CAPSS: A Framework for SNARK-Friendly Post-Quantum Signatures

Thibauld Feneuil and Matthieu Rivain

CryptoExperts, Paris, France
thibauld.feneuil@cryptoexperts.com
matthieu.rivain@cryptoexperts.com

Abstract. In this paper, we present a general framework for constructing SNARK-friendly post-quantum signature schemes based on minimal assumptions, specifically the security of an arithmetization-oriented family of permutations. The term "SNARK-friendly" here refers to the efficiency of the signature verification process in terms of SNARK constraints, such as R1CS constraints. Within the CAPSS framework, signature schemes are designed as proofs of knowledge of a secret preimage of a one-way function, where the one-way function is derived from the chosen permutation family. To obtain compact signatures with SNARK-friendly verification, we rely on SmallWood, a recently proposed hash-based zero-knowledge argument scheme well suited for statements arising in this context. From this proof system which we tweak towards SNARK-friendliness, the CAPSS framework offers a generic transformation of any arithmetization-oriented permutation family into a SNARK-friendly post-quantum signature scheme. We provide concrete instances built on permutations such as Rescue-Prime, Poseidon, Griffin, and Anemoi. For the Anemoi family, achieving 128-bit security, our approach produces signatures of sizes ranging from 9.5 to 15.5 KB, with R1CS constraints between 24 K and 35 K. This represents a 4–6× reduction in signature size and a 5–8× reduction in R1CS constraints compared to Loquat (CRYPTO 2024), a SNARK-friendly post-quantum signature scheme based on the Legendre PRF. We showcase CAPSS through aggregated signatures, achieving sub-kilobyte amortized size for large batches, and anonymous credentials, enabling presentation proofs under 150 KB, thus highlighting the practicality of SNARK-friendly, symmetric-based designs for post-quantum primitives.

1 Introduction

The advent of a quantum computer capable of breaking classical public-key cryptosystems (RSA, ECC) urges the cryptography research community to find reliable alternatives to common cryptosystems. One of the most ubiquitous cryptographic primitives that will need to be replaced is digital signature. While some post-quantum signature schemes have been proposed for general-purpose application, such as TLS communications, they might not be tailored to certain use-cases with specific constraints. One such area that requires specialized design is SNARK-friendly signatures. A SNARK (Succinct Non-Interactive Argument of Knowledge) is a type of cryptographic proof that enables a party to succinctly demonstrate the correctness of a potentially large computation, without the verifier needing to re-execute the entire computation themselves. For a digital signature to be SNARK-friendly, its verification process must be tweaked in a way to make it efficiently wrappable into a SNARK proof.

SNARK-friendly signatures have numerous potential applications, such as blind signatures and anonymous credentials, where zero-knowledge proofs for signatures are paramount [Cha82, Fis06, CL01]. Another particularly promising use case is aggregate signatures [BGLS03] –a cryptographic technique that combines multiple individual signatures into a single, compact signature for more efficient storage, transmission, and verification. In blockchains, aggregate signatures play a crucial role in enhancing scalability by allowing more transactions to fit into each block, optimizing bandwidth and reducing data overhead. They also speed up verification by enabling a single operation to validate multiple signatures at once, saving computational resources.

The design of a SNARK-friendly post-quantum signature scheme was recently explored in [ZSE$^+$24]. In this work, the authors introduced Loquat, a post-quantum signature scheme built on the Legendre PRF, explicitly tailored for SNARK compatibility. Loquat produces signatures of approximately 57 KB, with their verification requiring 149 K R1CS constraints. Another recent study proposed a signature scheme leveraging a STARK-based proof system applied to the Rescue-Prime permutation [ADK24] though it does not specifically target SNARK-friendliness. This scheme achieves signatures of either 80 KB or 100 KB, depending on the decoding regime used in the underlying STARK proof.

In the present paper, we introduce CAPSS,[1] a general framework for constructing SNARK-friendly post-quantum signature schemes based on a minimal assumption: the security of an arithmetization-oriented permutation modeled as an ideal permutation. SNARK-friendliness refers to the signature verification process being efficient in terms of SNARK constraints, such as Rank-1 Constraint System (R1CS) or Algebraic Intermediate Representation (AIR) constraints used in STARKs. This property enables efficient and generic aggregation schemes using post-quantum SNARKs, such as Aurora [BCR+19] or STARKs [BBHR19].

Our framework builds on the recently introduced SmallWood zero-knowledge argument scheme [FR25], a hash-based proof system that leverages techniques from both the Threshold-Computation-in-the-Head (TCitH) framework [FR23] and Brakedown [GLS+23]. SmallWood is particularly well-suited for producing compact proofs of arithmetization-oriented one-way functions. Although not originally designed for SNARK verification, its Merkle-tree-based structure makes it naturally compatible with such settings. We further propose a set of concrete modifications that enable efficient SNARK-based verification of SmallWood proofs, relying on lightweight arithmetic checks and a small number of Merkle path validations to achieve succinctness.

Leveraging these enhancements, the CAPSS framework extends SmallWood to construct post-quantum signature schemes from arithmetization-oriented permutation families. Specifically, the signature scheme proves knowledge of a preimage x for a one-way function output $y = f(x)$ using a SmallWood proof. Applying the Fiat-Shamir transform yields a non-interactive signature scheme where y serves as the public key and x as the secret key. Importantly, both the one-way function f and the hash function used in SmallWood are instantiated from the same underlying permutation family, ensuring that the scheme's security depends solely on the hardness of inverting the permutation. This unified design yields a conservative and modular approach to building SNARK-friendly, post-quantum secure signatures.

We provide general arithmetization techniques for arithmetization-oriented permutations in the PACS syntax underlying SmallWood arguments, which is ideally suited for this type of statement. Thanks to this, we can achieve signature sizes below 10 KB in a "short signature regime". We provide concrete instances built on arithmetization-oriented permutations such as Rescue-Prime [SAD20, AKM+22], Poseidon [GKR+21, GKS23], Griffin [GHR+23], and Anemoi [BBC+23], carefully selected for their suitability in SNARK-friendly contexts. For the Anemoi family, achieving 128-bit security, our approach produces signature sizes ranging from 9 to 13.3 KB, with the number of R1CS constraints between 19 K and 29 K. These results represent a 4–6$\times$ reduction in signature size and a 5–8$\times$ reduction in R1CS constraints compared to Loquat [ZSE+24].

We showcase CAPSS through two applications. First, we build aggregated signatures, where the amortized size becomes beneficial for $n > 16$ and drops to 260 bytes per signature for 1024 signatures. Second, we construct an anonymous credential system combining CAPSS with Aurora, yielding proofs below 150 KB and sub-second verification for disclosure circuits of size 2^{15}. These results demonstrate the practicality of CAPSS and the promise of SNARK-friendly, symmetric-based designs for post-quantum secure primitives.

Paper organization. Section 2 presents an overview of the CAPSS framework and its core components. Section 3 details the arithmetization of one-way function statements within the syntax of SmallWood arguments. Section 4 introduces the modifications required for SNARK-friendly verification. A comprehensive description of the general CAPSS signature scheme, along with its formal security under the EUF-CMA notion, is provided in Section 5. Section 6 presents concrete instantiations based on four families of permutations and offers benchmarks and comparisons with the current state of the art. Finally, Section 7 reports the results of using CAPSS in building aggregated signatures and anonymous credentials.

[1] CAPSS stands for "Compilation of Arithmetic-oriented Permutation into SNARK-friendly Signature".

2 Overview of the CAPSS Framework and its Core Components

The CAPSS framework compiles an arithmetization-oriented family of permutations $\mathcal{P}$ into a signature scheme. The transformation is overviewed in Figure 1. The family of permutations $\mathcal{P}$ is used to define three cryptographic primitives: a one-way function (OWF), Merkle trees (MT) and an extendable output hash function (XOF). The one-way function is the underlying hard problem for the signature scheme. Specifically, for an initialization value iv and secret input value w (for witness), the secret and public keys (sk, pk) of the scheme are defined as:

$$sk = w \quad \text{and} \quad pk = (iv, y) \quad \text{where} \quad y = \mathsf{OWF}_{iv}(w)\,.$$

A signature is a non-interactive zero-knowledge (ZK) argument of knowledge (ARK) of a secret input w which maps to the public output y through OWF_{iv}.

For the latter we use SmallWood-ARK, the ZK-ARK scheme for PACS statements introduced in [FR25] and presented in Section 2.2, which combines the PACS polynomial IOP with their DECS-based polynomial commitment scheme, SmallWood-PCS. The PACS syntax is ideal to efficiently arithmetize a permutation-based OWF; we provide general arithmetization techniques for such primitives. The Merkle trees in SmallWood-PCS and the XOF used to hash the leaves and for the Fiat-Shamir transform are both built from $\mathcal{P}$. The obtained ZK-ARK is further tweaked in several ways towards SNARK friendliness. The goal of those tweaks is to make the underlying verification algorithm efficient to arithmetize while giving rise to a low number of R1CS constraints.

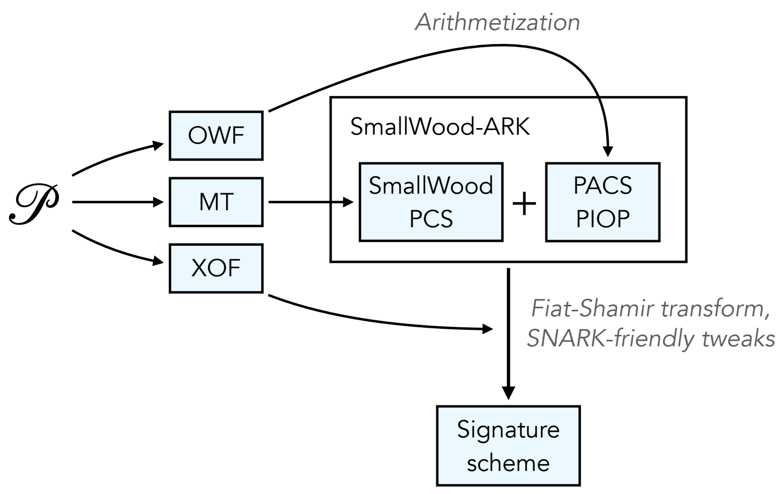

Fig. 1: Overview of the CAPSS framework to compile a permutation family $\mathcal{P}$ into a signature scheme.

The remainder of this section introduces the core components on which the CAPSS framework relies:

- the three permutation-based primitives (OWF, MT and XOF);
- the SmallWood-ARK zero-knowledge argument scheme.

2.1 Permutation-Based Primitives

As overviewed previously, the CAPSS framework relies on the following modes and permutation-based primitives:

– extendable output hash functions (XOF) using the Sponge mode [BDPV08],
– Merkle trees (MT) with XOF compression functions using the Jive mode [BBC$^+$23],
– one-way functions (OWF) using truncation.

We formally introduce the considered families of permutations and the aforementioned modes of operations hereafter.

Family of Permutations. An arithmetization-oriented family of permutations is a set of bijective functions

$$\mathcal{P} = \{P_{t,q} : \mathbb{F}_q^t \to \mathbb{F}_q^t \mid q \in \mathcal{Q}, t \in \mathcal{T}\}$$

defined with respect to a set of admissible field sizes $\mathcal{Q} \subseteq \mathbb{N}$ and a set of admissible state sizes $\mathcal{T} \subseteq \mathbb{N}$. In the following, we will sometimes keep the state size t and the field size q implicit and simply denote P the considered permutation from $\mathcal{P}$.

We specifically consider permutations which are constructed by iterating a number of rounds. We say that the permutation has a *regular iterated construction* with n_r rounds if it can be expressed as

$$P(x) = R_{n_r-1} \circ \cdots \circ R_1 \circ R_0(x) \quad \text{with} \quad R_i(x) = F(x, c_i)$$

for a round permutation $F : \mathbb{F}_q^t \times \mathbb{F}_q^{n_c} \to \mathbb{F}_q^t$ and round constants $c_0, \dots, c_{n_r-1} \in \mathbb{F}_q^{n_c}$. We say that the permutation has an *irregular iterated construction* with n_p partial rounds and n_f full rounds if it can be expressed as

$$P(x) = R_{n_f+n_p-1} \circ \cdots \circ R_0(x) \quad \text{with} \quad R_i(x) = \begin{cases} F_f(x, c_i) & \text{if } i \in [0, n_f/2) \cup [n_f/2 + n_p, n_f + n_p) \\ F_p(x, c_i) & \text{if } i \in [n_f/2, n_f/2 + n_p) \end{cases}$$

for a full round permutation $F_f : \mathbb{F}_q^t \times \mathbb{F}_q^{n_c} \to \mathbb{F}_q^t$, a partial round permutation $F_p : \mathbb{F}_q^t \times \mathbb{F}_q^{n_c} \to \mathbb{F}_q^t$ and round constants $c_0, \dots, c_{n_r-1} \in \mathbb{F}_q^{n_c}$. A round permutation F (resp. F_f, F_p) has verification function $G : \mathbb{F}_q^{|v|} \times \mathbb{F}_q^t \times \mathbb{F}_q^t \times \mathbb{F}_q^{n_c} \to \mathbb{F}_q^{n_G}$ (resp. G_f, G_p) if for all $x, y \in \mathbb{F}_q^t$ and $c \in \mathbb{F}^{|c|}$, we have

$$y = F(x, c) \quad \Leftrightarrow \quad \exists v \in \mathbb{F}_q^{|v|} : G(v, x, y, c) = 0 \ .$$

The arithmetization-oriented feature of a permutation $P \in \mathcal{P}$ is informally captured by requiring that a equality $y = P(x)$ can be efficiently verified using a small number of arithmetic constraints. In the iterated setting, with underlying verification function G, verifying $y = P(x)$ translates to verifying the constraint system

$$\begin{cases} x_0 = x, \ x_{n_r} = y \\ \forall i \in [0, n_r) \ , \ \exists v_i \ : \ G(v_i, x_i, x_{i+1}, c_i) = 0 \end{cases}$$

We now introduce the three modes that we consider in our framework (XOF, MT and OWF). They will all make use of the truncation function

$$\mathsf{Tr}_u : \mathbb{F}_q^t \to \mathbb{F}_q^u$$

which returns the u first coordinates of its input vector.

Sponge-Based (Extendable Output) Hash Functions. We consider the Sponge mode of operation [BDPV08] with the tweak from [Hir18].[2] Given a permutation family $\mathcal{P}$, a target security level λ and a field size q, we select a permutation $P \in \mathcal{P}$ of state size $t \geq \log_2 q/2\lambda + 1$ and define the *capacity* and the *rate* as the parameters

$$c = \left\lceil \frac{\log_2 q}{2\lambda} \right\rceil \quad \text{and} \quad r = t - c \ .$$

[2] The tweak proposed in [Hir18] consists in the introduction of the constant $\sigma \in \{0, 1\}$ in the capacity to deal with message of length n_{in} divisible by the rate r without adding a full additional block $(1, 0 \dots, 0)$.

For these parameters and for an input length n_{in} and an output length n_{out}, the Sponge-based XOF function

$$\mathsf{XOF}_{\mathcal{P}} : m \in \mathbb{F}_q^{n_{\text{in}}} \mapsto z \in \mathbb{F}_q^{n_{\text{out}}}$$

is defined as follows. The message is split into $n'_{\text{in}} := \lceil n_{\text{in}}/r \rceil$ blocks $m_0, \ldots, m_{n'_{\text{in}}-1} \in \mathbb{F}_q^r$ which are defined as:

$$\begin{cases} (m_0 \parallel m_1 \parallel \ldots \parallel m_{n'_{\text{in}}-1}) = m & \text{if } r \mid n_{\text{in}} \\ (m_0 \parallel m_1 \parallel \ldots \parallel m_{n'_{\text{in}}-1}) = (m \parallel (1, 0, \ldots, 0)) & \text{if } r \nmid n_{\text{in}} \end{cases}$$

From these blocks, the Sponge mode iteratively processes state variables $s_0, s_1, \ldots, s_{n'_{\text{in}}+n'_{\text{out}}} \in \mathbb{F}_q^t$ as follows:

$$s_i = \begin{cases} (m_0 \parallel 0^c\,) & \text{if } i = 0 \\ P(s_{i-1}) + (m_i \parallel 0^c\,) & \text{if } i \in [1, n'_{\text{in}}) \\ P(s_{n'_{\text{in}}-1}) + (m_{n'_{\text{in}}} \parallel \sigma\,) & \text{if } i = n'_{\text{in}} \\ P(s_{i-1}) & \text{if } i \in (n'_{\text{in}}, n'_{\text{in}} + n'_{\text{out}}] \end{cases} \qquad \text{with} \qquad \sigma = \begin{cases} 1 & \text{if } r \mid n_{\text{in}} \\ 0 & \text{if } r \nmid n_{\text{in}} \end{cases}$$

Finally the output is composed of $n'_{\text{out}} := \lceil n_{\text{out}}/r \rceil$ blocks $z_0, \ldots, z_{n'_{\text{out}}-1} \in \mathbb{F}_q^r$ such that

$$z = \mathsf{Tr}_{n_{\text{out}}}(z_0 \parallel \ldots \parallel z_{n'_{\text{out}}-1}) \quad \text{with} \quad z_i = \mathsf{Tr}_r(s_{n'_{\text{in}}+1+i})\,,\ \forall i \in [0, n'_{\text{out}})\ .$$

It is well known that if the permutation P is modeled as a random permutation, then $\mathsf{XOF}_{\mathcal{P}}$ is indifferentiable from a random oracle [BDPV08]. The security of our signature schemes shall thus hold under a random oracle assumption for $\mathsf{XOF}_{\mathcal{P}}$.

