extracts sequences of bits, each one $\lceil\log_2(z)\rceil$ long from SHAKE, reinterprets the output bits as the natural binary encoding of an integer and, if the resulting value is in $\mathbb{F}_z$, the value is concatenated to the sequence. If the resulting integer is larger, it is discarded, and a new $\lceil\log_2(z)\rceil$ long bit sequence is extracted from SHAKE. The procedure is repeated until enough elements of $\mathbb{F}_z$ are generated. The rationale for extracting sequences of $\lceil\log_2(z)\rceil$ bits from SHAKE and not longer ones, is that $\lceil\log_2(z)\rceil$ is the amount of binary digits on which all the numbers modulo z, i.e., all integers between 0 and $z-1$ can be encoded. Drawing only such an amount minimizes the amount of discarded bits from the SHAKE output to generate each value in $\mathbb{F}_z$.

$\mathsf{CSPRNG}\text{–}_{\mathbb{F}_p^{(n-k)\times k}}(\cdot)$: The same rejection sampling approach employed to generate multiple elements of $\mathbb{F}_z$ is also employed to generate multiple elements of $\mathbb{F}_p$, with the only difference being that the sequence of bits extracted from SHAKE at every attempt to generate an element of $\mathbb{F}_p$ is $\lceil\log_2(p)\rceil$ bits long.

$\mathsf{CSPRNG}\text{–}_{\mathbb{F}_z^{m\times(n-m)}\times\mathbb{F}_p^{(n-k)\times k}}(\cdot)$: Generating a pair of matrices, with elements coming from $\mathbb{F}_z$ (the first) and $\mathbb{F}_p$ (the second) is done sequentially by rejection sampling. Therefore, we first generate an element of $\mathbb{F}_z^{m\times(n-m)}$ by generating each of its elements drawing $\lceil\log_2(z)\rceil$ long bit strings from SHAKE until $m\times(n-m)$ elements of $\mathbb{F}_z$ are obtained, and subsequently, we generate $(n-k)\times k$ elements of $\mathbb{F}_p$.

$\mathsf{CSPRNG}\text{–}_{\mathbb{F}_z^m\times\mathbb{F}_p^n}(\cdot)$, $\mathsf{CSPRNG}\text{–}_{\mathbb{F}_z^n\times\mathbb{F}_p^n}(\cdot)$: Generating a pair of vectors with elements coming from $\mathbb{F}_z$ (the first) and $\mathbb{F}_p$ (the second) is done in the same fashion as generating matrices, i.e., generating their elements via rejection sampling from the SHAKE output.

$\mathsf{CSPRNG}\text{–}_{(\mathbb{F}_p^*)^t}(\cdot)$: Sampling t elements in $\mathbb{F}_p^*$ amounts to sampling uniformly numbers in $\{1, 2, \dots, p-1\}$. We achieve this via rejection sampling with the same strategy as before. Thus, numbers are uniformly drawn from $\{0, 1, \dots, p-2\}$ and deterministically adding 1 to the result.

$\mathsf{CSPRNG}\text{–}_{\mathcal{B}_{(t,w)}}(\cdot)$: Sampling uniformly from the Hamming ball of length t and radius w is done by observing that taking any of its elements and applying to it a random permutation of the coordinates (i.e., a permutation of the bits constituting the constant weight string representing it) amounts to drawing a random element of $\mathcal{B}_{(t,w)}$. We adopt this approach, and apply a uniformly randomly picked permutation over t elements to the binary string 1^w0^{t-w}, employing a Fisher-Yates shuffle [25]. The random indices for the Fisher-Yates shuffle are obtained via rejection sampling from output bits of SHAKE.

Domain separation: To preserve the domain separation between using SHAKE as CSPRNG and as Hash, we append a 16-bit integer to each input of CSPRNG and Hash. For values $\geq 2^{15}$ SHAKE is used as Hash, whereas each value $< 2^{15}$ denotes a CSPRNG call, respectively. That is, the most significant bit of this integer denotes the corresponding usage of SHAKE. In addition, we use the lower 15 bits to separate different CSPRNG and Hash instances, if necessary. In Algorithm 1 to Algorithm 3, the cases where the lower 15 bits are specifically used for further separation are indicated by the additional integer in the input of the corresponding Hash and CSPRNG calls.

3 Security

3.1 Hardness of Restricted Decoding

The security of CROSS relies on the hardness of restricted decoding problems. This section gives an overview of the state-of-the-art solvers for these problems. For further details, we refer to the detailed security guide [39].

3.1.1 Underlying Hardness Assumptions

CROSS relies on the hardness of restricted decoding problems which are defined as follows.

Problem 3. ***Restricted Syndrome Decoding Problem (R-SDP)***
Let $g \in \mathbb{F}_p^*$ be of order z, $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_p^{n-k}$, and $\mathbb{E} = \{g^i \mid i \in \{1, \dots, z\}\} \subset \mathbb{F}_p^*$.
Does there exists $\mathbf{e} \in \mathbb{E}^n$ with $\mathbf{eH}^\top = \mathbf{s}$?

R-SDP is tightly connected to other well-known decoding problems. In particular, for $z = p - 1$, we recover syndrome decoding with full weight; for $z = 2$, R-SDP is related to the subset sum problem over finite fields. CROSS uses R-SDP with $p = 127$ and $z = 7$. Nevertheless, it is unsurprising that the decisional version of R-SDP is NP-complete for arbitrary restriction $\mathbb{E}$.

Theorem 4. The decisional version of R-SDP (Problem 3) is NP-complete.

The proof for the NP-completeness can be found in [42], as well as in the security guide [39].

R-SDP can be generalized by considering a subgroup $(G, \star) \leq (\mathbb{E}^n, \star)$ as

$$G = \langle \mathbf{a}_1, \dots, \mathbf{a}_m \rangle = \left\{ \star_{i=1}^m \mathbf{a}_i^{\overline{u}_i} \mid \overline{u}_i \in \mathbb{F}_z \right\},$$

for some $m < n$, where the star denotes component-wise multiplication. A variant of CROSS relies on this generalization, to which we refer as R-SDP(G).

Problem 5. ***Restricted Syndrome Decoding Problem with subgroup G (R-SDP(G))***
Let $G = \langle \mathbf{a}_1, \dots, \mathbf{a}_m \rangle$, for $\mathbf{a}_i \in \mathbb{E}^n$, $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$, and $\mathbf{s} \in \mathbb{F}_p^{n-k}$.
Does there exist a vector $\mathbf{e} \in G$ with $\mathbf{eH}^\top = \mathbf{s}$?

CROSS uses R-SDP(G) with $p = 509$ and $z = 127$.

Uniqueness of solution: For instances with planted solution, the average number of solutions for R-SDP and R-SDP(G) is computed as $1 + (z^n - 1)p^{k-n}$ and as $1 + (z^m - 1)p^{k-n}$, respectively. In both cases, the CROSS parameters are chosen such that this average number of solutions is small.

3.1.2 Combinatorial Solvers for R-SDP

Combinatorial solvers for R-SDP are inspired by Information Set Decoding (ISD) algorithms [11, 12, 22, 38] for the syndrome decoding problem and the best-known algorithms for the subset sum problem [10, 30].

A framework for combinatorial solvers: A standard technique in generic decoders is bringing $\mathbf{H}$ into quasi-systematic form

$$\mathbf{H} = \begin{pmatrix} \mathrm{Id}_{n-k-\ell} & \mathbf{H}_1 \\ 0 & \mathbf{H}_2 \end{pmatrix},$$

where $\mathbf{H}_1 \in \mathbb{F}_p^{(n-k-\ell)\times(k+\ell)}, \mathbf{H}_2 \in \mathbb{F}_p^{\ell\times(k+\ell)}$. This inherently splits the unknown error vector into $\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2) \in \mathbb{E}^{n-k-\ell} \times \mathbb{E}^{k+\ell}$. Thus, we get the system of two equations

$$\begin{aligned} \mathbf{e}_1 + \mathbf{e}_2\mathbf{H}_1^\top &= \mathbf{s}_1 \text{ and} \\ \mathbf{e}_2\mathbf{H}_2^\top &= \mathbf{s}_2, \end{aligned}$$

where $\mathbf{s}_1 \in \mathbb{F}_p^{n-k-\ell}, \mathbf{s}_2 \in \mathbb{F}_p^{\ell}$. To solve this system, one enumerates solutions $\mathbf{e}_2$ of the second equation $\mathbf{e}_2\mathbf{H}_2^\top = \mathbf{s}_2$ and checks for each one if the remaining $\mathbf{e}_1 = \mathbf{s}_1 - \mathbf{e}_2\mathbf{H}_1^\top$ completes it to a valid, i.e., restricted, solution. In the following, we discuss methods for the enumeration of $\mathbf{e}_2$.

Collision search: Split $\mathbf{e}_2$ into $(\mathbf{e}_a, \mathbf{e}_b)$. Then, a pair $(\mathbf{e}_a, \mathbf{e}_b)$ solves $(\mathbf{e}_a, \mathbf{e}_b)\mathbf{H}_2^\top = \mathbf{s}_2$ if and only if $(\mathbf{e}_a,\ \mathbf{0})\mathbf{H}_2^\top = \mathbf{s}_2 - (\mathbf{0},\ \mathbf{e}_b)\mathbf{H}_2^\top$. To find such pairs, construct the lists

$$\begin{aligned} \mathcal{L}_a &\coloneqq \left\{(\mathbf{e}_a, \quad (\mathbf{e}_a,\ \mathbf{0})\mathbf{H}_2^\top) \mid \mathbf{x}_a \in \mathbb{E}^{\lfloor\frac{k+\ell}{2}\rfloor}\right\} \text{ and} \\ \mathcal{L}_b &\coloneqq \left\{(\mathbf{e}_b, \mathbf{s}_2 - (\mathbf{0},\ \mathbf{e}_b)\mathbf{H}_2^\top) \mid \mathbf{x}_b \in \mathbb{E}^{\lceil\frac{k+\ell}{2}\rceil}\right\}, \end{aligned}$$

and perform a collision search [22, 29, 38]. Using a hash table, this costs approximately $2z^{(k+\ell)/2} + z^{k+\ell}p^{-\ell}$ vector operations.

Multilevel solvers via representations: The best-known solvers for SDP and subset sum problems generalize the described collision search to multiple levels [10, 11, 28, 30]. The basic idea behind this improvement is to split $\mathbf{e}_2 = \mathbf{e}_a + \mathbf{e}_b$, which allows for several representations $\mathbf{e}_a, \mathbf{e}_b$ of a given $\mathbf{e}_2$. The effectiveness of such solvers depends on the number of representations, which is determined by the additive structure of $\mathbb{E}$ for R-SDP [5, 16]. The restriction $\mathbb{E} = \{1, 2, 4, 8, 16, 32, 64\}$ used by CROSS has no additive structure apart from $2e \in \mathbb{E}$ for all $e \in \mathbb{E}$. As a consequence, only minimal improvements over the basic collision search seem to be possible.

Shifting $\mathbb{E}$: An R-SDP instance can be transformed into an instance with a modified restriction. Denote the columns of $\mathbf{H}$ as $\mathbf{h}_0, \dots, \mathbf{h}_{n-1}$, set $\mathbf{x} = (x, \dots, x)$, and define

$$\widetilde{\mathbf{H}} = \begin{pmatrix} \mathbf{h}_0 \cdot g^{i_0}, & \dots, & \mathbf{h}_{n-1} \cdot g^{i_{n-1}} \end{pmatrix} \quad \text{and} \quad \tilde{\mathbf{s}} = \mathbf{s} - \mathbf{x}\mathbf{H}^\top.$$

Then, $\tilde{\mathbf{e}} = \mathbf{e} \star (g^{i_0}, \dots, g^{i_{n-1}}) - \mathbf{x}$ is a solution to $(\widetilde{\mathbf{H}}, \tilde{\mathbf{s}})$ with restriction $\widetilde{\mathbb{E}} = \{e - x \mid e \in \mathbb{E}\}$. The shifted instance can be solved by adapting the algorithms described above.

- *Weight distribution:* For $x \in \mathbb{E}$, the weight of the modified instance follows a binomial distribution instead of being full weight. Enumerating vectors of reduced weight decreases the cost of the described solvers.
- *Additive structure:* For parameters used in CROSS, $\widetilde{\mathbb{E}}$ does not possess additive structure when shifting with $x \in \mathbb{E}$. This reduces the effectiveness of representation-based solvers.

