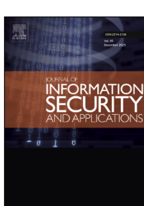
Journal of Information Security and Applications 97 (2026) 104331

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa



A high efficiency AVX2-optimized engineering of the post-quantum digital signature CROSS

Alessandro Barenghi *, Marco Gianvecchio, Gerardo Pelosi

Politecnico di Milano Department of Electronics, Information and Bioengineering (DEIB), Via G. Ponzio 34/5, Milano (MI), 20133, Italy

ARTICLE INFO

Keywords:
Post-quantum cryptosystems
Digital signatures
High performance realizations

ABSTRACT

Post-quantum cryptosystems are currently attracting a significant amount of research efforts due to the continuous improvements in quantum computing technologies, and the inherent high inertia characterizing the replacement of cryptographic standards. This situation has pushed large standardization bodies, such as the USA National Institute of Standards and Technology (NIST), to open standardization competitions to foster proposals and public scrutiny of new quantum-resistant cryptosystems and digital signatures. Whilst NIST has chosen, after four selection rounds (November 2017 - June 2023), three digital signature algorithms, in July 2023 it started a new selection process as the chosen candidates either rely exclusively on lattice-based computationally hard problems, or have unsatisfactory performance figures. In this work, we tackle the performance engineering of the Codes and Restricted Objects Signature Scheme (CROSS), which has been admitted to the second round of selection by NIST in October 2024. We propose a set of techniques to optimize software realizations of CROSS, targeting the AVX2 ISA extension by Intel, as requested by NIST; exploiting fully our choices on the signature scheme parameters, as part of the design team. We note that these techniques are general enough to be ported to other vector ISA extensions (e.g., ARM Neon). We provide a complete performance validation of our realization both with dedicated microbenchmarks as well as full end-to-end TLS benchmarks with realistic network delays. Our results show that CROSS is competitive with each of the already standardized post-quantum signature schemes as well as with the other schemes still under evaluation in the second selection round.

1. Introduction

Post-quantum cryptosystem design and engineering have seen a large amount of effort being dedicated to them, in the light of the strong societal push to the construction of a large scale quantum computer. Indeed large scale, reliable quantum computers have the potential to solve challenging computational problems in material design and chemistry, yielding significant benefits for natural sciences and engineering. However, the availability of such quantum computers also allows to solve two problems: integer factoring and the extraction of discrete logarithms in a cyclic group, which are considered computationally hard for a classical computer. This capability allows in turn to break essentially all the widely deployed asymmetric cryptographic primitives that are based on the RSA encryption and signature transformations, the Diffie-Hellman (DH) and Elliptic Curve DH key agreement protocols, as well as on the U.S. standards known as Digital Signature Algorithm (DSA) and Elliptic Curve DSA (ECDSA). While the construction of a large scale, reliable quantum computer is considered to be approximately a decade away [1], the slow pace at which the update of widely established cryptographic primitives proceeds, calls for a prompt effort towards quantum-resistant, also known as Post-Quantum Cryptography (PQC) primitives. In this light, a major push towards this goal was made by the standardization effort which was put forward by the USA National Institute of Standards and Technology (NIST). Indeed, NIST started in 2017 a public contest [2] with the intention of selecting a portfolio of post-quantum cryptographic algorithms, namely one or more Key Encapsulation Methods (KEMs) and one or more Digital Signature (DS) algorithms. In 2024, NIST released the Federal Information Processing Standards (FIPS) publications providing the specifications for the first set of PQC primitives. FIPS 203, 204, 205 are the three documents describing one KEM (CRYSTALS-Kyber a.k.a. ML-KEM in NIST's parliance) and two digital signatures (CRYSTALS-Dilithium and SPHINCS$^+$ [3] a.k.a. ML-DSA and SLH-DSA, respectively, in NIST's parliance), which are currently NIST standards. In particular, FIPS 204 [4] describes the ML-DSA cryptosystem, which is derived from the CRYSTALS-Dilithium lattice-based algorithm, and has been selected for general-purpose digital signature protocols, while FIPS 205 [5] describes the SLH-DSA cryptosystem, which is derived from the stateless hash-based digital signature

* Corresponding author.
E-mail addresses: alessandro.barenghi@polimi.it (A. Barenghi), marco.gianvecchio@polimi.it (M. Gianvecchio), gerardo.pelosi@polimi.it (G. Pelosi).

https://doi.org/10.1016/j.jisa.2025.104331

Available online 29 December 2025
2214-2126/© 2025 The Author(s). Published by Elsevier Ltd. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

scheme named SPHINCS+ [6]. In 2025, a draft for the FIPS 206 [7] built around the Falcon digital signature algorithm [8] is expected. It will describe the FFT (Fast Fourier Transform) over the Number Theorists R Us (NTRU) [9] lattice-based Digital Signature Algorithm (FN-DSA), which is derived from the hash-then-sign signature scheme employing the NTRU lattices that corresponds to the Falcon algorithm. Despite the selection of three options for digital signature algorithms, between September 2022 and July 2023, NIST called for additional digital signature proposals to be considered in an extra PQC standardization process to diversify its post-quantum signature portfolio [10]. The reason behind this additional call for further proposals is that the general purpose algorithms ML-DSA [4] and Falcon [8] rely on hard problems from discrete lattices, while SLH-DSA [5] (i.e., SPHINCS+ [3]) does not provide satisfactory performance to be employed in interactive scenarios such as TLS communications. This state-of-the-art, which leaves the authentication guarantees provided by post-quantum digital signatures in the vast majority of digital communications to rely on a single computational hard problem, was deemed unsatisfactory. As a consequence, NIST expressed interest in additional signature schemes based on security assumptions distinct from the ones on structured lattices, and outperforming SPHINCS+, leaving open room to lattice-based signatures only if they provide significant benefits over the standardized proposals. SPHINCS+ [6] is characterized by cryptographic key sizes in the range of tens of bytes; nonetheless, it incurs into a significant computational load, with execution times of its signature generation algorithm in the range between a hundredth of a second, and a pair of seconds (0.05 to 2 Gcycles), for an AES-128 equivalent security level on modern desktops, depending on the specific hash function employed in the inner constructions of the scheme.

Among the proposed alternatives which have passed the first round of scrutiny of NIST's additional call for signatures there are the Codes and Restricted Objects Signature Scheme (CROSS) [11] and the Linear Equivalence Signature Scheme (LESS) [12], which rely on computationally hard problems from coding theory, and build a signature out of a zero-knowledge identification scheme paired with the Fiat-Shamir transform [13]. In designing CROSS, the authors, among which we are counted, chose to forsake computationally demanding techniques that trade off a signature size reduction for a significant increase in the required computation time. This choice, in contrast with other schemes also relying on computationally hard problems that come from coding theory such as the Syndrome Decoding-in-the-Head (SDitH) scheme [14], which adopt techniques from the multiparty computation area, allows to obtain low latency signature creation and verification, while exhibiting public key and signature sizes comparable with the ones of SLH-DSA [5]. In particular, the public key sizes are between 77 and 153 bytes, depending on the security level, while signature sizes are between 9 kB and 75 kB.

Given its potential for general purpose applications, engineering efficient software implementations of CROSS for desktop, server and laptop grade x86 − 64 CPUs, and evaluating the practicality of its use in the Transport Layer Security (TLS) protocol is a matter of significant practical interest. In particular, NIST has chosen, as the reference instruction set for which optimizations on x86 − 64 should be tailored the Intel Advanced Vector eXtensions 2 (AVX2) Instruction Set Architecture (ISA) extension. This ISA extension is also fully supported by AMD CPUs, and is now widely present in CPUs of the last decade from both vendors.

Contributions. In this work we describe a set of techniques to realize an AVX2-optimized implementation of CROSS aimed at accelerating its signature and verification times and provide concrete benchmarks of its performance through both microbenchmarks and measurements on TLS protocol instances which employ it. While designed for the AVX2 ISA extension, our techniques can be adapted to other vector-ISA extensions such as the ARM Neon.

In particular, we provide the following contributions to the engineering of software implementations of CROSS.

- **Efficient arithmetic.** We describe optimized scalar arithmetic for the finite fields employed in CROSS, fully exploiting the design choices on the moduli values to speed up the computations. We complement this description with dedicated strategies for AVX2-targeted optimizations, gaining 4× to 15× in speed through parallelization and careful memory layout. All our arithmetic algorithms run in constant (i.e., input value independent) time.
- **Parallelizing SHAKE.** A significant amount of time of Fiat-Shamir based signatures is spent in the computation of Cryptographically Safe Pseudo Random Number Generators (CSPRNGs) and cryptographic hashes. In CROSS, we took part in choosing for both of these purposes the extensible output function named as Secure Hash Algorithm KECCAK-based variant (SHAKE) specified in the FIPS 202 standard documentation for the Secure Hash Algorithm-3 (SHA-3) family of functions. We exploited the SHAKE intrinsic parallelism to achieve a ≈ 4× speed-up in the AVX2 implementation, compared with the usual algorithm included by cryptographic libraries running on a x86 − 64 ISA, proposing an approach for vector computations of multiple SHAKE instances, while managing data dependencies arising from Merkle- and seed-trees data structures where it is largely employed as per the CROSS specification [11].
- **End-to-end TLS benchmarking.** While signature and public key size alone provide a first gauge of the usability of a cryptographic signature scheme, end-to-end TLS connection latencies and throughput (in terms of maximum sustained new connections per second) provide a clearer picture. We benchmark our AVX2 performance-oriented implementation of CROSS on this workbench, and compare against current standard-selected primitives and MAYO (Multivariate And Yet Other) [15], another prime competitor when it comes to low-latency signature schemes admitted to second round of analysis [16] in the current NIST standardization contest, with security guarantees based on the difficulty of solving systems of simultaneous multivariate quadratic equations.

Our aim in the proposed contribution is to show that it is possible to employ a Fiat-Shamir transform based signature scheme in interactive loads such as TLS communications with good performances despite the signature size being larger than lattice-based candidates. Indeed, this is the case, as we show that CROSS achieves a higher amount of TLS connections per second with respect to other on-ramp signature candidates such as, SNOVA or UOV, despite microbenchmarks on the signature primitives in isolation may suggest otherwise. As a final remark, our approach to the parallelization of SHAKE can be of general interest for all post-quantum signature schemes employing seed trees or Merkle trees, such as LESS [12].

2. Background

In this section, we provide preliminaries on the notation and algebraic structures involved in CROSS, describe briefly the auxiliary cryptographic primitives needed for a proper realization, and finally provide the signature scheme parameters as well as a procedural description of the three primitives: CROSS.KeyGen, CROSS.Sign, and CROSS.Verify. We will, for the sake of brevity, omit the full theoretical security rationale of CROSS. We point the reader interested to the theoretical security aspects of the scheme to the CROSS official specification [11], and the related security guide [17].

2.1. Fundamentals and notation

In the following, we will denote scalars with lowercase italic letters (e.g., a), vectors with lowercase boldface letters (e.g., $\mathbf{v}$, having elements $\mathbf{v} = [\mathbf{v}_0, \dots, \mathbf{v}_{n-1}]$), matrices with uppercase italic letters (M), the support of algebraic groups as uppercase boldface letters ($\mathbf{E}$), and the finite field with p elements as $\mathbb{F}_p$, where p is a prime number. CROSS constructs on linear block error-correcting codes: given a field $\mathbb{F}_p$, a linear block

code over $\mathbb{F}_p$ with length $n > 0$ and dimension k, $0 < k \leq n$, is identified with the set vectors (also known as *codewords*) having length n that are obtained by multiplying all the vectors over $\mathbb{F}_p$ with length k (also known as *information words*) by a matrix $G \in \mathbb{F}_p^{k\times n}$ with (row-)rank k, which is known as the *generator matrix*. A given code can be described by multiple generator matrices, which are all equivalent up to a linear combination of their rows. A generator matrix is in systematic form if $G = [I, V]$, where I is a $k \times k$ identity matrix and $V \in \mathbb{F}_p^{k\times(n-k)}$. Each generator matrix has an associated *parity check matrix* $H \in \mathbb{F}_p^{(n-k)\times n}$ such that $GH^T = N$, where N is a null, $k \times (n-k)$ matrix. Multiplying any codeword $\mathbf{c}$ by H yields a null vector $\mathbf{s} = H\mathbf{c}^T$ known as *syndrome*. Multiplying a random vector $\mathbf{v}$ of length n over $\mathbb{F}_p$ by H yields in general a non null syndrome $\mathbf{s} = H\mathbf{v}^T$, which can be thought, due to linearity, as $\mathbf{s} = H\mathbf{v}^T = H\mathbf{c}^T + H\mathbf{e}^T = H\mathbf{e}^T$, where $\mathbf{c}$ is a codeword, and $\mathbf{e}$ is an *error* vector. The computational problem of finding, given a uniformly randomly drawn parity check matrix H and a non null syndrome $\mathbf{s}$, any error vector $\mathbf{e}$ corresponding to the syndrome $\mathbf{s}$, i.e., such that $\mathbf{s} = H\mathbf{e}^T$ is easy, unless some constraints are imposed on the values of $\mathbf{e}$. The most common constraint is that the number of non null entries of $\mathbf{e}$, i.e., its Hamming weight, is below a constant fraction of n, giving rise to the canonical (search) *Syndrome Decoding Problem* (SDP), which is proven to be NP-hard.

CROSS relies on a variant of the SDP, also proven NP-hard, where the restriction on $\mathbf{e}$ mandates, besides a fixed Hamming weight, that all its non null entries belong to a multiplicative subgroup $\mathbf{E}$ of $\mathbb{F}_p^*$, generated by a public element g of prime order z, i.e.: $\langle g \rangle = \{g^i \mid i \in \{1, \dots, z\}\} = \mathbf{E} \subset \mathbb{F}_p^*$, therefore the error vector $\mathbf{e}$ belongs to $\mathbf{E}^n$. This variant of the SDP is known as the Restricted SDP (RSDP), and allows to have instances with a unique solution even if all entries of $\mathbf{e}$ are non-null. Given that the complexity of the best solvers for both SDP and RSDP is exponential in the Hamming weight of the error vector $\mathbf{e}$, building schemes relying on RSDP allows to employ codes with a shorter length than corresponding schemes based on SDP.

To further improve the efficiency of the scheme, CROSS considers also another restriction for the values of the error vector. To describe the restriction, observe that it is possible to build a bijection between elements of $\mathbf{E}^n$ and elements of $\mathbb{F}_z^n$: indeed, each component of $\mathbf{e} \in \mathbf{E}^n$ can be expressed as $g^{\overline{x}}$, with $\overline{x} \in \mathbb{F}_z$. Denoting, for the sake of clarity, all objects either in $\mathbb{F}_z$ or composed by elements of $\mathbb{F}_z$ with an overline, we consider the vector $\overline{\mathbf{e}} = [\overline{\mathbf{e}}_0, \dots, \overline{\mathbf{e}}_{n-1}] \in \mathbb{F}_z^n$ to be in correspondence with $\mathbf{e} = [\mathbf{e}_0, \dots, \mathbf{e}_{n-1}] = [g^{\overline{e}_0}, \dots, g^{\overline{e}_{n-1}}] \in \mathbf{E}$. For the sake of brevity, we will denote $[g^{\overline{e}_0}, \dots, g^{\overline{e}_{n-1}}]$ as $g^{\overline{\mathbf{e}}}$. This bijection is useful to manipulate and describe elements of (subsets of) $\mathbf{E}$, as it is straightforward to observe that sums of elements in $\mathbb{F}_z^n$ correspond to component-wise products of elements in $\mathbf{E}$, i.e., $(\mathbb{F}_z^n, +)$ is isomorphic to $(\mathbf{E}, \odot)$, where $\odot$ denotes the component-wise product of two vectors in $\mathbf{E}$, i.e, with $\mathbf{a}, \mathbf{b} \in \mathbf{E}^n$, $\mathbf{a} \odot \mathbf{b} = g^{\overline{\mathbf{a}}+\overline{\mathbf{b}}}$. The alternative restriction for the values of an error vector $\mathbf{e}$ employed by CROSS can be synthetically described as follows: valid error vectors $\mathbf{e}$ are such that the corresponding $\overline{\mathbf{e}} \in \mathbb{F}_z^n$ is a codeword of a code of length n and dimension m, i.e., they are obtained by multiplying a vector $\mathbb{F}_z^m$ by a generator matrix $\overline{M} \in \mathbb{F}_z^{m\times n}$. It is straightforward to observe that the set of vectors $\mathbf{G}$, composed by all the vectors in $\mathbf{E}^n$ corresponding to information words in $\mathbb{F}_z^m$ form a subgroup of $(\mathbf{E}^n, \odot)$ (as their corresponding vectors in $\mathbb{F}_z^n$ do, being codewords, with respect to $(\mathbb{F}_z^n, +)$). Therefore, the SDP where the admissible solutions belong to $\mathbf{G}$ is called Restricted SDP with subgroup $\mathbf{G}$ (RSDP($\mathbf{G}$)). Note that any element in $\mathbf{a} \in \mathbf{G}$ can be compactly represented by a vector in $\overline{\mathbf{v}} \in \mathbb{F}_z^m$, and computed as $\mathbf{a} = g^{\overline{\mathbf{v}}\overline{M}}$.

Auxiliary primitives. Besides modular arithmetic, CROSS employs in its construction multiple instances of two elementary symmetric cryptographic primitives, i.e., a CSPRNG and a cryptographic hash function (Hash). The CROSS designers chose to instantiate both the CSPRNG and the Hash function by employing a single concrete primitive, the extendable output function SHAKE [18], taking either SHAKE-128 or SHAKE-256 as appropriate to the security margin required. The choice is justified both in terms of performance, as SHAKE provides a high throughput CSPRNG, and in terms of reducing hardware resources to a minimum, employing a single module for both purposes. In order to instantiate multiple, independent CSPRNGs and Hashes, CROSS adopts a simple *domain separation* strategy consisting in appending a unique 16-bit integer to the input material of each distinct instance of SHAKE. In particular, a call to the j-th CSPRNG having output in a range defined as the set $\mathcal{S}$, denoted as CSPRNG-$\mathcal{S}$(seed | j), is realized providing as input to SHAKE the bitstring obtained from the concatenation of the binary representation of seed with the 16-bit natural binary encoding of j, $0 \leq j < 2t$, e.g., $2t = 2^{15}$, and extracting from SHAKE as many pseudorandom bits as needed to generate the element of $\mathcal{S}$, either directly (for sets of binary strings) or via rejection sampling for objects composed of elements of either $\mathbb{F}_p$ or $\mathbb{F}_z$.