Domain separation. To enforce domain separation in CAPSS, we additionally tweak the sponge mode by redefining σ as follows:

$$\sigma^{(i)} = 2 \cdot i + \begin{cases} 1 & \text{if } r \mid n_{\text{in}} \\ 0 & \text{if } r \nmid n_{\text{in}} \end{cases}$$

for the i-th call to the XOF denoted $\mathsf{XOF}_{\mathcal{P}}^{(i)}$ (assuming that $i < q/2$).

Merkle Trees with Jive Compression. Merkle trees are derived from $\mathcal{P}$ by using two underlying primitives:

- a (fixed-output) hash function for the leaves: we use the Sponge-based $\mathsf{XOF}_{\mathcal{P}}$ function as introduced above,
- a (α-arity) compression function for the nodes of the tree.

Several approaches have been considered in the literature to build such a compression function from aritmetization-oriented permutations. Such a function is a collision-resistant function which maps α hash digests $x_0, \ldots, x_{\alpha-1} \in \mathbb{F}_q^c$ to 1 hash digest $y \in \mathbb{F}_q^c$ (where here hash digests are tuples of field elements). A possible strategy proposed by Poseidon [GKR+21] is to rely on the Sponge mode with a single permutation, with capacity c and rate $r = bc$, so that $y = \mathsf{Tr}_c(P(x_0 \parallel \ldots \parallel x_{\alpha-1} \parallel 0^c))$, this strategy is not optimal as it requires a permutation of state size $t = (\alpha + 1)c$ to deal compress an input of size bc, which is due to the capacity parameter of the Sponge mode. A better strategy is to rely on Davies-Meyer construction to avoid this loss. This is the approach followed by the Jive mode [BBC+23].

For a compression parameter α, and a state size t divisible by α, the Jive mode turns a arithmetization-oriented permutation $P : \mathbb{F}_q^t \to \mathbb{F}_q^t$ into a α-to-1 compression function $\mathsf{Jive}_{\mathcal{P}} : \mathbb{F}_q^t \to \mathbb{F}_q^c$ with $c = t/\alpha$. It can be summarized as summing the α blocks of size c composing the output of P', the Davies-Meyer transformation of P. Formally:

$$\mathsf{Jive}_{\mathcal{P}}(x) = \sum_{i=0}^{\alpha-1} P'_i(x)$$

with $P'_i : \mathbb{F}_q^t \to \mathbb{F}_q^c$ the coordinate functions of P' defined as:

$$(P'_0(x), \dots, P'_{\alpha-1}(x)) = P'(x) = P(x) + x \ .$$

For a security parameter λ, our instantiations of the Jive mode must have an output size of at least 2λ bits. In other words, the output of the $\mathsf{Jive}_{\mathcal{P}}$ compression function must be of size $c = \left\lceil \frac{\log_2 q}{2\lambda} \right\rceil$ (which coincides with the capacity parameter of the Sponge mode, hence we keep the same notation). We then select a permutation with state size $t = \alpha \cdot c$ given the target compression parameter α. We might use several instantiations of the Jive mode corresponding to different compression parameter α. We shall denote $\mathsf{Jive}_P^{(\alpha)}$ the compression function with compression parameter α when we wish to make it explicit.

We consider Merkle trees using Jive compression functions of possibly different arities at the different layer of the tree. Namely, a Merkle tree with H layers is defined with respect to arities $\alpha_1, \dots, \alpha_L$. It hashes $N = \prod_{i=1}^{H} \alpha_i$ leaves from $\mathbb{F}_q^c$ into one root in $\mathbb{F}_q^c$:

$$\mathsf{MerkleTree} : (x_0, \dots, x_{N-1}) \in (\mathbb{F}_q^c)^N \mapsto y \in \mathbb{F}_q^c \ .$$

At layer $j \in [0, H]$, the state of the tree is composed of N_j digests on $\mathbb{F}_q^c$ where $N_0 = 1$ (the root), $N_L = N$ (the leaves) and $N_j = \prod_{i=1}^{j} \alpha_j$ for $j \in [1, H]$. The N_{j-1} digests of layer $j-1$, denoted $s_0^{(j-1)}$, …, $s_{N_{j-1}-1}^{(j-1)}$, are computed from the N_j digests of layer j, denoted $s_0^{(j)}$, …, $s_{N_j-1}^{(j)}$, as follows

$$s_i^{(j-1)} = \mathsf{Jive}_{\mathcal{P}}(s_{i\alpha_j}^{(j)} \parallel s_{i\alpha_j+1}^{(j)} \parallel \dots \parallel s_{(i+1)\alpha_j-1}^{(j)}) \ , \ \forall i \in [0, N_{j-1}) \ ,$$

with $(s_0^{(H)}, \dots, s_{N-1}^{(H)}) = (x_0, \dots, x_{N-1})$ and $y = s_0^{(0)}$.

One-Way Function using Truncation. To derive a one-way function from $\mathcal{P}$, we use truncation. Specifically, for an input size $|x|$ and an output size $|y|$, the one-way function is defined with respect to an initialization value $iv \in \mathbb{F}_q^{|iv|}$ as follows:

$$\mathsf{OWF}_{\mathcal{P},iv} \ : \ x \in \mathbb{F}_q^{|x|} \ \mapsto \ y = \mathsf{Tr}_{|y|}(P(iv, x)) \in \mathbb{F}_q^{|y|}$$

where $P \in \mathcal{P}$ is of state size $t = |iv| + |x|$. We define those sizes as follows:

$$|x| = |y| = |iv| = \left\lceil \frac{\lambda}{\log_2 q} \right\rceil \ .$$

These parameters ensure the λ-bit security of the above one-way function under the hardness of solving the *constrained-input constrained-output* (CICO) problem [BDPA11, BBL+24]. Moreover, assuming a number of users (substantially) lower than $2^{\lambda/2}$, we should get no (or very few) collisions on the initialization value and hence further obtain nearly λ bits of multi-user security for the one-way function.

2.2 SmallWood Zero-Knowledge Arguments

As previously explained, the signatures produced by the CAPSS framework are non-interactive zero-knowledge message-bound arguments of knowledge of a secret input w such that $y = \mathsf{OWF}_{\mathcal{P},iv}(w)$. Proving knowledge of a preimage under such a one-way function results in a relatively small witness. Therefore, it is desirable to use an argument system well suited for this setting.

When considering hash-based zero-knowledge arguments for relatively small instances, two main approaches can be considered: schemes based on GGM trees, and those based on Merkle trees. In the case of GGM trees, one could rely on frameworks such as VOLE-in-the-Head [BBD+23a] or the GGM-based variant of the Threshold-Computation-in-the-Head (TCitH) framework [FR23]. However, since CAPSS aims to produce SNARK-friendly signatures, GGM-based constructions are less suitable: their verification cost is significant, as verifying a GGM tree involves expanding all but one of its leaves, making verification nearly as expensive as proof generation. For this reason, we favor Merkle-tree-based argument systems, where the cost of verifying a Merkle path scales logarithmically with the tree size. In practice, the CAPSS framework uses the recent proof system SmallWood-ARK [FR25], a Merkle-tree-based scheme specifically designed for arithmetization-oriented statements with relatively small witnesses.

In what follows, we introduce SmallWood-ARK: we begin by describing the format of statements that SmallWood-ARK can prove, then detail the underlying polynomial IOP protocol, and finally present a high-level overview of the associated polynomial commitment scheme.

PACS Statements. SmallWood-ARK is an argument system for *parallel and aggregated constraint systems* (PACS), a generalization of LPPC systems from TCitH and Ligero-style statements. In this syntax, the witness w is arranged as a matrix:

$$w = \begin{bmatrix} w_{1,1} & \dots & w_{1,s} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \dots & w_{n,s} \end{bmatrix}$$

which satisfies two types of constraints:

1. *Parallel polynomial constraints:* for $f_1, \dots, f_{m_1}$ some polynomials from $\mathbb{F}[X_1, \dots, X_n, Y_1, \dots, Y_{n_c}]$ of maximal degree d and $\{\theta_{j,i,k}\}$ constants from $\mathbb{F}$, the witness matrix satisfies:
$$\forall j \in [1, m_1],\ \forall k \in [1, s] \ : \ f_j(w_{1,k}, \dots, w_{n,k}, \theta_{j,1,k}, \dots, \theta_{j,n_c,k}) = 0 \ .$$
2. *Aggregated parallel polynomial constraints:* for $f'_1, \dots, f'_{m_2}$ some polynomials from $\mathbb{F}[X_1, \dots, X_n, Y_1, \dots, Y_{n_c}]$ of maximal degree d' and $\{\theta'_{j,i,k}\}$ constants from $\mathbb{F}$, the witness matrix satisfies:
$$\forall j \in [1, m_2] \ : \ \sum_{k=1}^{s} f'_j(w_{1,k}, \dots, w_{n,k}, \theta'_{j,1,k}, \dots, \theta'_{j,n_c,k}) = 0 \ .$$

Such a constraint system is named a PACS statement [FR25]. The PACS statement is hence defined by the m_1 polynomials $\{f_j\}$ and their associated constants $\{\theta_{j,i,k}\}$ and the m_2 polynomials $\{f'_j\}$ and their associated constants $\{\theta'_{j,i,k}\}$. From this definition, we see that each polynomial constraint f_j is verified in parallel on each column of the witness matrix (possibly with different constants) while, for a constraint of the second type, the sum of f'_j applied to each column is constrained to equal 0.

In this work, we will also consider *global linear constraints* of the form $\sum_{i,k} a_{i,k} w_{i,k} = t$, for some constants $\{a_{i,k}\}$ and t. Let us stress that those linear constraints can be captured as aggregated parallel polynomial constraints of degree 2, with the constants embedded into the θ' arguments of the f'_j function.

PACS PIOP. We now describe the PACS polynomial interactive oracle proof (PIOP) underlying SmallWood-ARK. A PIOP is an interactive proof in which the prover can send a *polynomial oracle* $[P_1, \dots, P_n]$ to the verifier for polynomials $P_1, \dots, P_n \in \mathbb{F}[X]$. From such a polynomial oracle, the verifier can then query some evaluations. Namely, for any point $e \in \mathbb{F}$, a query e to the oracle provides the verifier with the polynomial evaluations $P_1(e), \dots, P_n(e)$. The number of queries made by the verifier is fixed by the definition of the PIOP protocol. The PACS PIOP is designed to prove knowledge of a witness satisfying a PACS statement.

Let $\Omega = \{\omega_1, \dots, \omega_s\}$ some fixed points of $\mathbb{F}$. Let $\mathbb{S} \subseteq \mathbb{F} \setminus \Omega$, the set of evaluation points that can be queried by the verifier (in practice the definition of $\mathbb{S}$ depends on the underlying PCS; $\mathbb{S}$ must exclude Ω for zero-knowledge to be achieved). Let $\{\Theta_{j,i}(X)\}$ and $\{\Theta'_{j,i}(X)\}$ be the degree-$(s-1)$ polynomials defined by interpolation such that $\Theta_{j,i}(\omega_k) = \theta_{j,i,k}$ and $\Theta'_{j,i}(\omega_k) = \theta'_{j,i,k}$ for all j, i, k. Let $\ell' \in \mathbb{N}$ be the number of queries from the verifier to the polynomial oracle and let $\rho \in \mathbb{N}$ some security parameter. Finally, we let d_Q be defined as:

$$d_Q = \max\left(d \cdot (\ell + s - 1) + s - 1\ ,\ d' \cdot (\ell + s - 1)\right) \ , \tag{1}$$

where, as introduced above, $d = \max_j \deg(f_j)$ and $d' = \max_j \deg(f'_j)$.

The PACS IOP is described in detail in Protocol 2. Let us outline its high-level structure. The prover begins by interpolating each row $(w_{i,1}, \ldots, w_{i,s})$ of the witness matrix into a random degree-$(\ell' + s - 1)$ polynomial $P_i \in \mathbb{F}[X]$ over the evaluation domain $\Omega = \{\omega_1, \ldots, \omega_s\}$. The prover additionally samples random polynomials $M_1, \ldots, M_\rho$ of degree d_Q (as defined in Equation (1)), subject to the constraint $\sum_{\omega \in \Omega} M_k(\omega) = 0$ for all $k \in [1, \rho]$. The prover then sends a polynomial oracle $[\boldsymbol{P}, \boldsymbol{M}]$ to the verifier, where $\boldsymbol{P} = (P_1, \ldots, P_n)$ and $\boldsymbol{M} = (M_1, \ldots, M_\rho)$.

Next, the verifier samples random polynomials $\Gamma'_{k,j}(X) \in \mathbb{F}[X]^{(\leq s-1)}$, for all $k \in [1, \rho]$ and $j \in [1, m_1]$, as well as random scalars $\gamma'_{k,j} \in \mathbb{F}$ for all $k \in [1, \rho]$ and $j \in [1, m_2]$. These are sent to the prover under the form of a string $\Gamma' \in \mathbb{F}^{|\Gamma'|}$ where $|\Gamma'| = \rho \cdot (m_1 \cdot s + m_2)$. Using these random values, the prover computes the vector of polynomials $\boldsymbol{Q} = (Q_1, \ldots, Q_\rho)$ where:

$$Q_k(X) = M_k(X) + \sum_{j=1}^{m_1} \Gamma'_{k,j}(X) \cdot F_j(X) + \sum_{j=1}^{m_2} \gamma'_{k,j} \cdot F'_j(X) \ , \tag{2}$$

with

$$F_j(X) := f_j(P_1(X), \ldots, P_n(X), \Theta_{j,1}(X), \ldots, \Theta_{j,n_c}(X)) \ ,$$

and

$$F'_j(X) := f'_j(P_1(X), \ldots, P_n(X), \Theta'_{j,1}(X), \ldots, \Theta'_{j,n_c}(X)) \ .$$

The prover sends $\boldsymbol{Q}$ to the verifier, who checks two conditions: (1) that the vector polynomial $\boldsymbol{Q}$ is consistent with the oracle $[\boldsymbol{P}, \boldsymbol{M}]$ on randomly chosen evaluation points, and (2) that the constraint $\sum_{\omega \in \Omega} Q_i(\omega) = 0$ holds for all $i \in [1, \rho]$.

1. $\mathcal{P}$ builds random polynomials $P_1, \ldots, P_n \in \mathbb{F}[X]^{(\leq \ell' + s - 1)}$ defined such that
$$\forall\, i \in [1, n],\ P_i(\omega_1) = w_{i,1}, \ldots, P_i(\omega_s) = w_{i,s} \ .$$
2. $\mathcal{P}$ samples ρ random polynomials $M_1, \ldots, M_\rho \in \mathbb{F}[X]^{(\leq d_Q)}$ such that $\sum_{\omega \in \Omega} M_j(\omega) = 0$ for all j.
3. $\mathcal{P}$ sends a polynomial oracle $[\boldsymbol{P}, \boldsymbol{M}]$ to $\mathcal{V}$, where $\boldsymbol{P} = (P_1, \ldots, P_n)$, $\boldsymbol{M} = (M_1, \ldots, M_\rho)$.
4. $\mathcal{V}$ samples $\Gamma' \leftarrow \mathcal{D}_{\Gamma'}$, for $\mathcal{D}_{\Gamma'}$ a probability distribution over $\mathbb{F}^{|\Gamma'|}$ with $|\Gamma'| = \rho \cdot (m_1 \cdot s + m_2)$. $\mathcal{V}$ sends Γ' to $\mathcal{P}$.
5. $\mathcal{V}$ and $\mathcal{P}$ parse Γ' as $\rho \cdot m_1$ polynomials $\Gamma'_{i,j}(X) \in \mathbb{F}[X]^{(\leq s-1)}$, for $i \in [1, \rho]$ and $j \in [1, m_1]$, and $\rho \cdot m_1$ field elements $\gamma'_{i,j} \in \mathbb{F}$ for $i \in [1, \rho]$ and $j \in [1, m_2]$.
6. $\mathcal{P}$ computes the polynomial Q_i defined by Equation (2), for all $1 \leq i \leq \rho$. $\mathcal{P}$ sends $\boldsymbol{Q} = (Q_1, \ldots, Q_\rho)$ to $\mathcal{V}$.
7. $\mathcal{V}$ samples $E' \leftarrow \binom{\mathbb{S}}{\ell'}$ (random subset $E' \subseteq \mathbb{S}$ of size $|E'| = \ell'$) and queries the oracle $[\boldsymbol{P}, \boldsymbol{M}]$ for evaluations of the polynomials at the points in E'.
8. $\mathcal{V}$ checks that
   (a) $\boldsymbol{Q}$ verifies Equation (2) for all $i \in [1, \rho]$ for the opened evaluations at the points in E',
   (b) $\boldsymbol{Q}$ verifies $\sum_{\omega \in \Omega} Q_i(\omega) = 0$, for every $i \in [1, \rho]$.

Fig. 2: The PACS Polynomial IOP.