Expected security strength: In Section 4, Table 5 summarizes the costs of the combinatorial solvers for R-SDP as utilized by CROSS. For these parameters, combining the representation technique with shifting the error set yields the best performance. For a detailed explanation of the attack parameters and formulae for bit-complexity estimation, we refer the reader to the security guide [39].

3.1.3 Algebraic Solvers for R-SDP

Similar as other decoding problems [1, 6], algebraic methods can be used to solve R-SDP.

Modeling R-SDP: R-SDP can be modeled as the system of polynomial equations

$$\begin{aligned} \mathbf{x}\mathbf{H}^\top &= \mathbf{s}, \\ \mathbf{x}_i^z &= 1 \quad \forall i \in \{1, \dots, n\}. \end{aligned}$$

Solving complexity: The polynomial system can be solved by computing a Gröbner basis of the corresponding ideal. State-of-the-art solvers include F4 [23], F5 [24] and the XL algorithms [21]. The cost of these algorithms has been studied extensively in literature, see, e.g., [18]. A detailed analysis of the polynomial system given above is provided in [15], which reaches the conclusion that this algebraic approach is not competitive with the combinatorial solvers.

Hybrid approach: The complexity of algebraic attacks can be improved by hybrid techniques. The basic idea is to add further equations to the system of polynomials. This reduces the complexity of solving the system at the cost of repeating the process several times. For CROSS, [15] observes that the cost of the hybrid attack is optimized by bruteforcing almost z^k entries of the error vector.

3.1.4 Solvers for R-SDP(G)

Incorporating G: The set of valid error vectors is $\{g^{\overline{\mathbf{e}}} \mid \overline{\mathbf{e}} \in \ker(\overline{\mathbf{H}})\}$ for $\overline{\mathbf{H}} \in \mathbb{F}_z^{(n-m)\times n}$. To incorporate this into the described collision search, $\overline{\mathbf{H}}$ is brought into quasi-systematic form

$$\overline{\mathbf{H}} = \begin{pmatrix} \mathrm{Id}_{n-k-\ell} & \overline{\mathbf{H}}_1 \\ \mathbf{0} & \overline{\mathbf{H}}_2 \end{pmatrix},$$

where $\overline{\mathbf{H}}_1 \in \mathbb{F}_z^{(n-k-\ell)\times(k+\ell)}, \overline{\mathbf{H}}_2 \in \mathbb{F}_z^{(k+\ell-m)\times(k+\ell)}$. Then, the lists are constructed as

$$\begin{aligned} \mathcal{L}_a &:= \left\{ \left(\overline{\mathbf{e}}_a, \quad (\overline{\mathbf{e}}_a, \mathbf{0})\overline{\mathbf{H}}_2^\top, \qquad (g^{\overline{\mathbf{e}}_a}, \mathbf{0})\mathbf{H}_2^\top \right) \mid \overline{\mathbf{e}}_a \in \mathbb{F}_z^{\lfloor \frac{k+\ell}{2} \rfloor} \right\} \text{ and} \\ \mathcal{L}_b &:= \left\{ \left(\overline{\mathbf{e}}_b, \ -(\mathbf{0}, \overline{\mathbf{e}}_b)\overline{\mathbf{H}}_2^\top, \ \mathbf{s}_2 - (\mathbf{0}, g^{\overline{\mathbf{e}}_b})\mathbf{H}_2^\top \right) \mid \overline{\mathbf{e}}_b \in \mathbb{F}_z^{\lceil \frac{k+\ell}{2} \rceil} \right\}. \end{aligned}$$

By matching the second and third entry of each list element, the number of collisions is reduced.

A minor improvement for weak keys: A small fraction of the codes spanned by matrices $\overline{\mathbf{H}}$ contain subcodes with small, disjoint supports. For the sake of a conservative analysis, we assume that subcodes that occur with probability at least $2^{-\lambda}$ are available to the solver. These subcodes can be used to reduce the list sizes moderately.

An alternative collision attack: An alternative collision attack is proposed in [15]. The van Oorschot-Wiener algorithm [40] enables a reduction in the required memory. The estimates for the time complexities confirm the security level of the parameters used by CROSS.

Expected security strength: In Section 4, Table 6 summarizes the costs of the combinatorial solvers for R-SDP(G) as utilized by CROSS. For a detailed explanation of the attack parameters and formulae for bit-complexity estimation, we refer the reader to the security guide [39].

3.2 Security of the Protocol

In the following, we present two forgery attacks derived from [9]. The former is adapted from [31] for weighted challenges, while the latter is a new attack. The parameter choice is based on the complexity of the latter. We then present a security proof for the protocol, showing that CROSS is EUF-CMA secure.

3.2.1 Forgery Attacks

In this section, we describe two forgeries. We conservatively estimate the cost of these forgeries in terms of CROSS operations. In our analysis, one elementary operation corresponds to simulating several instructions the prover would perform. In particular, we conservatively estimate the cost of a CROSS operation as 2^5 instructions, as detailed at the end of this section. As we argue in Section 4, this allows us to easily assess the cost of such attacks so that the recommended CROSS parameters meet the NIST security categories.

First forgery: The first forgery we describe is relatively intuitive and attempts, for each round, to guess the first challenge $\texttt{chall}_1$ or the second challenge $\texttt{chall}_2$ (or both). The cost of this attack is given in the following proposition.

Proposition 6. The attack runs in average time $O\left(\frac{1}{P_\alpha(t,w,p)}\right)$, where

$$P_\alpha(t,w,p) = \sum_{w'=\max\{0,w-t+\alpha\}}^{\min\{w,\alpha\}} \frac{\binom{\alpha}{w'}\binom{t-\alpha}{w-w'}}{\binom{t}{w}} \left(\frac{1}{p-1}\right)^{(\alpha-w')+(w-w')}.$$

The overall cost of the forgery is estimated by optimizing over $\alpha \in \{0,\dots,t\}$.

Notice that the cost of the forgery of the previous proposition is in agreement with the optimal cheating probability of a dishonest prover against the (t,w)-fixed-weight repetition of a $(2,2)$-out-of-$(p-1,2)$ special sound protocol, as detailed in the security guide [39].

Second forgery: We now consider another forgery inspired by the attack in [31] to 5-pass schemes and optimized for the fixed-weight variant in [9]. The attack makes use of the fact that the second challenge is generated after the first challenge, and, furthermore, it is possible to generate multiple second challenges without modifying the commitments or the first challenge value. This way, one can split the forgery into two separate phases, where the overall cost is given by the sum of the two associated costs. Again, we exploit the fixed weight of the second challenge to optimize the round selection.

Proposition 7. The attack runs in average time

$$O\left(\min_{t^*\in\{0,\dots,t\}}\left\{\frac{1}{P_1(t,t^*,p)} + \frac{1}{P_2(t,t^*,w,p)}\right\}\right),$$

where

$$P_1(t,t^*,p) = \sum_{j=t^*}^{t}\binom{t}{j}\left(\frac{1}{p-1}\right)^j\left(1-\frac{1}{p-1}\right)^{t-j},$$

$$P_2(t,t^*,w,p) = \max_{\alpha\in\{w,\dots,t\}}\sum_{j=t^*}^{t}\frac{\binom{t}{j}\left(\frac{1}{p-1}\right)^j\left(1-\frac{1}{p-1}\right)^{t-j}}{P_1(t,t^*,p)}\sum_{w^*=\max\{0,\alpha-j\}}^{\min\{t-j,w\}}\frac{\binom{t-j}{w^*}\binom{j}{\alpha-w^*}}{\binom{t}{\alpha}}\frac{\binom{j}{w-w^*}}{\binom{t}{w}}.$$

Expected security strength: In Section 4, Table 7 summarizes the bit costs of the forgery attack for the set of parameters provided for CROSS. For a detailed explanation of the forgery procedure and formulae for running time estimation, we refer the reader to the security guide [39].

Finite regime considerations on forgery complexity: Providing parameters to match the NIST security categories requires quantifying the effort of attacking CROSS in terms of Boolean operations for comparison with the benchmark effort to be matched (breaking AES). Noting that a single forgery attempt takes at least a SHAKE call, and SHAKE is more expensive than AES, we target forgery probabilities slightly higher than that of guessing an AES key. Quantitative details are reported in the security guide [39].

3.2.2 Security Proof

As shown in [8, 9], the Fiat-Shamir transform of an interactive proof that is special sound and honest-verifier zero-knowledge is EUF-CMA secure. Proposition 1 proves that CROSS-ID is honest-verifier zero-knowledge and $(2, 2)$ special sound.

Theorem 8. CROSS is EUF-CMA secure.

Remark 9. In [9], the security is stated in expected polynomial time and not *strict* polynomial time. This is due to the fact that for fixed-weight challenges, the knowledge extractors defined in [8, Lemma 3] and [2, Lemma 2], which are the basis of [9], work in expected polynomial time and are allowed to reach exponential time. However, both extractors can be modified to be strict polynomial time at the cost of a negligible loss in success probability, as shown in [9].

4 Parameters and Expected Security Strength

This section outlines the parameter selection process for CROSS. The primary concern was ensuring the security of the system. Subsequently, we focused on selecting parameters that allow for efficient arithmetic. Balancing signature size and speed, the parameter selection consists of two phases:

i) Select the code parameters p, n, k and restriction parameters z, m to meet the NIST categories 1, 3, and 5, defined via the cost of breaking AES with 128, 192, or 256-bit keys.

ii) Determine the optimal number of rounds t and weight w of the fixed-weight challenge vector $\texttt{chall}_2$.

Possible values for p, z as well as n, k and m: In the first phase of the parameter selection process, we determined all (p, z)-pairs for which p prime with $17 \leq p \leq 2477$ and z prime with $z \mid p-1$, i.e., $\mathbb{F}_p^*$ admits a multiplicative subgroup of order z. For each such pair and code rates R in the range $0.3 \leq R \leq 0.7$, the minimal required code length n was determined such that the solvers reported in Section 3.1 yield the targeted security levels. In the case of R-SDP(G), the parameter m, i.e., the size of the subgroup, was also optimized.

Selecting code and restriction parameters: For the R-SDP variant of CROSS, we selected $p = 127$ and $z = 7$. While this choice incurs a slight penalty to signature size, it enables efficient arithmetic: both p and z are Mersenne primes, enabling an efficient modular reduction without a divisor functional unit. Furthermore, the elements of $\mathbb{F}_p$ and $\mathbb{F}_z$ are efficiently representable within a single byte.

The parameter m of the R-SDP(G) variant of CROSS provides additional flexibility in selecting parameters. We selected $p = 509$, as $\mathbb{F}_p^*$ admits a subgroup of order $z = 127$, enabling efficient Mersenne arithmetic for computation over $\mathbb{F}_z$. We furthermore use $g = 2$ as generator for R-SDP and $g = 16$ as generator for R-SDP(G).

Possible values for t and w: In the second phase of the parameter selection process, we determined valid (t, w)-pairs by selecting, for each possible t, the minimal w such that the cost of a forgery attack exceeds the targeted security level. We limited t to a maximum of 1536 as this exceeds the global minimum in signature size achievable for all instances. This global minimum results from the fact that for a sufficiently large value of t, the compression obtained by bringing w closer to t is outweighed by the sole increase of t.

Pruning for efficiency: For each NIST category, large sets of parameters are equivalent from a security standpoint. This allows pruning the parameter sets according to efficiency considerations. Since public and secret key are inherently of small size for CROSS (see Section 2.2.2), we selected the signature size as the primary space parameter for balancing trade-offs. Indeed, considering signature plus public key sizes does not alter the final results. For this phase, we use the number of rounds t as a proxy of the execution time, as both the signature and verification time in CROSS are proportional to it, albeit through different multiplicative factors.

Selecting number of rounds and challenge weight: For each NIST category, we propose three parameter sets, serving three optimization corners: computational speed (referred to as *fast*) in the signature and verification procedures, a balanced version (referred to as *balanced*) which aims for stability, and a version aiming for small signature sizes (referred to as *small*). For the *fast* corner, we chose the minimal number of rounds t applicable to achieve the desired security level. For the *small* corner, we chose either the number of rounds t yielding the global minimum in signature size or a smaller number of rounds t, resulting in at most 1.5% increase in signature size while decreasing the number of rounds (and thus the runtime of signing and verification procedures) by up to 37%. For the *balanced* corner, we chose an intermediate number of rounds yielding a reasonable trade-off in size and runtime.