The i-th Hash function, denoted as HASH(bstr | i), is instantiated concatenating the input binary string bstr with the 16-bit binary representation of the value $2t + i$, $0 \leq i < 2t$, e.g., $2t = 2^{15}$, thus providing complete domain separation from CSPRNGs, i.e., making it so that HASH(bstr | i) and CSPRNG-$\mathcal{S}$(seed | j), with $0 \leq i, j < 2t$ never provide the same output even if $i = j$, and bstr and seed take the same value, unless a collision in the output of SHAKE occurs.

Building on CSPRNGs and Hash functions, CROSS instantiates also two additional auxiliary primitives, namely *seed-trees*, which are also known as either GGM trees or puncturable pseudorandom functions [19], and Merkle trees [20].

Seed-trees are a pseudo random function (PRF) construct, i.e., they are efficient (computable in polynomial time) deterministic functions that map two distinct sets of binary strings (domain and range) and look like truly random functions. A seed-tree allows to reveal only portions of its output binary string in a more compact fashion than revealing the portions of the output bit sequence themselves. To this end, seed-trees employ multiple calls to a CSPRNG that has a binary output string twice as long as the one employed as seed (*length-doubling CSPRNG*). The multiple calls are logically arranged into a binary tree, with each call taking the place of an inner node of the tree, while leaf nodes are associated with binary strings that, concatenated from left to right, represent the output of the seed-tree. Note that, in the seed-tree construction, no requirements on the balancedness of the tree are imposed, although balancing it improves efficiency. There are several ways to build such a binary tree, in CROSS the trees are full (every node has either 0 or 2 children) and every node has a perfect left subtree [21, Section 3.3].

Fig. 1 reports a running example of a seed-tree having its output sequence partitioned into 11 subsequences, $\{o_0, \dots, o_{10}\}$, of which o_4 and o_8 should not be revealed, and balanced according to the CROSS convention. Each trapezoidal (inner) node represents a length-doubling CSPRNG, while a rectangular (leaf) node represents a binary string. The observation allowing to reveal the values of all the green leaves, is that revealing the input to a CSPRNG reveals also the input to all nodes that are its descendants. It is therefore possible to compress the representation of a sequence of leaves that all have to be revealed, representing them with the input of the CSPRNG that is their farthest common ancestor. In the running example, it is thus possible to reveal all the values but o_4 and o_8, by revealing the inputs to CSPRNG 3 and CSPRNG 10, and the values o_5, o_9 and o_{10}. Having leaves and CSPRNG inputs the same size, we reveal only 5 bitstrings instead of 9. We will denote as *seed-tree path* the set of revealed values (5 in the previous example) which allow to reconstruct the leaves to be revealed, and the paths toward them wherever needed.

A Merkle tree is a cryptographic primitive that, at the ends, provides the same guarantees of a cryptographic hash function, i.e., they accept an arbitrarily long binary string and return a deterministically computed pseudorandom digest with a fixed bit length. The advantage provided by Merkle trees in the CROSS context is that it is possible to recompute the digest starting from a portion of the input string and a more compact representation of the remaining computation. Merkle trees are binary tree data structures that achieve this by adopting a strategy that can

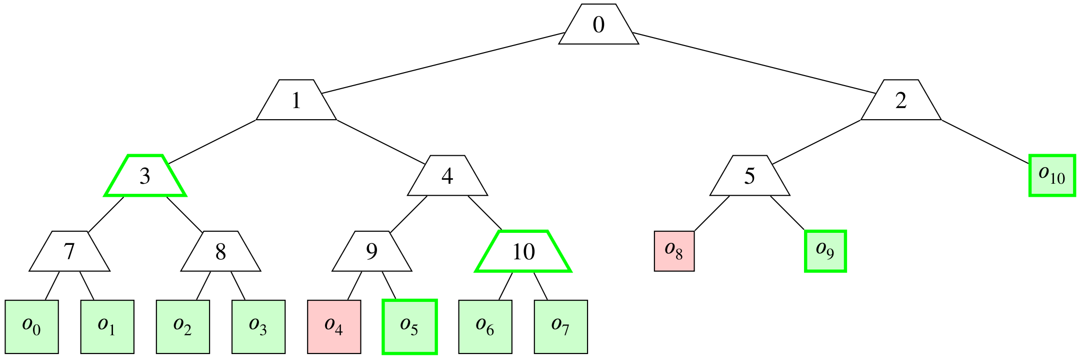


Fig. 1. Depiction of a seed-tree for the output of a pseudo random function partitioned into 11 substrings, $\{o_0, \dots, o_{10}\}$, of which o_4 and o_8 should not be revealed. Trapezoidal (inner) nodes are length-doubling CSPRNGs, while rectangular (leaf) nodes represent binary strings. A compact representation of the seed-tree output portions that has to be revealed is obtained revealing only the inputs to the CSPRNGs that have their border in green, and the strings associated with leaves also having green borders. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Algorithm 1: KeyGen()

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

be thought as the dual of seed-trees. The input message of a Merkle tree is split into as many portions as its leaves, and fed to the hash call associated to each leaf node, initializing the computation. Each inner node also contains a call to a cryptographic hash, taking as input the digests of its children. The final digest is yield by the hash call associated with the root node. In CROSS, we need to be able to compute the digest of a bit sequence without being in knowledge of portions of it. This can be efficiently done by knowing only the digests output by the farthest ancestors of the leaves that are meant to process portions of the input that should remain hidden. Indeed, the computation of the Merkle tree then moves on directly from these values, and the public portion of the input message, up to recomputing the tree root output. In the following, we will denote as a *Merkle tree proof*, the set of intermediate digests that, paired to the portion of the input message that can be known, allow a verifier to obtain a proof that the Merkle tree digest of the entire message is actually the one being claimed by someone.

2.2. The CROSS signature scheme

The CROSS signature scheme is built turning into a non interactive signature via Fiat-Shamir Transform a set of parallel repetitions of a Zero Knowledge Identification Scheme, named as CROSS-ID. From an operational standpoint this implies that the verification key corresponds to an instance of the RSDP/RSDP(G) problem, i.e., given a set of simultaneous equations as a parity check matrix-syndrome pair, the error vector having components that solve the said set of equations is the private signature key.

The CROSS-ID scheme allows the interactive prover to show that he knows the solution to the problem instance (i.e., the private portion of the keypair), without revealing the solution itself, by generating another random (solved) instance starting from it and showing the verifier either the solution, or the build process of the fresh random instance. To do so, the prover *commits* to the randomly generated problem instance and its solution (i.e., computes their hashes and sends them to the verifier), waits for the verifier to randomly pick if she chooses to see the solution, or the preparation of the procedure, and reveals whatever the verifier chose to inspect by sending her the input to the hash. The Fiat-Shamir transform turns this interactive procedure into a non interactive scheme by observing that the random pick from the verifier is made public, hence substituting it with the output of a cryptographic hash of a transcript containing all commitments and the message to be signed. The entity verifying the signature just needs to replay the interactions, and check that the pseudorandom challenge values are indeed obtained as digests of the appropriate values. Since a CROSS-ID protocol involves two commit-challenge-response interactions between the prover and the verifier, the Fiat-Shamir transform mandates the incorporation of the entire transcript of the first interaction, plus the commitments for the second one onto the hash input to generate the second challenge. Given that an attacker would have a non negligible chance of cheating when a single repetition of each CROSS-ID interaction takes place, the complete CROSS signature scheme requires $t > 1$ interactions to achieve a

cheating probability low enough to force an adversarial prover (attacker) to spend an unpractical computational effort to forge the signature (i.e., successfully pass the verification phase), matching the one associated to a desired security level [11], e.g., the same computational effort required for an exhaustive search of the value of the cryptographic key of an AES-128 cipher instance.

CROSS.KeyGen. The keypair generation procedure, reported in Algorithm 2.2, shows the steps exclusively done by the RSDP variant of CROSS in teal (on the right), while the ones for CROSS RSDP(G) are in orange (on the left). The key generation procedure starts by obtaining the private key sk as a truly random 2λ-bit seed, where $\lambda \in \{128, 192, 256\}$ denotes the prescribed security level. Such a value, $\texttt{seed}_{\text{sk}}$ (line 1), is in turn employed to derive both the CSPRNG seed from which the secret error vector $\mathbf{e}$ will be expanded and the CSPRNG seed $\texttt{seed}_{\text{pk}}$ equivalent to the public key material (i.e., a parity check matrix with maximal rank) (line 2). Indeed, given the random nature of the parity check matrix H, and for the RSDP(G) case the generator matrix of the code for the exponents $\overline{M}$, the CROSS key generation procedure proceeds to generate either one (for RSDP) or two (for RSDP(G)) matrices (line 3–4) directly in systematic form by generating only the non-identity portion(s) V (and $\overline{W}$ for RSDP(G)). After expanding the matrix(-ces) the exponent-vector $\bar{\mathbf{e}}$ corresponding to the restricted error vector is expanded via CSPRNG (line 5), and the corresponding restricted vector $\mathbf{e}$ is computed (lines 6). Finally, the syndrome corresponding to the second portion of the public key is computed (line 7), and returned together with the seed from which it is possible to expand the public parity-check matrix H (and M for RSDP(G)).

CROSS.Sign. The CROSS signature generation procedure is described in Algorithm 2.2. CROSS.Sign starts by expanding the secret key material, i.e., $\texttt{seed}_{\text{sk}}$ following the same procedure as CROSS.KeyGen, while retaining all intermediate results (line 1). The procedure then draws two uniformly random bit strings, seed and salt from the system True Random Number Generator (TRNG) (line 2); the former is employed to generate all ephemeral random elements in the signature, while the latter is meant to prevent advanced multi-target attacks [11] by differentiating two signatures made on the same message. The procedure expands the seed and salt values into a sequence of t seeds, to be used in the corresponding t repetitions of the CROSS-ID scheme. The t parallel repetitions are operatively computed sequentially in Algorithm 2.2 for the sake of clarity. To highlight this fact, we employ an array-index notation $[i]$ to denote that a given value is the one corresponding to the i-th iteration of the protocol (e.g., seed[0] is the seed corresponding to the first protocol iteration). The expansion of seed and salt through the computation of a *seed-tree* [22] with t leaves, $(\texttt{seed}[0], \dots, \texttt{seed}[t-1])$ (line 3). The loop at lines 4–12 is in charge of preparing all pairs of commitments for each parallel protocol iteration. In particular, the first commitment for the i-th iteration, $\texttt{cmt}_0[i]$ is prepared starting from the pseudorandom generation of a random vector $\mathbf{u}'[i]$ in $\mathbb{F}_p^n$ and one in the restricted vector set depending on the RSDP/RSDP(G) variant, $\mathbf{e}'[i]$ (line 5). The latter is employed to hide the value of the actual private key $\mathbf{e}$, transforming it into the ephemeral $\mathbf{v}[i]$ restricted error vector (lines 6–7).

The $\mathbf{v}[i]$ restricted error vector is thus component-wise multiplied by $\mathbf{u}'[i]$ and the syndrome $\mathbf{s}'[i]$ of the result through H is computed at line 9. The procedure then commits on the pair of values $\mathbf{s}'[i]$, $\overline{\mathbf{v}}[i]$ hashing them together with the salt (line 10), for the RSDP variant. In the RSDP(G) case, CROSS saves a small amount of hashing computation by committing on the compact representation of $\overline{\mathbf{v}}[i]$, $\overline{\mathbf{v}}_G[i]$ which exploits the fact that the random restricted vector $\overline{\mathbf{e}}'[i]$ belongs to the restriction within $\mathbf{E}$. The second commitment for the i-th iteration, $\texttt{cmt}_1[i]$, commits on the preparation of the solution of the randomized problem: to do so, it is sufficient for the signer to compute the digest of $\texttt{seed}[i]$, as all the pseudorandom values in the iteration are derived deterministically from it. After computing all $\texttt{cmt}_0[i]$ and $\texttt{cmt}_1[i]$ values for all t parallel iterations, CROSS.Sign hashes them into two digests, $\texttt{dig}_{\texttt{cmt}_0}$ and $\texttt{dig}_{\texttt{cmt}_1}$ (line 12): all $\texttt{cmt}_1[i]$ are simply concatenated and hashed, while $\texttt{cmt}_0[i]$ are hashed via a Merkle tree construction (i.e., a binary tree where each inner node is a hash computation, performed on the digests output by its children, and the leaves are input messages). The resulting digests are hashed (line 13) obtaining $\texttt{dig}_{\texttt{cmt}}$, in order to reduce the amount of data to be included in the signature (as they would be separately included otherwise), and $\texttt{dig}_{\texttt{cmt}}$ is hashed together with the message digest (obtained at line 14) to generate the pseudorandom salt to generate the first challenge $\texttt{dig}_{\texttt{chall}_1}$ (line 15). CROSS.Sign then proceeds to generate a vector of multiplicative factors in $\mathbb{F}_p^*$ constituting the first challenge, (line 16) and computing the t responses to the first challenge (loop at lines 17–19). The responses, computed adding the pseudorandom vectors $\mathbf{u}'[i]$ to a copy of the ephemeral restricted vector $\mathbf{e}'[i]$, rescaled by the first challenge $\texttt{chall}_1[i]$ are then hashed to obtain the second challenge $\texttt{chall}_2[i]$, that is a length t binary string with fixed Hamming weight w. The reason for the selection of a fixed weight string is a signature design trade-off allowing to reduce the signature size at the cost of a higher number of rounds [11]. CROSS.Sign completes the signature generation by packing into the signature itself the responses to the second challenge, for all rounds where the second challenge is null-valued. Indeed, in these cases, the verifier will check if the signer is able to provide a solution for the RSDP/RSDP(G) instance at hand. In all other cases, the verifier will check that the signer honestly prepared the problem instances: to do so, he will only need the $\texttt{seed}[i]$ values corresponding to rounds where the second challenge is not null: such values are contained in the path variable output by SeedPath (line 23). Finally, the signer adds to the signature all the nodes of the Merkle tree allowing the verifier to recompute its root (i.e., the proof variable).

CROSS.Verify. The CROSS.Verify procedure reported in Algorithm 2.2, starts the signature validation procedure by expanding the public matrices $\overline{W}, V$ (line 1), reconstructing the value of $\texttt{chall}_1$ and extracting $\texttt{chall}_2$ from the signature (lines 2–6). Once this step is complete, the verifier proceeds to reconstruct the values $\texttt{dig}_{\texttt{cmt}_0}, \texttt{dig}_{\texttt{cmt}_1}$, recomputing in turn the missing inputs leaves to the Merkle tree (for $\texttt{dig}_{\texttt{cmt}_0}$) and the missing $\texttt{cmt}_1$ (for $\texttt{dig}_{\texttt{cmt}_1}$) in the loop at lines 7–20. In particular, depending on whether $\texttt{chall}_2[i] = 0$ (lines 9–13) or $\texttt{chall}_2[i] = 1$ (lines 14–20) the value of $\texttt{cmt}_0[i]$ or $\texttt{cmt}_1[i]$ is recomputed, while the other one is directly extracted from the signature material. Besides reconstructing $\texttt{cmt}_0[i]$ or $\texttt{cmt}_1[i]$, the verifier also computes all the responses $\mathbf{y}[i]$ to the first challenge in the loop at lines 7–20. This allows the verifier to recompute the expected values $\texttt{dig}'_{\texttt{cmt}}, \texttt{dig}'_{\texttt{chall}_2}$ for $\texttt{dig}_{\texttt{cmt}}$ and $\texttt{dig}_{\texttt{chall}_2}$ (lines 21–22), and compare them to the latter ones contained in the signature (lines 23–24) and returning that the signature is valid if the reconstructed values match the ones in the signature.

CROSS parameter choices. CROSS employs fixed small primes as the choices for p and z, in particular $p = 127$ and $z = 7$ for RSDP and $p = 509$, $z = 127$ for RSDP(G), while the tuning on the security margin against attacks to RSDP/RSDP(G) is done by choosing different lengths and dimensions for the $[n, k, d]$ random linear codes. CROSS provides parameters for three different quantitative security targets, known as *categories* in NIST's parliance, namely category 1, 3 and 5, which imply that an attack against CROSS should take the same amount of computation that is required to attack the AES cipher with a 128, 192, or 256 bits long key, respectively. Finally, the designers of CROSS propose, for each security level three different engineering trade-offs between a short signature and a fast computation. The resulting optimization corners are a *fast* signature variant, one generating *small* signatures, and a *balanced* option in between the previous two choices. Thus, the CROSS specification defines 18 parameter sets [11] as triples CROSS-[RSDP, RSDP(G)]-category-optimization, e.g., CROSS-RSDP(G)-1-balanced refers to the CROSS variant relying on RSDP(G), with a security guarantee equivalent to AES-128, and tailored for a trade-off between signature size and speed.

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

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

```
// Expanding secret key
1  ē, ē_G, H, M̄ ← ExpandSK(Seed_sk)   |   ē, H ← ExpandSK(Seed_sk)
// Computing the commitments
2  Seed ← {0,1}^λ,   Salt ← {0,1}^{2λ}
3  (Seed[1], ..., Seed[t]) ← SeedLeaves(Seed, Salt)
// Compute v[i] such that v[i] ⋆ e'[i] = e
4  for i from 1 to t do
5      ē'_G[i], u'[i] ← CSPRNG_{F_z^m × F_p^n}(Seed[i] | Salt | i + c)   |   ē'[i], u'[i] ← CSPRNG_{F_z^n × F_p^n}(Seed[i] | Salt | i + c)
       v̄_G[i] ← ē_G − ē'_G[i]
       ē'[i] ← ē'_G[i]M̄
       v̄[i] ← ē − ē'[i]
6      for j from 1 to n do
7          v[i]_j ← g^{v̄[i]_j}
8      u[i] ← v[i] ⋆ u'[i],  s'[i] ← u[i]H^T
9      cmt_0[i] ← Hash(s'[i] | v̄_G[i] | Salt | i + c)   |   cmt_0[i] ← Hash(s'[i] | v̄[i] | Salt | i + c)
       cmt_1[i] ← Hash(Seed[i] | Salt | i + c)
10 digest_cmt0 ← MerkleTreeRoot(cmt_0[1], ..., cmt_0[t]),  digest_cmt1 ← Hash(cmt_1[1] | ... | cmt_1[t])
11 digest_cmt ← Hash(digest_cmt0 | digest_cmt1)
// Computing first challenge
12 digest_Msg ← Hash(Msg)
13 digest_chall1 ← Hash(digest_Msg | digest_cmt | Salt)
14 chall_1 ← CSPRNG_{(F_p^*)^t}(digest_chall1 | t + c)
// Computing first response
15 for i from 1 to t do
16     for j from 1 to n do
17         e'[i]_j ← g^{ē'[i]_j}
18     y[i] ← u'[i] + chall_1[i]e'[i]
// Computing second challenge
19 digest_chall2 ← Hash(y[1] | ... | y[t] | digest_chall1)
20 chall_2 ← CSPRNG_{B(t,w)}(digest_chall2 | t + c + 1)
// Computing second response
21 Proof ← MerkleTreeProof(cmt_0[1], ..., cmt_0[t], chall_2)
22 Path ← SeedPath(Seed, Salt, chall_2)
23 for i from 1 to t do
24     if chall_2[i] = 0 then
25         resp[i]_0 ← (y[i], v̄_G[i])   |   resp[i]_0 ← (y[i], v̄[i])
           resp[i]_1 ← cmt_1[i]
26 Sgn ← (Salt, digest_cmt, digest_chall2, Path, Proof, resp) // Assembling signature
27 return Sgn
```

2.3. Brief background on TLS

Since TLS will be providing our main use case and benchmark, we briefly recall its functioning and the challenges in transiting it to post-quantum primitives. TLS (currently, ver. 1.3 [23]) provides end-to-end security of data sent between client-server applications over the Internet, executing an ephemeral Diffie-Hellman key exchange with the use of several digital signatures that are in turn verified employing the public keys provided through X.509 certificates. The widespread use of TLS in secure web browsing, e-mail, file transfers, video/audioconferencing, instant messaging and voice-over-IP applications, as well as in the Domain Name System (DNS) protocol and in the NTP Network Time Protocol (NTP), make paramount its full transitioning to the use of post-quantum cryptographic schemes. The work for a post-quantum enhanced TLS started with the Internet draft [24] describing the use of a hybrid key exchange protocol (i.e., making use of pre- and post-quantum cryptographic tools). In particular, it describes a scheme to combine the CRYSTALS-Kyber KEM with a pre-quantum digital signature scheme, to the end of obtaining a secure key exchange mechanism executed in the TLS handshake phase as long as one of the underlying cryptographic algorithms remains unbroken. It is noteworthy that the above proposal is only for key exchange, and it does not tackle authentication algorithms.