SmallWood-PCS. [FR25] proposes a hash-based zero-knowledge polynomial commitment scheme tailored for relatively small polynomials, namely SmallWood-PCS. This PCS is built upon the degree-enforcing commitment scheme (DECS) from [FR23], using techniques from [BCG20, Lee21, GLS+23]. We refer the reader to [FR25] for details about SmallWood-PCS. Below, we briefly recall how this scheme makes use of Merkle trees.

The commitment procedure of SmallWood-PCS calls the DECS' commitment procedure as subroutine, with some polynomials $P'_1(X), \ldots P'_{n_{\mathsf{decs}}}(X)$ as inputs (which are related but different from the polynomials $P_1(X), \ldots, P_n(X)$ involved in Figure 2). This subroutine samples some masking polynomials $M'_1(X), \ldots, M'_\eta(X)$ and computes the hash digests

$$u_i \leftarrow \mathsf{XOF}(P'_1(e_i), \ldots, P'_{n_{\mathsf{decs}}}(e_i), M'_1(e_i), \ldots, M'_\eta(e_i)) \qquad (3)$$

for all $i \in [1, N]$, where N is a DECS parameter and e_i's are public fixed distinct evaluation points. It then computes the Merkle tree that has $\{u_i\}_i$ as leaves. In parallel, the verification algorithm of SmallWood-PCS (i.e. the algorithm that checks if a evaluation opening is valid) calls the DECS' verification procedure as subroutine, with some evaluations $\{P'_1(e_i), \ldots, P'_{n_{\mathsf{decs}}}(e_i), M'_1(e_i), \ldots, M'_\eta(e_i)\}_{i \in I}$ and an authentication path as inputs, where I is a subset of $[1, N]$ of size ℓ, with ℓ a SmallWood-PCS parameter. This subroutine computes u_i for all $i \in I$ using Equation 3 and then recomputes the root of the Merkle tree using the given authentication path.

SmallWood-ARK. The PACS PIOP can be compiled into an interactive proof system for PACS statements by replacing oracle accesses with polynomial commitments instantiated via SmallWood-PCS. Applying the Fiat-Shamir transform to this interactive protocol yields a non-interactive argument system, which corresponds to the SmallWood-ARK scheme. The security of the latter is analyzed in [FR25] and is stated below.

Theorem 1 ([FR25]). *In the Random Oracle Model (ROM), where the underlying hash and extendable-output functions are modeled as independent random oracles, SmallWood-ARK is straightline-extractable knowledge sound.*

We refer the reader to [FR25] for the explicit formula of the soundness error. In what follows, we instantiate SmallWood-ARK with concrete parameters that achieve a 128-bit security level, based on the soundness error expression provided in [FR25].

3 Arithmetization

As overviewed in Section 2, the CAPSS framework relies on SmallWood-ARK, the zero-knowledge argument scheme for PACS statements presented in Section 2.2. In the PACS syntax (c.f. Section 2.2), the witness is an $n \times s$ matrix which is proved to satisfy polynomial constraints applied to each column of the witness matrix in parallel and aggregated parallel polynomial constraints. In practice, we will consider global linear constraints, i.e. constraints applied globally on the (flattened) witness matrix, instead of aggregated parallel polynomial constraints. As stated in Section 2.2, such constraints are a subset of aggregated parallel polynomial constraints. In our context, the *arithmetization* of a permutation $P \in \mathcal{P}$ is the process of expressing a statement $(y, z) = P(iv, x)$ as an PACS statement, where x and z are secret (part of the witness) and iv and y are public (part of the PACS boundary constraints).

In what follows, we propose two different PACS arithmetization techniques applying to a wide-range of arithmetization-oriented families of permutations $\mathcal{P}$.

3.1 Arithmetization of Regular Permutations

We focus on the case of *regular permutations* which we define as permutations with a regular iterated round structure. Precisely, a regular permutation P has the following form:

$$P(\,\cdot\,) = R_{n_r-1} \circ \ldots R_1 \circ R_0(\,\cdot\,) \quad \text{with} \quad R_i(\,\cdot\,) = F(\,\cdot\,, c_i)$$

for a round permutation F and round constants $c_0, \ldots, c_{n_r-1}$, where n_r is the number of rounds. Many arithmetization-oriented families of permutations match this format, such as RescuePrime [AAB+20, SAD20], Griffin [GHR+23] and Anemoi [BBC+23]. In the following, we shall denote $x_0, \ldots, x_{n_r} \in \mathbb{F}^t$ the successive states arising in a statement $(y, z) = P(iv, x)$. Namely, we have:

$$\forall\, 0 \le i \le n_r,\ x_i = \begin{cases} (iv, x) & \text{if } i = 0 \\ F(x_{i-1}, c_{i-1}) & \text{otherwise} \end{cases}$$

and $x_{n_r} = (y, z)$. We further let G be a *verification function* of the round function F, that is a polynomial function satisfying:

$$x_{i+1} = F(x_i, c_i) \iff \exists\, v_i \in \mathbb{F}^{|v|} : G(v_i, x_i, x_{i+1}, c_i) = 0 \ ,$$

and denote α the degree of G. It is usual for arithmetization-oriented permutations to have such a verification function with low degree α. While an obvious verification function is $G(x_i, x_{i+1}, c_i) = x_{i+1} - F(x_i, c_i)$, which has same degree as F, some permutation designs rely on a round function F of large degree that has a verification function G of low degree α (making the SNARK verification of the function efficient).

For the sake of simplicity, let us first consider that the PACS packing factor s (i.e., the number of columns in the PACS witness matrix) divides the number of rounds. We will relax this assumption later on. Namely, there exists $b \in \mathbb{N}$ such that $n_r = b \cdot s$. We then define n, the number of rows in the PACS witness matrix, to be

$$n := (b + 1) \cdot t + b \cdot |v| \ .$$

The PACS witness matrix is defined as:

$$\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,s} \\ w_{2,1} & \cdots & & w_{2,s} \\ \vdots & & & \vdots \\ w_{n,1} & w_{n,2} & \cdots & w_{n,s} \end{bmatrix} := \begin{bmatrix} x_0 & x_b & \cdots & x_{(s-1)b} \\ x_1 & x_{b+1} & \cdots & x_{(s-1)b+1} \\ \vdots & \vdots & & \vdots \\ x_b & x_{2b} & \cdots & x_{s\cdot b} \\ v_0 & v_b & \cdots & v_{(s-1)b} \\ \vdots & \vdots & & \vdots \\ v_{b-1} & v_{2b-1} & \cdots & v_{s\cdot b-1} \end{bmatrix} .$$

In the right-hand side of the above equation, the x_i's are column vectors of length t and the v_i's are column vectors of length $|v_i| := |v|$, while the $w_{i,j}$ coefficients of the left-hand side are field elements. We well have a matrix with $n = (b + 1) \cdot t + b \cdot |v|$ rows.

Let us now define the PACS constraints to be verified on the above witness matrix to ensure that the statement $(y, z) = P(iv, x)$ is satisfied. We have $m_1 = b$ parallel polynomial constraints $f_0, \ldots, f_{b-1}$, defined as:

$$f_j : (\underbrace{\hat{x}_0, \ldots, \hat{x}_b, \hat{v}_0, \ldots, \hat{v}_{b-1}}_{\text{witness column}}, \underbrace{\hat{c}_0, \ldots, \hat{c}_{b-1}}_{\text{constants}}) \mapsto G(\hat{v}_j, \hat{x}_j, \hat{x}_{j+1}, \hat{c}_j)$$

for every $j \in [0, b - 1]$. Applied to the first column of the witness matrix with constants $c_0, \ldots, c_{b-1}$, these polynomial constraints check the correctness of the b first state transitions $x_0 \to x_1 \to \cdots \to x_b$. Indeed, by definition, we have:

$$f_j(x_0, \ldots, x_b, v_0, \ldots, v_{b-1}, c_0, \ldots, c_{b-1}) \quad \Leftrightarrow \quad x_{j+1} = F(x_j, c_j) \ .$$

In the same way, applied to the k^{th} column of the witness matrix with constants $c_{(k-1)b}, \ldots, c_{kb-1}$, these polynomial constraints check the correctness of state transitions $x_{(k-1)b} \to x_{(k-1)b+1} \to \cdots \to x_{kb}$.

This way, using $m_1 = b$ parallel polynomial constraints, we verify all the state transitions. However, we should still make sure that subsequent columns are consistent namely that the vector x_b of the first witness column is equal to the vector x_b of the second witness column and, more generally, that the vector x_{kb} in the k^{th} column equals the vector x_{kb} in the $(k+1)^{\text{th}}$ column for every $k \in [1, s - 1]$. To this purpose, we can use the following global linear constraints:

$$\forall\, i \in [0, t - 1], \forall\, k \in [1, s - 1] \ : \ w_{bt+i,k} - w_{i,k+1} = 0 \ .$$

Combining the previous parallel polynomial constraints and the above global linear constraints, the witness is ensured to satisfy $x_{n_r} = R_{n_r-1} \circ \ldots R_1 \circ R_0(x_0)$. An additional $|x| + |iv|$ global linear constraints, the *boundary constraints*, need to be added in order to check

$$\begin{cases} \mathsf{Tr}_{|iv|}(x_0) = iv \\ \mathsf{Tr}_{|y|}(x_{n_r}) = y \end{cases}$$

which finally ensures that there exists $x \in \mathbb{F}^{|x|}$ and $z \in \mathbb{F}^{t-|y|}$ such that $(y, z) = P(iv, x)$. We this obtain an PACS statement for $(y, z) = P(iv, x)$ with $m_1 = b$ parallel polynomial constraints and $m_2 = (s-1)\cdot t + |iv| + |y|$ global linear constraints.

Let us now assume that the packing factor s does not divide the number n_r of rounds. In that case, we take $b \in \mathbb{N}$ minimal such that $n_r \leq b \cdot s$ and we proceed exactly as previously while padding the witness for $x_{n_r+1}, \ldots, x_{b\cdot s}$. We should just be careful that the padded values do not prevent to the witness matrix of satisfying the polynomial constraints. One possible option is to define $x_{n_r+1}, \ldots, x_{b\cdot s}$ as

$$\begin{aligned} x_{n_r+1} &= F(x_{n_r}, 0), \\ &\vdots \\ x_{b\cdot s} &= F(x_{b\cdot s-1}, 0). \end{aligned}$$

3.2 S-Box-Centric Arithmetization

We propose an alternative, simpler, arithmetization technique for permutations which do not have a regular structure but rely on a single unitary S-box $S : \mathbb{F} \to \mathbb{F}$. Namely, we consider a permutation P which is solely composed of $\mathbb{F}$-linear operations and n_{sbx} calls to S. We can for instance express the Poseidon permutation [GKR+21] in this format (while it is not a regular permutation due to the usage of two different type of rounds: the full rounds and the partial rounds).

Let G be a degree-α verification function of the S-box S, which satisfies:

$$y_i = S(x_i) \iff \exists\, v_i \in \mathbb{F}^{|v|} : G(v_i, x_i, y_i) = 0$$

for some $|v| \in \mathbb{N}$. The idea of the S-box-centric arithmetization is that the witness contains the inputs and outputs of all the S-box calls in P. Then using the parallel polynomial constraints, we can batch the verification of the S-box relations, while using global linear constraints we can check the $\mathbb{F}$-linear operations and the boundary relations.

Let us assume that the packing factor divides the number of S-boxes, i.e. there exists $n' \in \mathbb{N}$ such that $n_{\mathsf{sbx}} = s \cdot n'$. We will relax this assumption later on. For the $((i-1)\cdot s + (j-1) + 1)^{\text{th}}$ S-box with $1 \leq i \leq n'$ and $1 \leq j \leq s$, we denote its input $x_{i,j}$, its output $y_{i,j}$ and its verification witness $v_{i,j}$ such that $G(v_{i,j}, x_{i,j}, y_{i,j}) = 0$. The PACS witness matrix of $\mathbb{F}^{n\times s}$ is defined as follows:

$$\begin{bmatrix} w_{1,1} & w_{1,2} & \ldots & w_{1,s} \\ w_{2,1} & \ldots & & w_{2,s} \\ \vdots & & & \vdots \\ w_{n,1} & w_{n,2} & \ldots & w_{n,s} \end{bmatrix} := \left[\begin{array}{c|c|c|c} v_{1,1} & v_{1,2} & & v_{1,s} \\ x_{1,1} & x_{1,2} & \ldots & x_{1,s} \\ y_{1,1} & y_{1,2} & & y_{1,s} \\ \hline v_{2,1} & v_{2,2} & & v_{2,s} \\ x_{2,1} & x_{2,2} & \ldots & x_{2,s} \\ y_{2,1} & y_{2,2} & & y_{2s} \\ \hline \vdots & \vdots & & \vdots \\ \hline v_{n',1} & v_{n',2} & & v_{n',s} \\ x_{n',1} & x_{n',2} & \ldots & x_{n',s} \\ y_{n',1} & y_{n',2} & & y_{n',s} \end{array}\right],$$

with $n := (2 + |v|) \cdot n'$. For $1 \leq j \leq n'$, the j^{th} parallel polynomial constraint is

$$\forall 1 \leq k \leq s,\ G(v_{j,k}, x_{j,k}, y_{j,k}) = 0\ .$$

The j^{th} parallel polynomial constraint simultaneously implies $y_{j,1} = S(x_{j,1}), \ldots, y_{j,s} = S(x_{j,s})$. The n' parallel polynomial constraints thus verify the input-output relation for all the S-boxes. It remains to check the $\mathbb{F}$-linear operations of the permutations. By definition of the permutation P, there exists two matrices $A_1^{(P)}, A_2^{(P)} \in \mathbb{F}^{(n_{\mathsf{sbx}}-t)\times n_{\mathsf{sbx}}}$ and a vector $b^{(P)} \in \mathbb{F}^{n_{\mathsf{sbx}}-t}$ such that

$$A_1^{(P)} \cdot (x_{1,1} \ldots x_{1,s} \ldots x_{n',1} \ldots x_{n',s})^\top = A_2^{(P)} \cdot (y_{1,1} \ldots y_{1,s} \ldots y_{n',1} \ldots y_{n',s})^\top + b^{(P)} \ . \tag{4}$$

The global linear constraints check the n_{sbx} linear relations induced by (4), together with the boundary linear constraints of the form

$$\begin{cases} iv = A_3^{(P)} \cdot (x_{1,1} \ldots x_{1,s} \ldots x_{n',1} \ldots x_{n',s})^\top + b'_P \\ y \ = A_4^{(P)} \cdot (y_{1,1} \ldots y_{1,s} \ldots y_{n',1} \ldots y_{n',s})^\top + b''_P \end{cases}$$

for some matrices $A_3^{(P)} \in \mathbb{F}^{|iv|\times n_{\mathsf{sbx}}}$, $A_4^{(P)} \in \mathbb{F}^{|y|\times n_{\mathsf{sbx}}}$ and vectors $b'_P \in \mathbb{F}^{|iv|}$, $b''_P \in \mathbb{F}^{|y|}$. This makes a total of $m_1 = n'$ parallel polynomial constraints and $m_2 = n_{\mathsf{sbx}} - t + |iv| + |y|$ global linear constraints.

Let us now assume that the packing factor s does not divide the number n_{sbx} of calls to S. In that case, we take $n' \in \mathbb{N}$ minimal such that $n_{\mathsf{sbx}} \leq s \cdot n'$ and we proceed exactly as previously while padding the witness for $(v_{i,j}, x_{i,j}, y_{i,j})$ when $(i-1)\cdot s + (j-1) + 1 > n_{\mathsf{sbx}}$. We should just be careful that the padded values do not prevent to the witness matrix of satisfying the polynomial constraints. One possible option is to define those $(v_{i,j}, x_{i,j}, y_{i,j})$ as

$$(v_{i,j}, x_{i,j}, y_{i,j}) = (v_0, 0, y_0) \ ,$$

where $y_0 := S(0)$ and v_0 is the value satisfying $G(v_0, 0, y_0) = 0$.

4 SNARK-Friendly Verification of SmallWood Arguments

As mentioned previously, the CAPSS framework relies on the SmallWood-ARK scheme recalled in Section 2.2. The goal of this framework is to build SNARK-friendly post-quantum signature schemes featuring lightweight verification algorithm while expressed as an arithmetic circuit or in terms of SNARK constraints for some arithmetization system, typically R1CS. By inspecting the verification process in a CAPSS signature, one can see that the bottleneck comes from the verification of the Merkle authentication paths as well as the XOF calls to hash the Merkle leaves and generate verifier challenges through Fiat-Shamir. In what follows, we propose several design tweaks to mitigate this bottleneck and reduce the number of SNARK constraints arising in the verification algorithm.

4.1 Parameter Trade-offs in Merkle Trees

The main part of the verification algorithm in terms of SNARK constraints is the decommitment of the polynomial evaluations in the DECS, i.e., the verification of the underlying Merkle authentication paths. Let us denote $P'_1, \ldots, P'_{n_{\mathsf{decs}}}$ the polynomials committed using the DECS. As explained in Section 2.2, the DECS verification gets as input ℓ evaluations

$$\{P'_1(e_i), \ldots, P'_{n_{\mathsf{decs}}}(e_i), M'_1(e_i), \ldots, M'_\eta(e_i)\}_{i\in I} \ ,$$

with $|I| = \ell$, where $M'_1, \ldots, M'_\eta$ are masking polynomials committed along with $P'_1, \ldots, P'_{n_{\mathsf{decs}}}$. It computes the hash digests $\{u_i\}_{i\in I}$ of these evaluations following (3) and verify they are the correct leaves of the committed Merkle root using the associated authentication paths.