Parameters sets: The final outcome of the parameter selection procedure is the set of parameters reported in Table 4.

Expected security strength: Table 5 and Table 6 present the computational cost of a key recovery attack against CROSS (see Section 3.1). The code parameters p, n, k and the restriction parameters z, m are selected to achieve NIST categories 1, 3, and 5, respectively. Table 7 illustrates the computational cost of forging a CROSS signature (see Section 3.2). The parameters t and w are selected to achieve the NIST categories 1, 3, and 5, respectively. For further details, the reader is referred to the security guide [39].

5 Implementation Techniques

5.1 Symmetric Primitives

The CSPRNG is used to generate pseudo-random bit-strings for the seed tree construction [14] or for sampling uniformly algebraic objects, such as vectors and matrices. For our choice, we performed a comparative benchmark of AES-CTR-DRBG [7] and SHAKE, the extendable output function standardized in NIST FIPS 202 [34].

The Hash function is used to construct a (Merkle-) tree of the commitments, to compute the digests from which challenges are sampled and to compute the commitments. As suitable candidates, we considered the NIST standard SHA-2 (standardized in [33]), SHA-3 and SHAKE (standardized in [34]) with digest sizes of 2λ for each security level. We chose FIPS-202 based primitives over SHA-2 since

Table 4: Parameter choices, keypair and signature sizes recommended for both CROSS-R-SDP and CROSS-R-SDP(G), assuming NIST categories 1, 3, and 5, respectively.

Algorithm and Security Category	**Optim. Corner**	p	z	n	k	m	t	w	**Pri. Key Size (B)**	**Pub. Key Size (B)**	**Signature Size (B)**
CROSS-R-SDP **1**	**fast**	127	7	127	76	-	157	82	32	77	18432
	balanced	127	7	127	76	-	256	215	32	77	13152
	small	127	7	127	76	-	520	488	32	77	12432
CROSS-R-SDP **3**	**fast**	127	7	187	111	-	239	125	48	115	41406
	balanced	127	7	187	111	-	384	321	48	115	29853
	small	127	7	187	111	-	580	527	48	115	28391
CROSS-R-SDP **5**	**fast**	127	7	251	150	-	321	167	64	153	74590
	balanced	127	7	251	150	-	512	427	64	153	53527
	small	127	7	251	150	-	832	762	64	153	50818
CROSS-R-SDP(G) **1**	**fast**	509	127	55	36	25	147	76	32	54	11980
	balanced	509	127	55	36	25	256	220	32	54	9120
	small	509	127	55	36	25	512	484	32	54	8960
CROSS-R-SDP(G) **3**	**fast**	509	127	79	48	40	224	119	48	83	26772
	balanced	509	127	79	48	40	268	196	48	83	22464
	small	509	127	79	48	40	512	463	48	83	20452
CROSS-R-SDP(G) **5**	**fast**	509	127	106	69	48	300	153	64	106	48102
	balanced	509	127	106	69	48	356	258	64	106	40100
	small	509	127	106	69	48	642	575	64	106	36454

Table 5: Bit-complexity estimates for solvers of R-SDP with parameters as used by CROSS. More details, such as the optimal attack parameters, can be found in the security guide [39].

Parameter set (p, z, n, k)	# solutions	Collision search	Representation technique	Shifted representations
Category 1 (127, 7, 127, 76)	2.1	150	162	143
Category 3 (127, 7, 187, 111)	1.0	213	229	207
Category 5 (127, 7, 251, 150)	1.4	281	301	274

Table 6: Bit-complexity estimates for solvers of R-SDP(G) with parameters as used by CROSS. More details, such as the optimal attack parameters, can be found in the security guide [39].

Parameter set (p, z, n, k, m)	# solutions	Collision search	Collision search with small-support subcodes	Analysis in [15]
Category 1 (509, 127, 55, 36, 25)	15.7	152	143	145
Category 3 (509, 127, 79, 48, 40)	2.8	217	210	212
Category 5 (509, 127, 106, 69, 48)	7.8	286	272	276

Table 7: Bit-complexity estimates for signature forgery with parameters as used by CROSS. More details, such as the optimal attack parameters, can be found in the security guide [39].

R-SDP	Category 1			Category 3			Category 5		
	fast	balanced	short	fast	balanced	short	fast	balanced	short
t	157	256	520	239	384	580	321	512	832
w	82	215	488	125	321	527	167	427	762
forgery	128	128	128	192	192	192	256	256	256

R-SDP(G)	Category 1			Category 3			Category 5		
	fast	balanced	short	fast	balanced	short	fast	balanced	short
t	147	256	512	224	268	512	300	356	642
w	76	220	484	119	196	463	153	258	575
forgery	128	128	128	192	192	193	256	256	256

- they have a smaller executable code size in memory-constrained devices such as microcontrollers,
- they have a reduced area consumption in FPGA/ASIC implementations, thanks to the possibility of sharing the SHA-3/SHAKE inner state logic between the CSPRNG and the Hash,
- they minimize the Boolean degree of the round function, allowing for greater degree of protection against power side-channel attacks.

Choice for CSPRNG: For benchmarking the AES-CTR-DRBG, we consider a software implementation of the AES block cipher and the usage of Intel AES-NI ISA extensions. Our benchmark results show that the SHAKE extendable output functions yield better overall performances compared to the use of AES-CTR-DRBG. CROSS hence uses SHAKE-128 for NIST security category 1 and SHAKE-256 for NIST security categories 3 and 5.

Choice for Hash: Our benchmarks obtained a small execution time gain by employing SHA-2 (in the few percentage points range) over SHA-3. To ensure collision resistance we use of SHAKE128 with a 256 bit output for category 1, and SHAKE256 with 384 and 512 bit output for categories 3 and 5, respectively.

The selected SHAKE functions share the same collision resistance of SHA-3 instances with the same output length (as stated in [34]), while processing the input information faster (thanks to their larger *rate* parameter). SHAKE further improves on the required code complexity in software implementations and reduces the number of dedicated hardware components for hashing and random number generation to a single SHAKE128/SHAKE256 module.

We summarize the chosen primitives in Table 3.

Domain separation: To preserve the domain separation between SHAKE and SHA-3, which is built-in in the primitive definitions in FIPS-202, we append a 16-bit integer to each input of CSPRNG and Hash as mentioned in Section 2.3. For values $\geq 2^{15}$ SHAKE is used as Hash, and for values $< 2^{15}$ SHAKE is used as CSPRNG. The lower 15 bits are used to separate different CSPRNG and Hash instances if necessary. In the cases where the lower 15 bits are specifically used for further separation are indicated by the additional integer appended in the input of the corresponding Hash and CSPRNG calls. The 16 bit values are encoded in little endian byte order.

Constant time: We compute the amount of randomness which should be extracted from the CSPRNG such that the rejection sampling processes we perform fail with a probability $2^{-\lambda}$. We provide in the submission package a Python script which computes such values automatically for all our parameter sets.

Sampling elements: For sampling objects like vectors or matrices with elements in a particular finite field, we perform rejection sampling using a fixed amount of randomness. In order to generate $\texttt{chall}_2$ with a fixed weight, we shuffle a fix array using the Fisher-Yates algorithm [25].

Hashing elements: Whenever we need to hash one or multiple objects, we absorb them in bit-packed representation if possible. More specifically, all vectors with elements in $\mathbb{F}_p$ or $\mathbb{F}_z$ are compressed as explained in Section 5.4 when used as input to Hash.

ExpandSK: Given as input the seed of the secret key $\texttt{Seed}_{\texttt{sk}}$, the function ExandSK outputs the secret $\overline{\mathbf{e}} \in \mathbb{F}_z^n$, as well as the public key $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$ and in case of R-SDP(G) also $\overline{\mathbf{e}}_G \in \mathbb{F}_z^m$ and $\overline{\mathbf{M}} \in \mathbb{F}_z^{m\times n}$. The function ExpandSK performs exactly the same computations as KeyGen, with the only difference that we do not need the syndrome $\mathbf{s}$ and do not need to compute $\mathbf{e} \in \mathbb{F}_p^n$.

In the case of R-SDP(G), we sample first $\overline{\mathbf{M}}$ and then $\mathbf{H}$. Furthermore, we sample $\mathbf{V}$ in transposed form, i.e., column-wise, for more efficient access during multiplication. The pseudo-code for ExpandSK is given in Algorithm 4.

Algorithm 4: ExpandSK($\texttt{Seed}_{\texttt{sk}}$)

```
Input: $\texttt{Seed}_{\texttt{sk}}$: the seed of the secret key;
Output: $\overline{\mathbf{e}} \in \mathbb{F}_z^n$ secret vector;
        $\mathbf{H} \in \mathbb{F}_p^{(n-k)\times n}$ $\overline{\mathbf{M}} \in \mathbb{F}_z^{m\times n}$ public matrices;
Data: $\lambda$: security parameter;
      $g \in \mathbb{F}_p^*$: generator of $\mathbb{E}$;
1 $(\texttt{Seed}_{\texttt{e}}, \texttt{Seed}_{\texttt{pk}}) \leftarrow \text{CSPRNG}_{\{0,1\}^{2\lambda}\times\{0,1\}^{2\lambda}}(\texttt{Seed}_{\texttt{sk}} \mid 3t+1)$
  // Sampling random matrices $\mathbf{H}$ and $\overline{\mathbf{M}}$
2 $(\overline{\mathbf{W}}, \mathbf{V}) \leftarrow \text{CSPRNG}_{\mathbb{F}_z^{m\times(n-m)}\times\mathbb{F}_p^{(n-k)\times k}}(\texttt{Seed}_{\texttt{pk}} \mid 3t+2)$ | $\mathbf{V} \leftarrow \text{CSPRNG}_{\mathbb{F}_z^{m\times(n-m)}\times\mathbb{F}_p^{(n-k)\times k}}(\texttt{Seed}_{\texttt{pk}} \mid 3t+2)$
3 $\mathbf{H} \leftarrow [\mathbf{V} \mid \mathrm{Id}_{n-k}]$
  // Computing e
  $\overline{\mathbf{M}} \leftarrow [\overline{\mathbf{W}} \mid \mathrm{Id}_m]$
4 $\overline{\mathbf{e}}_G \leftarrow \text{CSPRNG}_{\mathbb{F}_z^m}(\texttt{Seed}_{\texttt{e}} \mid 3t+3)$ | $\overline{\mathbf{e}} \leftarrow \text{CSPRNG}_{\mathbb{F}_z^n}(\texttt{Seed}_{\texttt{e}} \mid 3t+3)$
  $\overline{\mathbf{e}} \leftarrow \overline{\mathbf{e}}_G\overline{\mathbf{M}}$
  // Return secret vector and public matrices
5 return $(\overline{\mathbf{e}}, \overline{\mathbf{e}}_G, \mathbf{H}, \overline{\mathbf{M}})$ | $(\overline{\mathbf{e}}, \mathbf{H})$
```

5.2 Seed- and Merkle Tree

5.2.1 Tree Structures

We instantiate two tree structures for efficiency reasons. One instance is the seed tree (or GGM tree [27]) to derive t round seeds $\texttt{Seed}[i]$. In this instance, the root of the tree consists of a randomly sampled $\texttt{Seed}$, which is then expanded. Thus, the seed tree is computed from top to bottom.

The second instance is used in a Merkle tree fashion with the commitments $\texttt{cmt}_0[i]$ on its leaves, which are then hashed to compute the root of the tree. Thus, this second tree instance is computed from bottom to top. For the balanced and small versions, the trees serve the purpose of compressing the elements required in the signature by computing a `Path` and (Merkle-) `Proof`, since w is close to t. In the fast version, however, this technique yields no real benefit since $w \sim t/2$. Nevertheless, we also employ two trees with a different structure for computational efficiency.

Tree structures for balanced and small: In these versions the tree is constructed as a classical binary tree where each parent node has two children resulting in a total of $2t-1$ nodes. As the number of rounds t in CROSS are not always a power of two, the trees are truncated and constructed such that the whole tree consists of multiple full binary sub-trees. Figure 3 depicts such a tree for the case of $t = 11$. In this example, the tree consists of three sub-trees starting at nodes 1, 5 and 6 and are then combined from right to left, i.e., from the smallest sub-tree to the largest. The leaves are marked with double circles and the leftmost leaf (index 13) corresponds to the first round and the rightmost leaf (index 6) corresponds to the last round in the ID-loop of the protocol.