Willing to analyze how many instances of a post-quantum KEM and of a digital signature are needed in TLS 1.3, a brief summary of the main steps executed during the protocol is as follows. A server listens for new connections on the port 443 of the Transmission Control Protocol (TCP), a client connects to port 443 and initiates the handshake process with a `ClientHello` message to the server. Such a message includes a list of 5 AEAD (Authenticated Encryption and Auxiliary Data) cipher suites in descending order of preference that the client supports, a list of ephemeral public keys that the server might find suitable for establishing a shared secret by executing the Diffie-Hellman protocol with one set of parameters in a list of options, which in turn includes both elliptic curve and finite field algebraic structures, and the protocol versions that the client can support. Server composes and transmits back to client the `ServerHello` message, which includes an ephemeral public key for key exchange, the selected cipher suite (e.g., TLS_AES_256_GCM_SHA384), and the negotiated protocol version followed by an AEAD encrypted payload. In fact, the server after the reception of the `ClientHello` message (equivalently for the client after the reception of the `ServerHello` message), is able to generate the shared secret, ss, and via a Key Derivation Function (KDF) fed with ss and the SHA384 of the `ClientHello` and `ServerHello` messages, a group of symmetric-keys employed for various purposes during the execution of the protocol, among which the

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

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

// Recovering public key
1. $(\overline{\mathbf{W}}, \mathbf{V}) \leftarrow \mathsf{CSPRNG}\text{–}_{\mathbb{F}_z^{m\times(n-m)} \times \mathbb{F}_p^{(n-k)\times k}}\left(\mathsf{Seed}_{\mathtt{pk}} \mid 3t+2\right)$ | $\mathbf{V} \leftarrow \mathsf{CSPRNG}\text{–}_{\mathbb{F}_p^{(n-k)\times k}}\left(\mathsf{Seed}_{\mathtt{pk}} \mid 3t+2\right)$
2. $\mathbf{H} \leftarrow [\mathbf{V} \mid \mathbf{I}_{n-k}]$
3. $\overline{\mathbf{M}} \leftarrow [\overline{\mathbf{W}} \mid \mathbf{I}_m]$

// Computing challenges
4. $\mathtt{digest}_{\mathtt{Msg}} \leftarrow \mathsf{Hash}(\mathtt{Msg})$
5. $\mathtt{digest}_{\mathtt{chall}_1} \leftarrow \mathsf{Hash}(\mathtt{digest}_{\mathtt{Msg}} \mid \mathtt{digest}_{\mathtt{cmt}} \mid \mathtt{Salt})$
6. $\mathtt{chall}_1 \leftarrow \mathsf{CSPRNG}\text{–}_{(\mathbb{F}_p^*)^t}\left(\mathtt{digest}_{\mathtt{chall}_1} \mid t+c\right)$
7. $\mathtt{chall}_2 \leftarrow \mathsf{CSPRNG}\text{–}_{B_{(t,w)}}\left(\mathtt{digest}_{\mathtt{chall}_2} \mid t+c+1\right)$

// Computing commitments
8. $(\mathsf{Seed}[i])_{i:\mathtt{chall}_2[i]=1} \leftarrow \mathsf{RebuildLeaves}(\mathtt{Path}, \mathtt{chall}_2, \mathtt{Salt})$
9. **for** i **from** 1 **to** t **do**
10.   **if** $\mathtt{chall}_2[i] = 1$ **then**
11. $\mathtt{cmt}_1[i] \leftarrow \mathsf{Hash}(\mathsf{Seed}[i] \mid \mathtt{Salt} \mid i+c)$
12. $\mathbf{e}'_G[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG}\text{–}_{\mathbb{F}_z^m \times \mathbb{F}_p^n}(\mathsf{Seed}[i] \mid \mathtt{Salt} \mid i+c)$; $\overline{\mathbf{e}}'[i] \leftarrow \overline{\mathbf{e}}'_G[i]\overline{\mathbf{M}}$ | $\overline{\mathbf{e}}'[i], \mathbf{u}'[i] \leftarrow \mathsf{CSPRNG}\text{–}_{\mathbb{F}_z^n \times \mathbb{F}_p^n}(\mathsf{Seed}[i] \mid \mathtt{Salt} \mid i+c)$
    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. $(\mathbf{y}[i], \overline{\mathbf{v}}_G[i]) \leftarrow \mathtt{resp}[i]_0$; Check if $\overline{\mathbf{v}}_G[i] \in \mathbb{F}_z^m$; $\overline{\mathbf{v}}[i] \leftarrow \overline{\mathbf{v}}_G[i]\overline{\mathbf{M}}$ | $(\mathbf{y}[i], \overline{\mathbf{v}}[i]) \leftarrow \mathtt{resp}[i]_0$; Check if $\overline{\mathbf{v}}[i] \in \mathbb{F}_z^n$
    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. $\mathtt{digest}_{\mathtt{cmt}_0} \leftarrow \mathsf{RecomputeRoot}(\mathtt{cmt}_0, \mathtt{Proof}, \mathtt{chall}_2)$, $\mathtt{digest}_{\mathtt{cmt}_1} \leftarrow \mathsf{Hash}(\mathtt{cmt}_1[1] \mid \cdots \mid \mathtt{cmt}_1[t])$
23. $\mathtt{digest}'_{\mathtt{cmt}} \leftarrow \mathsf{Hash}(\mathtt{digest}_{\mathtt{cmt}_0} \mid \mathtt{digest}_{\mathtt{cmt}_1})$
24. $\mathtt{digest}'_{\mathtt{chall}_2} \leftarrow \mathsf{Hash}(\mathbf{y}[1] \mid \cdots \mid \mathbf{y}[t] \mid \mathtt{digest}_{\mathtt{chall}_1})$
25. **if** $\mathtt{digest}_{\mathtt{cmt}} = \mathtt{digest}'_{\mathtt{cmt}}$ **and** $\mathtt{digest}_{\mathtt{chall}_2} = \mathtt{digest}'_{\mathtt{chall}_2}$ **then**
26.   **return** True
27. **return** False

most notable ones can be denoted as $\mathtt{k}$ (handshake secret key), $\mathtt{k}'$(server application key and IV-initialization vector), $\mathtt{k}'''$(client application key and IV), and $\mathtt{k}''$ (client handshake key and IV).

After the ServerHello message, the server sends a leaf certificate authenticating its public key, a signature computed on a hashed transcript of the connection, and a key-confirmation bitstring. A further AEAD encrypted message from server to client, employing the auxiliary key $\mathtt{k}'$ allows the validation checking on all parts of the exchanged messages to be performed. On the client side, the reception of ss and of the ServerHello message allows to re-compute all auxiliary keys, while $\mathtt{k}''$ is employed to transmit back to the server the AEAD encrypted ciphertext of the key-confirmation bitstring. Finally, the client application key $\mathtt{k}'''$ is used by the client to encrypt the data transmitted from it.

It is worth noting that the above mentioned leaf certificate is signed by a Certification Authority (CA), which is usually an intermediate CA, thus in most of the cases a chain with at least two certificates is needed to provide a root certificate. Furthermore, it is increasingly common that a leaf certificate includes at least two Signed Certificate Timestamps (SCTs). These SCTs are signatures created by Certificate Transparency (CT) logs [25,26] to attest they've been publicly logged. In the future three or more SCTs might be required. Finally, servers may also send an Online Certificate Status Protocol (OCSP) *staple* [27] to demonstrate a certificate hasn't been revoked: It allows the server to bear the resource cost involved in providing (possible) responses of the Online Certificate Status Protocol (OCSP) by appending ("stapling") a time-stamped OCSP response signed by the CA to the initial TLS handshake – enhancing security and performance, with no need for clients to contact the CA.

Thus, when an intermediate CA is present, there is *a minimum of five signatures* (not counting the OCSP staple) and two public keys transmitted across the network *to establish a new TLS connection*. Only the handshake transcript signature is created online; the other signatures appearing in a certificate are created ahead of time. For these signatures, fast verification is much more important than fast signing. On the other hand, for the handshake signature, the desiderata is to minimize the sum of signing and verification time. Only the public keys of the leaf and intermediate certificates are transmitted on the communication channel during the handshake, and for those the desiderata is to minimize the combined size of the signature and the public key. For the other signatures (e.g., the two SGTs), the public key is not transmitted during the handshake, and thus a scheme with larger public keys would be tolerable, and preferable if it trades larger public keys for smaller signatures.

3. Engineering CROSS for AVX2 platforms

In this section, we present the optimization techniques for AVX2 endowed platforms, starting by a computational complexity and bottleneck analysis for CROSS, and then describing the realization of scalar

and vector arithmetic operations. Finally, we report our parallelization strategy for the execution of SHAKE instances involved in the computation of CSPRNGs in seed- and Merkle trees. The presented optimizations do not alter the underlying CROSS protocol, whose security properties have been extensively analyzed in [17]. Furthermore, Section 3.5 provides evidence of constant-time behavior.

3.1. Computational complexity and performance bottleneck analysis

Designing an optimized implementation of CROSS starts from an analysis of the expected computational bottlenecks for each of the CROSS primitives, and an estimate of their parallelizability. We note that, after conducting the synthetic analysis, we validated the fact that its results allows to identify the actual bottlenecks of the software computation in CROSS, as they account for more than 95% of the total computation time.

CROSS.KeyGen. The keypair generation primitive in CROSS is constituted of one or two matrix multiply operations (for RSDP and RSDP(G), respectively), a vector-exponentiation and two calls to CSPRNGs to expand the matrices. Noting that one of the CSPRNG calls is expected to yield enough material to fill the V matrix (and the $\overline{W}$ matrix for RSDP(G)) thus yielding an $\mathcal{O}(n^2)$ amount of field elements, we have that the vector-matrix multiply operations and CSPRNG calls take the largest amount of computation, with the vector-exponentiation completing the picture.

CROSS.Sign and CROSS.Verify. The CROSS signature computation is dominated by the elements that are constituting the t parallel iterations of the CROSS identification protocol, and the computation of the seed- and Merkle trees. Indeed, the seed- and Merkle tree computations involve $\approx 2t$ calls, respectively, to a CSPRNG and a Hash function (both implemented with SHAKE in CROSS), while the elements of the identification protocol (i.e., the instructions of loops at lines 4–9 and 15–18) are iterated t times too. The loop at lines 4–9 is constituted by one matrix-vector multiplication, one vector exponentiation and one vector point wise multiplication, plus, for RSDP(G), only one additional vector-matrix multiplication and one vector subtraction. These computations are complemented by CSPRNG and Hash calls emitting and processing an amount of bits linear in the code size n. In addition to these computations, the ones in the loop body at lines 15–18 are also a vector-exponentiation and a vector addition, where one of the elements is scaled by a multiplicative factor. The CROSS verification algorithm shares with the CROSS signature algorithm its computation profile, as the verification essentially executes only a portion of the signature generation algorithm, given known intermediate values.

As a summary, the computation of all CROSS primitives is largely dominated by vector operations (exponentiation, addition and addition of a rescaled addend), matrix-vector multiplications, and calls to SHAKE used as either a CSPRNG or a Hash. All vector operations are expected to obtain gains by a linear factor in the number of elements computed in parallel, while SHAKE calls involved in the seed- and Merkle trees exhibit some degree of data dependency due to the hierarchy of the trees themselves. However, all the SHAKE calls belonging to the same tree level can also be fruitfully executed in parallel, thus leaving ample margin for parallelization. One case that needs separate attention is the one of the fast-signature optimized versions of CROSS, which do not employ a binary seed- and Merkle tree to reduce the amount of calls to SHAKE as either a CSPRNG or a Hash. In this optimization corner, as designers of the scheme, we opted to swap the full binary tree with a shorter one in which the root has four children, and the four children are the parents of all the leaves. The reason for avoiding a simple substitution of the seed-tree with a single call to a CSPRNG, and the Merkle tree with a (linear) hash lies in the possibility to expose a degree-four parallelization possibility on the computations, which we indeed exploit in the following.

3.2. Scalar arithmetic

CROSS operates over three different finite fields, namely $\mathbb{F}_7$, $\mathbb{F}_{127}$ and $\mathbb{F}_{509}$. In all cases, modular reductions following arithmetic operations on integer values associated with field elements are required; we differentiate the optimization strategy for modular reductions depending on whether the prime modulus p is a Mersenne prime (7, 127) or not (509).

Mersenne primes modular reductions. Both Mersenne primes employed in CROSS allow for a fast modular reduction algorithm, since $p = 2^i - 1$, $i \in \{3, 7\}$. Given this observation, we have that $2^i \bmod p = 1$, and therefore, if we represent a number a, $0 \leq a \leq (p-1)^2$ as two digits in base 2^i, $a = (a_1, a_0)_{2^i}$ it is easy to observe that $(a_1, a_0)_{2^i} \equiv_p a_1 2^i + a_0 \equiv_p (a_1 \cdot 1 + a_0) \bmod p$, yielding a first step for a fast reduction by simply adding the most significant digit to the least significant one and reducing the result. This would already provide a fast strategy, requiring only a right shift (to extract a_1), an addition, and a reduction that can be performed by a single conditional subtraction, by observing that $0 \leq a_1 + a_0 \leq 2(2^i - 1)$ exceeds $p = 2^i - 1$ by at most $2^i - 1$. To reduce the number of times in which such a conditional subtraction is performed, we keep the elements of $\mathbb{F}_p$ in a redundant representation employing one or two bytes (depending on the amount of operations before a reduction) to store them. Considering, for the sake of discussion, the case where $p = 2^7 - 1 = 127$, moving from the one-byte-per-value redundant representation to a non redundant representation takes one final reduction step done through adding the value of the most significant bit of the byte to the number represented by the seven least significant bits, and computing the conditional subtraction on the result. To perform the conditional subtraction with minimal overhead and no timing information leakage, we observe that the first step of the reduction leaves a value in the $\{0, \dots, 2(2^7 - 1)\}$ range (encoded in 1 byte). Since all the values that require a subtraction of $p = 2^7 - 1$ lie in the $\{(2^7 - 1), \dots, 2(2^7 - 1)\}$ range, we observe that, by adding 1 to the number (thus obtaining a value in the $\{1, \dots, 2(2^7) - 1\}$ range), and testing if the seventh most significant bit is set (thus checking if the result is in the $\{(2^7), \dots, 2(2^7) - 1\}$ range) we obtain 1 if and only if the value at hand needs to be reduced. If this is the case, 2^7 should be subtracted from the value at hand; however, observing that $1 \equiv_{127} -2^7$ allows us to add 1 instead of subtracting 2^7 to the value. As a consequence, the final conditional subtraction amounts to adding to the value to be reduced the seventh most significant bit of the integer obtained by adding 1 to the value itself. This in turn costs just two additions and a shift per modular reduction.

As a last point, we observe that computing the opposite of a number modulo a Mersenne prime amounts to toggling all of the bits in its binary representation. Indeed, since a Mersenne prime is in the form $2^i - 1$, no borrows occur whenever a number smaller than the prime itself is subtracted from it to compute its modular opposite.

Optimizing reductions modulo 509. The arithmetic for the field $\mathbb{F}_m$, where $m = 509$ is a prime integer, cannot benefit from the specialized techniques for Mersenne primes. To accelerate the computation in such a case, we adopted a specialized variant of the generic (in the value of the modulus) reduction method introduced by Paul Barrett in [28]. In particular, the Barrett's method allows to compute a modular reduction without the execution of a complete division operation; indeed it achieves this result by approximating the quotient of the division using a precomputed value, a multiplication and a bit-shift operation, which in turn can be realized with highly efficient CPU instructions.

Barrett's method is often presented assuming that the reduction modulo m is applied to a value $x \in \{0, \dots, (m-1)^2\}$, as it is usually applied just after a multiplication between two operands in the range $\{0, \dots, m-1\}$.