This verification computation cost can be lowered in two different ways:

- *Lowering the leaf hash computation.* In the verification algorithm, since we open ℓ evaluations in the DECS (used as subroutine of the PCS), we need to recompute ℓ hash digests. To mitigate this cost, we can adapt the parameters of SmallWood-ARK to decrease the number n_{decs} of polynomials committed using the DECS, while increasing a bit the signature size. Reducing n_{decs} lightens the hash computation, as the size of the hashed elements scales with n_{decs}.

– *Shortening the Merkle authentication paths.* While binary Merkle trees are optimal in terms of size – minimizing the number of hash digests in an authentication path – Merkle trees with larger arity reduce the path length in terms of the number of hash nodes. This, in turn, decreases the number of SNARK constraints required for verification. We hence consider Merkle trees of arity possibly greater than 2, which provides a trade-off between signature size (shortened with binary trees) and SNARK constraints (reduced with larger arity). We further consider that the arity parameter might vary with the node depth in the tree for more flexibility in the parameter selection.

4.2 Trimming Authentication Paths

Once the verifier recomputed all the opened leaves (by hashing the opened values), they need to check those leaves using the authentication paths. Since verifying authentication paths in a Merkle tree is not constant time due to the path merging, preventing us to write the verification algorithm as an arithmetic circuit, we will need to verify each path one by one. To proceed, one needs to decompress the merged authentication paths into ℓ individual authentication paths. This decompression step can be made outside the verification circuit. Then, the verification circuit take those ℓ individual authentication paths as inputs and check that the correct Merkle root is recomputed from each leaf and its associated Merkle path.

However, this strategy is not optimal in terms of computation. Since it checks each leaf independently, some nodes are recomputed several times. For example, we recompute the root from its children ℓ times (at each independent checking). In fact, the more a node is close to the root, the more often it is recomputed. To avoid this redundancy, we rely on an alternative strategy. We introduce an additional parameter $\gamma_{\mathsf{MT}} \in \{0, \ldots, H\}$. Instead of taking the ℓ individual authentication paths, the circuit inputs shall contain all the nodes of depth γ_{MT}, together with the ℓ truncated authentication paths. The verification circuit then recomputes the root from the depth-γ_{MT} nodes *only once* and checks each truncated individual authentication path independently. This strategy saves the computation of $\ell \cdot \gamma_{\mathsf{MT}}$ nodes (trimmed parts of the ℓ paths) at the cost of computing $\sum_{i=0}^{\gamma_{\mathsf{MT}}-1} \prod_{j=0}^{i-1} \alpha_j$ nodes (full bottom of the tree), where α_j denotes the arity of the nodes at depth j in the tree. The parameter γ_{MT} is chosen to optimize this trade-off.

To enable the verifier to compute all the nodes of depth γ_{MT}, we should slightly tweak the algorithm that generates the authentication paths. We describe all the routines for the Merkle tree in Figure 3:

– The routine MerkleTree computes the root from the N leaves. It does not involve any tweak.
– The routine GetAuthPath computes the authentication to the depth-γ_{MT} nodes. The only difference with the non-tweaked algorithm is that the loops (Steps 2 and 4) over the Merkle layers should stop earlier, at the depth-γ_{MT} layer instead than at the root layer. Even if those tweaked path enables the verifier to recompute the depth-γ_{MT} nodes, let us stress that it does not prevent him to recompute the root (by simply recomputing the head of the Merkle tree).
– The routine RetrieveRootFromPath recomputes the root from the tweaked authentication paths. As the previous routine, the only difference is that the layer loops at Steps 2 and 5 stop earlier.

4.3 Decomposition of the DECS Opening Challenge

At some point in SmallWood-PCS, the verifier needs to choose a random set I of size ℓ, which corresponds to the indexes of the opened evaluation points $\{e_i\}_{i\in I}$ for the DECS scheme. Since we rely on the Fiat-Shamir transformation, it implies that the verification algorithm of the signature scheme needs to derive this challenge from the output of an extendable-output hash function (XOF).

Let us investigate how to arithmetize this opening challenge when working over a 256-bit field and targeting a 128-bit soundness. In that setting, the verifier's challenge is a single field element $v \in \mathbb{F}_q$, i.e. the corresponding sponge-based XOF outputs a single field element $v \in \mathbb{F}_q$. We want to transform v as a set $I \subseteq [1, N]$ of size ℓ (without duplicates). Moreover, we want this hashing to integrate the grinding parameter κ_4, namely this hashing should integrate a κ_4-bit proof-of-work as in SmallWood-ARK, see [FR25,

```
MerkleTree($c_1, \dots, c_N$):
On input $N$ leaves $c_1, \dots, c_N$:
  1. Set $\mathsf{tree}[h_{\mathsf{MT}}][0] = c_1, \dots, \mathsf{tree}[h_{\mathsf{MT}}][N-1] = c_N$.
  2. For $h$ from $h_{\mathsf{MT}} - 1$ downto 0, for $i$ from 0 to $N_h$, compute
       $\mathsf{tree}[h][i] = \mathsf{Jive}(\mathsf{tree}[h+1][i \cdot \alpha_h] \| \dots \| \mathsf{tree}[h+1][(i+1) \cdot \alpha_h - 1])$.
  3. Set $\mathsf{root} = \mathsf{tree}[0][0]$.
  4. Return $(\mathsf{tree}, \mathsf{root})$.

GetAuthPath($\mathsf{tree}, I$):
On input a Merkle tree $\mathsf{tree}$ and a query index set $I := \{i_1, \dots, i_\ell\}$:
  1. Set $\mathsf{missing} \leftarrow \{(h_{\mathsf{MT}}, i), i \notin I\}$.
  2. For $h$ from $h_{\mathsf{MT}} - 1$ downto $\gamma_{\mathsf{MT}}$, for $i$ from 0 to $N_h$,
     – If $L := \{(h+1, i \cdot \alpha_h + j), 0 \le j < \alpha_h\} \subset \mathsf{missing}$, update $\mathsf{missing}$ as
         $\mathsf{missing} \leftarrow (\mathsf{missing} \backslash L) \cup \{(h, i)\}$.
  3. Set $\mathsf{auth} \leftarrow \emptyset$.
  4. For $h$ from $h_{\mathsf{MT}} - 1$ downto $\gamma_{\mathsf{MT}}$, for $i$ from 0 to $N_h$,
     – If $(h, i) \in \mathsf{missing}$, update $\mathsf{auth}$ as
         $\mathsf{auth} \leftarrow (\mathsf{auth} \| \mathsf{tree}[h][i])$.
  5. Return $\mathsf{auth}$.

RetrieveRootFromPath($\{c_{i_j}\}_j, \mathsf{auth}, I$):
On input a query index set $I := \{i_1, \dots, i_\ell\}$, the opened leaves $\{c_{i_j}\}_j$ and an authentication path:
  1. Set $\mathsf{missing} \leftarrow \{(h_{\mathsf{MT}}, i), i \notin I\}$.
  2. For $h$ from $h_{\mathsf{MT}} - 1$ downto $\gamma_{\mathsf{MT}}$, for $i$ from 0 to $N_h$,
     – If $L := \{(h+1, i \cdot \alpha_h + j), 0 \le j < \alpha_h\} \subset \mathsf{missing}$, update $\mathsf{missing}$ as
         $\mathsf{missing} \leftarrow (\mathsf{missing} \backslash L) \cup \{(h, i)\}$.
  3. Set $\mathsf{tree}[h][i] = \mathsf{null}$ for all $(h, i)$.
  4. Set $\mathsf{tree}[h_{\mathsf{MT}}][i_1 - 1] = c_{i_1}, \dots, \mathsf{tree}[h_{\mathsf{MT}}][i_\ell - 1] = c_{i_\ell}$.
  5. For $h$ from $h_{\mathsf{MT}} - 1$ downto $\gamma_{\mathsf{MT}}$, for $i$ from 0 to $N_h$,
     – If $(h, i) \in \mathsf{missing}$, update $\mathsf{auth}$ as
         $(\mathsf{tree}[h][i] \| \mathsf{auth}) \leftarrow \mathsf{auth}$.
  6. For $h$ from $h_{\mathsf{MT}} - 1$ downto 0, for $i$ from 0 to $N_h$
     – If $\mathsf{tree}[h+1][i \cdot \alpha_h] \neq \mathsf{null}$, compute
         $\mathsf{tree}[h][i] = \mathsf{Jive}(\mathsf{tree}[h+1][i \cdot \alpha_h] \| \dots \| \mathsf{tree}[h+1][(i+1) \cdot \alpha_h - 1])$.
  7. Set $\mathsf{root} = \mathsf{tree}[0][0]$.
  8. Return $\mathsf{root}$.
```

Fig. 3: Merkle-tree routines in the CAPSS framework. We denote N the number of leaves, H the height, α_j the arity at depth j, N_j the number of nodes at depth j ($N_0 = 1, N_1 = \alpha_0, \dots, N_H = \alpha_1 \times \dots \times \alpha_{H-1} = H$). Moreover γ_{MT} is an additional parameter allowing the verifier to recover all the depth-γ_{MT} nodes from the authentication paths.

Section 5.3]. Let us recall that α_j denotes the arity of the nodes at depth j in the Merkle tree, while N is the number of tree leaves. We decompose v as follows:

$$v = \sum_{j=0}^{\ell-1} \left(\sum_{i=0}^{h_{\text{MT}}-1} \left(\sum_{k=0}^{\alpha_i - 1} b_{j,i,k} \cdot k \right) \cdot \prod_{k=0}^{i-1} \alpha_i \right) N^j + \left(\sum_{j=0}^{n_{\text{decomp}}-1} b'_j \cdot 2^j \right) N^\ell$$

where

- $b_{j,i,k}$ and b'_j are binary values for all (i,j,k),
- $b_{j,i,0} + \ldots + b_{j,i,\alpha_i} = 1$ for all (i,j),
- $\sum_{i=0}^{h_{\text{MT}}-1} \left(\sum_{k=0}^{\alpha_i-1} b_{j,i,k} \cdot k \right) \cdot \prod_{k=0}^{i-1} \alpha_i$ are distinct for all j,
- $N = \alpha_0 \times \ldots \times \alpha_{h_{\text{MT}}-1}$ is the number of leaves of each Merkle tree,
- n_{decomp} is the largest value such that $2^{\kappa_4 + n_{\text{decomp}}} \cdot N^\ell \leq |\mathbb{F}|$.

Using this decomposition, the j^{th} opened leaf in the Merkle tree is the leaf of index

$$\mathsf{ind}_j := \sum_{i=0}^{h_{\text{MT}}-1} \left(\sum_{k=0}^{\alpha_i-1} b_{j,i,k} \cdot k \right) \cdot \prod_{k=0}^{i-1} \alpha_i \ .$$

The goal this decomposition is to ease the verification of the authentication path within an arithmetic circuit. The revealed child of the node at depth i is the child number $\sum_{k=0}^{\alpha_i-1} b_{j,i,k} \cdot k$, where $(b_{j,i,0}, \ldots, b_{j,i,\alpha_i})$ is an elementary vector (all the coefficients are zero, except one which is one). Let us denote x the value of the child number $\sum_{k=0}^{\alpha_i-1} b_{j,i,k} \cdot k$ and $y_1, \ldots, y_{\alpha_i}$ all the children of the parent node. To check that x is indeed well located among $y_1, \ldots, y_{\alpha_i}$, we can check:

$$\forall 0 \leq k < \alpha_i, \ b_{j,i,k} \cdot (y_k - x) = 0 \ .$$

This way, we can decompose any number from $\mathcal{S} := \{0, \ldots, 2^{n_{\text{decomp}}} \cdot N^\ell - 1\}$ except those leading two identical opened leaves (i.e. where $\mathsf{ind}_i = \mathsf{ind}_j$ for some $i \neq j$). It means that all the other values will be rejected. The rejection rate is then $\frac{|\mathcal{S}|}{|\mathbb{F}|} \approx \frac{N^\ell \cdot 2^{n_{\text{decomp}}}}{|\mathbb{F}|}$, which, by definition of n_{decomp}, satisfies

$$2^{-\kappa_4 - 1} \leq \frac{N^\ell \cdot 2^{n_{\text{decomp}}}}{|\mathbb{F}|} \leq 2^{-\kappa_4}$$

hence providing κ_4-bit proof of work complying to the selected grinding parameter κ_4.

When working on smaller fields (smaller than 256-bit fields), the verifier's challenge is not a single field element anymore, but a vector of field elements. We can proceed as previously: each field element is decomposed as a small subset of $[1, N]$, which together form the subset $I \subset [1, N]$ of size ℓ.

4.4 Batching using Powers

Besides the DECS opening challenge – which we addressed above – and the PIOP opening challenge E' (see Figure 2) – which is simple to handle –, the proof system includes a two random combination challenges from the verifier: the batching challenge Γ' used to build the polynomials $\boldsymbol{Q}$ in the PIOP and another challenge involved inside the DECS (namely, the challenge Γ of the degree-enforcing round – see [FR25, Section 3]). After application of the Fiat-Shamir transform, all the underlying random coefficients are generated through a sponge-based XOF. The lower the number of those generated random coefficients, the lower the number of permutation calls in these Fiat-Shamir XOF computations.

The random challenge Γ' in the PIOP protocol aims to batch all the PACS constraints. Γ' is composed of ρ random vectors $\bar{\gamma}^{(1)}, \ldots, \bar{\gamma}^{(\rho)}$ of size $m_1 \cdot s + m_2$. For every $k \in [1, \rho]$, the polynomial Q_k is built such that

$$\sum_{i=1}^{s} Q_k(\omega_i) = \sum_{i=1}^{s} M_k(\omega_i) + \sum_{j=1}^{m_1} \sum_{i=1}^{s} \bar{\gamma}^{(k)}_{(j-1)\cdot s + i} \cdot F_j(\omega_i) + \sum_{j=1}^{m_2} \bar{\gamma}^{(k)}_{m_1 \cdot s + j} \cdot \sum_{i=1}^{s} F'_j(\omega_i) \ ,$$

where

$$F_j(\omega_i) := f_j(P_1(\omega_i), \ldots, P_n(\omega_i), \theta_{j,1,i}, \ldots, \theta_{j,n_c,i}) \ ,$$

and

$$F'_j(\omega_i) := f'_j(P_1(\omega_i), \ldots, P_n(\omega_i), \theta_{j,1,i}, \ldots, \theta_{j,n_c,i}) \ .$$

A witness satisfying the PACS statement implies $F_j(\omega_i) = 0$ for all (j, i) and $\sum_{i=1}^{s} F'_j(\omega_i) = 0$ for all j. Therefore, $\sum_{i=1}^{s} Q_k(\omega_i)$ is equal to zero independently on the values in $\bar{\gamma}^{(k)}$, assuming that $\sum_{i=1}^{s} M_k(\omega_i) = 0$. In case of an invalid witness, there would exist a (j^*, i^*) such that $F_{j^*}(\omega_{i^*}) \neq 0$ or a j^* such that $\sum_{i=1}^{s} F'_{j^*}(\omega_i) \neq 0$. The value of $\sum_{i=1}^{s} Q_k(\omega_i)$ will then be non-zero with high probability. The exact probability that $\sum_{i=1}^{s} Q_k(\omega_i)$ is zero depends on the probability distribution of the vector $\bar{\gamma}^{(k)}$: it corresponds to the probability that $\langle \bar{\gamma}^{(k)}, u \rangle = 0$ for a non-zero vector $u \in \mathbb{F}_q^{m_1 \cdot s + m_2}$. While this technique has been introduced where $\{\bar{\gamma}_j^{(k)}\}_j$ are chosen uniformly at random, we could use the common strategy to define $\{\bar{\gamma}_j^{(k)}\}_j$ as $\{\gamma^j\}_j$ for some random value γ.[3] This strategy drastically reduces the number of random coefficients generated by the XOF but also impacts the soundness of the protocol. Namely, the first-round soundness error of the PIOP would become

$$\varepsilon_2 := \left(\frac{m_1 \cdot s + m_2}{|\mathbb{F}|}\right)^{\rho}$$

instead of $1/|\mathbb{F}|^{\rho}$. For a field of size $\approx 2^{2\lambda}$, this degradation has no impact since we still have $\varepsilon_2 < 2^{-\lambda}$ while setting the repetition parameter in the PIOP to $\rho = 1$. On the other hand, if $\mathbb{F}$ is a smaller field, the incurred loss of soundness might require to be compensated by a larger ρ (increasing the verifier computation and the proof size). To mitigate this, we instead proceed as follows: one derives ρ random vectors $v^{(k)} \in \mathbb{F}^{\rho+1}$ and $\rho + 1$ values $\hat{\gamma}_1, \ldots, \hat{\gamma}_{\rho+1}$, and one defines $\bar{\gamma}_j^{(k)}$ as

$$\bar{\gamma}_j^{(k)} := \sum_{i=1}^{\rho+1} v_i^{(k)} \cdot \hat{\gamma}_i^{j}$$

for all $1 \leq j \leq m_1 \cdot s + m_2$ and $1 \leq k \leq \rho$. Using this challenge, the soundness error now becomes

$$\varepsilon_2 := \frac{1}{|\mathbb{F}|^{\rho}} \cdot \left(1 + \frac{(m_1 \cdot s + m_2)^{\rho+1}}{|\mathbb{F}|}\right) ,$$

implying that we only have a degrading factor of $1 + \frac{(m_1 \cdot s + m_2)^{\rho+1}}{|\mathbb{F}|}$, which is close to 1 whenever $(m_1 \cdot s + m_2)^{\rho+1}$ is small compared to $|\mathbb{F}|$. This soundness error can be proved by observing that it corresponds to the product of a $\big((m_1 \cdot s + m_2)/|\mathbb{F}|\big)^{\rho+1}$-almost universal linear hash family and of a $1/|\mathbb{F}|^{\rho}$-almost one.