Because of the truncation, moving through the tree is not as straightforward as in a full binary case since not all leaves are on the same level. This truncated structure must also be taken into account when moving from a parent to its children. To ease that, we make use of some small pre-computed arrays and constants that are solely depending on t and define the trees structure. They are called

- `npl` for nodes per level;
- `lpl` for leaves per level;
- `off` for offsets, used to compute the parent/child index when moving between levels;
- `lsi` for leaves start indices;
- `ncl` for number consecutive leaves.

The `lsi` and `ncl` are small helper arrays that only have one entry per sub-tree. For instance, given the example in Figure 3, the arrays would be defined as `lsi` $= [13, 11, 6]$ and `ncl` $= [8, 2, 1]$.

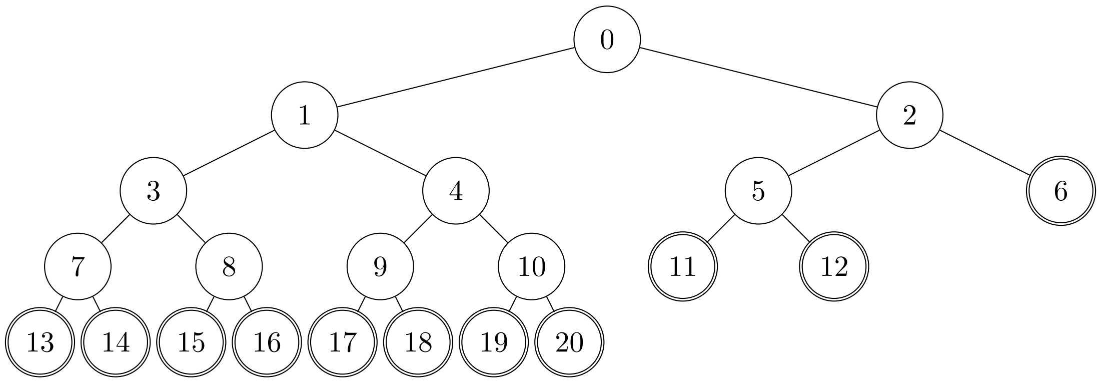

Figure 3: Exemplary tree structure for balanced and small versions for $t = 11$.

Tree structures for fast: For the fast version, we use a different tree construction, shown in Figure 4. This tree consists of $t + 5$ nodes and exactly three levels: A root node, then 4 intermediate nodes and t leaves. This implies, that each intermediate node has $\lfloor t/4 \rfloor$ children, plus 1 optional child depending on the value of t. More precisely, the remaining $t \mod 4$ children are equally distributed among the first three intermediate nodes, that is:

- node 1 has $\lfloor t/4 \rfloor$ plus 1 if $t \mod 4 > 0$ children;
- node 2 has $\lfloor t/4 \rfloor$ plus 1 if $t \mod 4 > 1$ children;

- node 3 has $\lfloor t/4 \rfloor$ plus 1 if $t \mod 4 > 2$ children.

Although in this version, we do not make use of the compression mechanism, in the sense of sending a standard Merkle proof or path in the seed tree, this structure allows to compute the four sub-trees in parallel on a CPU with a wide vector register set.

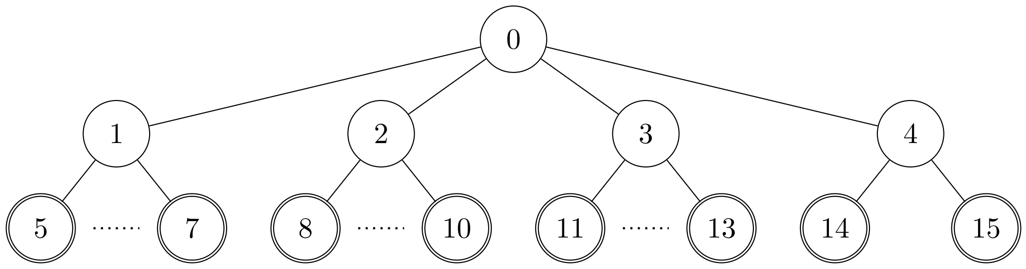

Figure 4: Exemplary tree structure for the fast version for $t = 11$.

5.2.2 Tree Algorithms

SeedLeaves: Algorithm 5 describes the implementation of SeedLeaves for the balanced and small versions. The function takes as input a root seed `Seed` and a `Salt` and computes t round seeds `Seed`$[i]$ from it. SeedLeaves internally computes a tree of nodes with the structure described in Section 5.2.1. To do so, the root is initialized with the root `Seed`. Then, proceeding from top to bottom and left to right, each node is expanded into two children by appending the `Salt` and the 16-bit index of the node to the seed of the node and feeding it into the CSPRNG that produces two seeds of λ bits. The 16-bit index is passed in little endian byte order and helper arrays are used as described in Section 5.2.1 for proper indexing within the truncated tree structure. Finally, the round seeds `Seed`$[i]$ are composed of the leaves of the tree.

Algorithm 5: SEEDLEAVES(Seed, Salt) – balanced and small versions

```
Input: Seed: the $\lambda$-bit root seed from which the whole tree is generated
       Salt: a $2\lambda$-bit salt
Output: (Seed[0], ..., Seed[$t-1$]): the $t$ round seeds
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a seed is $\lambda$ bits long)
      npl[...]: number of nodes per level
      lpl[...]: number of leaves per level
      off[...]: offsets required to move between two levels in the unbalanced tree
1  $\mathcal{T}$[0] ← Seed
2  startNode ← 0
3  for level from 0 to $\lceil \log_2(t) \rceil - 1$ do
4      for $i$ from 0 to npl[level] − lpl[level] − 1 do
5          parent ← startNode + $i$
6          leftChild ← LeftChild(parent) − off[level]
7          rightChild ← leftChild + 1
           // expand parent seed, salt and parent index
8          $\mathcal{T}$[leftChild], $\mathcal{T}$[rightChild] ← CSPRNG−$_{\{0,1\}^\lambda \times \{0,1\}^\lambda}$ ($\mathcal{T}$[parent] | Salt | parent)
9      startNode ← startNode + npl[level]
   // return the leaves of the tree as round seeds
10 return Leaves($\mathcal{T}$)
```

For the fast version of CROSS, SeedLeaves generates a seed tree of only three levels, as shown in Algorithm 6. It expands the root seed `Seed` into four intermediate seeds as shown in line 2, and then each of the intermediate seeds into $\lfloor t/4 \rfloor (+1)$ round seeds. Using four separate intermediate seeds allows to parallelize the expansion of the t round seeds on a CPU with sufficiently wide vector registers.

```
Algorithm 6: SeedLeaves(Seed, Salt) – fast version
    Input: Seed: the $\lambda$-bit root seed from which the whole tree is generated
           Salt: a $2\lambda$-bit salt
    Output: (Seed[0], ..., Seed[$t-1$]): the $t$ round seeds
    Data: $t$: number of leaves (corresponds to the number of protocol rounds)
          $\lambda$: security parameter (a seed is $\lambda$ bits long)
1   $\mathcal{T}[0] \leftarrow$ Seed
    // expand root seed, salt and parent index, with co-domain $\{0,1\}^{4\lambda}$
2   $\mathcal{T}[1:4] \leftarrow \mathsf{CSPRNG}\text{–}_{\{0,1\}^{4\lambda}}(\mathcal{T}[0] \mid \mathtt{Salt} \mid 0)$
    // expand each intermediate seed with appended salt and index into final round seeds
    // each $\mathcal{T}_i$ denotes a subset of $\mathcal{T}$ of size $a\lambda$ where $a = \lfloor t/4 \rfloor$ when $t \bmod 4 = 0$, or
       $a = \lfloor t/4 \rfloor + 1$, when $i < t \bmod 4$, as described in Section 5.2.1
3   for $i$ from 0 to 3 do
4       $\mathcal{T}_i \leftarrow \mathsf{CSPRNG}\text{–}_{\{0,1\}^{a\lambda}}(\mathcal{T}[i+1] \mid \mathtt{Salt} \mid i+1)$
5   return $\mathcal{T}[5 : t+4]$
```

SeedPath: Algorithm 7 describes SeedPath for the small and balanced versions of CROSS. This function takes the challenge $\mathtt{chall}_2$ to label a reference tree $\mathcal{T}'$ which indicates which nodes to pack into `Path`. More specifically, it places the challenge bits $\mathtt{chall}_2[i]$ on the leaves and updates the reference tree such that a parent node is labeled to be published if both of its children are to be published. Afterwards, SeedPath iterates through the tree from top to bottom and left to right and packs a node into the `Path` if the node itself is to be revealed, while its parent is not to be revealed.

In the fast version of CROSS, SeedPath simply returns the leaves Seed$[i]$, for i such that $\mathtt{chall}_2[i] = 1$, of the squashed tree, as described in Algorithm 8.

In the actual C reference implementation, it is not required to re-generate the full seed tree again, but it is given a pointer to the tree/leaves as constructed in SeedLeaves. The description given here is chosen for a unified notation for both, fast and small/balanced versions of CROSS.

Algorithm 7: SeedPath(Seed, Salt, $\texttt{chall}_2$) – balanced and small versions

```
Input: Seed: the $\lambda$-bit root seed from which the whole tree is generated
       Salt: a $2\lambda$-bit salt
       $\texttt{chall}_2$: the $t$-bit challenge denoting which leaves need to be revealed
Output: Path: the subset of nodes that allow re-computing the leaves corresponding to
        $\texttt{chall}_2[i] = 1$
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a seed is $\lambda$ bits long)
      npl[...]: number of nodes per level
      off[...]: offsets required to move between two levels in the unbalanced tree
   // Generate seed tree $\mathcal{T}$
1  $\mathcal{T} \leftarrow$ ComputeSeedTree(Seed, Salt)
   // Use flag tree $\mathcal{T}'$ to indicate which nodes to reveal
2  $\mathcal{T}' \leftarrow$ ComputeNodesToPublish($\texttt{chall}_2$)
3  startNode $\leftarrow 0$, pubNodes $\leftarrow 0$, Path $\leftarrow \emptyset$
4  for level from 1 to $\lceil \log_2(t) \rceil$ do
5      for $i$ from 0 to npl[level] $- 1$ do
6          node $\leftarrow$ startNode $+ i$
7          parent $\leftarrow$ Parent(node) + off[level $- 1$]/2
           // Reveal node if it is to publish but its parent is not
8          if $\mathcal{T}'$[node] $= 1$ and $\mathcal{T}'$[parent] $= 0$ then
9              Path[pubNodes] $\leftarrow \mathcal{T}$[node]
10             pubNodes $\leftarrow$ pubNodes + 1
11     startNode $\leftarrow$ startNode + npl[level]
12 return Path
```

Algorithm 8: SeedPath(Seed, Salt, $\texttt{chall}_2$) – fast version

```
Input: Seed: the $\lambda$-bit root seed from which the whole tree is generated
       Salt: a $2\lambda$-bit salt
       $\texttt{chall}_2$: the $t$-bit challenge denoting which leaves need to be revealed
Output: $(\texttt{Seed}[i])_{i:\texttt{chall}_2[i]=1}$: The round seeds Seed[$i$] for which $\texttt{chall}_2[i] = 1$
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a seed is $\lambda$ bits long)
1  Seed[0], ..., Seed[$t - 1$] $\leftarrow$ SeedLeaves(Seed | Salt)
2  return $(\texttt{Seed}[i])_{i:\texttt{chall}_2[i]=1}$
```

RebuildLeaves: This function re-generates the round-seeds Seed[i] given the Path, Salt and $\texttt{chall}_2$. As shown in Algorithm 9 for the balanced and small versions of CROSS, the reference tree is created just as in SeedPath. The procedure starts to rebuild the tree from top to bottom, left to right by expanding the nodes in the tree that were either given by the Path, or computed from expanding nodes from the Path. Finally, it returns the corresponding leaves Seed[i] where i is such that $\texttt{chall}_2[i] = 1$.

The fast version of RebuildLeaves is shown in Algorithm 10, which simply returns the corresponding nodes from Path.