Let $r = x \bmod m = (x - q \cdot m) \in \{0, \dots, m-1\}$ be the remainder to be computed, where $q = \left\lfloor \frac{x}{m} \right\rfloor$ is the corresponding integer quotient. Assuming a positional notation of the integer values, it is necessary to specify the radix of the representation and the number of digits needed for operands, results and intermediate computation values. To this end,

let us denote as $w \geq 2$ the bit-length of a CPU word, and as $b = 2^w$ the radix for representing a (multiple precision) integer value composed by a sequence of digits, each of which is stored in a single CPU word. Being $m = 509_{\texttt{decimal}}$, it well fits into a single CPU word with $w \in \{16, 32, 64\}$ bits. As a consequence, in the following we consider m associated to a single digit value in radix b. With the given premises, the precomputed value prescribed by Barrett's reduction method is defined as $\mu = \left\lfloor \frac{b^2}{m} \right\rfloor < 2^{w+1}$, which is an integer encoded with at most 2 digits in radix b (where the most significant digit may be needed to store the value 1 only). Indeed, it is obtained from the integer division of the three digits value b^2 (i.e., $(100)_b = 1 \cdot b^2 + 0 \cdot b + 0$) by the single digit integer m. The approximation of the quotient $q = \left\lfloor \frac{x}{m} \right\rfloor$, denoted as $\widetilde{q}$, is derived as a single digit value from a fraction exhibiting as a numerator $(x \cdot \mu)$ with four digits and a denominator (b^2) with three digits due to the following derivation.

$$q = \left\lfloor \frac{x}{m} \right\rfloor = \left\lfloor \frac{x}{m} \cdot \frac{b^2}{b^2} \right\rfloor \geq \left\lfloor \frac{x \cdot \left\lfloor \frac{b^2}{m} \right\rfloor}{b^2} \right\rfloor = \left\lfloor \frac{x \cdot \mu}{b^2} \right\rfloor = \widetilde{q}, \quad \widetilde{q} \leq q.$$

To understand the degree of approximation introduced with the previous derivation, let us denote as $\beta = b^2 \bmod m$ the remainder related to the computation of μ, and observe that $\widetilde{q} = \left\lfloor \frac{x \cdot \mu}{b^2} \right\rfloor = \left\lfloor \frac{x \cdot (b^2 - \beta)}{b^2 \cdot m} \right\rfloor = \left\lfloor \frac{x}{m} - \frac{x \cdot \beta}{m \cdot b^2} \right\rfloor$.
Remembering that for any pair of integer values $a, b \in \mathbb{Z}$ the sum $\lfloor a + b \rfloor$ is upper bounded ($\leq$) by $\lfloor a \rfloor + \lfloor b \rfloor + 1$, the following inequalities hold: $q = \left\lfloor \frac{x}{m} \right\rfloor \leq \left\lfloor \frac{x}{m} - \frac{x \cdot \beta}{m \cdot b^2} \right\rfloor + \left\lfloor \frac{x \cdot \beta}{m \cdot b^2} \right\rfloor + 1 = \widetilde{q} + \left\lfloor \frac{x \cdot \beta}{m \cdot b^2} \right\rfloor + 1$.
Observing that $\frac{x \cdot \beta}{m \cdot b^2} < \frac{b^2 \cdot m}{m \cdot b^2} = 1$ and $\left\lfloor \frac{x \cdot \beta}{m \cdot b^2} \right\rfloor = 0$, a constraint on q and $\widetilde{q}$ values can be inferred, i.e.: $\widetilde{q} \leq q \leq \widetilde{q} + 1$ and $q - 1 \leq \widetilde{q} \leq q \Leftrightarrow \widetilde{q} \in \{q - 1, q\}$.

Finally, the computation of the (single digit) remainder $r = x - q \cdot m \in \{0, \ldots, m-1\}$ is obtained by computing $\widetilde{r} = (x - \widetilde{q} \cdot m) \in \{r, r + m\}$, followed by the execution of a conditional single subtraction of the modulus value m, in case the condition $\widetilde{r} > m$ is verified to be true.
In order to improve the efficiency of the Barrett modular reduction and avoid the final subtraction, the exact value of the quotient (i.e., $\widetilde{q} = q$) must be derived, while maintaining the efficiency with which the sequence of operations used to compute it is executed. In order to achieve this objective, we take advantage of the specific size of the modulus as well as of the instruction latencies and register size available with an x86 − 64 ISA.
In particular, instead of approximating the quotient $q = \left\lfloor \frac{x}{m} \right\rfloor$, with $0 \leq x < b^2$, $0 \leq m < b$, and $b = 2^w$, as $\widetilde{q} = \left\lfloor \frac{x \cdot \mu}{b^2} \right\rfloor$, with $\mu < 2^{w+1} < b^2$, which implies to handle an intermediate numerator value composed by at most two digits and a denominator value represented with three digits, we explicitly assume to work with a word-size w large enough such that both x and m can be accommodated in a single w-bit word word, i.e., $0 \leq x < 2^w$, $0 \leq m < 2^w$, and replace the terms μ and b^2 with two new parameters $\mu' < 2^{w+1}$ and $\ell \geq w$ (because originally $b^2 = 2^{2w}$), aiming to find the least values for μ' and ℓ such that:

$$\left\lfloor \frac{x \cdot \mu'}{2^\ell} \right\rfloor = \left\lfloor \frac{x}{m} \right\rfloor. \tag{1}$$

Since x and m are in the same range, the equality in Eq. (1) when $x = m$ implies that: $\frac{m \cdot \mu'}{2^\ell} \geq 1$.
Denoting as $\bar{x}$ the largest value of x such that its remainder modulo m equals $m - 1$, it is easy to observe that $\bar{x} \geq m - 1$ and $\max(x) - (m-1) \leq \bar{x} \leq \max(x) \Leftrightarrow 2^w - m \leq \bar{x} \leq 2^w - 1$.
The substitution of $\bar{x}$ in Eq. (1) allows the following derivation: $\left\lfloor \frac{\bar{x} \cdot \mu'}{2^\ell} \right\rfloor = \left\lfloor \frac{\bar{x}}{m} \right\rfloor = \frac{\bar{x} - (m-1)}{m} = \frac{\bar{x}+1}{m} - 1$.
Finally, being $\frac{\bar{x} \cdot \mu'}{2^\ell} - 1 < \left\lfloor \frac{\bar{x} \cdot \mu'}{2^\ell} \right\rfloor$, the following relation is also true: $\frac{\bar{x} \cdot \mu'}{2^\ell} < \frac{\bar{x}+1}{m}$.

Considering simultaneously the inequalities derived from Eq. (1) with no flooring operation and involving μ' it is possible to derive that $\frac{2^\ell}{m} \leq \mu' \leq \frac{2^\ell}{m} \frac{\bar{x}+1}{\bar{x}}$. The lower bound in such a chain of inequalities allows to derive the expression to compute the smallest admissible value for μ' as a function of ℓ and m, while replacing the said expression for μ' in $\frac{\bar{x} \cdot \mu'}{2^\ell} < \frac{\bar{x}+1}{m}$ allows to state a constraint between ℓ and m:

$$\begin{cases} \mu' &= 1 + \dfrac{(2^\ell - 1) - ((2^\ell - 1) \bmod m)}{m}, \\ 2^\ell &> \bar{x} \cdot \left(m - 1 - ((2^\ell - 1) \bmod m)\right). \end{cases} \tag{2}$$

From the standpoint of an implementation targeting the $x86-64$ ISA, considering the specific modulus employed in the CROSS specification $m = 509_{\texttt{decimal}}$ led us to select a word size $w = 32$ (i.e., to use `uint32_t C11` variables) for the representation of intermediate operations in $\mathbb{F}_m$ and to realize the reduction operation on a value x obtained after a multiplication between elements in $\mathbb{F}_m$ or in any case in the range $\{0, \ldots, 2^{32} - 1\}$. In particular, for the computation of the remainder $r = x \bmod m = x - q \cdot m$, the quotient $q = \left\lfloor \frac{x}{m} \right\rfloor = \left(\frac{x \cdot \mu'}{2^\ell} \right)$ was precomputed assuming $\ell = 40$ and $\mu' = 2160140723_{\texttt{decimal}}$ (32-bit value). The former value was obtained as the smallest $\ell \in \{w, \ldots, 2w\}$ that satisfies the second relation in Eq. (2), while the latter was derived from the first relation in Eq. (2).

The computation of the quotient (q) benefits also from extending the intermediate numerator value $(x \cdot \mu')$ to be a 64-bit unsigned integer, which is subsequently fit into a 32-bit word by executing the right shift of $\ell = 40$ bit-positions, as prescribed by the denominator 2^ℓ. In the computation of r, the subsequent multiplication between 32-bits factors $(q \cdot m)$ also implies an intermediate 64-bit result, which is subsequently subtracted from an intermediate copy of the original integer x that is temporarily accommodated in a 64-bit word. Finally, storing the result of the said subtraction in a 32-bit word is guaranteed to handle the correct values of r due to the actual variability ranges of the involved variables. Overall, the sequence of two 32-bit multiplications, one 64-bit right shift and one 64-bit subtraction provides a constant time computation of a modulo operation that is efficiently performed on the $x86-64$ ISA.

3.3. Vector arithmetic

Vector arithmetic in CROSS provides three opportunities for optimization with AVX2 by parallelizing *(i)* vector modular additions (whether the second operand is rescaled or not), *(ii)* vector modular exponentiations (to convert an exponent into its corresponding restricted field element), and *(iii)* vector-matrix multiplications.

***(i)* Vector modular additions.** We perform efficient parallel vector modular additions by encoding elements of $\mathbb{F}_7$ and $\mathbb{F}_{127}$ as 16-bit unsigned values (`epi16` in Intel's parliance) and employing the vector equivalent operations corresponding to addition and Mersenne-style reduction.

In the case of the addition by a rescaled addend, we first broadcast the value of the scalar to a full register by means of a `_mm256_set1_epi16` intrinsic[1], and then value-wise multiply each copy of it employing a vector 16×16 bit multiplication, with extraction of the low 16 bits of the results, done in a single AVX2 instruction when the `_mm256_mullo_epi16` intrinsic is employed.

Vector modular additions on $\mathbb{F}_{509}$ are instead managed by lifting the representations of its elements from vectors of unsigned 16-bit values (`epi16`) onto vectors containing half of the elements, encoded on 32 bits, whenever needed. We perform such lifting in parallel, by a bitmask composed of alternating 16-bit long runs of ones and zeroes, and a single shift to extract the elements in odd positions (counting, as per

[1] a C-style function to use low level instructions without writing assembly code – its implementation is directly handled by the compiler.

Intel's convention from the element in the rightmost position). Once the elements of $\mathbb{F}_{509}$ are represented in 32 bit, we proceed as per the previous description on the realization of reductions modulo 509.

(ii) **Vector modular exponentiations.** The execution of modular exponentiation operations ($a = b^c \bmod m$) is in general expensive, as it involves a *square and multiply* (S&M) approach, which in turn requires a number of modular multiplications and squares linearly proportional to the bit-size of the exponent. Even for small exponents, such as the ones in the case of CROSS, such an approach requires 5 modular scaring and at most 5 modular multiplications for the RSDP case, while the RSDP(G) case would require 6 modular scariness and at most 6 modular multiplications. The square and multiply approach is furthermore affected by a possible timing information leakage, if only the minimum amount of multiplications is performed each time. For these reasons, we chose to adopt a different strategy to perform vector exponentiations. We observe that, in CROSS, the base value b and the modulus m are fixed, while the exponent c is known to be within a certain range $c \in \{c_{\mathtt{min}}, \dots, c_{\mathtt{max}}\}$.

Vector exponentiation of elements in $\mathbb{F}_{127}$ with exponents in $\mathbb{F}_7$. In the RSDP case, when we operate modulo 127, we have that $\langle 2 \rangle \subset \mathbb{F}^*_{127}$, and $|\langle 2 \rangle| = 7$. Therefore, the base value is $b = 2$, while the exponent c can take values in $\{0, \dots, 6\}$. As a consequence, the most efficient strategy to compute modular exponentiations is realized via a full precomputation of $b^c \bmod 127$ values and a table lookup. However, a naive implementation of such a strategy would present a significant vulnerability due to timing information leakage: the address of the element being looked-up in the table reveals the value of the exponent, which is to be kept secret in CROSS. To securely implement such a table-based approach, we observe that tabulating the entire set of possible exponentiations amounts to building a lookup table with 7 entries:

$$T = \{2^0 \bmod 127, \dots, 2^6 \bmod 127\} = \{1, 2, 4, 8, 16, 32, 64\}$$

This lookup table fits, whenever each element is represented as a single byte, in a single 64-bit register, also allowing room to store the value of $2^7 \bmod 127 = 1$ as its topmost byte. This in turn allows the use of the table on a redundant representation of $\mathbb{F}_7$ elements, where the zero-exponent is represented by either 0 or 7, yielding the correct, reduced mod127 exponentiation result. Packing the values of the said table into a single 64-bit constant in C programming language, considering the little-endian ordering of Intel platforms is done as follows:

```
#define T ((uint64_t) (0x0140201008040201))
```

A single, constant time lookup into this table can be performed by loading the whole unsigned integer, shifting it to its right by an amount equal to the exponent, and extracting a single byte of the result via bitmask, making the entire exponentiation process doable by executing one load, one shift and one "bitwise and" operation.

While this approach does not present a straightforward vectorization, through a creative use of the byte shuffling instruction in Intel's AVX2 ISA extension, we are able to achieve it. The bytewise shuffle instruction `vpshufb`, emitted in correspondence to the intrinsic call `_mm256_shuffle_epi8(x, y)`, takes two operands corresponding to two 256-bit registers and considers them as made of two vectors (also referred to as *lanes*) of 16 bytes each. The intrinsic `_mm256_shuffle_epi8` returns a value, representing the 256-bit register containing the result of the `vpshufb` operation: `vpshufb` considers each lane separately and stores in the destination register copies of the values contained in `x`, in positions determined by the value of the corresponding byte in the same position in register `y`. The main purpose of the instruction is thus to apply, for each lane, the permutation, having destination indexes contained in the lane of `y`, to the contents of the corresponding lane of `x`. However, the usefulness of the `vpshufb` for our purposes comes from the fact that the indices in `y` do not need to be unique; therefore the instruction can be repurposed as a parallel lookup of bytes contained in the lane of `x`, according to the indices contained in the lane of `y`. We therefore initialize the two lanes of a 256-bit register with two copies of the aforementioned table, one per lane, placing the values in the positions corresponding to indices in $\{0, \dots, 7\}$ for the `vpshufb` instruction. The duplication of the table is mandated by the fact that `vpshufb` does not perform inter-lane shuffling. This allows us to perform 32 modular exponentiations of the kind required in CROSS-RSDP in a single clock cycle, by filling two lanes of a 256-bit register `c` with 32 exponent values, one per byte, and passing it as the input of the following function

```
__m256i 32_parallel_exp_mod_127(__m256i c) {
    __m256i t = _mm256_setr_epi8(1,2,4,8,16,32,64,1,0,0,0,0,0,0,0,0,
                                 1,2,4,8,16,32,64,1,0,0,0,0,0,0,0,0);
    return _mm256_shuffle_epi8(t, c);
}
```

We note that this approach also outperforms the one which exploits the fact that the generator of the restricted subgroup of $\mathbb{F}_{127}$ was chosen to be $b = 2$ so that its powers are actually power of two, and can be directly obtained by taking the value 2 and shifting it to the left by an amount equivalent to the exponent. However, vectorizing this approach yields a less efficient realization on Intel's AVX2 as vector-shift operations only act on vectors of `epi16` elements, thus only allowing to perform 16 shifts in parallel.

Vector exponentiation of elements in $\mathbb{F}_{509}$ with exponents in $\mathbb{F}_{127}$. Vectorizing exponentiations in $\mathbb{F}_{509}$ for CROSS requires an approach different from the one previously described, as the size of the restricted multiplicative subgroup, and therefore the amount of possible exponent values is 127. Specifically, $\langle 16 \rangle \subset \mathbb{F}^*_{509}$, and $|\langle 16 \rangle| = 127$. A straightforward tabulation would either place a significant pressure on the register file, or, resort to memory, opening up to timing side channel attacks.

We adopt an approach trading off computation and table size, inspired by a *windowed square and multiply (S&M)* approach, and proceeds as follows. To compute $a = b^c \bmod m$ using the classical S&M, we would scan (from right to left) the bits of the binary encoding of the exponent $c = (c_6c_5c_4c_3c_2c_1c_0)_{\mathtt{bin}}$, and multiply an accumulator by 1 if $c_i = 0$, or by b^{2^i} if $c_i = 1$, to avoid timing side channel leakage. This amounts to considering the exponentiation decomposed as follows:

$$b^c \bmod m = b^{\left(\sum_{i=0}^{6}(c_i 2^i)\right)} = \left(\prod_{i=0}^{6}\left(b^{2^i}\right)^{c_i}\right) \bmod m.$$

With this method the cost of one exponentiation is 6 multiplications and 3 modular reductions, assuming to use the method described in the previous section which allows to reduce any unsigned 32-bit integer modulo 509.

As depicted in Fig. 2, our computation-memory trade-off approach involves logically partitioning the seven bits of the exponent $0 \le c < 127$ into two portions, constituted by the low-4 bits $(c_3c_2c_1c_0)_{\mathtt{bin}}$ and the high-3 $(c_6c_5c_4)_{\mathtt{bin}}$ bits, in turn considering the value $a = b^c \bmod m$ as:

$$a = b^c \bmod m = \left(b^{(c_6c_5c_4)_{\mathtt{bin}}2^4} \cdot b^{(c_3c_2c_1c_0)_{\mathtt{bin}}}\right) \bmod m.$$

We now observe that tabulating all the values of $b^{(c_3c_2c_1c_0)_{\mathtt{bin}}} \bmod m$ requires a 16 elements look-up table, while tabulating all $b^{(c_6c_5c_4)_{\mathtt{bin}}2^4} \bmod m$ requires an 8 element lookup table. This in turn allows us to adapt the approach employed for the modular exponentiation of elements in $\mathbb{F}_{127}$. We build two lookup tables containing elements of $\mathbb{F}_{509}$: table `l4` (depicted in the bottom (right) part of Fig. 2 in correspondence with the 4-bit exponent values in yellow) with 16 elements, and table `h3` (depicted in the top (right) part of Fig. 2 in correspondence with

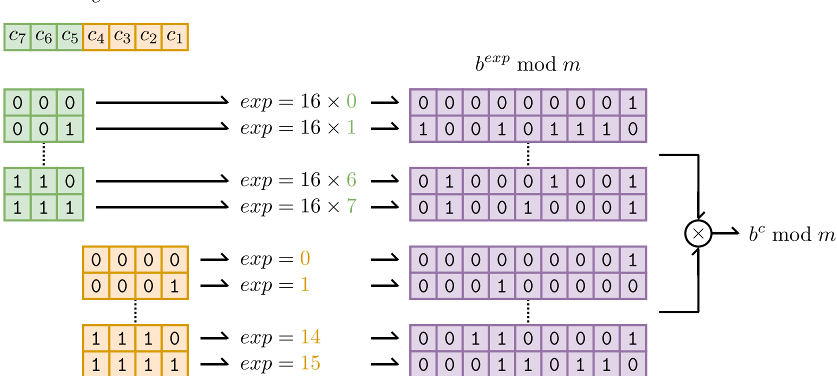


Fig. 2. Modular exponentiation using a single multiplication. Relies on two sets of **precomputed constants**, one for the **high-3 bits** of the exponent c and one for the **low-4 bits**. At runtime use bits $c_7c_6c_5$ to select the first constant, and bits $c_4c_3c_2c_1$ to select the second, then multiply them together and reduce. The result is 16^c mod 509..

the 3-bit exponent values in green), with 8 elements. To perform a full modular exponentiation, we simply multiply together the results of two look-ups, one in `l4`, one in `h3`, each done with an index corresponding to the natural binary interpretation of the appropriate portion of exponent bits, and then perform a single modular reduction. We therefore perform a single modular exponentiation at the cost of two look-ups, one multiplication and one reduction, when done sequentially.