We can use the same strategy for the randomness involved in the degree-enforcing test of the DECS. In that case, the degree-enforcing soundness error of the PIOP becomes

$$\varepsilon_1 := \binom{N}{d_\beta + 2} \cdot \frac{1}{|\mathbb{F}|^{\eta}} \cdot \left(1 + \frac{n_{\mathsf{decs}}^{\eta+1}}{|\mathbb{F}|}\right)$$

instead of $\binom{N}{d_\beta+2}/|\mathbb{F}|^{\eta}$, where n_{decs} is the number of input polynomials of the DECS scheme (see Section 2.2).

For a large enough field $\mathbb{F}$ (namely $|\mathbb{F}| > n_{\mathsf{decs}}^{\eta+1}$), the loss of soundness is negligible while the XOF workload is reduced to generate

- $(\rho + 1)^2$ field elements instead of $\rho \cdot (m_1 \cdot s + m_2)$ for the PIOP batching challenge,
- $(\eta + 1)^2$ field elements instead of $\eta \cdot n_{\mathsf{decs}}$ for the degree-enforcing challenge.

4.5 Illustration of the Saving

Let us give concrete numbers to illustrate the typical saving in terms of SNARK constraints we obtain using our tweaks. We consider the Anemoi permutation family [BBC+23] over a 256-bit prime field and round

[3] This strategy is frequent in SNARKs to obtain a verification time independent of (or sublinear in) the size of the circuit. This strategy was not considered in SmallWood-ARK since its main target was building arguments for small circuits without much care about asymptotic complexity.

verification degree $\alpha = 3$, with a Merkle tree of 4096 leaves. Without any tweaks for SNARK-friendliness, an application of SmallWood-ARK would lead to signature size of around 9.6 KB with 28 700 R1CS constraints, from which around 3300 are due to hashing the leaves and around 18 000 are due to the verification of the authentication paths.

Using the parameter trade-offs in Merkle trees (less hashing for the leaves and greater node arity), we can reduce the number of constraints to 23 000 at the cost of increasing the signature size to 11.4 KB. From now on, 12 300 are due to the verification of the authentication paths in the Merkle tree. While further applying the trimming tweak, the signature size remains 11.4 KB but the number of constraints for verifying the authentication paths drop to 9300, making a total of 20 000 constraints. Finally, using an alternative distribution of the random challenges for the PIOP and DECS batching saves an additional 1000 constraints without changing the signature size.

To summarize, the tweaks decreased the number of constraints by 34%, from 28 700 to 19 000 while increasing the signature size of 19%, from 9.6 KB to 11.4 KB. Of course, a lot of trade-offs are possible given the many different parameters and tweaks but this gives an illustration of a typical trade-off.

5 CAPSS Signature Scheme

In this section, we present a detailed description of the general signature scheme produced by the CAPSS framework and formally define its security under the notion of existential unforgeability under chosen-message attacks (EUF-CMA). By general, we mean that the description remains compatible with any PACS statement and does not assume a particular permutation family or arithmetization technique.

5.1 Description of the Signature Scheme

The reader is referred to Table 1 for a summary of the different parameters of SmallWood-ARK, which are also used in the signature scheme. We further denote by c the "capacity parameter" introduced in Section 2.1 which is the smallest integer such that $|\mathbb{F}|^c \geq 2^{2\lambda}$. Hash digests arising in the application of Fiat-Shamir belong to $\mathbb{F}^c$. The signature description further uses three fixed subsets of $\mathbb{F}$, namely the DECS evaluation domain $\mathbb{E} = \{e_1, \ldots, e_N\}$, the LVCS witness support $\Omega = \{\omega_1, \ldots, \omega_{n_{\mathsf{cols}}}\}$ and the LVCS randomness support $\Omega' = \{\omega'_1, \ldots, \omega'_\ell\}$. Also, the DECS evaluations to be opened are queried through an index set $I = \{i_1, \ldots, i_\ell\}$, which means that the queried set of evaluation points is $E = \{e_{i_1}, \ldots, e_{i_\ell}\}$. For the sake of simplicity, the domain separation index as well as the output format is left implicit in the calls to the XOF primitive.

We first describe the signing and verification algorithms and then the underlying PCS routines.

Signing and Verification Algorithms. The signing and verification algorithms are depicted respectively in Figure 4 and Figure 5. The public key pk consists of the PACS statement, which is composed of the polynomial constraints $\{f_j\}_j$ and $\{f'_j\}_j$, together with the associated constants $\{\theta_{j,i,k}\}$ and $\{\theta'_{j,i,k}\}$. The secret key sk is composed of the latter PACS statement and the associated witness matrix $(w_{i,j})_{i,j}$.

The signing algorithm follows the PIOP protocol described in Figure 2. After building the polynomials $P_1, \ldots, P_n$ and sampling the polynomials $M_1, \ldots, M_\eta$, it commits to those polynomials using the polynomial commitment scheme SmallWood-PCS (instead of using an oracle as the PIOP protocol). It then derives the batching PIOP challenge $\{\Gamma_{k,j}(X)\}_{k,j}$ and $\{\gamma'_{k,j}\}_{k,j}$ by hashing the message to sign msg and the commitment digest $\mathsf{transcript}_{\mathsf{pcs}}$. Finally, it computes the polynomials $\boldsymbol{Q}$, derives the PIOP opening queries and opens the committed polynomials on those queries. The verification algorithm of the signature follows the verification procedure of the PIOP protocol: after checking that the opened evaluations are consistent with the polynomial commitment, it checks that those evaluations satisfy the polynomial relation (2) and that the polynomials $\boldsymbol{Q}$ verify $\sum_{\omega\in\Omega} Q_i(\omega) = 0$ for all i.

We employ a standard optimization technique to reduce the signature size by selectively revealing elements. Specifically, the polynomials $\boldsymbol{Q}$ from the PIOP proof are only partially disclosed in the signature as

Table 1: Parameters of SmallWood-ARK (from [FR25]).

Parameters of the PACS Statement:		
$\mathbb{F}$	Base field	
s	Number of columns in the witness matrix (packing factor)	
n	Number of rows in the witness matrix	
d	Maximal degree of parallel polynomial constraints	
m_1	Number of parallel polynomial constraints	
m_2	Number of global linear constraints	
Parameters of the PACS PIOP:		
ℓ'	Number of oracle queries (polynomial evaluations)	*chosen*
ρ	Number of parallel repetitions	*chosen*
d_Q	Degree of $\boldsymbol{Q}$	(1)
Parameters of the PCS:		
n_{pcs}	Number of committed polynomials	$n_{\mathsf{pcs}} = n + 2\rho$
$\{d_j\}_{1\leq j\leq n}$	Degrees of the witness polynomials	$d_j = \ell' + s - 1$
$\{d_j\}_{n<j\leq n+\rho}$	Degrees of the masking polynomials $\{M_{k,1}\}_k$	$d_j = d \cdot (\ell' + s - 1) - s$
$\{d_j\}_{n+\rho<j\leq n+2\rho}$	Degrees of the masking polynomials $\{M_{k,2}\}_k$	$d_j = \ell' + 2s - 2$
μ	Number of coefficient rows in the matrices $\{\boldsymbol{A}_j\}$	*chosen*
$\{\nu_j\}$	Number of columns of the matrices $\{\boldsymbol{A}_j\}$	$\nu_j = \lceil (d_j + 1 - \ell')/\mu \rceil$
β	Stacking factor (number of $(\mu + \ell')$-row layers in $\boldsymbol{A}$)	*chosen*
Parameters of the LVCS:		
n_{rows}	Number of committed row vectors	$n_{\mathsf{rows}} = \beta \cdot (\mu + \ell')$
n_{cols}	Size of committed row vectors	$n_{\mathsf{cols}} = \lceil (\sum_j \nu_j)/\beta \rceil$
m	Number of LVCS queries	$m = \beta \cdot \ell'$
ℓ	Number of DECS evaluation queries	*chosen*
Parameters of the DECS:		
n_{decs}	Number of committed polynomials	$n_{\mathsf{decs}} = n_{\mathsf{rows}}$
d_{decs}	Degree of committed polynomials	$d_{\mathsf{decs}} = n_{\mathsf{cols}} + \ell - 1$
N	Size of the evaluation domain (number of Merkle leaves)	*chosen*
η	Number of parallel repetitions of the degree-enforcing round	*chosen*
$\kappa_1, \kappa_2, \kappa_3, \kappa_4$	Grinding parameters	*chosen*

the opened polynomial evaluations enable to reconstruct them in full. We must still provide hash commitment of these polynomials for further application of the Fiat-Shamir transform. This is done through the variable h_{piop}. The PIOP proof is then implicitly checked in the verification algorithm by recomputing h_{piop} (hashing $\boldsymbol{Q}$).

$\underline{\mathsf{Sign}(\mathsf{sk}, \mathsf{msg})}$:

1. *Initialization.*
   (a) Sample a random salt salt.
   (b) Parse sk as $(\mathsf{pacs\_stat}, \mathsf{pacs\_wit})$.
   (c) Parse $\mathsf{pacs\_wit}$ as $(w_{i,j})_{i,j}$ and $\mathsf{pacs\_stat}$ as $\big(\{f_j, \{\theta_{j,i,k}\}_{i,k}\}_j, \{f'_j, \{\theta'_{j,i,k}\}_{i,k}\}_j\big)$.
2. *Polynomial commitment.*
   (a) *Building witness polynomials.* For all $j \in [1, n]$, generate a random degree-$(s+\ell'-1)$ polynomial $P_j(X)$ such that $P_j(\omega_1) = w_{j,1}, \ldots, P_j(\omega_s) = w_{j,s}$. Let $\boldsymbol{P} = (P_1, \ldots, P_n)$.
   (b) *Sampling masking polynomials.* Sample ρ random degree-$(d \cdot (s+\ell'-1)+s)$ polynomials $(M_1, \ldots, M_\rho)$ such that $\sum_{i=1}^{s} M_k(\omega_i) = 0$ for all k. Let $\boldsymbol{M} = (M_1, \ldots, M_\rho)$.
   (c) *Computing polynomial commitment.* Run:
   $$(\mathsf{transcript}_{\mathsf{pcs}}, \mathsf{key}_{\mathsf{pcs}}) \leftarrow \mathsf{PCS.Commit}(\mathsf{salt}, \boldsymbol{P}, \boldsymbol{M}) \ .$$
3. *Polynomial IOP.* Let $\mathsf{transcript} = \mathsf{msg} \parallel \mathsf{transcript}_{\mathsf{pcs}}$.
   (a) Compute $h_{\text{fpp}} = \mathsf{XOF}(\mathsf{transcript})$.
   (b) Compute $(\{\Gamma_{k,j}(X)\}_{k,j}, \{\gamma'_{k,j}\}_{k,j}) = \mathsf{XOF}(h_{\text{fpp}})$.
   (c) For all $1 \leq k \leq \rho$, compute $Q_k(X) = M_k(X) + \sum_{j=1}^{m_1} \Gamma_{k,j}(X) \cdot F_j(X) + \sum_{j=1}^{m_2} \gamma'_{k,j} \cdot F'_j(X)$, where
   $$\begin{cases} F_j(X) & := f_j(P_1(X), \ldots, P_n(X), \Theta_{j,1}(X), \ldots, \Theta_{j,n_c}(X)) \ , \\ F'_j(X) & := f'_j(P_1(X), \ldots, P_n(X), \Theta'_{j,1}(X), \ldots, \Theta'_{j,n_c}(X)) \ . \end{cases}$$
   Let $\boldsymbol{Q} = (Q_1, \ldots, Q_\rho)$.
   (d) Set $\pi_{\mathsf{piop}} = (q_k^{(H)})_k$, where $q_k^{(H)}$ is the $\deg Q_k + 1 - \ell'$ higher coefficients of Q_k.

   Let $\mathsf{transcript}_{\mathsf{piop}} = (h_{\text{fpp}}, \boldsymbol{Q})$. Derive the PIOP challenge:
   $$h_{\mathsf{piop}} = \mathsf{XOF}(\mathsf{transcript}_{\mathsf{piop}}) \in \mathbb{F}^c \ ,$$
   $$E' = \mathsf{XOF}(h_{\mathsf{piop}}) \in \mathbb{F}^{\ell'} \ .$$
4. *PCS opening.* Let $\mathsf{transcript} = h_{\mathsf{piop}}$. Run:
   $$((\boldsymbol{P}, \boldsymbol{M})|_{E'}, \pi_{\mathsf{pcs}}) = \mathsf{PCS.Open}(\mathsf{key}_{\mathsf{pcs}}, E', \mathsf{transcript}) \ .$$
5. *Signature assembling.* Return
   $$\sigma := (\mathsf{salt}, h_{\mathsf{piop}}, \pi_{\mathsf{piop}}, (\boldsymbol{P}, \boldsymbol{M})|_{E'}, \pi_{\mathsf{pcs}}).$$

Fig. 4: Signature scheme in the CAPSS framework – Signing algorithm.

$\underline{\mathsf{Verify}(\mathsf{pk}, \mathsf{msg}, \sigma)}$:

1. *Initialization.* Parse σ as $(\mathsf{salt}, h_{\mathsf{piop}}, \pi_{\mathsf{piop}}, (\boldsymbol{P}, \boldsymbol{M})|_{E'}, \pi_{\mathsf{pcs}})$ and pk as $\big(\{f_j, \{\theta_{j,i,k}\}_{i,k}\}_j, \{f'_j, \{\theta'_{j,i,k}\}_{i,k}\}_j\big)$.
2. *PIOP challenge recomputation.* Run:
$$E' = \mathsf{XOF}(h_{\mathsf{piop}}) \in \mathbb{F}^{\ell'} \ .$$
3. *Polynomial commitment recomputation.* Let $\mathsf{transcript} = h_{\mathsf{piop}}$. Run:
$$\mathsf{transcript}_{\mathsf{pcs}} = \mathsf{PCS.RecomputeTranscript}(\mathsf{salt}, E', (\boldsymbol{P}, \boldsymbol{M})|_{E'}, \pi_{\mathsf{pcs}}, \mathsf{transcript})$$
4. *PIOP transcript recomputation.* Let $\mathsf{transcript} = \mathsf{msg} \parallel \mathsf{transcript}_{\mathsf{pcs}}$.
   (a) Compute $h_{\mathrm{fpp}} = \mathsf{XOF}(\mathsf{transcript})$.
   (b) Compute $(\gamma'_1, \ldots, \gamma'_\rho) = \mathsf{XOF}(h_{\mathrm{fpp}})$.
   (c) For all $1 \le k \le \rho$, compute $Q_k(q) = M_k(q) + \sum_{j=1}^{m_1} \Gamma_{i,j}(q) \cdot F_j(q) + \sum_{j=1}^{m_2} \gamma'_{i,j} \cdot F'_j(q)$, where
$$\begin{cases} F_j(q) & := f_j(P_1(q), \ldots, P_n(q), \Theta_{j,1}(q), \ldots, \Theta_{j,n_c}(q)) \ , \\ F'_j(q) & := f'_j(P_1(q), \ldots, P_n(q), \Theta'_{j,1}(q), \ldots, \Theta'_{j,n_c}(q)) \ . \end{cases}$$
   for all $q \in E'$ and restore $Q_k(X)$ from $q_k^{(H)}$ and $\{Q_k(q)\}_{q \in E}$.
5. *Final verification.* Let $\mathsf{transcript}_{\mathsf{piop}} = (h_{\mathrm{fpp}}, \boldsymbol{Q})$. If $h_{\mathsf{piop}} = \mathsf{XOF}(\mathsf{transcript}_{\mathsf{piop}})$ and if $\sum_{i=1}^{s} Q_k(\omega_i) = 0$ for all $1 \le k \le \rho$, return ACCEPT. Otherwise return REJECT.

Fig. 5: Signature scheme in the CAPSS framework – Verification algorithm.

Polynomial Commitment Routines. We describe hereafter the routines for the non-interactive version of the $\mathsf{SmallWood\text{-}PCS}$ scheme introduced in [FR25]. This scheme is based on three layers, from bottom to top: the DECS layer, the LVCS layer and the PCS layer. We describe hereafter the routines of these three layers successively.

DECS routines. The DECS routines are formally described in Figure 6.The routine $\mathsf{DECS.Commit}$ is called in the signing algorithm (through the LVCS layer) to compute the DECS commitment transcript (including the degree-enforcing round). The $\mathsf{DECS.Open}$ routine is called in the signing algorithm (through the LVCS layer) to compute the opened evaluations and their opening proof. The $\mathsf{DECS.RecomputeTranscript}$ routine is called in the verification algorithm (through the LVCS layer) to recompute the commitment transcript –in particular the polynomial $\boldsymbol{R}$– from the opened evaluations. The $\mathsf{DECS.OpeningChallenge}$ is called in the signing algorithm (through the LVCS layer) to derive the opening challenge, i.e., the set of open DECS evaluations. The $\mathsf{DECS.RecomputeOpeningChallenge}$ is called in the verification algorithm (through the LVCS layer) with the same purpose as the latter routine, but taking the grinding counter as argument to avoid the proof-of-work computation.