Algorithm 9: RebuildLeaves($\texttt{Path}, \texttt{chall}_2, \texttt{Salt}$) – balanced and small versions

```
Input: Path: the subset of nodes that allow re-computing the leaves corresponding to
       $\texttt{chall}_2[i] = 1$
       $\texttt{chall}_2$: the $t$-bit challenge denoting which leaves need to be regenerated
       Salt: a $2\lambda$-bit salt
Output: $(\texttt{Seed}[i])_{i:\texttt{chall}_2[i]=1}$: the leaves corresponding to $\texttt{chall}_2[i] = 1$
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a seed is $\lambda$ bits long)
      $\texttt{npl}[\dots]$: number of nodes per level
      $\texttt{lpl}[\dots]$: number of leaves per level
      $\texttt{off}[\dots]$: offsets required to move between two levels in the unbalanced tree
// Use flag tree $\mathcal{T}'$ to indicate which nodes have been revealed
1  $\mathcal{T}' \leftarrow \mathsf{ComputeNodesToPublish}(\texttt{chall}_2)$
2  $\mathcal{T} \leftarrow \emptyset$
3  $\texttt{startNode} \leftarrow 0, \texttt{pubNodes} \leftarrow 0, \texttt{Path} \leftarrow \emptyset$
4  for level from $1$ to $\lceil \log_2(t) \rceil$ do
5      for $i$ from $0$ to $\texttt{npl}[\texttt{level}] - 1$ do
6          $\texttt{node} \leftarrow \texttt{startNode} + i$
7          $\texttt{parent} \leftarrow \mathsf{Parent}(\texttt{node}) + \texttt{off}[\texttt{level} - 1]/2$
8          $\texttt{leftChild} \leftarrow \mathsf{LeftChild}(\texttt{node}) - \texttt{off}[\texttt{level}]$
9          $\texttt{rightChild} \leftarrow \texttt{leftChild} + 1$
           // If node is in Path, copy it to tree
10         if $\mathcal{T}'[\texttt{node}] = 1$ and $\mathcal{T}'[\texttt{parent}] = 0$ then
11             $\mathcal{T}[\texttt{node}] \leftarrow \texttt{Path}[\texttt{pubNodes}]$
12             $\texttt{pubNodes} \leftarrow \texttt{pubNodes} + 1$
           // Expand it if node is in the tree and not a leaf, with co-domain $\{0,1\}^\lambda \times \{0,1\}^\lambda$
13         if $\mathcal{T}'[\texttt{node}] = 1$ and $i < \texttt{npl}[\texttt{level}] - \texttt{lpl}[\texttt{level}]$ then
14             $\mathcal{T}[\texttt{leftChild}], \mathcal{T}[\texttt{rightChild}] \leftarrow \mathsf{CSPRNG}\text{–}_{\{0,1\}^\lambda \times \{0,1\}^\lambda}(\mathcal{T}[\texttt{node}] \mid \texttt{Salt} \mid \texttt{node})$
15     $\texttt{startNode} \leftarrow \texttt{startNode} + \texttt{npl}[\texttt{level}]$
16 return $\mathsf{Leaves}(\mathcal{T})[i]_{i:\texttt{chall}_2[i]=1}$
```

Algorithm 10: RebuildLeaves($\texttt{Path}, \texttt{chall}_2, \texttt{Salt}$) – fast version

```
Input: Path: the subset of leaves corresponding to $\texttt{chall}_2[i] = 1$
       $\texttt{chall}_2$: the $t$-bit challenge denoting which leaves need to be regenerated
       Salt: a $2\lambda$-bit salt, unused in the fast version
Output: $(\texttt{Seed}[i])_{i:\texttt{chall}_2[i]=1}$: the leaves corresponding to $\texttt{chall}_2[i] = 1$
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a seed is $\lambda$ bits long)
1  return $\mathsf{Path}[i]_{i:\texttt{chall}_2[i]=1}$
```

TreeRoot: TreeRoot as given in Algorithm 11 (for balanced and small versions) and Algorithm 12 (for the fast version) computes a root $\mathcal{T}[0]$ of a tree as described in Section 5.2.1, through iterative hashing. By doing so, the commitments $\texttt{cmt}_0[i]$ are placed on the tree leaves and hashed either pairwise (balanced and small versions) or in larger groups (fast version) from bottom to top.

Algorithm 11: $\text{TreeRoot}(\texttt{cmt}_0[0], \ldots, \texttt{cmt}_0[t-1])$ – balanced and small versions

```
Input: $\texttt{cmt}_0[i]$: the $2\lambda$-bit commitments of each of $t$ rounds
Output: $\texttt{digest}_{\texttt{cmt}_0}$: the Merkle root of the commitment tree
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a digest is $2\lambda$ bits long)
      $\texttt{npl}[\ldots]$: number of nodes per level
      $\texttt{lsi}[\ldots]$: leaves start indices for a set of consecutive leaves
      $\texttt{off}[\ldots]$: offsets required to move between two levels in the unbalanced tree
1  $\mathcal{T} \leftarrow \text{PlaceOnLeaves}(\texttt{cmt}_0[0], \cdots, \texttt{cmt}_0[t-1])$
2  $\texttt{startNode} \leftarrow \texttt{lsi}[0]$
3  for $\texttt{level}$ from $\lceil \log_2(t) \rceil$ to $1$ do
4      for $i$ from $\texttt{npl}[\texttt{level}] - 2$ to $0$ step $-2$ do
5          $\texttt{leftChild} \leftarrow \texttt{startNode} + i$
6          $\texttt{rightChild} \leftarrow \texttt{leftChild} + 1$
7          $\texttt{parent} \leftarrow \text{Parent}(\texttt{leftChild}) + \texttt{off}[\texttt{level} - 1]/2$
8          $\mathcal{T}[\texttt{parent}] \leftarrow \text{Hash}(\mathcal{T}[\texttt{leftChild}] \mid \mathcal{T}[\texttt{rightChild}])$
9      $\texttt{startNode} \leftarrow \texttt{startNode} - \texttt{npl}[\texttt{level} - 1]$
10 return $\mathcal{T}[0]$
```

Algorithm 12: $\text{TreeRoot}(\texttt{cmt}_0[0], \ldots, \texttt{cmt}_0[t-1])$ – fast version

```
Input: $\texttt{cmt}_0[i]$: the $2\lambda$-bit commitments of each of $t$ rounds
Output: $\texttt{digest}_{\texttt{cmt}_0}$: the Merkle root of the commitment tree
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a digest is $2\lambda$ bits long)
1  $\mathcal{T}[5 : t+4] \leftarrow (\texttt{cmt}_0[0], \cdots, \texttt{cmt}_0[t-1])$
   // each $\mathcal{T}_i$ denotes a subset of $\mathcal{T}$ of size $a2\lambda$ where $a = \lfloor t/4 \rfloor$ when $t \bmod 4 = 0$, or
      $a = \lfloor t/4 \rfloor + 1$, when $i < t \bmod 4$, as described in Section 5.2.1
2  for $i$ from $0$ to $3$ do
3      $\mathcal{T}[i+1] \leftarrow \text{Hash}(\mathcal{T}_i)$
4  $\mathcal{T}[0] \leftarrow \text{Hash}(\mathcal{T}[1:4])$
5  return $\mathcal{T}[0]$
```

TreeProof: Algorithm 13 shows the implementation of TreeProof for the small and balanced versions. By utilizing a reference tree $\mathcal{T}'$ using ComputeNodesToPublish, the function packs those nodes into the `Proof` that are required by the verifier to reconstruct the tree root, given their own subset of computed leaf nodes. That is, only if one of the two children in the tree is being labelled to be re-computed by the verifier, the corresponding sibling is packed into `Proof`. In doing so, the function moves through the tree from bottom to top, right to left. It is noteworthy that the indices of nodes being packed into `Proof` correspond exactly to the indices of the nodes in the `Path` as computed by TreePath.

Like for SeedPath, the re-computation of the Merkle tree in line 1 of Algorithm 13 is just for clarity, but not done in the actual implementation.

Analogously to the seed tree functions, the fast version of TreeProof as shown in Algorithm 14 simply selects a subset of the leaves $\texttt{cmt}_0[i]$, for i such that $\texttt{chall}_2[i] = 1$.

Algorithm 13: TreeProof($\texttt{cmt}_0, \texttt{chall}_2$) – balanced and small versions

```
Input: $\texttt{cmt}_0[i]$: the $2\lambda$-bit leaves for which a Merkle tree is computed
       $\texttt{chall}_2$: the $t$-bit challenge denoting which leaves need to be revealed
Output: $\texttt{Proof}$: the subset of nodes that allow re-computing the the Merkle root given the
        leaves where $\texttt{chall}_2[i] = 0$
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a digest is $2\lambda$ bits long)
      $\texttt{npl}[\dots]$: number of nodes per level
      $\texttt{lsi}[\dots]$: leaves start indices for a set of consecutive leaves
      $\texttt{off}[\dots]$: offsets required to move between two levels in the unbalanced tree
// Generate Merkle tree $\mathcal{T}$
1  $\mathcal{T} \leftarrow \mathsf{ComputeMerkleTree}(\texttt{cmt}_0)$
// Use flag tree $\mathcal{T}'$ to indicate which nodes to reveal
2  $\mathcal{T}' \leftarrow \mathsf{ComputeNodesToPublish}(\texttt{chall}_2)$
3  $\texttt{startNode} \leftarrow \texttt{lsi}[0], \texttt{pubNodes} \leftarrow 0, \texttt{Proof} \leftarrow \emptyset$
4  for level from $\lceil \log_2(t) \rceil$ to 1 do
5      for $i$ from $\texttt{npl}[\texttt{level}] - 2$ to 0 step $-2$ do
6          $\texttt{node} \leftarrow \texttt{startNode} + i$
7          $\texttt{parent} \leftarrow \mathsf{Parent}(\texttt{node}) + \texttt{off}[\texttt{level} - 1]/2$
           // add left sibling only if right one was computed but left was not
8          if $\mathcal{T}'[\texttt{node}] = 0$ and $\mathcal{T}'[\texttt{node} + 1] = 1$ then
9              $\texttt{Proof}[\texttt{pubNodes}] \leftarrow \mathcal{T}[\texttt{node}]$
10             $\texttt{pubNodes} \leftarrow \texttt{pubNodes} + 1$
           // add left sibling only if right one was computed but left was not
11         if $\mathcal{T}'[\texttt{node}] = 1$ and $\mathcal{T}'[\texttt{node} + 1] = 0$ then
12             $\texttt{Proof}[\texttt{pubNodes}] \leftarrow \mathcal{T}[\texttt{node} + 1]$
13             $\texttt{pubNodes} \leftarrow \texttt{pubNodes} + 1$
14     $\texttt{startNode} \leftarrow \texttt{startNode} - \texttt{npl}[\texttt{level} - 1]$
15 return Proof
```

Algorithm 14: TreeProof($\texttt{cmt}_0, \texttt{chall}_2$) – fast version

```
Input: $\texttt{cmt}_0[i]$: the $2\lambda$-bit leaves for which a squashed tree is computed
       $\texttt{chall}_2$: the $t$-bit challenge denoting which leaves need to be revealed
Output: $\texttt{Proof}$: the subset of nodes that allow re-computing the the root given the leaves
        where $\texttt{chall}_2[i] = 0$
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a digest is $2\lambda$ bits long)
1  return $(\texttt{cmt}_0[i])_{i:\texttt{chall}_2[i]=1}$
```

RecomputeRoot: Given the Proof from the signature and $\texttt{chall}_2$, the version of RecomputeRoot in the balanced and small versions of CROSS recomputes the root of the Merkle tree as shown in Algorithm 15. To do so, the commitments $\texttt{cmt}_0[i]$, which are computed by the verifier, are first placed on the tree. Afterwards, using the reference tree $\mathcal{T}'$, several nodes are hashed from bottom to top, right to left by using nodes from the tree or the Proof.

The corresponding fast version of RecomputeRoot is shown in Algorithm 16. The verifier moves the nodes from Proof to the corresponding locations within the array of commitments $\texttt{cmt}_0[i]$ and then returns the root of the tree as generated in Algorithm 12.