Vectorizing this approach requires to tackle three remaining challenges. The first one is the fact that each of the 16 values in the `l4` table are $\lceil \log_2(509) \rceil = 9$ bits wide; as a consequence, the 16 values in the `l4` do not fit into a single lane of a 256-bit AVX2 register (indeed, they require two). While partitioning the exponent into a different split may allow all tables to fit, we observe that an alternative approach allows to retain almost the same computation efficiency as the one of two tables, without adding further pressure on the register file due to additional tables being built. We solve this challenge splitting the elements of `l4` in two sub-tables, one applicable when bit c_3 is zero (i.e., $(c_3c_2c_1c_0)_{\texttt{bin}} < 8_{\texttt{dec}}$) and the other when $c_3 = 1$. We perform the lookup from the correct table by means of a constant-time conditional move (implemented by means of the Boolean formula of a 2-to-1 multiplexer), which is fully parallel.

The second challenge comes from the fact that the `_mm256_shuffle_epi8` intrinsic performs byte-wise look-ups, while our values in `l4` and `h3` are encoded in two bytes each. This challenge can be solved observing that a bytewise lookup instruction can be used to perform a pairs-of-bytes lookup, generating an appropriate sequence of lookup indexes. In particular, consider a sequence of lookup indexes addressing 16-bit portions of a 256-bit register, i.e., having values between 0 and 15, with 0 indexing the leftmost pair of bytes and 15 the rightmost pair. Starting from the 16-bit-oriented lookup index value, say, $i \in \{0, \dots, 15\}$, it is straightforward to observe that the actual pair of indices pointing to the two bytes in the register constituting it (counting bytes from left to right) are $2i$ and $2i + 1$. We therefore transform a vector of exponent portions to be used as indexes for 16-bit values, into the appropriate pairs of byte-wise indices in four instructions, acting as follows. We first prepare the starting indexes, encoding 16 of them, each one into 16 bits of a 256-bit register. We then compute their double in parallel, adding the contents of the register to itself, placing the result in a separate 256-bit vector register. We interleave them byte-wise with their duplicates by means of a `vpshufb` instruction, obtaining a 256-bit register which is filled with pairs of equal indices in each pair of odd- even- positioned bytes. Finally, we add the original register contents to the result of the `vpshufb`. This indeed yields a final result which is a 256-bit register where the values are pairwise $2i$ and $2i + 1$, given that i was a 16-bit encoded value in the original register. The cost of such a duplication operation is 4 cycles, as all the involved operations are single-cycle.

The final challenge comes from the fact that the multiplication combining the two results from the table look-ups yields an intermediate result that requires more than 16 bits to be encoded. Our approach to the solution of this problem is to employ a temporary 256-bit register and multiply half of the 16-bit elements obtained from the two look-ups at once, reducing the result, and packing it back into 16-bit encoded numbers after the modular reduction.

The complete function performing vector-exponentiation mod 509 is shown in the Appendix.

***(iii)* Vector-matrix multiplication.** Parallelizing vector-matrix multiplications requires a combination of two factors: choosing the most efficient data-parallel operation to be spread across vector units, and employ a memory layout allowing coalesced load and store operations.

To analyze which data-parallel operation is best distributed across vector units, we consider the multiplication of a vector $\mathbf{e} \in \mathbb{F}_p^n$ by a matrix $\mathbf{M} \in \mathbb{F}_p^{n\times m}$, where $\mathbf{M} = [\mathbf{V} \mid \mathbf{I}]$ is made of an arbitrarily valued $(n - m) \times m$ block $\mathbf{V}$ and an m-sized identity matrix $\mathbf{I}$. Note that this computation pattern fits both the syndrome computations (e.g., $\mathbf{s}'[i] \leftarrow \mathbf{u}'[i]\mathbf{H}$ at line 8 of the CROSS signature generation algorithm), and systematic codeword encoding required in the RSDP(G) variant (e.g., $\mathbf{e}'[i] \leftarrow \mathbf{e}'_G[i]\mathbf{M}$ at line 5 in CROSS signature generation). We will, in first instance, assume that reductions are performed after each modular operation, and consider separately their cost, as they are independent of the parallelization strategy. Note that, due to the form of $\mathbf{M}$, the multiplication of the m-elements long portion of $\mathbf{e}$ by the identity submatrix will yield the portion of $\mathbf{e}$ itself, which can thus be used to initialize the result $\mathbf{s}$, in which, the results of the other operations are addition-accumulated. A vector-matrix multiplication is essentially a double nested loop, iterating on the rows and columns of $\mathbf{V}$, and adding the result of a scalar multiplication into an element of the result. Sweeping over the rows of $\mathbf{V}$ with the inner loop, and over its columns with the outer loop, amounts to computing a sequence of m inner products between (the first $n - m$ elements of) $\mathbf{e}$ and a column of $\mathbf{V}$. Due to the commutativity of the addition operation employed in accumulating the scalar results, it is possible, in line of principle, to parallelize either the outer or the inner loop.

Parallelizing the inner loop leads to computing the scalar multiplication whose results constitute the addends in a scalar product in parallel. This approach can be implemented exploiting Intel's `vphaddw` instruction that adds all the `epi16` elements contained in a 256-bit register, together with a latency of 2 to 3 cycles. Performing a full scalar product will thus take $\left\lceil \frac{n-m}{16} \right\rceil$ vector multiplications and `vphaddw` operations, plus a final $\left\lceil \frac{n-m}{16} \right\rceil$ additions to combine together the partial results from the scalar product via addition. Note that parallelizing these last additions is challenging, as the elements to be added lie, by construction, in different registers, as they are the result of different `vphaddw`

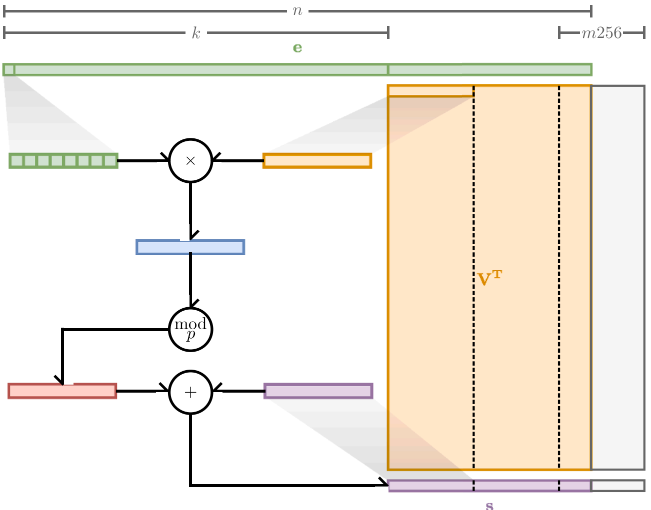


Fig. 3. Fast vector by matrix multiplication parallelized with Intel AVX2. The syndrome $\mathbf{s} \in \mathbb{F}_p^{n-k}$ is the product of the error vector $\mathbf{e} \in \mathbb{F}_p^n$ and the transposed parity check matrix $\mathbf{H}^T$, of which only the non-identity portion ($\mathbf{V}^T \in \mathbb{F}_p^{k\times(n-k)}$) is materialized. An element of $\mathbf{e}$ is first broadcast into a `m256` register, then SIMD-multiplied with the corresponding section of $\mathbf{V}^T$. The **result** gets reduced modulo p and accumulated into the **syndrome**. .

instructions, and packing them together into a single register to parallelize the sum with `vphaddw` results in a slower implementation (shift-and-or instruction pairs being slower than scalar add instructions). This results in a total latency of $\left\lceil \frac{n-m}{16} \right\rceil (c_{\texttt{vmul}} + c_{\texttt{add}} + c_{\texttt{vphaddw}})$, where $c_{\texttt{vmul}}, c_{\texttt{add}}$, and $c_{\texttt{vphaddw}}$ are the latencies in clock cycles of vector multiplications, additions and `vphaddw`s, respectively.
Choosing to parallelize the outer loop amounts instead to computing multiple scalar products in parallel, and to accumulate their results in a vector that will end up constituting a portion of the final result $\mathbf{s}$. A graphical representation of this approach is reported in Fig. 3. In this case, a vector full of elements from $\mathbf{V}^T$ is multiplied by a vector register filled with the replicas of a single element of $\mathbf{e}$, and accumulated into a portion of $\mathbf{s}$. The procedure costs only three vector operations (a multiplication, an addition, and a scalar broadcast, all single cycle operations), and, once repeated $n - m$ times performs one vectorful of scalar products without the need for further adjustments. This in turn allows to perform the entire vector-matrix multiplication in $\left\lceil \frac{m}{16} \right\rceil (c_{\texttt{vmul}} + c_{\texttt{add}} + c_{\texttt{broadcast}})$ immediately gaining on the alternative from the replacement of the `vphaddw` with a faster operation. We therefore select this second strategy as the one of choice for vector-matrix multiplications.

Concerning modular reductions, we choose to perform them after each accumulation of a product between scalars when operating with elements of $\mathbb{F}_{127}$. This allows us to encode all the intermediate results in `epi16` values, thus performing 16 operations in parallel. When operating with elements of $\mathbb{F}_{509}$, even a single multiplication result does not fit into an `epi16` value, therefore we exploit the additional room provided by the encoding in 32-bit values to delay the reductions when accumulating values in a matrix multiplication. This allows to perform as few as three reduction operations in an entire vector-matrix multiplication without exceeding the allotted range.

Having fully chosen the parallelization structure, we now consider the memory layout. The key point in doing so is to execute the `load` operations employed to fetch vectors of elements processed in parallel, to minimize the amount of cache misses, and to allow for the (faster) vector-register-size aligned `load` operations. To this end, we layout the $\mathbf{V}^T$ matrix in memory by rows, making it so that the prefetcher is able to infer the highly regular access pattern. Furthermore we add a padding

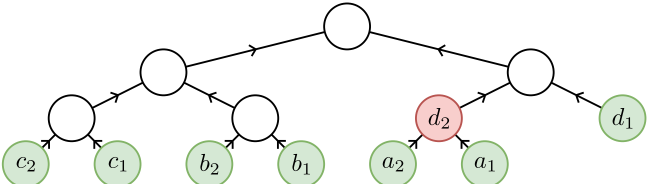


Fig. 4. Data independence violation when using 4-way Keccak on a Merkle tree. Normally, two nodes are concatenated and hashed to produce their parent digest in the tree. When the hash calls are parallelized, then 2 pairs are hashed at the same time (e.g., c, b, or a). If a batch span across two tree levels, a node, like d_2, might be both an input and an output. .

to each row so that its length is a multiple of 256 bits, which in turn allows the use of aligned vector loads. The padding zones are depicted in grey in Fig. 3, where we report the size of a 256-bit register (m256 marker, top right), for scale.

3.4. Parallelizing SHAKE computations

CROSS relies on SHAKE in a significant fashion, as the algorithm is employed to provide both a CSPRNG and a hash function functionality. The amount of SHAKE calls in a CROSS execution is in the thousands range, therefore its optimization has a significant potential in terms of speeding the algorithm up.

SHAKE is an extendable-output function, i.e., a one-way, collision-resistant function whose output can be extended to any length. SHAKE is standardized in the same publication where SHA-3 is included [18], and has also two variants, SHAKE-128 and SHAKE-256. They provide an output digest of 256 and 512 bits, respectively, and ensure a security margin against collision attacks equivalent to the one of bruteforcing AES-128 and AES-256, respectively. CROSS employs SHAKE instead of SHA-3 as a hash, due to its higher throughput in hashing relatively long input messages (kilobits range), and employs it as a CSPRNG, as it outperforms the AES-CTR alternative on long (kilobits range) outputs. The core component of SHAKE is a Boolean bijective function, known as Keccak-f, acting on a set of 1600 bits, known as the *state*. In the same fashion as per the design of SHA-3, the execution of SHAKE has two phases: in the first phase input bits are *absorbed* into the state, by adding a portion of them (1344 bits for SHAKE-128, 1088 bits for SHAKE-256) via `xor` to the state, and applying the Keccak-f to it. The process is repeated until no more input is available. The second phase of the computation *squeezes* the output out of the state by simply copying a quantity of bits from the state in output: the copied bits are in the same amount as the ones absorbed in a step during the absorption phase. The absorption and squeezing phase are strictly sequential, and data dependencies prevent from parallelizing them. The inner structure of the Keccak-f operates on a $5 \times 5 \times 64$ bit matrix, applying for 24 times a set of simple transformations, known as *round*. The function therefore offers a maximum parallelism, which is represented by the possibility of working on 64-bit words of the state at once.
While this inner parallelism is significant, it is not enough to fully profit from the large vector units available in the AVX2 ISA. Our approach is therefore to compute in parallel four instances of SHAKE, packing into a 256-bit AVX2 register four 64-bit words, each one coming from a different SHAKE execution. Any time we have many calls to SHAKE performed on different inputs, the vector implementation allows to benefit from a performance improvement: instead of performing the calls one after the other, we can batch them and process multiple inputs at the same time. To do so, however, two requirements have to be satisfied: SHAKE calls should have equal input and output lengths. These are necessary conditions since the multiple SHAKE instances will be computed in step-lock, and the inputs to each call of the algorithm should not be data-dependent on the output of another instance of the algorithm.

Parallelizing computations of Seed- and Merkle Trees. Parallelizing SHAKE computation, whether they are involved in a seed- or in a Merkle tree, requires to cope with the data dependencies induced by the computation of a tree structured sequence of CSPRNGs or hash functions.

We start by considering the parallelization of a Merkle tree computation, and provide a graphical depiction of an example in Fig. 4. The sequential computation associated with a Merkle tree typically starts by hashing equally sized portions of the message to obtain the digests bound to the leaves of the tree (see green nodes in Fig. 4). Such a sequential pass is followed by another one hashing the digests obtained from the hash computations bound to the leaves, to obtain the digests bound to the parent nodes, and so on towards the root. Four-way parallelizing the computations of distinct SHAKE instances, which are done from left to right, would yield a situation where c_2, c_1, b_2 and b_1 labeled hashes are computed in parallel, followed by a_2, a_1, d_1 and d_2. However, the fact that the nodes in the same layer are not a multiple of four, leads to hash computations belonging to nodes in different layers to be computed in parallel. This, in turn, may lead to data dependency violations, such as the case of d_2, which would be computed in parallel with the hashes a_2, a_1 that are meant to return its inputs, making the parallel computation impossible. To solve this problem, we realize a queue of nodes, where all the elements in queue have their data dependency already cleared by previous computations. We keep a constant size, four elements queue, and whenever it is full, trigger a fourfold computation of SHAKE, updating the queue. If no more computations can be added to the queue due to data dependencies, and the queue is not empty, the computations inside it are completed to avoid locking. Note that this situation is expected only towards the root of the tree, where, structurally, there may be not enough SHAKE computations left to be done (e.g., when the root is computed, only a single SHAKE needs to be run).

An analogous approach is adopted to parallelize SHAKE calls in seed-trees. The main distinction is that the computations start sequentially (as the root CSPRNG must be run before its children are), but, after two layers of the tree, at least four nodes of the seed-tree can be computed in parallel, fully exploiting our quadruple parallel SHAKE approach. This approach also allows us to fully exploit our design choice for the fast variants of CROSS. Indeed, the described approach is independent from the specific tree structure. This allows us to employ it with the quasi-flat tree from CROSS-[RSDP,RSDP(G)]-[1, 3, 5]-fast, which is characterized by a root node having four children, and subsequently, each of the four children generates a quarter of the pseudorandom material needed. In our approach, we simply run the root CSPRNG sequentially, and then, the four children, which are generating the bulk of the pseudorandom bit strings, are just run in parallel.

3.5. Timing analysis

To verify that our implementation of CROSS is free of secret-dependent control flows, we used a source-aware analysis. This approach was first proposed by [29] and consists of running sensitive operations (key generation and signing) on random inputs, but declaring this randomness as uninitialized memory. Valgrind (a popular suite of tools for debugging and profiling memory [30]), can then be used to find branches or memory accesses that depend on the uninitialized data (corresponding to secret values), with the `memcheck` tool. Two popular libraries (`liboqs` [31] and `SUPERCOP` [32]), use this same approach to verify that the included algorithms are free of timing information leakage. Incidentally, the latest release of CROSS (version 2.2, which incorporates all the AVX2 optimizations discussed in Section 3) is included in both libraries, allowing for public verification of its constant-time properties.

The analysis shows a few instances of suspected variable-time behavior in CROSS, none of which depend on the optimizations we discussed. These behaviors are all related to rejection sampling, which CROSS uses to obtain vectors and matrices with the necessary properties (belonging to a finite field or having constant weight) from the output of SHAKE.

Table 1
Speed comparison between the AVX2-optimized primitives described in Section 3 and their reference implementation.

Primitive measured	Parameters	Clock Cycles		Speedup
		Ref.	AVX2	
$2^c \bmod 127$ with $c \in [1,7]$	251 $\mathbb{F}_{127}$ elements	531	35	15.17×
$16^c \bmod 509$ with $c \in [1,127]$	251 $\mathbb{F}_{509}$ elements	6 497	473	13.74×
Vector times matrix (mod 127)	$n = 251, k = 150$	11, 319	2, 665	4.25×
Vector times matrix (mod 509)	$n = 106, k = 69$	5, 000	867	5.77×
Compute Merkle tree	832 leaves	1, 691 976	570, 528	2.97×
Compute seed tree	832 leaves	1, 528 030	527, 756	2.90×

When the sampling procedure draws an element which does not satisfy the desired property, it discards the corresponding bits of randomness and samples again. While this does produce a branch that depends on a secret input, we consider it safe, as an attacker would only learn how many samples were rejected and when. This type of analysis enables a more fine-grained approach compared to source-agnostic tools like `dudect` [33], as it allows pinpointing exactly which instruction causes a timing variation, and to identify false positives accurately.

4. Experimental results

In this section, we report the experimental evaluation on the performance improvements of our AVX2 optimized realization of CROSS, and we report full TLS handshake performance figures.

4.1. AVX2 benchmarks

Our experimental workbench is a system equipped with an Intel i3 − 8350K, clocked at 4 GHz, equipped with 32 GiB of DDR4 − 2400, running Debian 12 with `gcc` 12.2.0. Clock cycles (CCs) measurements were performed with `libcpucycles` [34] version 2024.03.18, a public domain library exploiting the hardware performance counters present in the CPU to obtain accurate clock cycle counts. To reduce the effect of system-wide disturbances such as context changes, all the obtained figures are obtained as the average over 10, 000 runs. The source code for these experiments is publicly available on `GitHub` [35].