LVCS routines. The LVCS routines are formally described in Figure 7. The $\mathsf{LVCS.Commit}$ routine is used in the signing algorithm (through the PCS layer) to compute the LVCS commitment transcript. The $\mathsf{LVCS.Open}$ routine is used in the signing algorithm (through the PCS layer) to compute the opened evaluations and their opening proof. The $\mathsf{LVCS.RecomputeTranscript}$ is used in the verification algorithm (through the PCS layer) to recompute the LVCS commitment transcript.

PCS routines. The PCS routines are formally described in Figure 8. The $\mathsf{PCS.Commit}$ routine is used in the signing algorithm to compute the PCS commitment transcript. The $\mathsf{PCS.Open}$ routine is used in the signing algorithm to compute the opened evaluations and their opening proof. The $\mathsf{PCS.RecomputeTranscript}$ is used in the verification algorithm to recompute the PCS commitment transcript. These routines satisfy the following property: for any $\mathsf{salt}, \boldsymbol{P}, E', \mathsf{transcript}$ of the right format, running:

$$\begin{aligned}
&(\mathsf{transcript}_{\mathsf{pcs}}, \mathsf{key}_{\mathsf{pcs}}) \leftarrow \mathsf{PCS.Commit}(\mathsf{salt}, \boldsymbol{P}) \\
&(\boldsymbol{P}|_{E'}, \pi_{\mathsf{pcs}}) \leftarrow \mathsf{PCS.Open}(\mathsf{key}_{\mathsf{pcs}}, E', \mathsf{transcript}) \\
&\mathsf{transcript}'_{\mathsf{pcs}} \leftarrow \mathsf{PCS.RecomputeTranscript}(\mathsf{salt}, E', \boldsymbol{P}|_{E'}, \pi_{\mathsf{pcs}}, \mathsf{transcript})
\end{aligned}$$

always yields $\mathsf{transcript}_{\mathsf{pcs}} = \mathsf{transcript}'_{\mathsf{pcs}}$. Checking the latter equality amounts to implicitly checking the correctness of the PCS opening proof π_{pcs} for the opened evaluations $\boldsymbol{P}|_{E'}$. Namely, the equality only holds if the opening proof is correct.

```
DECS.Commit(salt, $\boldsymbol{P}$):

On input a salt salt and a vector polynomial $\boldsymbol{P} := (P_1, \ldots, P_{n_{\mathsf{decs}}}) \in \left(\mathbb{F}[X]^{(\le d_{\mathsf{decs}})}\right)^{n_{\mathsf{decs}}}$:
  1. Sample $\boldsymbol{M} = (M_1, \ldots, M_\eta)$ from $\left(\mathbb{F}[X]^{(\le d_{\mathsf{decs}})}\right)^{\eta}$.
  2. For all $i \in [1, N]$, compute the leaves $u_i = \mathsf{XOF}(\mathsf{salt}, \boldsymbol{P}(e_i), \boldsymbol{M}(e_i))$.
  3. Compute $\mathsf{tree}, \mathsf{root} = \mathsf{MerkleTree}(u_1, \ldots, u_N)$.
  4. Compute $h_{\mathrm{mt}} = \mathsf{XOF}(\mathsf{salt}, \mathsf{root}) \in \mathbb{F}^c$.
  5. Derive the challenge $\{\gamma_k\}_{k\in[1,\eta]} = \mathsf{XOF}(h_{\mathrm{mt}}) \in \mathbb{F}^\eta$.
  6. For all $k \in [1, \eta]$, compute the polynomial $R_k(X) = M_k(X) + \sum_{i=1}^{n_{\mathsf{decs}}} \gamma_k^i \cdot P_i(X) \in \mathbb{F}[X]^{(\le d_{\mathsf{decs}})}$.
  7. Set $\mathsf{transcript}_{\mathsf{decs}} = (h_{\mathrm{mt}}, \boldsymbol{R})$ and $\mathsf{key}_{\mathsf{decs}} = (\boldsymbol{M}, \boldsymbol{P}, \boldsymbol{R}, \mathsf{tree})$, where $\boldsymbol{R} = (R_1, \ldots, R_\eta)$.
  8. Return $(\mathsf{transcript}_{\mathsf{decs}}, \mathsf{key}_{\mathsf{decs}})$.

DECS.Open($\mathsf{key}_{\mathsf{decs}}, I$):

On input an opening key $\mathsf{key}_{\mathsf{decs}} := (\boldsymbol{M}, \boldsymbol{P}, \boldsymbol{R}, \mathsf{tree})$ and a query index set $I := \{i_1, \ldots, i_\ell\}$:
  1. Compute $\mathsf{auth} = \mathsf{GetAuthPath}(\mathsf{tree}, I)$.
  2. For all $j \in [1, \ell]$, compute $\boldsymbol{p}^{(\mathrm{eval},j)} = \boldsymbol{P}(e_{i_j})$ and $\boldsymbol{m}^{(\mathrm{eval},j)} = \boldsymbol{M}(e_{i_j})$.
  3. Set $\boldsymbol{r}^{(\mathrm{high})}$ as the $d_{\mathsf{decs}} + 1 - \ell$ higher coefficients of $\boldsymbol{R}$, and $\pi_{\mathsf{decs}} = (\mathsf{auth}, \{\boldsymbol{m}^{(\mathrm{eval},j)}\}_j, \boldsymbol{r}^{(\mathrm{high})})$.
  4. Return $(\{\boldsymbol{p}^{(\mathrm{eval},j)}\}_j, \pi_{\mathsf{decs}})$.

DECS.RecomputeTranscript(salt, $I, \{\boldsymbol{p}^{(\mathrm{eval},j)}\}_j, \pi_{\mathsf{decs}}$):

On input a salt salt, a query index set $I := \{i_1, \ldots, i_\ell\}$, a set of evaluations $\{\boldsymbol{p}^{(\mathrm{eval},j)}\}_j$ and a proof $\pi_{\mathsf{decs}} := (\mathsf{auth}, \{\boldsymbol{m}^{(\mathrm{eval},j)}\}_j, \boldsymbol{r}^{(\mathrm{high})})$:
  1. For all $j \in [1, \ell]$, compute $u_{i_j} = \mathsf{XOF}(\mathsf{salt}, \boldsymbol{p}^{(\mathrm{eval},j)}, \boldsymbol{m}^{(\mathrm{eval},j)})$.
  2. Compute $\mathsf{root} = \mathsf{RetrieveRootFromPath}(\{u_{i_j}\}_j, \mathsf{auth}, I)$.
  3. Compute $h_{\mathrm{mt}} = \mathsf{XOF}(\mathsf{salt}, \mathsf{root}) \in \mathbb{F}^c$.
  4. Derive the challenge $\{\gamma_k\}_{k\in[1,\eta]} = \mathsf{XOF}(h_{\mathrm{mt}}) \in \mathbb{F}^\eta$.
  5. For all $k$, compute $r_k^{(\mathrm{eval},j)} = m_k^{(\mathrm{eval},j)} + \sum_{i=1}^{n_{\mathsf{decs}}} \gamma_k^i \cdot p_i^{(\mathrm{eval},j)}$.
  6. Restore $\boldsymbol{R}$ from $\boldsymbol{r}^{(\mathrm{high})}$ and $\{\boldsymbol{r}^{(\mathrm{eval},j)}\}_j$.
  7. Set $\mathsf{transcript}_{\mathsf{decs}} = (h_{\mathrm{mt}}, \boldsymbol{R})$.
  8. Return $\mathsf{transcript}_{\mathsf{decs}}$.

DECS.OpeningChallenge(trans_hash):

On input a transcript hash trans_hash:
  1. Initialize counter = 0.
  2. Compute $v = \mathsf{XOF}(\mathsf{counter}, \mathsf{trans\_hash})$. If $v > t_{\mathrm{pow}}$, increment counter and repeat.
  3. Decompose $v$ as $\left(\sum_{j=1}^{\ell} v_j \cdot N^{j-1}\right) + N^\ell \cdot v'$, with $v_1, \ldots, v_\ell \in \{0, \ldots, N-1\}$ and $v' \in \{0, \ldots, t_{\mathrm{pow}}/N^\ell - 1\}$.
  4. If there exists $i \ne j$ such that $v_i = v_j$, increment counter and repeat.
  5. Set $I = \{v_j\}_{j\in[1,\ell]}$.
  6. Return $(I, \mathsf{counter})$.

DECS.RecomputeOpeningChallenge(counter, trans_hash):

On input a counter counter and a transcript hash trans_hash:
  1. Compute $v = \mathsf{XOF}(\mathsf{counter}, \mathsf{trans\_hash})$. Check that $v \le t_{\mathrm{pow}}$.
  2. Decompose $v$ as $\left(\sum_{j=1}^{\ell} v_j \cdot N^{j-1}\right) + N^\ell \cdot v'$, with $v_1, \ldots, v_\ell \in \{0, \ldots, N-1\}$ and $v' \in \{0, \ldots, t_{\mathrm{pow}}/N^\ell - 1\}$.
  3. Check that there are no $i \ne j$ such that $v_i = v_j$.
  4. Set $I = \{v_j\}_{j\in[1,\ell]}$.
  5. Return $(I, \mathsf{counter})$.
```

Fig. 6: DECS routines in the CAPSS framework.

```
$\underline{\mathsf{LVCS.Commit}(\mathsf{salt}, \boldsymbol{r}_1, \ldots, \boldsymbol{r}_{n_{\mathsf{rows}}})}$
On input a salt $\mathsf{salt}$ and $n_{\mathsf{rows}}$ row vectors $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_{n_{\mathsf{rows}}} \in \mathbb{F}^{n_{\mathsf{cols}}}$:
  1. For all $j \in [1, n_{\mathsf{rows}}]$, sample $\bar{\boldsymbol{r}}_j$ from $\mathbb{F}^{\ell}$.
  2. For all $j \in [1, n_{\mathsf{rows}}]$, interpolate the degree-$(n_{\mathsf{cols}} + \ell - 1)$ polynomial $P_j(X)$ such that
       $(P_j(\omega_1), \ldots, P_j(\omega_{n_{\mathsf{cols}}})) = \boldsymbol{r}_j$  and  $(P_j(\omega'_1), \ldots, P_j(\omega'_\ell)) = \bar{\boldsymbol{r}}_j$.
  3. Run $(\mathsf{transcript}_{\mathsf{decs}}, \mathsf{key}_{\mathsf{decs}}) = \mathsf{DECS.Commit}(\mathsf{salt}, \boldsymbol{P})$.
  4. Set $\mathsf{transcript}_{\mathsf{lvcs}} = \mathsf{transcript}_{\mathsf{decs}}$ and $\mathsf{key}_{\mathsf{lvcs}} = (\mathsf{key}_{\mathsf{decs}}, \{\boldsymbol{r}_j, \bar{\boldsymbol{r}}_j\}_j)$.
  5. Return $(\mathsf{transcript}_{\mathsf{lvcs}}, \mathsf{key}_{\mathsf{lvcs}})$.

$\underline{\mathsf{LVCS.Open}(\mathsf{key}_{\mathsf{lvcs}}, \mathcal{C}, \mathsf{transcript})}$:
On input an opening key $\mathsf{key}_{\mathsf{lvcs}} := (\mathsf{key}_{\mathsf{decs}}, \{\boldsymbol{r}_j, \bar{\boldsymbol{r}}_j\}_j)$, a set of coefficients $\mathcal{C} := \{c_{k,1}, \ldots, c_{k,n_{\mathsf{rows}}}\}_{k \in \{1,\ldots,m\}}$ and a transcript $\mathsf{transcript}$:
  1. For all $k \in [1, m]$, compute $\boldsymbol{v}_k = \sum_{j=1}^{n_{\mathsf{rows}}} c_{k,j} \cdot \boldsymbol{r}_j$ and $\bar{\boldsymbol{v}}_k = \sum_{j=1}^{n_{\mathsf{rows}}} c_{k,j} \cdot \bar{\boldsymbol{r}}_j$.
  2. Compute $\mathsf{trans\_hash} = \mathsf{XOF}(\mathsf{transcript}, \{\boldsymbol{v}_k, \bar{\boldsymbol{v}}_k\}_k) \in \mathbb{F}^c$.
  3. Compute $(I, \mathsf{counter}) = \mathsf{DECS.OpeningChallenge}(\mathsf{trans\_hash})$.
  4. Run $(\{\boldsymbol{p}^{(\mathrm{eval},j)}\}_j, \pi_{\mathsf{decs}}) = \mathsf{DECS.Open}(\mathsf{key}_{\mathsf{decs}}, I)$.
  5. Set $\pi_{\mathsf{lvcs}} = (\mathsf{counter}, \pi_{\mathsf{decs}}, \{\bar{\boldsymbol{v}}_k\}_k, \{\mathsf{drop}_m(\boldsymbol{p}^{(\mathrm{eval},j)})\}_j)$.
  6. Return $(\{\boldsymbol{v}_k\}_k, \pi_{\mathsf{lvcs}})$.

$\underline{\mathsf{LVCS.RecomputeTranscript}(\mathsf{salt}, \mathcal{C}, \{\boldsymbol{v}_k\}_k, \pi_{\mathsf{lvcs}}, \mathsf{transcript})}$:
On input a salt, a query set $\mathcal{C} = \{(c_{k,1}, \ldots, c_{k,n_{\mathsf{rows}}})\}_{k \in \{1,\ldots,m\}}$, a set of LVCS evaluations $\{\boldsymbol{v}_k\}_k$ and a proof $\pi_{\mathsf{lvcs}} := (\mathsf{counter}, \pi_{\mathsf{decs}}, \{\bar{\boldsymbol{v}}_k\}_k, \{\mathsf{drop}_m(\boldsymbol{p}^{(\mathrm{eval},j)})\}_j)$:
  1. Compute $\mathsf{trans\_hash} = \mathsf{XOF}(\mathsf{transcript}, \{\boldsymbol{v}_k, \bar{\boldsymbol{v}}_k\}_k) \in \mathbb{F}^c$.
  2. Compute $I = \mathsf{DECS.RecomputeOpeningChallenge}(\mathsf{counter}, \mathsf{trans\_hash})$.
  3. For all $k \in [1, m]$, compute $Q_k$ as the degree-$(n_{\mathsf{cols}} + \ell + 1)$ polynomial satisfying:
       $(Q_k(\omega_1), \ldots, Q_k(\omega_{n_{\mathsf{cols}}})) = \boldsymbol{v}_k$  and  $(Q_k(\omega'_1), \ldots, Q_k(\omega'_\ell)) = \bar{\boldsymbol{v}}_k$.
  4. For all $k \in [1, m]$, compute $\boldsymbol{v}'_k = (Q_k(e_{i_1}), \ldots, Q_k(e_{i_\ell}))$ .
  5. Compute $\{(\boldsymbol{p}_h^{(\mathrm{eval},j)})_{h \le m}\}_j$ such that $(\boldsymbol{v}'_k)_j = \sum_h c_{k,h} \cdot \boldsymbol{p}_h^{(\mathrm{eval},j)}$ for all $j \in [1, \ell]$ and $k \in [1, m]$.
  6. Return $\mathsf{DECS.RecomputeTranscript}(\mathsf{salt}, I, \{\boldsymbol{p}^{(\mathrm{eval},j)}\}_j, \pi_{\mathsf{decs}})$.
```

Fig. 7: LVCS routines in the CAPSS framework.