Algorithm 15: RecomputeRoot($\mathtt{cmt}_0$, Proof, $\mathtt{chall}_2$) – balanced and small versions

```
Input: $\mathtt{cmt}_0[i]$: the $2\lambda$-bit leaves for which a Merkle tree is computed
       Proof: the subset of nodes that allow re-computing the Merkle root given the leaves
       where $\mathtt{chall}_2[i] = 0$
       $\mathtt{chall}_2$: the $t$-bit challenge denoting which leaves have been revealed
Output: $\mathtt{digest}_{\mathtt{cmt}_0}$: the root of the recomputed Merkle tree
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a digest is $2\lambda$ bits long)
      $\mathtt{npl}[\ldots]$: number of nodes per level
      $\mathtt{lsi}[\ldots]$: leaves start indices for a set of consecutive leaves
      $\mathtt{off}[\ldots]$: offsets required to move between two levels in the unbalanced tree
// Initialize Merkle tree $\mathcal{T}$
1  $\mathcal{T} \leftarrow$ PlaceCmtOnLeaves($\mathtt{cmt}_0$, $\mathtt{chall}_2$)
   // Use flag tree $\mathcal{T}'$ to indicate which nodes were revealed
2  $\mathcal{T}' \leftarrow$ ComputeNodesToPublish($\mathtt{chall}_2$)
3  startNode $\leftarrow \mathtt{lsi}[0]$, pubNodes $\leftarrow 0$
4  for level from $\lceil \log_2(t) \rceil$ to 1 do
5      for $i$ from npl[level] $- 2$ to 0 step $-2$ do
6          node $\leftarrow$ startNode $+ i$
7          parent $\leftarrow$ Parent(node) + off[level $- 1$]/2
           // Skip if both siblings are unused
8          if $\mathcal{T}$[node] $= 0$ and $\mathcal{T}$[node $+ 1$] $= 0$ then
9              continue
           // add left sibling from tree or Proof
10         if $\mathcal{T}'$[node] $= 1$ then
11             leftChild $\leftarrow \mathcal{T}$[node]
12         else
13             leftChild $\leftarrow$ Proof[pubNodes]
14             pubNodes $\leftarrow$ pubNodes $+ 1$
           // add right sibling from tree or Proof
15         if $\mathcal{T}'$[node $+ 1$] $= 1$ then
16             rightChild $\leftarrow \mathcal{T}$[node $+ 1$]
17         else
18             rightChild $\leftarrow$ Proof[pubNodes]
19             pubNodes $\leftarrow$ pubNodes $+ 1$
20         $\mathcal{T}$[parent] $\leftarrow$ Hash(leftChild | rightChild)
21     startNode $\leftarrow$ startNode $-$ npl[level $- 1$]
22 return $\mathcal{T}[0]$
```

Algorithm 16: RecomputeRoot($\texttt{cmt}_0$, $\texttt{Proof}$, $\texttt{chall}_2$) – fast version

```
Input: $\texttt{cmt}_0[i]$: the $2\lambda$-bit leaves for which a squashed tree is computed
       $\texttt{Proof}$: the subset of nodes that allow re-computing the root given the leaves where
       $\texttt{chall}_2[i] = 0$
       $\texttt{chall}_2$: the $t$-bit challenge denoting which leaves have been revealed
Output: $\texttt{digest}_{\texttt{cmt}_0}$: the root of the re-computed tree
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a digest is $2\lambda$ bits long)
// Initialize the leaves.  For $\texttt{chall}_2[i] = 0$, the $\texttt{cmt}_0[i]$ are already computed
1 pubNodes ← 0
2 for $i$ from 0 to $t-1$ do
3     if $\texttt{chall}_2[i] = 1$ then
4         $\texttt{cmt}_0[i] \leftarrow \texttt{Proof}[\texttt{pubNodes}]$
5         pubNodes ← pubNodes + 1
6 return TreeRoot($\texttt{cmt}_0$)
```

5.3 Parallelization of SHAKE

Given SHAKE's central role as both a hash function and a CSPRNG in CROSS, we introduced support for its AVX2-optimized implementation in round 2 [26]. SHAKE is built on the Keccak-f primitive, which can be parallelized by leveraging the SIMD (Single Instruction, Multiple Data) capabilities of modern CPUs.

The official Keccak repository [13] provides a four-way parallel implementation, enabling four independent instances of SHAKE to be computed simultaneously. We utilized this implementation to speed up three key areas in CROSS where many calls to SHAKE are performed in sequence: hashing commitments at the end of each round in the identification protocol, the seed tree, and the Merkle tree.

Hashing commitments: The first case requires a simple queue: in the Fiat-Shamir transformation, the rounds are inherently parallel and independent of one another, so we can enqueue the calls and hash the commitments every four rounds, rather than during each individual round. When the number of rounds, t, is not a multiple of four, the queue will have fewer than four calls at the end of the protocol. In such cases, the serial version of Keccak is used to compute the remaining digests.

The other two cases (the seed tree and the Merkle tree) follow the same queuing principle, with only slight adjustments to account for the tree structure.

Parallelized trees: Algorithm 5 demonstrates how the seed tree is constructed in the reference implementation of CROSS. We start by placing the root seed in the first position of the tree, which is linearized as a list of seeds. The tree is then traversed level by level, node by node, skipping the leaves. Every node is input into SHAKE and the output is divided into its left and right children.

Algorithm 17 illustrates the same procedure in the AVX2-optimized implementation of CROSS, with the differences highlighted in magenta. A queue is constructed by storing the positions of up to four parent nodes ($\texttt{ins}[\dots]$) and their corresponding left children ($\texttt{outs}[\dots]$). The variable `toExpand` keeps track of how many calls to SHAKE are currently in the queue.

When the queue is full, we empty it by calling the parallel version of SHAKE (called ParCSPRNG here). The queue is also emptied when transitioning between tree levels to prevent expanding a parent seed that has not yet been generated. In such cases, `toExpand` indicates the number of calls to the serial version of the CSPRNG that remain to be executed.

Note that both the seed and Merkle tree structures in CROSS are unbalanced, as the number of leaves t is not a power of two, meaning not all leaves are on the last level. Since the two tree structures are equal, parallelizing the hashing operations in the Merkle tree follows the same approach. The key difference is that calls to SHAKE are enqueued by traversing the tree in the opposite direction (from the leaves to the root) hashing sibling nodes together to compute their parent as a digest.

Fast variants: Another operation that can benefit from the SIMD implementation of Keccak is the seed tree generation for the fast variants of CROSS, described by Algorithm 6. In the AVX2-optimized implementation, the for-loop performing four separate calls to the CSPRNG (at line 4) is replaced with a single call to ParCSPRNG. Being able to parallelize these four calls is the reason for using the squashed tree structure in CROSS-fast instead of using a fully linearized approach.

Algorithm 17: ParallelSeedLeaves(Seed, Salt) – balanced and small, AVX2

```
Input: Seed: the $\lambda$-bit root seed from which the whole tree is generated
       Salt: a $2\lambda$-bit salt
Output: (Seed[0], ..., Seed[$t-1$]): the $t$ round seeds
Data: $t$: number of leaves (corresponds to the number of protocol rounds)
      $\lambda$: security parameter (a seed is $\lambda$ bits long)
      npl[...]: number of nodes per level
      lpl[...]: number of leaves per level
      off[...]: offsets required to move between two levels in the unbalanced tree
1  $\mathcal{T}$[0] ← Seed
2  startNode ← 0
   // Enqueue the calls to the CSPRNG
3  toExpand ← 0
4  ins ← [0, 0, 0, 0]
5  outs ← [0, 0, 0, 0]
6  for level from 0 to $\lceil \log_2(t) \rceil - 1$ do
7      for $i$ from 0 to npl[level] − lpl[level] − 1 do
8          toExpand ← toExpand + 1
9          parent ← startNode + $i$
10         leftChild ← LeftChild(parent) − off[level]
11         rightChild ← leftChild + 1
12         ins[toExpand − 1] ← parent
13         outs[toExpand − 1] ← leftChild
           // add Salt and domain separator to the CSPRNG inputs
14         ...
           // Call CSPRNG in batches of 4 (or less when changing tree level)
15         if toExpand = 4 or $i$ = (npl[level] − lpl[level] − 1) then
16             $\mathcal{T}$[outs[0]], ..., $\mathcal{T}$[outs[3]] ← ParCSPRNG(toExpand, $\mathcal{T}$[ins[0]], ⋯, $\mathcal{T}$[ins[3]])
17             toExpand ← 0
18     startNode ← startNode + npl[level]
19 return Leaves($\mathcal{T}$)
```

5.4 Packing and Unpacking:

The syndrome in the public key $\mathbf{s}$ and the response vectors $\texttt{resp}_0$, which are part of the signature, consist of elements in $\mathbb{F}_p$ or $\mathbb{F}_z$. For the chosen values of p and z, the maximum number of bits needed to store

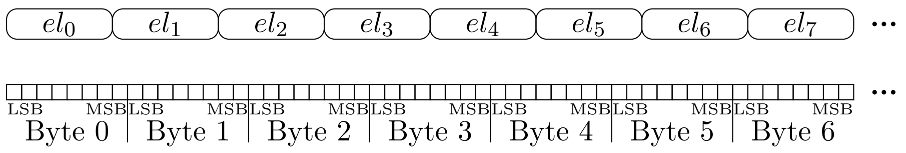

Figure 5: Packing of elements with $p = 127$ or $z = 127$, $\mathbf{s} = \{el_0, ..., el_{n-k-1}\}$, $\mathbf{y} = \{el_0, ..., el_{n-1}\}$ and $\overline{\mathbf{v}}_G = \{el_0, ..., el_{m-1}\}$

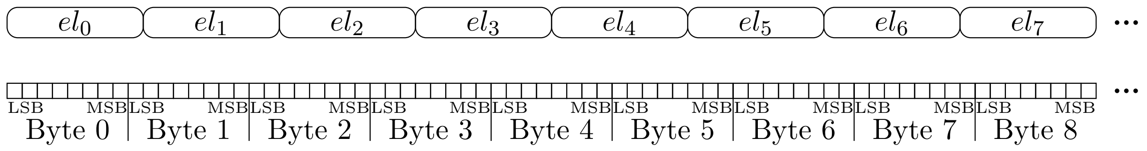

Figure 6: Packing of elements with $p = 509$, $\mathbf{s} = \{el_0, ..., el_{n-k-1}\}$, $\mathbf{y} = \{el_0, ..., el_{n-1}\}$

values in $\mathbb{F}_p$ (respectively $\mathbb{F}_z$) does not require a number of bits that is a multiple of eight in general.

It is reasonable to store these values bit-packed to reduce signature and public key size. For the R-SDP variant of CROSS, we therefore need

- $\lceil (n-k) \cdot 7/8 \rceil$ bytes for the syndrome $\mathbf{s}$;
- $\lceil n \cdot 7/8 \rceil$ bytes per $\mathbf{y}$ in $\mathtt{resp}_0$;
- $\lceil n \cdot 3/8 \rceil$ bytes per $\overline{\mathbf{v}}$ in $\mathtt{resp}_0$.

For the R-SDP(G) variant of CROSS, we need

- $\lceil (n-k) \cdot 9/8 \rceil$ bytes for the syndrome $\mathbf{s}$;
- $\lceil n \cdot 9/8 \rceil$ bytes per $\mathbf{y}$ in $\mathtt{resp}_0$;
- $\lceil m \cdot 7/8 \rceil$ bytes per $\overline{\mathbf{v}}_G$ in $\mathtt{resp}_0$.

The elements are packed little endian, i.e. the least significant bit of the first element aligns with the least significant bit of the first packed byte with all subsequent elements starting at the least significant bit position unoccupied in the packed array.

The bit-packed pattern for $\mathbb{F}_p$ elements in the R-SDP variant of CROSS and $\mathbb{F}_z$ elements in the R-SDP(G) variant of CROSS is shown in Figure 5, while the bit-packed pattern for $\mathbb{F}_p$ elements in the R-SDP(G) variant of CROSS is depicted in Figure 6. Finally, Figure 7 shows the bit-packed pattern for $\mathbb{F}_z$ elements in the R-SDP variant of CROSS.

We pad each packed vector with 0 to the next byte boundary and also check for this padding when unpacking any vector.