We start by providing the results of microbenchmarks aimed at evaluating the speedups obtained by our vector exponentiations, parallelized vector-matrix multiplications and parallel SHAKE computations (described in Sections 3.2–3.4, respectively). Table 1 reports the results of the microbenchmarks, comparing them against the performance of the reference `C` implementation. For the sake of highlighting the results, given the very low latency of single operations, we chose to report the results on the parameters used by the most computationally demanding CROSS variant, i.e., CROSS-RSDP-5-small. Vector modular exponentiation attains performance gains close to the maximum theoretical speedup, i.e., 16×, which is obtained considering that, employing the 256-bit wide registers in AVX2, we are able to process 16 elements at once. Vector-matrix multiplications obtain lower speedups with respect to what could be expected in a comparison against a naive, sequential vector-matrix multiplication, namely, between 4.25× and 5.77×. We investigated the reason for this result, and, disassembling the binary emitted by `gcc`, we observed that the compiler was able to infer a (partial) vectorization strategy, in turn obtaining some gains from emitting (partially) parallel code. Finally, we report gains in the 2.90× to 2.97× range for the parallel execution of four instances of SHAKE. We ascribe the larger distance from the infinite-register linear scaling (which would yield a 4× speedup) for SHAKE to the increased register pressure in the quadruple-parallelized implementation. Indeed, the four states of the four SHAKE executions take up to 25 architectural registers (due to the 5×5 lane structure of SHAKE), leaving a significantly reduced elbow room for the register allocator.

Table 2
Speedups for the AVX2-optimized implementations of CROSS vs. CROSS reference implementation.

CROSS Variant			Reference (kCC)			AVX2-Opt. (kCC)			Speedup (times)		
			kg	sign	verif	kg	sign	verif	kg	sign	verif
RSDP	1	**fast**	80	2,697	1,564	75	1,873	1,184	1.07	1.44	1.32
		balanced	85	5,189	3,041	73	3,066	2,216	1.16	1.69	1.37
		small	79	10,053	5,856	73	6,210	4,614	1.09	1.62	1.27
	3	**fast**	196	7,657	4,415	183	4,182	2,642	1.07	1.83	1.67
		balanced	197	13,179	6,612	169	6,989	4,859	1.16	1.89	1.36
		small	202	19,654	10,084	183	10,281	7,504	1.10	1.91	1.34
	5	**fast**	349	12,334	7,159	294	7,411	4,832	1.19	1.66	1.48
		balanced	348	20,366	11,157	247	12,139	8,184	1.40	1.68	1.36
		small	323	33,097	17,762	248	19,483	13,783	1.30	1.70	1.29
RSDP(G)	1	**fast**	41	1,993	1,349	41	1,035	700	1.01	1.93	1.93
		balanced	45	4,323	2,732	43	2,015	1,436	1.06	2.15	1.90
		small	41	8,123	5,145	41	3,913	2,896	1.00	2.08	1.78
	3	**fast**	81	4,315	2,792	83	2,488	1,655	0.98	1.73	1.69
		balanced	81	5,821	3,609	82	3,021	2,195	0.98	1.93	1.64
		small	86	11,366	7,001	84	5,672	4,289	1.03	2.00	1.63
	5	**fast**	155	8,186	5,313	130	4,168	2,816	1.19	1.96	1.89
		balanced	143	10,504	6,477	129	5,070	3,524	1.11	2.07	1.84
		small	139	18,552	11,397	137	8,989	6,354	1.01	2.06	1.79

Table 2 reports the speedups obtained in our optimized implementation of CROSS, with respect to the reference code provided in the NIST submission package [11]. The reported latencies show that CROSS-[RSDP,RSDP(G)]-1-[fast,balanced] consistently achieves sub millisecond signature and verification latencies, with the fastest option (CROSS-RSDP(G)-1-fast) computing a signature in $258\,\mu$s and verifying it in $175\,\mu$s. Signing with the optimized implementation is, on average, 1.85 times faster, while verification also improves considerably, by 1.59 times on average. Key(pair) generation shows more limited gains, due to the fact that its computation is dominated by the (sequential) calls to SHAKE as a CSPRNG to expand the public matrix (or matrices for RSDP(G)), therefore partly overshadowing the gains from a parallelized vector-matrix multiplication. Nonetheless, besides being $\approx 1.1\times$ faster than the reference code on average, it also completes in less than $80\,\mu$s for all parameter sets, with the fastest one (CROSS-RSDP(G)-1-fast) finishing in $\approx 10\,\mu$s.

Finally, Fig. 5 compares the signing and verification times of CROSS with those of other signature schemes. The measurements were obtained on the same Debian 12 system, using the benchmarking framework provided by the Open Quantum Safe (OQS) project [36], in particular `liboqs` [31] (0.13.0 commit `b02d0c9`). OQS hosts both reference and optimized implementations for several standardized and on-ramp candidates from the NIST competition. It also enforces the use of common implementations of Keccak and other cryptographic primitives, ensuring a fair comparison. For each signature scheme, we used its AVX2-optimized implementation, continuously executed signing and verification operations for 3 seconds, and recorded the mean number of CPU cycles. To keep the plot readable, we report results for the first and second NIST security categories, while the complete dataset is available on GitHub [35]. All CROSS variants achieve faster signing than SPHINCS$^+$, while maintaining competitive verification performance. We note that the other depicted candidates all rely on the hardness of different computationally hard problems (either solving multivariate quadratic equation sets or, problems from discrete lattices). While lattice based signatures are also characterized by very compact public key-signature pairs, multivariate quadratic equation set signatures have small signatures at the cost of significantly large public keys (indeed, the public key is an instance of the hard-to-solve simultaneous equations set), which make their choice a trade-off between public key size and signature speed. Furthermore, we note that UOV-Ip and MAYO-2 are affected by a recent attack [37] which will likely require them to change their parameters, and thus, performance profile. Additional insight into how these results translate to real-world performance is provided in the next section, which presents our TLS experiment.

4.2. TLS benchmark

We now provide the results of the evaluation of CROSS, and a comparison with other signature schemes in a TLS handshake, building on top of the tools provided by the Open Quantum Safe (OQS) project [36]. OQS collects and tests quantum-safe algorithms in its main library, `liboqs` [31], and it also makes them available for use in TLS, thanks to its OQS-provider [38] project. We first describe our experimental benchmark choices to provide reproducible and reliable results, and follow up by providing the obtained figures of merit.

Experimental workbench. OpenSSL, the most widespread implementation of TLS, starting with version 3.0 introduced the concept of a provider: a modular component that supplies specific implementations of cryptographic algorithms and operations. These include tasks like encryption, decryption, hashing, digital signatures, and key exchange mechanisms; using a provider these can all be added or replaced without modifying the core library. Thanks to OQS-provider [38] we can use the post-quantum algorithms available in `liboqs` [31] to create X.509 certificates, establish TLS connections, and even mix them with classical algorithms, producing hybrid key exchanges and signatures. In particular, `liboqs` aims at hosting all the NIST competition standardized choices, such as ML-KEM and HQC (which are key encapsulation methods), and ML-DSA (formerly CRYSTALS-Dilithium) and SLH-DSA (formerly SPHINCS$^+$), and host other algorithms for experimentation. Among the latter, submissions from the current participants to the additional call for signatures by NIST are welcome, and currently, the teams of MAYO (Multivariate And Yet Other) [16], UOV (Unbalanced Oil and Vinegar) [39], and SNOVA (Simple Non commutative-ring based UOV with key-randomness Alignment) [40] have contributed their implementation. All three schemes rely on the computational hardness of solving multivariate quadratic simultaneous equation sets. At the moment, UOV and the NIST Category 2 parameters for MAYO do not take into account the recent attack from [37]. It is therefore expected for them to obtain worse performances than the ones in our current experiment once the parameters are updated to compensate from the effects of the attack.

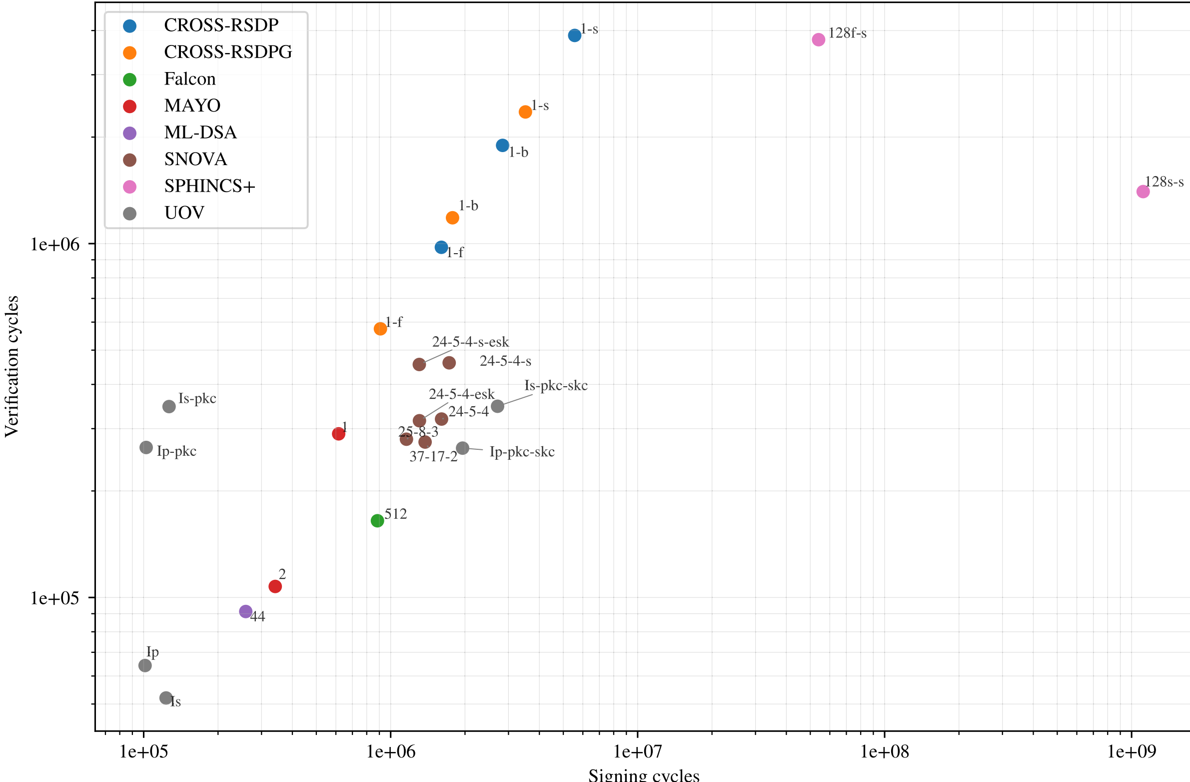


Fig. 5. Comparison of signing and verification cycles for the AVX2-optimized implementations of signature schemes available in the Open Quantum Safe project. Only NIST Categories 1 and 2 are shown. .

We structure our experiments to analyze the impact of post-quantum signature schemes on the TLS 1.3 handshake considering different network scenarios. In particular, we shape the latency and throughput of the network communication by means of the Linux Traffic Control suite (driven via the `tc` command-line utility), an established practice when performing network simulations [41]. We consider four scenarios, obtained by adding a Gaussian distributed latency to the outgoing queue of the packets of both the server and client endpoints in our setup: a first scenario is a local loopback connection with no added latency, the second one considers peer-to-peer traffic in a large LAN or small city, with an added delay of 2 ms; the third scenario considers 20 ms of added latency to simulate traffic between hosts in the same country, while our fourth scenario adds 120 ms to simulate intercontinental traffic.

Our experiment involves four participants: Client, Server, Intermediate Certification Authority (CA), and Root CA. The Root CA has self-signed a X.509 certificate, which is already stored in Client's Trust Store. The Intermediate CA owns another certificate, issued by the Root CA. Finally, the Server has its own certificate, signed by the Intermediate CA, which links the Server's public key to its identity. The result is a certificate chain (Root→Intermediate→Server) such as the one which has now become ubiquitous thanks to the widespread use of Let's Encrypt as a Root CA [42]. This chain will be used by the Client to authenticate the Server. Fig. 6 shows how the handshake involves three signature generations: one chosen by the Root CA, one by the Intermediate CA, and one by the Server. Two public keys and two signatures are transmitted on the wire as part of the certificate chain, and the Server also sends an additional signature to authenticate a hashed transcript of the conversation (this is the `CertificateVerify` message). One signing operation (Server-side) and three verifications (Client-side) are performed. Note that the figure omits the two "Change Cipher Spec" messages, as they exist only for compatibility with earlier versions of TLS. Also, the X.509 certificates are shown as a simple *pk-sig* pair, while in reality they contain extensive data about validity, issuer, and subject.

In our experiment we first generate the certificate chain, setting all three signature schemes to the same choice. Then the measurement is carried out as follows: the Client requests a TLS connection to the Server, waits for the Server to respond and, when the handshake is complete, it requests a new connection. This is repeated for 10 seconds, where the Client tries to handshake the server as many times as possible. The statistics are collected using the `openssl s_time` utility, also a well established method to proceed ([43], [44], [45]). In order to clarify how the latency and throughput figures are computed, consider the following sample output from one of the measurements:

```
Collecting connection statistics for 10 seconds
*****************************************************
53 connections in 0.38s; 139.47 connections/user sec, bytes read 0
53 connections in 11 real seconds, 0 bytes read per connection
```

The "`11 real seconds`" represent the wall clock time, the actual time it took to perform the test from start to finish, rounded to the nearest integer. On the other hand, `0.38s` is the user time (measured on the client side), which excludes for example the time it takes to transfer data over the network and the time spent by the server to process the requests. In total a sequence of 53 handshakes were performed sequentially by the client (this is the number we report in Fig. 7), providing a measure of the number of sustained TLS connection setups which is independent from the number of available CPUs on the ma-

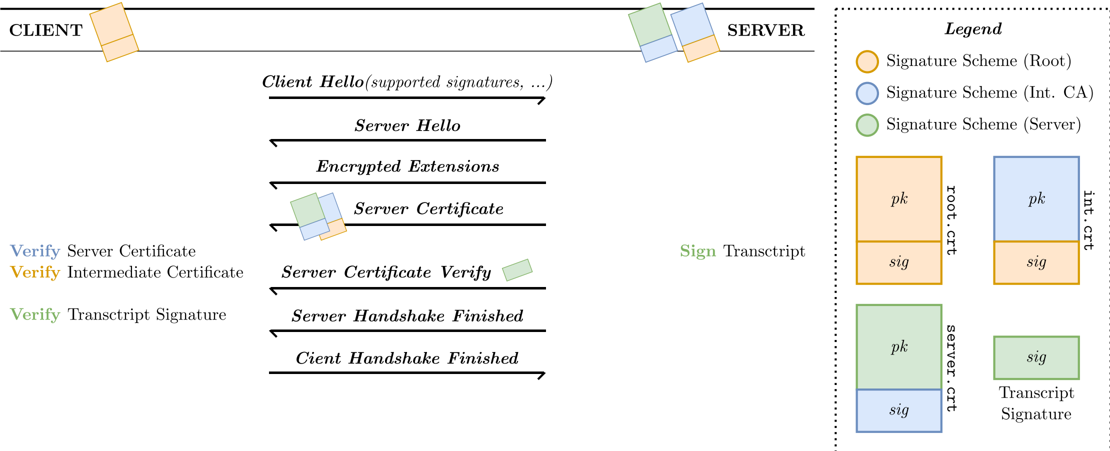


Fig. 6. Use of signature schemes in a typical TLS 1.3 handshake. The **first** is used by the Root CA to self-sign its certificate and, later, to sign the intermediate certificate. The **second** is used by the Intermediate CA to sign the server certificate. The **third** is used by the Server to sign a hashed transcript of the previous messages during the handshake. All three schemes are also used by the Client to verify the signatures it received..

Fig. 7. Number of connections to a Server within 10 seconds, where the Server's certificate chain is generated using various classical and post-quantum signature schemes. Four network latency conditions are simulated using Traffic Control on an **Intel i3-8350K**. The measurements are normalized by the F variant of SPHINCS⁺.

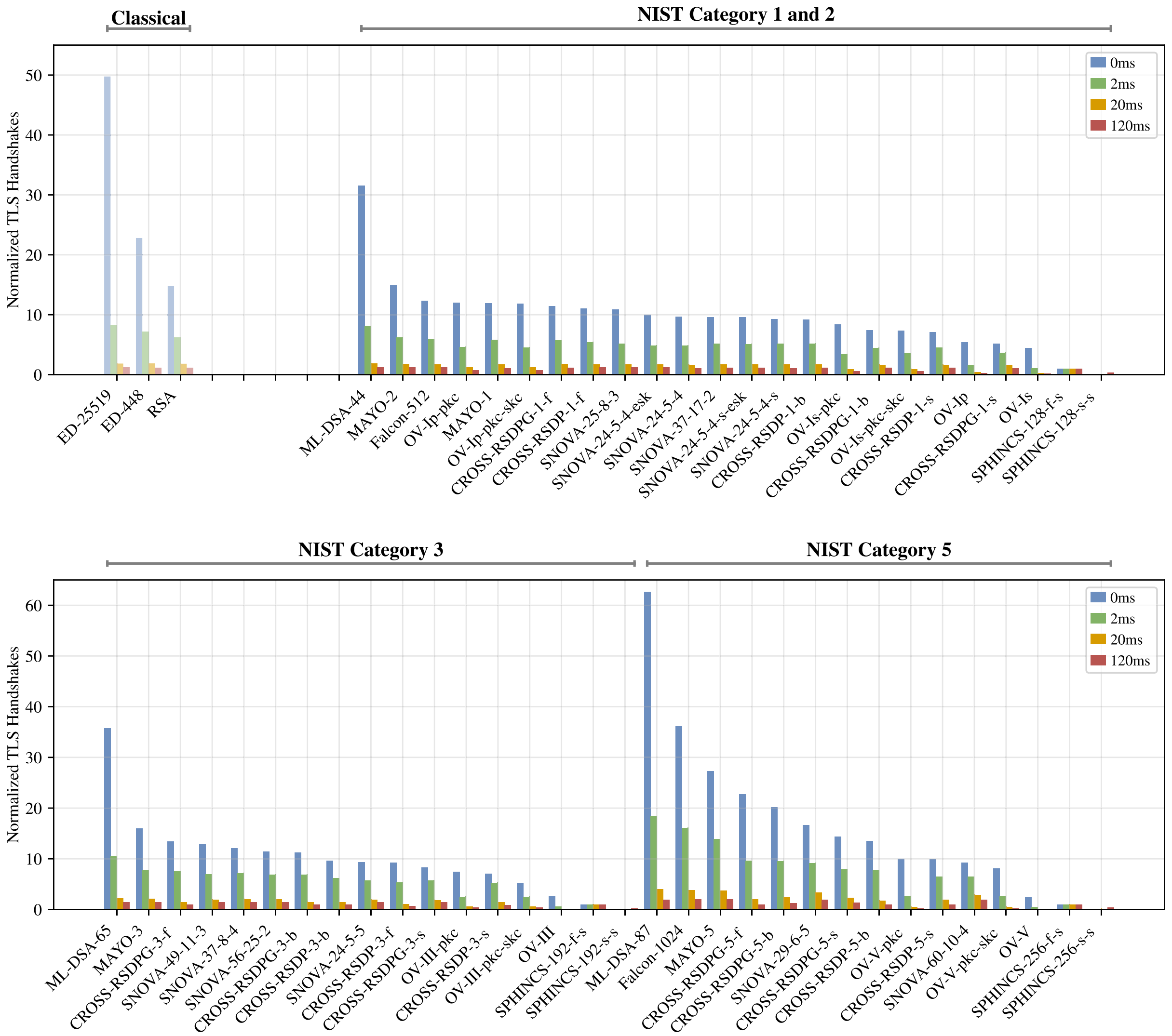


Fig. 8. Number of connections to a Server within 10 seconds, where the Server's certificate chain is generated using various classical and post-quantum signature schemes. Four network latency conditions are simulated using Traffic Control on a **Raspberry Pi 5**. The measurements are normalized by the F variant of SPHINCS^{+}.
.

chine. Finally, `139.47` is the number of connections per "user second" (i.e., 53 ÷ 0.38 = 139.47).