$\underline{\mathsf{PCS.Commit}(\mathsf{salt}, (P_1, \ldots, P_{n_{\mathsf{pcs}}}))}$:

1. For all $1 \leq j \leq n_{\mathsf{pcs}}$, sample $\ell'(\nu_j - 1)$ values $r_{j,1,1}, \ldots, r_{j,\nu_j-1,\ell'}$ uniformly at random from $\mathbb{F}$ and compute:

$$\boldsymbol{A}_j := \left[\begin{array}{ccc|c}
a_{j,0} & \cdots & a_{j,(\nu_j-2)\mu} & \left.\begin{matrix}0\\ \vdots\\ 0\end{matrix}\right\}\begin{matrix}\delta_j\\ \text{times}\end{matrix} \\
\vdots & \ddots & \vdots & \\
\vdots & \ddots & \vdots & \\
\vdots & \ddots & \vdots & a_{j,(\nu_j-1)\mu} \\
a_{j,\mu-1} & \cdots & a_{j,(\nu_j-1)\mu-1} & \vdots \\
\cline{1-3}
0 & \cdots & 0 & a_{j,d_j-\ell'+1} \\
\vdots & \ddots & \vdots & \vdots \\
0 & \cdots & 0 & a_{j,d_j}
\end{array}\right] + \left[\begin{array}{cccc|c}
0 & -r_{j,1,1} & \cdots & -r_{j,1,\nu_j-2} & \left.\begin{matrix}0\\ \vdots\\ 0\end{matrix}\right\}\begin{matrix}\delta_j\\ \text{times}\end{matrix} \\
\vdots & \vdots & \ddots & \vdots & \\
0 & -r_{j,\ell',1} & \cdots & -r_{j,\ell',\nu_j-2} & \\
0 & 0 & \cdots & 0 & -r_{j,1,\nu_j-1} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & 0 & -r_{j,\ell',\nu_j-1} \\
\cline{1-4}
r_{j,1,1} & r_{j,1,2} & \cdots & r_{j,1,\nu_j-1} & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
r_{j,\ell',1} & r_{j,\ell',2} & \cdots & r_{j,\ell',\nu_j-1} & 0
\end{array}\right]$$

   with $\delta_j := (\mu \cdot \nu_j + \ell') - (d_j + 1)$ and where $\{a_{j,i}\}_i$ are the coefficients of P_j, i.e., $P_j := \sum_{i=0}^{d_j} a_{j,i} X^i$.
2. Set $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_{n_{\mathsf{rows}}} \in \mathbb{F}^{n_{\mathsf{cols}}}$ such that

$$\begin{bmatrix} \boldsymbol{r}_1^\top \\ \boldsymbol{r}_2^\top \\ \vdots \\ \boldsymbol{r}_{n_{\mathsf{rows}}}^\top \end{bmatrix} = \mathsf{Stack}_\beta\big(\boldsymbol{A}_1 | \ldots | \boldsymbol{A}_{n_{\mathsf{pcs}}}\big) ,$$

   where Stack_β is the mapping which consists in splitting the columns of a matrix in β subsequent groups and stacking them.
3. $(\mathsf{transcript}_{\mathsf{lvcs}}, \mathsf{key}_{\mathsf{lvcs}}) = \mathsf{LVCS.Commit}(\mathsf{salt}, \boldsymbol{r}_1, \ldots, \boldsymbol{r}_{n_{\mathsf{rows}}})$.
4. Return $(\mathsf{transcript}_{\mathsf{pcs}}, \mathsf{key}_{\mathsf{pcs}}) = (\mathsf{transcript}_{\mathsf{lvcs}}, \mathsf{key}_{\mathsf{lvcs}})$

$\underline{\mathsf{PCS.Open}(\mathsf{key}_{\mathsf{lvcs}}, E', \mathsf{transcript})}$:

1. Set $\mathcal{C} = \{\boldsymbol{u}_k \otimes (1, r, \ldots, r^{\mu+m})\}_{r \in E', k \in [1,\beta]}$.
2. Compute $(\{\boldsymbol{v}_k^{(r)}\}_{r \in E', k \in [1,\beta]}, \pi_{\mathsf{lvcs}}) = \mathsf{LVCS.Open}(\mathsf{key}_{\mathsf{lvcs}}, \mathcal{C}, \mathsf{transcript})$.
3. For all $r \in E'$, parse $(\boldsymbol{v}_1^{(r)} \mid \ldots \mid \boldsymbol{v}_\beta^{(r)})$ as $(\hat{\boldsymbol{v}}_1^{(r)} \mid \ldots \mid \hat{\boldsymbol{v}}_{n_{\mathsf{pcs}}}^{(r)})$, where $\hat{\boldsymbol{v}}_1^{(r)}, \ldots, \hat{\boldsymbol{v}}_n^{(r)}$ are vectors of size $\nu_1, \ldots, \nu_{n_{\mathsf{pcs}}}$.
4. For all $r \in E'$ and all $1 \leq j \leq n_{\mathsf{pcs}}$, $p_j^{(r)} = \langle \hat{\boldsymbol{v}}_j^{(r)}, (1, r^\mu, \ldots, r^{(\nu_j-2)\mu}, r^{(\nu_j-1)\mu-\delta_j}) \rangle$.
5. Set $\pi_{\mathsf{pcs}} = (\pi_{\mathsf{lvcs}}, \{\mathsf{drop\_first}(\hat{\boldsymbol{v}}_j^{(r)})\}_{r \in E', 1 \leq j \leq n_{\mathsf{pcs}}})$, where $\mathsf{drop\_first}$ removes the first coordinate of the input vector.
6. Return $(\{p_j^{(r)}\}_{r \in E', 1 \leq j \leq n_{\mathsf{pcs}}}, \pi_{\mathsf{pcs}})$.

$\underline{\mathsf{PCS.RecomputeTranscript}(\mathsf{salt}, E', \{p_j^{(x)}\}_{x \in E', 1 \leq j \leq n_{\mathsf{pcs}}}, \pi_{\mathsf{pcs}}, \mathsf{transcript})}$:

1. Set $\mathcal{C} = \{\boldsymbol{u}_k \otimes (1, r, \ldots, r^{\mu+m})\}_{r \in E', k \in [1,\beta]}$.
2. Parse π_{pcs} as $(\pi_{\mathsf{lvcs}}, \{\mathsf{drop\_first}(\hat{\boldsymbol{v}}_j^{(r)})\}_j)$.
3. For $1 \leq j \leq n_{\mathsf{pcs}}$, compute $(\hat{\boldsymbol{v}}_j)_1 = p_j^{(r)} - \Big[(\hat{\boldsymbol{v}}_j)_{\nu_j} \cdot (r^{(\nu_j-1)\mu-\delta_j}) + \sum_{i=2}^{\nu_j-1} (\hat{\boldsymbol{v}}_j)_i \cdot (r^\mu)^{i-1}\Big]$.
4. For all $r \in E'$, parse $(\boldsymbol{v}_1^{(r)} \mid \ldots \mid \boldsymbol{v}_\beta^{(r)})$ as $(\hat{\boldsymbol{v}}_1^{(r)} \mid \ldots \mid \hat{\boldsymbol{v}}_{n_{\mathsf{pcs}}}^{(r)})$.
5. Return $\mathsf{LVCS.RecomputeTranscript}(\mathsf{salt}, \mathcal{C}, \{\boldsymbol{v}_k^{(r)}\}_{r \in E', k \in [1,\beta]}, \pi_{\mathsf{lvcs}}, \mathsf{transcript})$.

Fig. 8: PCS routines in the CAPSS framework.

5.2 Unforgeability of CAPSS Signatures

Signature schemes in the CAPSS framework aim at providing *unforgeability against chosen message attacks* (EUF-CMA). In this setting, the adversary is given a public key pk and they can ask an oracle (called the *signature oracle*) to sign messages $(\mathsf{msg}_1, \ldots, \mathsf{msg}_r)$ of their choice. The goal of the adversary is to generate a pair (msg, σ) such that msg is not one of the requests to the signature oracle and such that σ is a valid signature of msg with respect to pk.

We prove hereafter the EUF-CMA security of the CAPSS signature scheme in the Random Oracle Model (ROM). Our security statement is based on the one-wayness of the function $\mathsf{OWF}_{\mathcal{P},iv}$ defined in Section 2.1 (which is related to the hardness of CICO for the underlying permutation family) and random oracle heuristic for the (extandable output) hash functions Hash and XOF as random oracles.

Theorem 2. *Let the extendable-output function be modelled as a random oracle. Assume $\mathsf{OWF}_{\mathcal{P},iv}$ is $(t, \varepsilon_{\text{OW}})$-one-way. Let $\mathcal{A}$ be an adversary against the EUF-CMA security of the CAPSS signature scheme making a total of Q_{RO} random oracle queries and Q_{sign} signing oracle queries. The advantage of $\mathcal{A}$ in the EUF-CMA game is upper-bounded as:*

$$\varepsilon_{\text{EUF-CMA}} \leq \varepsilon_{\text{SE-KS}} + \varepsilon_{\text{ZK}} + \varepsilon_{\text{OW}} \ ,$$

with

$$\varepsilon_{\text{ZK}} \leq \frac{Q_{\mathsf{sign}}^2}{2^{2\lambda}} + \frac{Q_{\mathsf{RO}}}{2^{\lambda}} + \frac{Q_{\mathsf{sign}} \cdot Q_{\mathsf{RO}}}{2^{2\lambda}}$$

and

$$\varepsilon_{\text{SE-KS}} \leq \frac{Q_{\mathsf{RO}}^2}{2^{2\lambda}} + \frac{Q_{\mathsf{RO}} \cdot \varepsilon_1}{2^{\kappa_1}} + \frac{Q_{\mathsf{RO}} \cdot \varepsilon_2}{2^{\kappa_2}} + \frac{Q_{\mathsf{RO}} \cdot \varepsilon_3}{2^{\kappa_3}} + \frac{Q_{\mathsf{RO}} \cdot \varepsilon_4}{2^{\kappa_4}} \tag{5}$$

where

$$\varepsilon_1 = \binom{N}{d_\beta + 2} \cdot \frac{1}{|\mathbb{F}|^\eta} \cdot \left(1 + \frac{n_{\mathsf{decs}}^{\eta+1}}{|\mathbb{F}|}\right) \ , \quad \varepsilon_2 = \frac{1}{|\mathbb{F}|^\rho} \cdot \left(1 + \frac{(m_1 \cdot s + m_2)^{\rho+1}}{|\mathbb{F}|}\right) \ , \quad \varepsilon_3 = \frac{\binom{d_Q}{\ell'}}{\binom{|\mathbb{F}|}{\ell'}} \ , \quad \varepsilon_4 = \frac{\binom{n_{\mathsf{cols}}+\ell-1}{\ell}}{\binom{N}{\ell}} \ . \tag{6}$$

Proof. We first prove that the CAPSS signature scheme is secure against existential unforgeability with key only (EUF-KO), where the adversary has no access to a signing oracle. We then show how this result extends to standard existential unforgeability under chosen-message attacks (EUF-CMA).

EUF-KO security. The EUF-KO security of the CAPSS signature scheme follows from the straightline-extractable knowledge soundness of SmallWood-ARK and the one-wayness of the function $\mathsf{OWF}_{\mathcal{P},iv}$. Recall that a CAPSS signature is, by construction, a SmallWood-ARK proof for the statement $\mathsf{OWF}_{\mathcal{P},iv}(x) = y$ expressed in PACS syntax, where iv and y are public and x is the secret witness.

The SNARK-friendly verification tweaks introduced in Section 4 do not affect the soundness of SmallWood-ARK, except for one modification that alters the distributions $\mathcal{D}_\Gamma$ and $\mathcal{D}_{\Gamma'}$ used to sample batching challenges. As explained in Section 4, this change impacts the definitions of the round soundness errors ε_1 and ε_2 of SmallWood-ARK, which become:

$$\varepsilon_1 = \binom{N}{d_\beta + 2} \cdot \frac{1}{|\mathbb{F}|^\eta} \cdot \left(1 + \frac{n_{\mathsf{decs}}^{\eta+1}}{|\mathbb{F}|}\right) \quad \text{and} \quad \varepsilon_2 = \frac{1}{|\mathbb{F}|^\rho} \cdot \left(1 + \frac{(m_1 \cdot s + m_2)^{\rho+1}}{|\mathbb{F}|}\right) \ ,$$

(as in the present theorem statement) in contrast to the original (non-tweaked) case, for which $\varepsilon_1 = \binom{N}{d_\beta+2}/|\mathbb{F}|^\eta$ and $\varepsilon_2 = 1/|\mathbb{F}|^\rho$. Therefore, the tweaked version of SmallWood-ARK remains $(t, \varepsilon_{\text{SE-KS}})$-straightline-extractable knowledge sound, with the updated soundness error $\varepsilon_{\text{SE-KS}}$ satisfying the bounds given in Equations (5) and (6).

Assume there exists an EUF-KO adversary $\mathcal{A}$ that runs in time t and outputs a valid forgery with probability $\varepsilon_{\text{EUF-KO}}$. We construct a reduction algorithm $\mathcal{R}$ that breaks the one-wayness of $\mathsf{OWF}_{\mathcal{P},iv}$. The reduction $\mathcal{R}$ operates as follows: it runs $\mathcal{A}$ while simulating the random oracle using lazy sampling, and then invokes the straightline extractor Ext on the set of random oracle queries made by $\mathcal{A}$. If $\mathcal{A}$ outputs a valid forgery (i.e., a valid CAPSS signature or a valid tweaked $\mathsf{SmallWood\text{-}ARK}$ proof), then –by the straightline-extractable knowledge soundness of the tweaked $\mathsf{SmallWood\text{-}ARK}$ scheme– Ext fails to produce a valid witness for the statement $\mathsf{OWF}_{\mathcal{P},iv}(x) = y$ with probability at most $\varepsilon_{\text{SE-KS}}$. Therefore, the probability $\varepsilon_{\mathcal{R}}$ that $\mathcal{R}$ succeeds in inverting $\mathsf{OWF}_{\mathcal{P},iv}$ satisfies:

$$\varepsilon_{\text{EUF-KO}} - \varepsilon'_{\text{SE-KS}} \leq \varepsilon_{\mathcal{R}} \leq \varepsilon_{\text{OW}} ,$$

which implies the bound:

$$\varepsilon_{\text{EUF-KO}} \leq \varepsilon'_{\text{SE-KS}} + \varepsilon_{\text{OW}} .$$

EUF-CMA security. Assume there exists an EUF-CMA adversary $\mathcal{A}$ that runs in time t and outputs a valid forgery with probability $\varepsilon_{\text{EUF-CMA}}$. We construct a reduction algorithm $\mathcal{R}$ that breaks the EUF-KO security game. The reduction $\mathcal{R}$ operates by running $\mathcal{A}$, answering its random oracle queries via lazy sampling, and responding to its signature queries using the zero-knowledge simulator of $\mathsf{SmallWood\text{-}ARK}$ [FR25]. We note that the SNARK-friendly tweaks (described in Section 4) do not affect the zero-knowledge property of $\mathsf{SmallWood\text{-}ARK}$. The simulator provides perfect simulations of CAPSS signatures, except when one of the following failure events occurs:

1. *Salt collision:* Two different simulated signatures happen to use the same salt value. Since salts are sampled uniformly from $\{0,1\}^{2\lambda}$, this occurs with probability at most $Q^2_{\mathsf{sign}}/2^{2\lambda}$.
2. *Failure in DECS simulation:* As detailed in the proof of hiding property of the DECS scheme in [FR25], this occurs if the adversary makes a query of the form $(\mathsf{salt}, \boldsymbol{p}, \boldsymbol{m}, j, \rho_j)$ to Hash where:
   - the salt value salt was previously used in a signature simulation,
   - the query targets a leaf index j which was *not* opened in the simulated signature,
   - and the same tape ρ_j was used.

   For each such query, this occurs with probability at most $1/2^{\lambda}$ due to the randomness of the tapes ρ_j. Over all Q_{RO} hash queries, the failure probability is bounded by $Q_{\mathsf{RO}}/2^{\lambda}$. If this event does not occur, the DECS commitments and openings in the simulated signatures are perfectly indistinguishable from real ones (c.f. the theorem stating the DECS' hiding property in [FR25]).
3. *Failure in Fiat–Shamir oracle programming:* As shown in the proof of the zero-knowledge property of $\mathsf{SmallWood\text{-}ARK}$ in [FR25], a failure may occur when programming the random oracles used for Fiat–Shamir challenges during signature generation. The probability of such a failure is at most $Q_{\mathsf{sign}} \cdot Q_{\mathsf{RO}}/2^{2\lambda}$.

In total, the probability that $\mathcal{R}$ fails to simulate signatures correctly (i.e., any of the above events occur) is bounded by:

$$\varepsilon_{\text{ZK}} \leq \frac{Q^2_{\mathsf{sign}}}{2^{2\lambda}} + \frac{Q_{\mathsf{RO}}}{2^{\lambda}} + \frac{Q_{\mathsf{sign}} \cdot Q_{\mathsf{RO}}}{2^{2\lambda}} .$$

If no failure occurs, the signature oracle is perfectly simulated, and the adversary $\mathcal{A}$ produces a valid forgery with probability $\varepsilon_{\text{EUF-CMA}}$. Hence, the reduction succeeds in the EUF-KO game with probability:

$$\varepsilon_{\text{EUF-KO}} \geq \varepsilon_{\text{EUF-CMA}} - \varepsilon_{\text{ZK}} ,$$

which concludes the proof. □

6 Instances and Benchmarks

This section presents concrete instances of signature schemes built from the CAPSS framework and compare their performances to other signature schemes from the literature. We apply the CAPSS framework to four families of permutations: Anemoi [BBC+23], Griffin [GHR+23], Poseidon [GKR+21], and RescuePrime [AAB+20, SAD20]. For each family, we propose three instances with different trade-offs between signature size, signing time and verification complexity (running time and number of constraints). Each trade-off corresponds to a different size for the Merkle tree $N \in \{2^{10}, 2^{12}, 2^{14}\}$:

– "Short" trade-off: We use $N = 2^{14}$ and trees of small arities. This trade-off aims at small signature sizes.
– "Default" trade-off: We use $N = 2^{12}$. This trade-off aims at a good balance between signing time, signature size and verification complexity.
– "Fast" trade-off: We use $N = 2^{10}$. This trade-off aims at fast signing time.

We stress that for any of the above trade-offs, verification is much faster than signing.