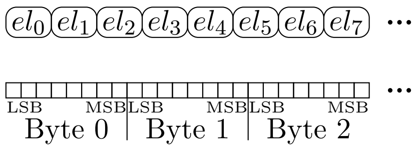

Figure 7: Packing of elements with $z = 7$, $\overline{\mathbf{v}} = \{el_0, ..., el_{n-1}\}$

5.5 Efficient arithmetic for $\mathbb{F}_7$, $\mathbb{F}_{127}$, and $\mathbb{F}_{509}$

Implementing CROSS requires, besides the auxiliary CSPRNG and Hash function, a set of arithmetic primitives which act on collections of either $\mathbb{F}_p$ or $\mathbb{F}_z$ elements. The simple nature of the arithmetic operations allows for a straightforward constant time implementation. In particular, vector additions, vector subtractions, and point-wise vector multiplications are realized by countable loops, with a compile-time determined trip-count. Similarly, matrix-vector multiplications by either $\mathbf{H}$ or $\overline{\mathbf{M}}$ are characterized by countable nested loops sharing the data-independent execution time of the vector operations.

The only arithmetic operation which may be affected by a variable time implementation is the computation which, given a vector $\overline{\mathbf{e}} = (\overline{\mathbf{e}}_0, \dots, \overline{\mathbf{e}}_{n-1})$ in $\mathbb{F}_z$, computes the vector $\mathbf{e} = (\mathbf{e}_0, \dots, \mathbf{e}_{n-1})$ in $\mathbb{F}_p$ such that for all $0 \le i < n$ we have $\mathbf{e}_i = g^{\overline{\mathbf{e}}_i}$, where g is the generator of the restricted subgroup $\mathbb{E}$.

A straightforward implementation would employ a square-and-multiply strategy, which is affected by timing side-channel vulnerabilities. To avoid this issue, we resorted to two different techniques, depending on whether $z = 7$ or $z = 127$, which are the only two values which we need to treat.

In the $z = 7$ case, we have that $p = 127$, and therefore its elements can be stored in a single byte, encoded as in natural binary encoding. As a consequence, it is possible to fit the entire look-up table for the seven values $\{g^0, g^1, \dots, g^6\}$ in a single, 64-bit register. A look-up in this single-register-sized table takes constant time as the entire table is loaded, regardless of the value being looked up.

In the $z = 127$ case, we have that $p = 509$. As a consequence, for software implementations, two bytes are required to represent an $\mathbb{F}_p$ element, and the table-based approach cannot be applied in the same straightforward fashion, as for $p = 127$. To this end, we implement the g^i operation through a square-and-multiply approach, where all the values $\{g^{2^0}, g^{2^1}, \dots, g^{2^6}\}$ mod p are precomputed constants, which are composed through a single arithmetic expression, where each power of two is selected via an arithmetic predicated expression. The modular reductions are performed tree-wise to reduce their number to a minimum.

A final note on the arithmetic employed to implement computations on both $\mathbb{F}_7$ and $\mathbb{F}_{127}$ concerns the runtime data representation. We work, in both cases, performing reductions modulo 8 and 128 respectively, thus resulting in a double representation of the zero value (as 0 and 7 for $\mathbb{F}_7$, and as 0 and 127 for $\mathbb{F}_{127}$). This, in turn, effectively reduces the cost of the modular reductions to, at most, two shift and add operations. The values with the double-zero representation are then normalized via a constant time arithmetic expression before emission.

Reductions modulo 509: The `%` operator can be used in C to perform modular reductions, however, some compilers translate it into an unsafe division instruction (especially when compiling with options like `-Os`, i.e., optimize for size).

To avoid this, we implement reductions modulo $p = 509$ as a constant-time sequence of instructions: first, we approximate the quotient with a multiplication and a bit shift by precomputed constants, then perform a multiplication by p and a subtraction to find the remainder.

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

This Barrett-like reduction is described in detail in [41, Chapter 10-15]. For CROSS we use $\beta = 40$ and $\mu = 2160140723$, so that the operation works on all 32-bit positive integers.

Table 8: Computation time expressed in clock cycles for all CROSS primitives and variants. Measurements collected via `rtdscp` on an Intel Core i7-12700K, clocked at 4.9GHz. The figures are the results of the average of 10k tests (standard deviation below 1%), and were obtained pinning the process to a `P`-core. The computer was running Debian GNU/Linux 12.

NIST Cat.	Parameter Set	KeyGen (Mcycles)	Sign (Mcycles)	Verify (Mcycles)
1	CROSS-R-SDP-f	0.038	1.007	0.572
	CROSS-R-SDP-b	0.040	2.013	1.270
	CROSS-R-SDP-s	0.039	4.048	2.725
	CROSS-R-SDP-(G)-f	0.020	0.687	0.422
	CROSS-R-SDP-(G)-b	0.020	1.579	0.985
	CROSS-R-SDP-(G)-s	0.020	3.137	1.971
3	CROSS-R-SDP-f	0.090	2.324	1.398
	CROSS-R-SDP-b	0.089	4.171	2.776
	CROSS-R-SDP-s	0.089	6.254	4.277
	CROSS-R-SDP-(G)-f	0.041	1.555	0.982
	CROSS-R-SDP-(G)-b	0.040	2.240	1.446
	CROSS-R-SDP-(G)-s	0.040	4.195	2.832
5	CROSS-R-SDP-f	0.136	4.116	2.512
	CROSS-R-SDP-b	0.136	7.042	4.752
	CROSS-R-SDP-s	0.135	11.356	7.765
	CROSS-R-SDP-(G)-f	0.068	2.580	1.634
	CROSS-R-SDP-(G)-b	0.065	3.482	2.248
	CROSS-R-SDP-(G)-s	0.067	6.197	4.059

5.6 Implementation Attacks

Currently, there are two works [37] and [32] investigating implementation attacks which both attack the reference implementation of CROSS version 1.2 and target embedded platforms.

The first work proposes a passive power side-channel attack, that targets the input of the syndrome computation [37]. One can recover single elements from $\mathbf{u}$ via a horizontal attack mounted on one round of the protocol. It is then possible to recompute elements of the secret key vector $\mathbf{e}$ by using information published with the signature.

This attack successfully recovers the entire secret key $\mathbf{e}$ from a single signing procedure for most parameter sets and requires two signing operations for the R-SDP(G) 1 fast parameter set. The attack can be impeded by either shuffling the execution order of the multiplications in the syndrome computation or by masking the input data to the syndrome computation.

The second work proposes a fault attack on the reference tree $\mathcal{T}'$ used to determine which leaves are to be published [32]. This attack recovers the entire secret key using a single fault by obtaining the responses for both cases of the second challenge $\texttt{chall}_2$ in a single round.

6 Detailed Performance Analysis

We benchmarked the performance of CROSS on aIntel Core i7-12700K, clocked at 4.9GHz, with 64GiB of DDR5. The computer was running Debian GNU/Linux 12, and the benchmark binaries were compiled with gcc 12.2.0 (Debian 12.2.0-14). The computation times are measured in clock cycles, the clock cycle count has been gathered employing the `rtdscp` instruction, which performs instruction fencing. All numbers of clock cycles reported were obtained as the average of 10k runs of the same primitive. All the timings for CROSS were taken with respect to the current AVX2 optimized implementation.

We report in Table 8 the required number of clock cycles to compute the Keygen, Sign and Verify signature algorithms. For CROSS the "f" letter in the parameter set denotes a "fast" optimization corner, "b" the balanced one and "s" denotes a short (signature) optimization corner.

To provide a concrete grounding for practical use, we observe that the fast optimization corner of CROSS achieves sub-millisecond signing and verification times for all categories and both R-SDP and R-SDP(G) variants. Furthermore CROSS-R-SDP, for NIST security category 1 is well below a single millisecond for signature creation (205 μs) and verification (116 μs) on the platform we employed for the benchmarks. CROSS-R-SDP(G) performs even better, signing in 140 μs, and verifying in 86 μs.

Concerning the computational load of CROSS-R-SDP, about $\approx 60\%$ of the time taken by the signature primitive is spent computing either hashes or CSPRNGs. This computational load profile is essentially the same during verification, as a result, in both cases, of the optimization of the arithmetic operations with AVX2 vector instructions.

7 Known Answer Tests

Known Answer Tests (KAT) have been generated and are a separate archive. The submission package contains facilities (in the Additional Implementation folder) to regenerate them, following the instructions in the `README` file. We include the SHA-2-512 digests of the KAT requests and responses in the following.

```
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_77_13152.req
f4cfa7f574000affa3be37924cc6effa42f500d25dc86948797789bd240aeb65731b3fd91eee3b5cccecc0618bf8c2fc979fde7828718ae09b2ec8fc3c26f03a  ./PQCsignKAT_106_48102.rsp
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_106_48102.req
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_153_50818.req
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_106_40100.req
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_83_22464.req
c5b8957851480247de83cd7048e6413b01f5c0690c7755609ddce2794db4d90d5e0168d10b461c159c85d7cae14963d16561c7ecbfdcf982289717f7eea42ca6  ./PQCsignKAT_77_12432.rsp
83f9e38b119901a2edb4e7705b670ecde5dca9323f3dc3809ad8676fac5aa134d2f69def08ec1e39c3cab70b41c0d5c01b9e1fd1a2efeb1db72cd2146d801c19  ./PQCsignKAT_153_53527.rsp
5f4c14bc96f59a4a8f8db755db82577522f854d43b0d5739bc621593a44a98d5db53f1464a51ee335deb0af893c3c0ebde4264b4b3b64fd6714ba6f317c5d8a4  ./PQCsignKAT_106_40100.rsp
db9b3a0d271039f014d3e6c2d836e20d9e36567275abbbe3e70692966a0eef72e60d2d3717978ef62334176d29a93cdb75a2e32016c50b50c95e6c4eb0b7fa56  ./PQCsignKAT_83_22464.rsp
aefb71960f7d0c698acd4270291101fbc0c4f0e845a80bd3e5f800ef28cf2275c01ed1373a1e847dbc20ba1fd296ecfad814101a147655e1d46092b13e3f55cd  ./PQCsignKAT_54_11980.rsp
1d71834e43fff7bdae9aa5c7c267cd6603125a28a7dfb1bd7aa130beb4462efeebf139a9b992b4fd795dc9a3610d9715c1ddf91f735def8821da42bb13ed4049  ./PQCsignKAT_115_41406.rsp
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_106_36454.req
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_153_74590.req
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_115_41406.req
3e2d13a4da52aa164c33512a1c32ef28e13d0d1f2cffe834b3abbf3e96e20c7b32fedffde28a27e9bc4c63b3f9c8ad7ebb0c9599fda4131d7094113901ef5d61  ./PQCsignKAT_77_18432.rsp
01d95620f0ce69f43c7eebd43b862c709bfd640fd3de3752f09cf13ec7d0a90e484aab0c78e8998ce223e31b72e1df69e83da3fb449b3ce545b85a8df0f0395e  ./PQCsignKAT_153_50818.rsp
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_54_9120.req
76d95454ca79b7b043912aaf2cf4542dbec8f170778bf807d144b9becd809614d0a68c017cd43162abf8a5d5cd944fff017560bd2fdbfa38bde9044ee3816a64  ./PQCsignKAT_83_20452.rsp
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_115_29853.req
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_77_12432.req
071f8947de8bb58e7f28c715312631640390 4b15fb4eb7dacadceece5711e3320cfd2d7436c2e2bb151ce699fa5135caa14ae408ed74155419923dcf8e0f35e4  ./PQCsignKAT_54_9120.rsp
e7f9ddc7f1b61df38cb9d43e9411107cc4f7d69d3ee3690a3a0916df072a5da4de9fd88fd638b409f91bf212da32bc3f2e2ffc014ab4bacc6f07f3fe63326478  ./PQCsignKAT_54_8960.rsp
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_54_11980.req
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_54_8960.req
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_83_26772.req
67efaa028f9cf997301bc4d3003979e445e2c26e5a34ff55c53f093b3efc13633e3fc9e99cb135a173c5fac897c0aa9245d262abe98ab70930877de44b218945  ./PQCsignKAT_153_74590.rsp
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_83_20452.req
6368e6508aa7fcd70861f667cdb15086d00f15e9575f582a02ea147f5f82b5835ec4653b510df2196f168c74b011699928ee13c65987995d0b4cb4f36290f5b6  ./PQCsignKAT_83_26772.rsp
983717e527ae81eaa2ed40d9134daaddaad561964ee1e3d41158b590f0922d09a4c5331c75fb65b6e646dfdb7d60afaafcdb0ed09054c46b3fc80337fda04dcd  ./PQCsignKAT_77_13152.rsp
73dead39ebb6a80505cb91ed3982011e5e5c323fc928a4c1188ee8d5bdd90e82bf524aa7fe12da548c9a000af70750172856997090b2fd3b070fa4c40f09f3fb  ./PQCsignKAT_115_29853.rsp
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_115_28391.req
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_77_18432.req
a87eccf3d19fd50883d3a2c21435ac031e998c7d20f9ba81da57a70b9709f99b77fef37cae8856740002e15c46d2873348a9b37ad07a59659076b5e8a46a8458  ./PQCsignKAT_153_53527.req
d3bfb762e670ec93468cfe55a37485cb5734243083209a664690bb7ffdc5a13d08f9e4d2147f386659e0602ece6e51515a4a20f1e77d952e59973b8245ad9cce  ./PQCsignKAT_115_28391.rsp
03f78bb762b144465adddbf0454e1d581c0bf5f1a0d97845018ed2a7b162dc685eb2ce41555feb27d4e92913a182bc7182243d7d2df0d48d4b6840d5a1562489  ./PQCsignKAT_106_36454.rsp
```