Unfortunately, during the experiments we found a bug[2] in the widely employed `s_time` utility which caused the Client-side validation of the certificate chain to be skipped entirely. This is particularly relevant when comparing signature schemes because the client must perform as many signature verifications as the number of CAs in the chain, plus one (to verify the hashed transcript of the handshake). We fixed this bug and used our patched version of OpenSSL 3.5.0 (commit `cdd01b5`) for the experiment. We also opened a pull request to OpenSSL, where the OpenSSL maintainers reviewed our feedback and acknowledged the bug. A second modification to OpenSSL was required to increase the maximum size of the certificate chain that a client application accepts. The TLS specification [23] sets this limit at 16 MiB, but OpenSSL adopts a more conservative 100 KiB upper bound[3]. Raising the limit to the theoretical maximum allows the experiment to include some signature algorithms whose public key and signature size result in a large certificate chain (e.g., OV-V, CROSS-RSDP-5-b).

We used the most recent versions of `liboqs` (0.13.0 commit `b02d0c9`), and `oqs-provider` (0.10.0 commit `8603d0d`). These both host CROSS version 2.2 (which includes all the optimizations discussed in Section 3) together with AVX2-optimized implementations for the onramp and standardized signature schemes we compare CROSS to. We modified a configuration file in `oqs-provider` to enable more signature algorithms in the experiment (e.g., only one CROSS variant was enabled by default) and compiled all software libraries with their default optimization options.

The measurements are taken on the same system as the one from the microbenchmarks, a Debian 12 system with an i3 − 8350K. The experiment takes place inside a Docker container (version 28.3.1) with Alpine 3.21, where `liboqs`, `oqs-provider`, `OpenSSL`, and `curl` 8.11.1 are compiled from source. All the patches, the source code, and the full results are hosted on GitHub [35]. Our analysis excludes the impact of certificate revocation mechanisms (CRL, OCSP) and certificate transparency checks (SCT). While these do involve signature schemes, they deserve their own analysis, which should also account for more complex

[2] https://github.com/openssl/openssl/issues/27869
[3] https://github.com/openssl/openssl/issues/28720

mechanisms like caching. Regarding the choice of analyzed algorithms, we included most of the ones available in `oqs-provider`. Notably, one CROSS variant (CROSS-RSDP-256-f) is incompatible with the TLS specification, as its signature exceeds the maximum allowed size for the `CertificateVerify` message (16 KiB). We show only the *padded* variants of Falcon (i.e., the ones with fixed signature sizes): the *compressed* variants are close in terms of performance but are less likely to be employed in a scenario such as TLS, where saving a few bytes will likely not have a significant impact [8]. Finally, concerning SPHINCS$^+$, we only report the SHAKE variants, as we measured similar performances for the SHA-2 [46] and Haraka [47] variants. Since the purpose is to compare signature algorithms, we fixed the KEM choice to ML-KEM-512 [48]. It seemed a good candidate as it is already standardized, but we could have chosen any other classical, post-quantum, or hybrid KEM. Interestingly, other experiments have shown that, although the choices of KEM and signature schemes are independent in TLS, different combinations can interact and influence how TLS messages are buffered [49, Section 5.2].

Experimental evaluation. We report the results of our experimental evaluations in Fig. 7. We split the signature schemes in Classical, and (post-quantum) NIST Category 1 and 2, 3, and 5. The reason for coalescing categories 1 and 2 is that the security guarantees provided by a TLS exchange hinge on the weakest link, and, since most block ciphers, among which AES, only provide a security level comparable with Category 1, 3, and 5, Category 2 signatures are typically used in lieu of category 1 ones if there is a non-security related advantage (as there is with MAYO-2 which is faster than MAYO-1 in our benchmarks). On the vertical axis we observe the number of connections the server was capable of sustaining in a 10-second interval, the four colors indicate the different latency conditions. The results are normalized to highlight the relative performance between schemes. We chose SPHINCS$^+$ as the reference since NIST has indicated that new candidates should demonstrate performance improvements over it.

The first observation is that, especially for Category 1, higher network latency scenarios tend to flatten the differences between algorithms. We can quantify this using the Relative Standard Deviation (RSD): the ratio of the standard deviation to the mean, expressed as a percentage. In the local network scenario (0 ms latency) the RSD between all the post-quantum signatures is 74%. In comparison, the other network scenarios (2 ms, 20 ms, 120 ms) yield a progressively lower RSD: 54%, 52%, and 48%, respectively. Signing time, verification time, public key size, and signature size all play a role during the handshake, and TLS appears to be rewarding signature schemes that struck a balance between the three. The key generation time, on the other hand, is irrelevant for the handshake, as all the keys are generated offline before any exchange of messages. The performance figures provided by CROSS are competitive, especially for NIST Category 1, where its best performing parameter set, RSDP-1-f it is outperformed only by MAYO-2 (with currently security-insufficient parameters) and ML-DSA. This behavior is interesting as CROSS has signature sizes comparable with SPHINCS$^+$, and therefore larger than both the lattice-based and multivariate based competitors. However, in a full TLS test, the combination of small public keys and fast signature and verification computations allows CROSS to match or exceed the performance of other competitors. This is especially true for the case of SPHINCS$^+$, which is able to achieve about one decade less connections per second with respect to CROSS, providing strong evidence on the fact that CROSS fulfills NIST's requirements in terms of exceeding the performances provided by SPHINCS$^+$.

An unexpected finding from our experiments was that, in NIST Category 1, RSDPG-1-fast achieved a higher number of connections than Falcon-512. This result is surprising, as Falcon exhibits a signing time comparable to CROSS (~900 Kcycles), but a significantly faster verification time (164 vs. 581 Kcycles). Moreover, the combination of two public keys and three signatures exchanged during the handshake results in a smaller total payload for Falcon (3.7 KiB) compared to CROSS (35.2 KiB). A packet-capture analysis revealed the cause of this apparent discrepancy: the TLS message buffering behavior in OpenSSL. For CROSS, the messages are transmitted as shown in Fig. 6. Once the Server finishes sending the certificate chain, it immediately begins signing the hashed transcript of the conversation; meanwhile the Client, having received the chain, can begin verifying it. Importantly, this allows the Server-side signing and the two Client-side verifications to proceed in parallel. In contrast, Falcon's certificate chain is small enough that OpenSSL chooses to bundle it with the `CertificateVerify` message, sending both together. Consequently, the Client must wait until the Server completes the signing process before starting verification, removing this opportunity for parallelism. To validate this hypothesis, we artificially inflated Falcon's certificate (by adding a dummy extension) just enough to force OpenSSL to transmit it separately, before the `CertificateVerify` message. In this modified configuration, the number of completed handshakes for Falcon increased (despite the larger message size) thereby confirming our hypothesis.

Fig. 8 repeats the TLS experiment on a Raspberry Pi 5. Unlike the Intel system used for Fig. 7, the Raspberry Pi lacks AVX2 support, so the reference implementation of CROSS is employed by default. In contrast, schemes such as Falcon and MAYO, which include ARM-optimized implementations in OQS, tend to perform better in this setting. Future work could explore adapting the optimizations discussed in Section 3 to the NEON instruction set architecture. Nonetheless, the experiment demonstrates that the fastest CROSS variants remain competitive even when the reference implementation is used. For instance, in Category 5, RSDPG-5-f is more than twenty times faster than the SPHINCS$^+$ variant used for normalization (256-f-simple).

Appendix A presents additional figures that provide further insight into the experiments. Figs. A.9 and A.10 show the same results (obtained on Intel and Raspberry, respectively) but in absolute values instead of normalized form. Note the use of a logarithmic scale, a choice dictated by the vast difference in number of the connections between the fastest and slowest network scenarios. We also repeated our experiment on the AWS Cloud infrastructure. We launched four EC2 instances of type `m7i-flex.large` (2 vCPU, 8 GiB RAM), all running Debian 13. The first was a Server, located in Paris (France). The other three were Clients, located in Paris, Frankfurt (Germany), and the Bay Area (USA). The AWS regions are `eu-west-3`, `eu-central-1`, and `us-west-1`, respectively. Then we repeated our measurements on the infrastructure following the same praxes as the one on the simulated one. Using AWS virtual machines, physically located in different regions, helped to validate the results we had collected on a local network with Docker and Traffic Control. The number of handshakes measured on AWS follows closely the ones we obtained on the simulated network (Fig. A.9), and the relations between schemes stay essentially unchanged. The setup, however, is inherently more difficult to reproduce, as it depends on the underlying AWS infrastructure, which is subject to change. For this reason, we include the AWS measurements as the supplementary Fig. A.11 in Appendix A. The complete set of results, together with the source code for all experiments described in Section 4.2, is available on GitHub [35].

5. Conclusion

We presented a set of optimization techniques to obtain a computationally efficient implementation of the CROSS post-quantum signature scheme. We validate the efficiency of our implementation, showing that CROSS provides largely sub-millisecond signature generation and verification for security levels matching the one provided by AES-128 against confidentiality threatening attacks. Furthermore, we validated the use of CROSS in TLS, showing that it is ready for practical use, and provides competitive performances.

Data availability

Data are fully available, link in the paper.

CRediT authorship contribution statement

Alessandro Barenghi: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization; **Marco Gianvecchio:** Writing – review & editing, Writing – original draft, Validation, Software, Investigation, Data curation; **Gerardo Pelosi:** Writing – review & editing, Writing – original draft, Supervision, Software, Methodology, Investigation, Formal analysis, Conceptualization.

Acknowledgments

This work was supported in part by project SERICS (PE00000014) under the Italian NRRP MUR program funded by the EU - NGEU.

Appendix A. Appendix

In the following, the AVX2 algorithm realizing a fast and constant-time modular exponentiation is reported. It takes as input sixteen 16-bit integers packed into a 256-bit vector and computes $16^c \bmod 509$.

```
__m256i mm256_exp16m509_epu16(__m256i c) {
    /* high 3 bits */
    __m256i     h3 = _mm256_srli_epi16(c, 4);
    __m256i pre_h3 = _mm256_setr_epi16(1,302,93,91,505,319,137,145,
                                       1,302,93,91,505,319,137,145);
    __m256i h3_shu = mm256_shuffle_epi16(pre_h3, h3);
    /* low 4 bits */
    __m256i      mask_l4 = _mm256_set1_epi16(0x0F);//0b1111
    __m256i           l4 = _mm256_and_si256(c, mask_l4);
    __m256i mask_l4_bit4 = _mm256_set1_epi16(0x8); //0b1000
    __m256i      l4_bit4 = _mm256_and_si256(c, mask_l4_bit4);
                 l4_bit4 = _mm256_srli_epi16(l4_bit4, 3);
    __m256i      l4_sub8 = _mm256_sub_epi16(l4, _mm256_set1_epi16(8));
    __m256i     pre_l4_0 = _mm256_setr_epi16(1,16,256,24,384,36,67,54,
                                             1,16,256,24,384,36,67,54);
    __m256i     l4_shu_0 = mm256_shuffle_epi16(pre_l4_0, l4);
    __m256i     pre_l4_1 = _mm256_setr_epi16(355,81,278,376,417,55,371,337,
                                             355,81,278,376,417,55,371,337);
    __m256i     l4_shu_1 = mm256_shuffle_epi16(pre_l4_1, l4_sub8);
    __m256i       l4_shu = mm256_cmov_epu16(l4_bit4, l4_shu_1, l4_shu_0);
    /* multiply */
    __m256i  r = mm256_mulmod509_epu16(h3_shu, l4_shu);
    return r;
}
```

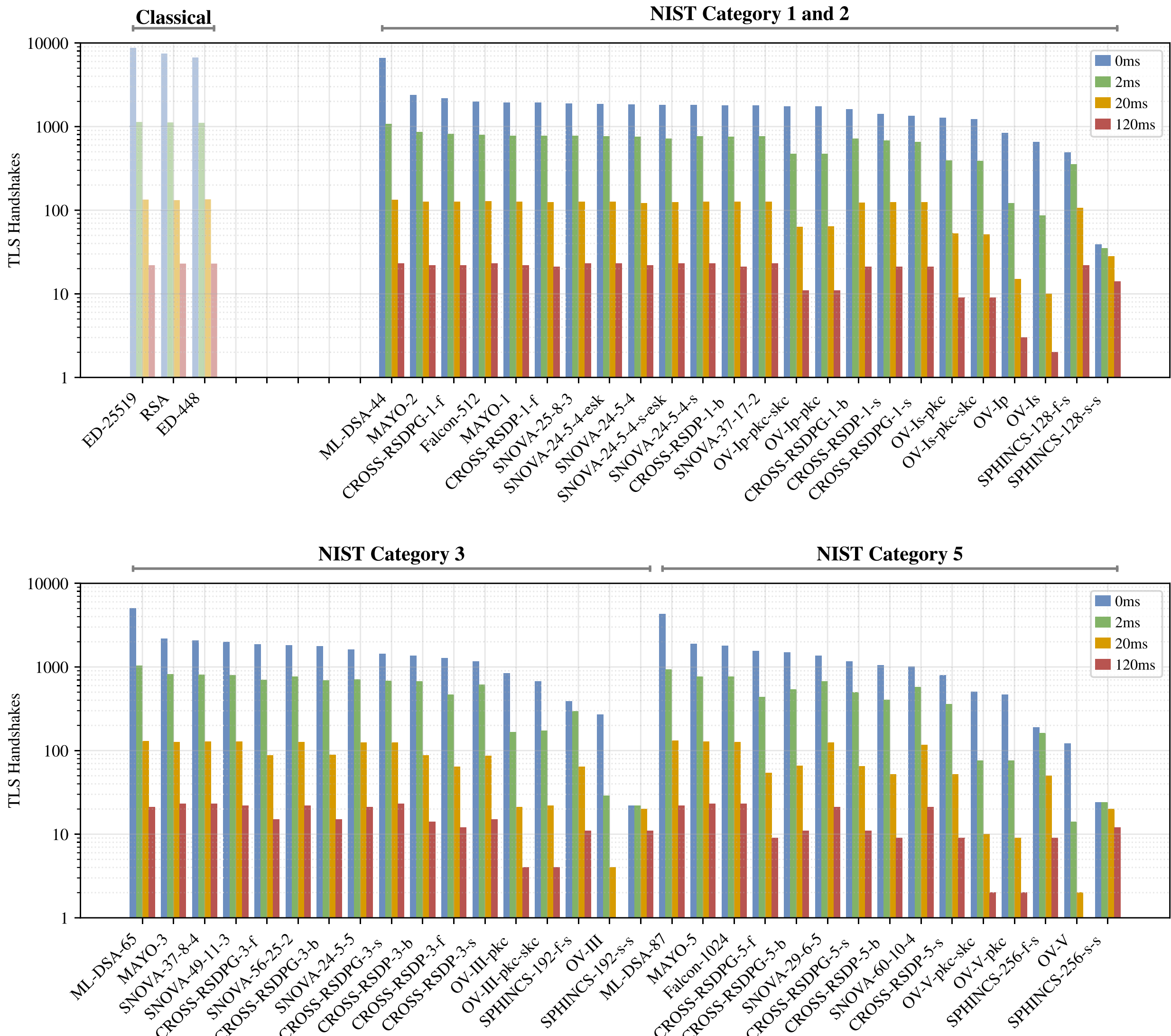


Fig. A.9. Number of connections to a Server within 10 seconds, where the Server's certificate chain is generated using various classical and post-quantum signature schemes. Four network latency conditions are simulated using Traffic Control on an **Intel i3-8350K**. .

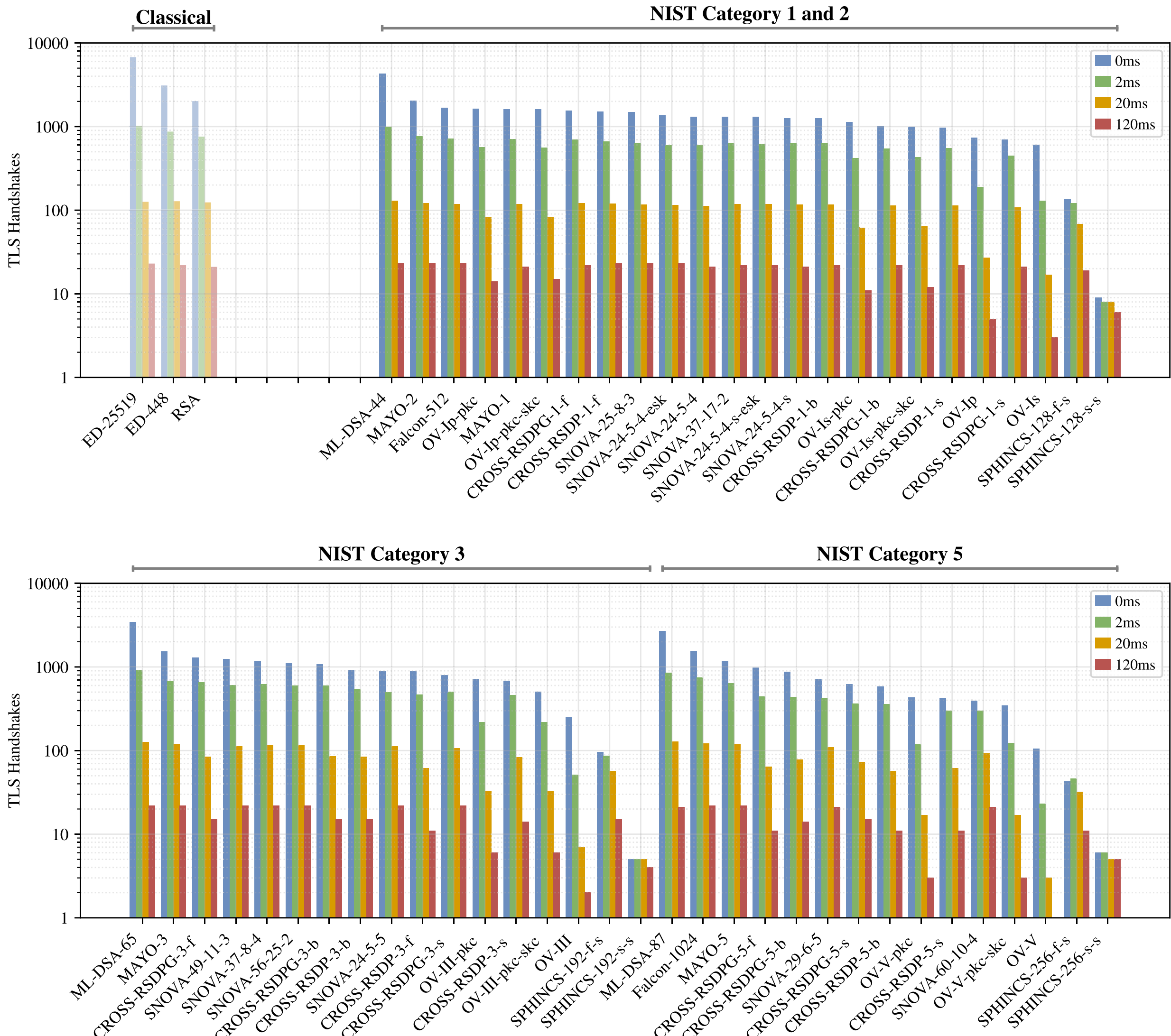


Fig. A.10. Number of connections to a Server within 10 seconds, where the Server's certificate chain is generated using various classical and post-quantum signature schemes. Four network latency conditions are simulated using Traffic Control on a **Raspberry Pi 5**. .