8 Advantages and Limitations

Advantages:

- Due to the use of restricted errors, generic decoders have an increased cost compared to generic decoders in the Hamming metric. As our thorough security analysis [39] shows, this allows us to choose smaller parameters to achieve the same security level. We have adapted the best-known techniques from classical ISD algorithms, subset-sum solvers and considered algebraic attacks.
- By leveraging a ZK protocol, we do not require any code with algebraic structure and thus do not rely on any indistinguishability assumption. The used code is chosen uniformly at random and is made public. Since the secret is given by the randomly chosen restricted error vector, an adversary faces an NP-hard problem: either R-SDP or R-SDP(G).
- The ZK protocol CROSS-ID follows the well-established structure of CVE [19], which is a well-known and studied protocol. The resulting signature scheme is provable EUF-CMA secure.
- The choice of a ZK protocol allows for a flexible choice of parameters, trading performance for signature size and vice versa.
- We considered the attack in [31] and a novel forgery attack for fixed-weight challenges from [9]. We considered the computational improvements of this work and designed the system parameters conservatively.
- Restricted error vectors and their transformations can be compactly represented, which significantly reduces the signature sizes compared to other settings, such as when using fixed Hamming weight error vectors.
- The fully random parity-check matrix can be derived on the fly from a small seed using a CSPRNG. This allows us to compress the public key to ≤ 153 B, which means the signature scheme is suitable for highly memory-constrained devices such as smartcards. Furthermore, the small public key size and sub-10 kB signature sizes endorse its use in X.509 certificates.
- The transformations of restricted vectors do not require permutations, which ensures a simplified constant-time implementation.
- Since roughly half of the operations are performed in a smaller field, $\mathbb{F}_z$, the computations are less expensive than in other schemes which use the full ambient space.
- Due to the order of the ambient spaces $\mathbb{F}_p$ and $\mathbb{F}_z$ being either a Mersenne prime or close to one, CROSS enjoys fast arithmetic and achieves fast signature generation and verification.
- Since CROSS only chooses two different ambient spaces, namely $(p = 127, z = 7)$ and $(p = 509, z = 127)$, the code size and area of its realization are more compact concerning schemes that require tailored arithmetic for each NIST security category.
- For the R-SDP variant of CROSS, the choice of z is small enough to allow expensive operations to be performed via a constant-time table lookup, as the entire table fits into a (64-bit) register.
- CROSS only requires simple operations, such as symmetric primitives (CSPRNGs and cryptographic hashes) and vector/matrix operations among small elements. This also allows for a straightforward constant-time implementation of the scheme.
- The nature of the arithmetic operation in CROSS allows efficient vectorization with ISA extensions such as Intel's AVX2: the computation of the arithmetic operations, when vectorized, reduces the amount of time spent in them to a minority in the overall signature time
- Only a single standardized primitive (SHAKE, as per FIPS-202) is required in each CROSS implementation, reducing both hardware and software implementation complexity.

Limitations:

- The achieved signature sizes are still in the range of 9 kB for NIST category 1, which is larger than the standardized signatures Falcon and Dilithium but only slightly larger than those of SPHINCS+. This range of signature sizes is to be expected from a signature scheme derived through a ZK protocol.
- The restricted syndrome decoding problem is relatively new [4], but closely related to the classical syndrome decoding problem and the subset sum problem, both of which are well studied in literature [10, 11]. Due to this relation, the best-known solvers for R-SDP [5, 16] are modifications of the best-known solvers for SDP and the subset sum problem.

9 Bibliography

[1] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In *International Colloquium on Automata, Languages, and Programming*, pages 403–415. Springer, 2011.

[2] Thomas Attema and Serge Fehr. Parallel repetition of $(k_1, \ldots, k_\mu)$-special-sound multi-round interactive proofs. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 415–443, Cham, 2022. Springer Nature Switzerland.

[3] Thomas Aulbach, Samed Düzlü, Michael Meyer, Patrick Struck, and Maximiliane Weishäupl. Hash your keys before signing: BUFF security of the additional NIST PQC signatures. In *International Conference on Post-Quantum Cryptography*, pages 301–335. Springer, 2024.

[4] Marco Baldi, Massimo Battaglioni, Franco Chiaraluce, Anna-Lena Horlemann, Edoardo Persichetti, Paolo Santini, and Violetta Weger. A new path to code-based signatures via identification schemes with restricted errors. *Advances in Mathematics of Communications*, 2025.

[5] 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. *PKC 2024*, 2024.

[6] Magali Bardet, Pierre Briaud, Maxime Bros, Philippe Gaborit, Vincent Neiger, Olivier Ruatta, and Jean-Pierre Tillich. An algebraic attack on rank metric code-based cryptosystems. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 64–93. Springer, 2020.

[7] Elaine Barker and John Kelsey. NIST SP 800-90A Rev. 1 - Recommendation for Random Number Generation Using Deterministic Random Bit Generators. https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final, 2015.

[8] Michele Battagliola, Riccardo Longo, Federico Pintore, Edoardo Signorini, and Giovanni Tognolini. Security of fixed-weight repetitions of special-sound multi-round proofs. Cryptology ePrint Archive, Paper 2024/884, 2024.

[9] Michele Battagliola, Riccardo Longo, Federico Pintore, Edoardo Signorini, and Giovanni Tognolini. A revision of CROSS security: Proofs and attacks for multi-round fiat-shamir signatures. Cryptology ePrint Archive, Paper 2025/127, 2025.

[10] Anja Becker, Jean-Sébastien Coron, and Antoine Joux. Improved generic algorithms for hard knapsacks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 364–385. Springer, 2011.

[11] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in $2^{n/20}$: How 1+ 1= 0 improves information set decoding. In *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*, pages 520–536. Springer, 2012.

[12] Daniel J Bernstein, Tanja Lange, and Christiane Peters. Smaller decoding exponents: ball-collision decoding. In *Advances in Cryptology–CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings 31*, pages 743–760. Springer, 2011.

[13] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. eXtended Keccak Code Package. https://github.com/XKCP/XKCP.

[14] Ward Beullens. Sigma protocols for MQ, PKP and SIS, and fishy signature schemes. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 183–211. Springer, 2020.

[15] Ward Beullens, Pierre Briaud, and Morten Øygarden. A security analysis of restricted syndrome decoding problems. Cryptology ePrint Archive, Paper 2024/611, 2024.

[16] Sebastian Bitzer, Alessio Pavoni, Violetta Weger, Paolo Santini, Marco Baldi, and Antonia Wachter-Zeh. Generic decoding of restricted errors. In *2023 IEEE International Symposium on Information Theory (ISIT)*, pages 246–251. IEEE, 2023.

[17] Giacomo Borin, Edoardo Persichetti, Paolo Santini, Federico Pintore, and Krijn Reijnders. A guide to the design of digital signatures based on cryptographic group actions. Cryptology ePrint Archive, Paper 2023/718, 2023.

[18] Alessio Caminata and Elisa Gorla. Solving degree, last fall degree, and related invariants. *Journal of Symbolic Computation*, 114:322–335, 2023.

[19] Pierre-Louis Cayrel, Pascal Véron, and Sidi Mohamed El Yousfi Alaoui. A zero-knowledge identification scheme based on the q-ary syndrome decoding problem. In *International Workshop on Selected Areas in Cryptography*, pages 171–186. Springer, 2010.

[20] André Chailloux. On the (In) security of optimized Stern-like signature schemes. In *Proceedings of WCC 2022: The Twelfth International Workshop on Coding and Cryptography, March 7 - 11, 2022, Rostock (Germany). URL: https://www.wcc2022.uni-rostock.de/storages/uni-rostock/Tagungen/WCC2022/Papers/WCC_2022_paper_54.pdf*, 2022.

[21] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 392–407. Springer, 2000.

[22] Il'ya Isaakovich Dumer. Two decoding algorithms for linear codes. *Problemy Peredachi Informatsii*, 25(1):24–32, 1989.

[23] Jean-Charles Faugere. A new efficient algorithm for computing Gröbner bases (F4). *Journal of pure and applied algebra*, 139(1-3):61–88, 1999.

[24] Jean Charles Faugere. A new efficient algorithm for computing Gröbner bases without reduction to zero (F 5). In *Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 75–83, 2002.

[25] Ronald Aylmer Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research.* Oliver and Boyd, London, 3rd ed., rev. and enl edition, 1948.

[26] Marco Gianvecchio, Alessandro Barenghi, and Gerardo Pelosi. Towards efficient post-quantum signatures: parallelizing keccak in CROSS and contributing to open-source libraries. Master's thesis, Politecnico di Milano, October 2024. https://hdl.handle.net/10589/227057.

[27] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, August 1986.

[28] Cheikh Thiécoumba Gueye, Jean Belo Klamti, and Shoichi Hirose. Generalization of BJMM-ISD using May-Ozerov nearest neighbor algorithm over an arbitrary finite field $\mathbb{F}_q$. In *International Conference on Codes, Cryptology, and Information Security*, pages 96–109. Springer, 2017.

[29] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)*, 21(2):277–292, 1974.

[30] Nick Howgrave-Graham and Antoine Joux. New generic algorithms for hard knapsacks. In *Advances in Cryptology – EUROCRYPT 2010*, pages 235–256. Springer, 2010.

[31] Daniel Kales and Greg Zaverucha. An attack on some signature schemes constructed from five-pass identification schemes. In *Cryptology and Network Security: 19th International Conference, CANS 2020, Vienna, Austria, December 14–16, 2020, Proceedings*, pages 3–22. Springer, 2020.

[32] Puja Mondal, Supriya Adhikary, Suparna Kundu, and Angshuman Karmakar. ZKFault: Fault attack analysis on zero-knowledge based post-quantum digital signature schemes. Cryptology ePrint Archive, Paper 2024/1422, 2024.

[33] National Institute of Standards and Technology. FIPS 180-4 - Secure Hash Standard (SHS). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf, 2015.

[34] National Institute of Standards and Technology. FIPS 202 - SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf, 2015.

[35] NIST Post quantum standardization effort mailing list. Footguns as an axis for security analysis. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/l2iYk-8sGnI/m/sHWyfvfNDAAJ.

[36] OQS Team. Open Quantum Safe. https://openquantumsafe.org/liboqs/algorithms/.

[37] Jonas Schupp and Georg Sigl. A horizontal attack on the codes and restricted objects signature scheme (CROSS). Cryptology ePrint Archive, Paper 2025/116, 2025.

[38] Jacques Stern. A method for finding codewords of small weight. In *International Colloquium on Coding Theory and Applications*, pages 106–113. Springer, 1988.

[39] The CROSS Team. CROSS: Security details. https://www.cross-crypto.com/resources.html, 2025. Included in the submission package.

[40] Paul C Van Oorschot and Michael J Wiener. Parallel collision search with cryptanalytic applications. *Journal of cryptology*, 12:1–28, 1999.

[41] Henry S. Warren. *Hacker's Delight.* Addison-Wesley Professional, 2nd edition, 2012.

[42] Violetta Weger, Karan Khathuria, Anna-Lena Horlemann, Massimo Battaglioni, Paolo Santini, and Edoardo Persichetti. On the hardness of the Lee syndrome decoding problem. *Advances in Mathematics of Communications*, 2022.