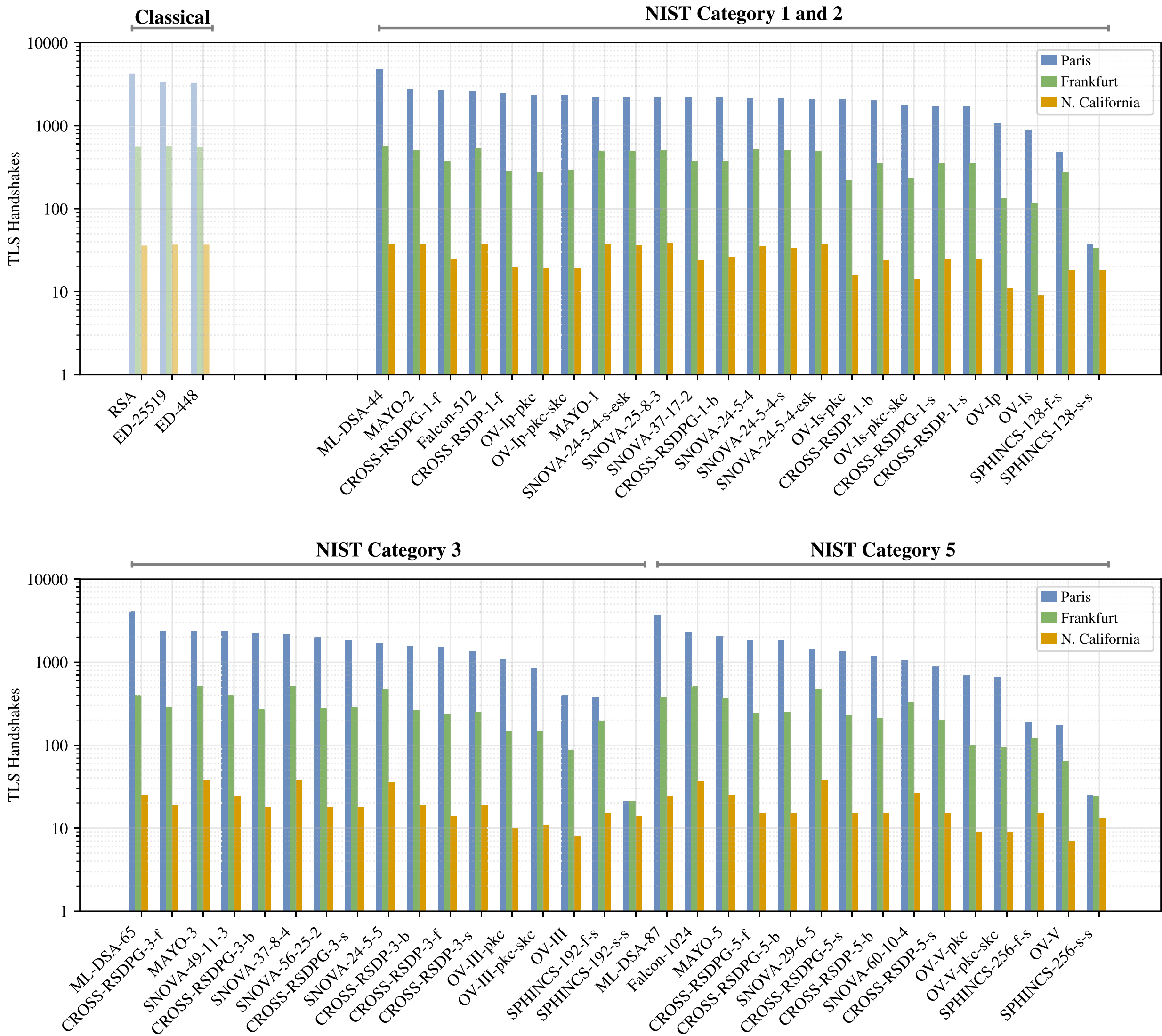


Fig. A.11. Number of connections to a Server within 10 seconds, where the Server's certificate chain is generated using various classical and post-quantum signature schemes. The Clients are **AWS instances** located in Paris, Frankfurt, and Northern California, all handshaking a Server in Paris. .

References

[1] evolutionQ Inc.. Quantum threat timeline 2025: executive perspectives on barriers to action. White paper: https://globalriskinstitute.org/mp-files/quantum-threat-timeline-2025-executive-perspectives-on-barriers-to-action.pdf/; 2025.

[2] NIST. Post-quantum cryptography standardization. National Institute of Standards and Technology - Information Technology Lab. - Computer Security Resource Center document: https://csrc.nist.gov/pqc-standardization; 2017. [Online accessed on July 2025].

[3] Bernstein DJ, Hülsing A, Kölbl S, Niederhagen R, Rijneveld J, Schwabe P. The SPHINCS+ signature framework. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. CCS '19; New York, NY, USA: Association for Computing Machinery. ISBN 9781450367479; 2019, p. 2129–46. https://doi.org/10.1145/3319535.3363229

[4] NIST. Fips 204 - module-lattice-based digital signature standard. U.S. National Institute of Standards and Technology (NIST) standard: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.204.pdf; 2024. https://doi.org/10.6028/NIST.FIPS.204

[5] NIST. FIPS 205 - stateless hash-based digital signature standard. U.S. National Institute of Standards and Technology (NIST) standard: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.205.pdf; 2024.

[6] Hülsing A, Aumasson J-P, Bernstein DJ, Beullens W, Dobraunig C, Eichlseder M, et al. SPHINCS+ – submission to the 3rd round of the NIST post-quantum project. v3.1. Technical report: https://sphincs.org/data/sphincs+-r3.1-specification.pdf; 2022.

[7] NIST. FFT over NTRU-lattice-based digital signature algorithm. U.S. National Institute of Standards and Technology (NIST) standard preparation document: https://csrc.nist.gov/csrc/media/Presentations/2024/falcon/images-media/prest-falcon-pqc2024.pdf; 2024.

[8] Prest T, Fouque P-A, Hoffstein J, Kirchner P, Lyubashevsky V, Pornin T, et al. Falcon: fast-fourier lattice-based compact signatures over NTRU - specification v1.2 – 01/10/2020. https://falcon-sign.info/falcon.pdf; 2020.

[9] Hoffstein J, Pipher J, Silverman JH. Ntru: a ring-based public key cryptosystem. In: Buhler J, editor. Algorithmic number theory, third international symposium, ANTS-III, Portland, Oregon, USA, June 21–25, 1998, proceedings; 1423 of *Lecture Notes in Computer Science*. Springer; 1998, p. 267–88. https://doi.org/10.1007/BFB0054868

[10] NIST. Post-quantum cryptography: additional digital signature schemes. National Institute of Standards and Technology - Information Technology Lab. - Computer Security Resource Center document: https://csrc.nist.gov/projects/pqc-dig-sig; 2024. [Online accessed on July 2025].

[11] Baldi M, Barenghi A, Battagliola M, Bitzer S, Gianvecchio M, Karl P, et al. CROSS: codes and restricted objects signature scheme - algorithm specifications and supporting documentation version 2.2 - July 31, 2025. https://www.cross-crypto.com/CROSS_Specification_v2.2.pdf; 2025.

[12] Baldi M, Barenghi A, Beckwith L, Chou T, Biasse J-F, Esser A, et al. LESS: linear equivalence signature scheme - specification v2.0 – Feb. 7, 2025. https://www.less-project.com/LESS-2025-02-07.pdf; 2025.

[13] Fiat A, Shamir A. How to prove yourself: practical solutions to identification and signature problems. In: Odlyzko AM, editor. Advances in cryptology — CRYPTO' 86. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-47721-1; 1987, p. 186–94. https://doi.org/10.1007/3-540-47721-7_12.

[14] Melchor CA, Bettaieb S, Bidoux L, Feneuil T, Gaborit P, Gama N, et al. The syndrome decoding in the head (SD-in-the-head) signature scheme algorithm specifications and supporting documentation – version 2.0. February 5, 2025. Technical report: https://sdith.org/docs/sdith-v2.0.pdf; 2025.

[15] Beullens W. Mayo: practical post-quantum signatures from oil-and-vinegar maps. In: AlTawy R, Hülsing A, editors. Selected areas in cryptography - 28th international conference, SAC 2021, Virtual Event, September 29, - October 1, 2021, revised selected papers; vol. 13203 of *Lecture Notes in Computer Science*. Springer; 2021, p. 355–76. https://doi.org/10.1007/978-3-030-99277-4_17

[16] Beullens W, Campos F, Celi S, Hess B, Kannwischer MJ. MAYO - 5th of Feb, 2025. Submitted to round-2. https://pqmayo.org/assets/specs/mayo-round2.pdf; 2025.

[17] Baldi M, Barenghi A, Battagliola M, Bitzer S, Gianvecchio M, Karl P, et al. CROSS: codes and restricted objects signature scheme - security details version 2.2 - July 31, 2025. https://www.cross-crypto.com/CROSS_SecurityDetails_v2.2.pdf; 2025.

[18] NIST. FIPS 202 - SHA-3 standard: permutation-based hash and extendable-output functions. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf; 2015.

[19] Goldreich O, Goldwasser S, Micali S. How to construct random functions. J ACM 1986;33(4):792–807. https://doi.org/10.1145/6490.6503

[20] Merkle RC. A digital signature based on a conventional encryption function. In: Pomerance C, editor. Advances in cryptology - CRYPTO '87, a conference on the theory and applications of cryptographic techniques, Santa Barbara, California, USA, August 16–20, 1987, proceedings; vol. 293 of *Lecture Notes in Computer Science*. Springer; 1987, p. 369–78. https://doi.org/10.1007/3-540-48184-2_32

[21] Borin G, Persichetti E, Pintore F, Reijnders K, Santini P. A guide to the design of digital signatures based on cryptographic group actions. J Cryptol 2025;38(3):23. https://doi.org/10.1007/S00145-025-09542-9

[22] Beullens W, Katsumata S, Pintore F. Calamari and falafl: logarithmic (linkable) ring signatures from isogenies and lattices. In: Moriai S, Wang H, editors. Advances in cryptology - ASIACRYPT 2020 - 26th international conference on the theory and application of cryptology and information security, daejeon, South Korea, December 7–11, 2020, proceedings, part II; vol. 12492 of *Lecture Notes in Computer Science*. Springer; 2020, p. 464–92. https://doi.org/10.1007/978-3-030-64834-3_16

[23] Rescorla E. The transport layer security (TLS) protocol version 1.3. RFC 8446 - https://www.rfc-editor.org/info/rfc8446; 2018. https://doi.org/10.17487/RFC8446

[24] Stebila D, Fluhrer S, Gueron S. Hybrid key exchange in TLS 1.3. https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/12/; 2025. Work in Progress.

[25] Laurie B. Certificate transparency. Commun ACM 2014;57(10):40–6. https://doi.org/10.1145/2659897

[26] Laurie B, Messeri E, Stradling R. Certificate transparency version 2.0. RFC 9162 - https://www.rfc-editor.org/info/rfc9162; 2021. https://doi.org/10.17487/RFC9162

[27] Eastlake 3rd DE. Transport layer security (TLS) extensions: Extension definitions. RFC 6066; 2011. https://doi.org/10.17487/RFC6066

[28] Barrett P. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko AM, editor. Advances in cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, proceedings; vol. 263 of *Lecture Notes in Computer Science*. Springer; 1986, p. 311–23. https://doi.org/10.1007/3-540-47721-7_24

[29] Langley A. ctgrind - checking that functions are constant time with Valgrind. https://github.com/agl/ctgrind; 2017.

[30] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation. PLDI '07; New York, NY, USA: Association for Computing Machinery. ISBN: 9781595936332; 2007, p. 89–100. https://doi.org/10.1145/1250734.1250746

[31] OQS. Open quantum safe project: the liboqs library. https://github.com/open-quantum-safe/liboqs; 2025.

[32] Bernstein D. eBACS: ECRYPT benchmarking of cryptographic systems. https://bench.cr.yp.to/supercop.html; 2025.

[33] Reparaz O, Balasch J, Verbauwhede I. Dude, is my code constant time? In: Design, automation and test in Europe conference and exhibition (DATE), 2017. 2017, p. 1697–702. https://doi.org/10.23919/DATE.2017.7927267.

[34] Bernstein DJ. The libcpucycles microlibrary. https://cpucycles.cr.yp.to/; 2025.

[35] Barenghi A, Gianvecchio M, Pelosi G. CROSS-experiments. https://github.com/rtjk/CROSS-experiments; 2025.

[36] Stebila D, Mosca M. Post-quantum key exchange for the internet and the open quantum safe project. In: Avanzi R, Heys H, editors. Selected areas in cryptography – SAC 2016. Cham: Springer International Publishing. ISBN: 978-3-319-69453-5; 2017, p. 14–37. https://openquantumsafe.org.

[37] Ran L. Wedges, oil, and vinegar – an analysis of UOV in characteristic 2. Cryptology ePrint Archive, Paper 2025/1143; 2025. https://eprint.iacr.org/2025/1143.

[38] OQS. Open quantum safe project: the liboqs-provider TLS library. https://github.com/open-quantum-safe/oqs-provider; 2025.

[39] Beullens W, Chen M-S, Ding J, Gong B, Patarin M J KJ, Peng B-Y, et al. UOV: unbalanced oil and vinegar - algorithm specifications and supporting documentation - version 2.0 - February 5, 2025. https://drive.google.com/file/d/1lMRVyBFoPD1AT3D-uUa_hL0g3ldvuJgL/view?usp=drive_link; 2025.

[40] Wangand L-C, Chouand C-Y, Dingand J, Kuanand Y-L, Leegwater JA, Liand M-S, et al. SNOVA proposal for NISTPQC: additional digital signature schemes - version 2.0 - January 25, 2025. https://snova.pqclab.org/files/SNOVA_Round2.zip; 2025.

[41] Goertzen J, Stebila D. Post-quantum signatures in DNSSEC via request-based fragmentation. In: Johansson T, Smith-Tone D, editors. Post-quantum cryptography - 14th international workshop, PQCrypto 2023, college park, MD, USA, August 16–18, 2023, proceedings; vol. 14154 of *Lecture Notes in Computer Science*. Springer; 2023, p. 535–64. https://doi.org/10.1007/978-3-031-40003-2_20

[42] Orlando S, Barenghi A, Pelosi G. Investigating the health state of x.509 digital certificates. In: IEEE International conference on cyber security and resilience, CSR 2024, London, UK, September 2–4, 2024. IEEE; 2024, p. 222–7. https://doi.org/10.1109/CSR61664.2024.10679412

[43] Paquin C, Stebila D, Tamvada G. Benchmarking post-quantum cryptography in TLS. In: Ding J, Tillich J, editors. Post-Quantum cryptography - 11th international conference, PQCrypto 2020, Paris, France, April 15–17, 2020, proceedings; vol. 12100 of *Lecture Notes in Computer Science*. Springer; 2020, p. 72–91. https://doi.org/10.1007/978-3-030-44223-1_5

[44] Montenegro JA, Rios R, Lopez-Cerezo J. A performance evaluation framework for post-quantum TLS. Fut Gen Comput Syst 2026;175:108062. https://www.sciencedirect.com/science/article/pii/S0167739X25003577.

[45] Alnahawi N, Müller J, Oupický J, Wiesmaier A. A comprehensive survey on post-quantum TLS. Commun Cryptograp 2024;Volume 1, Issue 2. https://doi.org/10.62056/ahee0iuc

[46] NIST. FIPS 180-4 - secure hash standard (SHS). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf; 2015.

[47] Kölbl S, Lauridsen MM, Mendel F, Rechberger C. Haraka v2 - efficient short-input hashing for post-quantum applications. Cryptology ePrint Archive, Paper 2016/098: https://eprint.iacr.org/2016/098; 2016.

[48] NIST. FIPS 203 - module-lattice-based key-encapsulation mechanism standard. https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.203.pdf; 2024.

[49] Sosnowski M, Wiedner F, Hauser E, Steger L, Schoinianakis D, Gallenmüller S, et al. The performance of post-quantum TLS 1.3. In: Rossi D, Secci S, Bonaventure O, Qiu L, editors. Companion of the 19th international conference on emerging networking EXperiments and technologies, coNEXT 2023, Paris, France, December 5–8, 2023. ACM; 2023, p. 19–27. https://doi.org/10.1145/3624354.3630585

Alessandro Barenghi is an associate professor at Politecnico di Milano, Italy. His interests include computer and network security, formal languages and compilers and he has published more than 100 papers in international peer reviewed venues.

Marco Gianvecchio is a researcher at Politecnico di Milano, Italy. He received his M.Sc. degree in Computer Sicence and Engineering from the same university in 2024. His research interests include post-quantum cryptography and secure software development. He is a member of the CROSS team, where he works on high-performance software implementations, with a focus on optimizing arithmetic operations and the integration into open source libraries.

Gerardo Pelosi is an associate professor at Politecnico di Milano, Italy. His main research interests are in computer security, cryptography, security in hardware and in the area of data security and privacy. He has published more than 100 papers in international peer reviewed journals and conference proceedings and is co-inventor of 10 patents concerning the design of cryptographic systems.