Performance of our instances. Table 2 summarizes the performance of our instances for a 256-bit field, where "Perm-α" denotes the family of permutations Perm with α the degree of the round verification function introduced in Section 3. The signature size is mainly determined by the witness size, i.e., the size $s \cdot n$ of the witness matrix. For Anemoi, Griffin and RescuePrime, we rely on the arithmetization for regular permutations. This arithmetization benefits from small witnesses due to its low redundancy and the small number of rounds in these permutations. Specifically, the number of witness coefficients is 48 for Anemoi-3, 56 for Anemoi-5, 60 for Griffin-3/5, 75 for RescuePrime-3, and 60 for RescuePrime-5. Anemoi yields slightly smaller witnesses because we can use a permutation with state of size 2 to instantiate the one-way function (while we need at least 3 for the others). Unfortunately, we cannot use the same arithmetization for Poseidon because of its irregular structure. We then use the S-box-centric arithmetization for this family, but it leads to larger witnesses: Poseidon-3/5 requires 180 witness coefficients. This is due to the higher degree of redundancy in the S-box-centric arithmetization which includes both the input and output of each S-box to the witness. For this reason, we achieve signature sizes between 10 KB and 17 KB for Anemoi, Griffin and RescuePrime, while we achieve sizes between 16 KB and 26 KB for Poseidon.

Table 2 also provides the SNARK-friendliness of each instance by measuring the number of R1CS constraints for the verification algorithms. These constraints have been counted based on a Python proof-of-concept implementation[4] explicitly building the verification arithmetic circuit. We can see that the (non-fast) instances derived from Anemoi and Griffin have a number of R1CS constraints ranging between 20 K and 30 K, while the instances derived from Poseidon and RescuePrime are between 40 K and 55 K. This is not surprising, as the verification complexity is primarily dictated by the underlying permutations. Consequently, the number of R1CS constraints in signature verification is largely determined by the constraints dedicated to these permutations. For example, in the default trade-off of Anemoi-3, 84% of the R1CS constraints are devoted to verifying the permutations. Since Anemoi and Griffin are more efficient in terms of R1CS constraints compared to Poseidon and RescuePrime, the resulting CAPSS signature naturally inherits this efficiency hierarchy.

We implemented two specific instances of the CAPSS framework, both relying on the Anemoi permutation.[5] The first instance is defined over the BN254's field (a 254-bit field), while the second one is defined over the 64-bit Goldilocks field $\mathbb{F}_{2^{64}-2^{32}+1}$. Table 3 provides the obtained running times. The schemes have been compiled with Debian clang version 19.1.7 and benchmarked on an AMD Ryzen Threadripper PRO 7995WX. While we do not have a precise number of R1CS constraints for the verification algorithm when working over the 64-bit field, we estimate this number in the range $80\,000 - 100\,000$.

To provide further insights on the R1CS cost, we give in Table 4 the distribution of R1CS constraints among the different components of the verification algorithm for the CAPSS-Anemoi instances for a 256-bit field. We observe that the bottleneck is the verification of the authentication paths in the Merkle tree –from 34% to 53%– despite the tweaks described in Section 4. The next bottlenecks come from hashing the leaves and generating the Fiat-Shamir challenges. While hashing the leaves is expensive (18%–28%), generating the Fiat-Shamir challenges is also far from being negligible, notably the one that require hashing the polynomials $\boldsymbol{R}$ in the DECS (11%–13%) because of the relatively high degree of this vector polynomial.

Comparison to the state of the art. Table 5 compares our signature schemes with the other symmetric-based post-quantum signature schemes from the state of the art. Besides the CAPSS instances, all the numbers of R1CS constraints are taken from the estimates given in [ZSE+24].

[4] The proof-of-concept implementation is available at `https://github.com/CryptoExperts/smallwood-python`.
[5] The source code is available at `https://github.com/CryptoExperts/smallwood`.

Table 2: Parameters, sizes and number of R1CS constraints (for the verification) of CAPSS signature instances.

Permutation Family	**PACS Statement**					**Proof System**						**Sig. Size**	**#R1CS**
	t	n_r	s	b	n	Trade-off	ℓ	N	Arity	η	w		
Anemoi-3 (256-bit field)	2	21	3	7	16	Short	13	16 384	2^{14}	2	8	9 504 B	24 671
						Default	17	4 096	4^6	2	7	12 640 B	23 484
						Fast	24	1 024	4^5	2	8	14 368 B	29 086
Anemoi-5 (256-bit field)	2	21	4	6	14	Short	13	16 384	2^{14}	2	8	10 304 B	30 719
						Default	17	4 096	4^6	2	7	13 568 B	28 991
						Fast	24	1 024	4^5	2	8	15 520 B	35 485
Griffin-3 (256-bit field)	3	16	4	4	15	Short	12	19 683	3^9	2	13	10 720 B	20 717
						Default	17	4 096	4^6	2	7	13 216 B	24 855
						Fast	25	1 024	4^5	2	5	15 680 B	32 142
Griffin-5 (256-bit field)	3	14	4	4	15	Short	12	19 683	3^9	2	13	11 168 B	21 994
						Default	17	4 096	4^6	2	7	13 664 B	23 724
						Fast	25	1 024	4^5	2	5	16 128 B	30 915
Poseidon-3 (256-bit field)	3	-	15	-	16	Short	12	19 683	3^9	2	13	16 448 B	39 790
						Default	17	3 888	$4^2 \cdot 3^5$	2	7	19 616 B	46 567
						Fast	25	972	$4 \cdot 3^5$	2	5	25 056 B	60 223
Poseidon-5 (256-bit field)	3	-	15	-	12	Short	12	19 683	3^9	2	13	17 216 B	47 750
						Default	17	3 888	$4^2 \cdot 3^5$	2	7	20 384 B	55 428
						Fast	25	972	$4 \cdot 3^5$	2	5	25 824 B	71 145
RescuePrime-3 (256-bit field)	3	18	5	4	15	Short	12	19 683	3^9	2	13	11 232 B	40 128
						Default	17	4 096	4^6	2	7	13 888 B	43 388
						Fast	25	1 024	4^5	2	5	16 608 B	54 643
RescuePrime-5 (256-bit field)	3	14	5	3	12	Short	12	19 683	3^9	2	13	11 456 B	47 026
						Default	17	4 096	4^6	2	7	14 112 B	50 549
						Fast	25	1 024	4^5	2	5	16 832 B	63 812

Table 3: Running times of some CAPSS signature instances, in milliseconds.

Permutation Family	**Trade-off**	**Sig. Size**	**KeygGen Time**	**Signing Time**	**Verif. Time**
Anemoi-5 (BN254's field)	Short	9 092 B	0.1	9900	29
	Default	11 486 B	0.1	2460	29
	Fast	12 337 B	0.1	710	41
Anemoi-7 (Goldilocks field)	Short	9 084 B	< 0.01	394	1.2
	Default	9 715 B	< 0.01	101	1.4
	Fast	10 729 B	< 0.01	29	1.6

The first part of the table provides known schemes which do not specifically target at SNARK-friendliness. As our signature schemes, SPHINCS$^+$ verification also involves verifying Merkle paths. While this scheme could be also made SNARK-friendly by using an arithmetization-oriented hash function, this would result in

Table 4: Distribution of R1CS constrains for 256-bit-field CAPSS-Anemoi instances.

		Anemoi-3			Anemoi-5		
		Short	Default	Fast	Short	Default	Fast
Verify	$\mathsf{XOF}(h_{\mathsf{piop}})$	112	112	112	140	140	140
	$\mathsf{XOF}(\mathsf{transcript}_{\mathsf{piop}})$	560	560	560	1260	1260	1260
PCS	Computation of $\{(\hat{\boldsymbol{v}}_j)_1\}_j$	9	9	9	12	12	12
LVCS	$\mathsf{XOF}(\mathsf{transcript}, \{\boldsymbol{v}_k, \bar{\boldsymbol{v}}_k\}_k)$	1344	1456	1680	1680	1820	2100
	$\mathsf{RecomputeOpeningChallenge}(\mathsf{counter}, \mathsf{trans\_hash})$	254	291	394	282	319	425
	Computation of $\{\boldsymbol{v}'_k\}_k$	819	1207	2040	819	1207	2040
	Computation of $\{(\boldsymbol{p}_h^{(\mathrm{eval},j)})_{h \le m}\}_j$	53	69	97	66	86	121
DECS	$\mathsf{XOF}(\mathsf{salt}, \boldsymbol{p}^{(\mathrm{eval},j)}, \boldsymbol{m}^{(\mathrm{eval},j)})$	4 368	5 712	8 064	5 460	7 140	10 080
	$\mathsf{RetrieveRootFromPath}(\{u_{i_j}\}_j, \mathsf{auth}, I)$	13 116	9 264	9 968	16 161	11 308	12 124
	$\mathsf{XOF}(\mathsf{salt}, \mathsf{root})$	112	112	112	140	140	140
	$\mathsf{XOF}(h_{\mathrm{mt}})$	118	118	118	148	148	148
	Computation of $\boldsymbol{R}$	962	1394	2304	988	1428	2352
PIOP	$\mathsf{XOF}(\mathsf{msg} \parallel \mathsf{transcript}_{\mathsf{pcs}})$	2735	3071	3519	3415	3835	4395
	Computation of $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$	108	108	108	147	147	147
Total		24 671	23 484	29 086	30 719	28 991	35 485

Table 5: Comparison of post-quantum signatures based on symmetric-key primitives. Except for the CAPSS instances, the numbers of R1CS constraints are from [ZSE+24]. The CAPSS instances are based on a 256-bit field. Asterisks indicate estimated timings.

Signature Scheme	**Sig. Size**	**#R1CS**	**Signing Time**	**Verif. Time**	**Assumptions**
SPHINCS$^+$s	8 KB	$\approx$ 460 K	$\approx$ 200 ms	$\approx$ 1 ms	Hash
SPHINCS$^+$f	16 KB	$\approx$ 1 400 K	$\approx$ 14	$\approx$ 2 ms	Hash
Picnic1	32 KB	$\approx$ 3 500 K	$\approx 1-2$ ms	$\approx 1-2$ ms	LowMC + Hash
Picnic3	12 KB	$\approx$ 21 600 K	$\approx$ 5 ms	$\approx$ 4 ms	LowMC + Hash
LegRoast	16 KB	$\approx$ 1 100 K	$\approx$ 3 ms	$\approx$ 3 ms	Leg. PRF + Hash
Banquet	12 KB	$\approx$ 11 800 K	$\approx$ 40 ms	$\approx$ 40 ms	AES + Hash
Rainer	8 KB	$\approx$ 26 100 K	$\approx$ 1 ms	$\approx$ 1 ms	Rain + Hash
FAEST	$\approx 4-5$ KB	–	$\approx 0.5-4$ ms	$\approx 0.5-4$ ms	AES + Hash
Loquat-128 (Keccak) [ZSE+24]	57 KB	–	5.0 s	0.2 s	Legendre PRF + Keccak
Loquat-128 (Griffin) [ZSE+24]	57 KB	$\approx$ 150 K	105 s	11 s	Legendre PRF + Griffin
Loquat*-128 (Keccak) [ZSE+24]	114 KB	–	5.0 s	0.2 s	Legendre PRF + Keccak
Loquat*-128 (Griffin) [ZSE+24]	114 KB	$\approx$ 300 K	214 s	25 s	Legendre PRF + Griffin
RescuePrime + STARKs [AdSGK24]	80–100 KB	–	9–23 ms	1 ms	RescuePrime (+ Blake3)
RescuePrime + STARKs [AdSGK24]	80–100 KB	–	94–370 ms	21–27 ms	RescuePrime
CAPSS-Anemoi	9–16 KB	24 K – 36 K	400 ms – 6 s *	22–28 ms *	Anemoi
CAPSS-Griffin	10–16 KB	20 K – 32 K	200 ms – 6 s *	12–17 ms *	Griffin
CAPSS-Poseidon	16–26 KB	40 K – 71 K	100 ms – 3 s *	4–7 ms *	Poseidon
CAPSS-RescuePrime	11–17 KB	40 K – 64 K	500 ms – 14 s *	31–47 ms *	RescuePrime

very slow signing times because of the large number of hash computations required in a SPHINCS$^+$ signature computation.[6]

The next schemes in the first part of the table are all based on the MPC-in-Head paradigm [IKOS07]. In these schemes, the signature verification requires the re-computation of GGM trees, which results in a large

[6] According to [ZSE+24], a SPHINCS$^+$ signature requires more than 100 K hash computations for its fast instance and more than 2000 K hash computations for its short instance. This is greater than the number of hash computations in our fast and short instances by two order of magnitudes.

number of R1CS constraints. FAEST [BBD$^+$23b] is the most recent scheme in this category, which achieves signature sizes around 5 KB using the VOLE-in-the-Head technique [BBD$^+$23a]. While the number of R1CS constraints for this scheme is not estimated in [ZSE$^+$24], it should be above the other ones, given the fact that FAEST makes greedier use of GGM trees.

The second part of the table includes two recent schemes. The first one, Loquat [ZSE$^+$24], is a signature scheme based on the Legndre PRF and targeting SNARK-friendliness. Our schemes clearly outperforms Loquat in terms of signature size, timings and R1CS constraints for the verification. For instance, our CAPSS-Anemoi instances achieve a 4–6× reduction in signature size and a 5–8× reduction in R1CS constraints compared to Loquat. Another advantage of our approach compared to Loquat is that the security of our schemes solely relies on the underlying family of permutations, whereas Loquat further relies on the security of the Legendre PRF. The second scheme is from a recent independent work [AdSGK24] which applies a STARK proof system to the RescuePrime permutation. While this methodology is similar to ours (namely applying a hash-based proof system to an arithmetization-oriented permutation), our work goes further in the optimization of this approach, notably thanks to the SmallWood-ARK proof system, which is specifically designed to obtain small proofs. For this reason, we obtain much shorter signatures than [AdSGK24].

7 Applications

7.1 Aggregated Signatures

A typical use case for SNARK-friendly signatures is the construction of aggregated signatures. In contexts where storage space is limited or costly (for instance, in blockchain environments), it is often desirable to combine multiple signatures into a single compact one. The goal is that the size of this aggregated signature be significantly smaller than the total size of the individual signatures taken separately. Thanks to their structure, CAPSS signatures can be aggregated efficiently using a generic SNARK. In what follows, we describe the methodology we adopted to achieve such an aggregation.

Since the CAPSS framework produces quantum-resistant signature schemes, it is natural to ensure that this property is preserved in the aggregation process. To that end, we rely on a post-quantum SNARK. Among the available options, we focused on the hash-based SNARK Aurora [BCR$^+$19], because a practical and user-friendly C++ library is available for it, namely `libiop` [BCR$^+$19].

Our approach is as follows: starting from our Python proof-of-concept implementation of the CAPSS framework, we construct an R1CS encoding that corresponds to the verification of a CAPSS signature. We then extend this system with n R1CS assignments, each one representing the verification of an individual signature transcript. These data are subsequently imported[7] into Aurora, which allows us to generate a succinct proof π. This proof certifies that we know n valid CAPSS signatures for given messages $m_1, \ldots,$ m_n (that might all be equal) and given public keys $pk_1, \ldots pk_n$, hence acting as their aggregated signature. The storage benefit comes from the fact that, whenever $|\pi|/n$ is smaller than the size of a single CAPSS signature, the aggregated construction reduces memory requirements. In Table 6 we report the measured running times for both the prover and the verifier, as well as the size of the proof π, when considering the CAPSS signature scheme based on Anemoi-3 over a 256-bit field. All benchmarks were conducted on an AMD Ryzen Threadripper PRO 7995WX. Our experiments show that the aggregation becomes advantageous when $n > 16$. For $n = 1024$, the amortized size of an aggregated CAPSS signature drops to 260 bytes. In comparison, Loquat [ZSE$^+$24], a recent post-quantum SNARK-friendly signature scheme, requires around 125,000 R1CS constraints to verify a single signature at a 100-bit security level (versus 128-bit in our context). For a given n, the underlying number of R1CS constraints is roughly four times larger with Loquat, which results in higher aggregation and verification costs. For example, aggregating 1024 CAPSS signatures can be done with similar efficiency as aggregating only 256 Loquat signatures.

A recent work has highlighted the use of SNARKs to aggregate hash-based signatures, with a concrete application to a post-quantum variant of Ethereum's proof-of-stake consensus [DKKW25]. While their approach shares similarities with ours, it relies on "synchronized" hash-based signatures, which limit each key

[7] The source code is available at `https://github.com/CryptoExperts/capss-aggregation`.

n	1	4	16	64	256	1024
# R1CS	$\leq 2^{15}$	$\leq 2^{17}$	$\leq 2^{19}$	$\leq 2^{21}$	$\leq 2^{23}$	$\leq 2^{25}$
t_A (s)	2.6	12	49	208	906	3872
t_V (s)	0.5	1.9	7.8	31	127	513
$\|\pi\|$ (KB)	128	153	181	209	241	271
$\|\pi\|/n$ (KB)	128	38	11	3.3	0.94	0.26

Table 6: Aggregation of CAPSS signatures (based on Anemoi-3 over a 256-bit field, "default" trade-off) using Aurora. The used internal Aurora's parameter: the RS extra dimensions is set as 3 and the FRI location parameter is set as 3. t_A=aggregation time, t_V=verification time, n =number of signatures being aggregated.

pair to a bounded number of signatures and require maintaining some state. In contrast, our work builds signature schemes that can be used without such restrictions.

7.2 Anonymous Credentials

An anonymous credential is a cryptographic primitive that allows a user to obtain a credential from an issuer and later prove possession of certain attributes (e.g., "over 18", "member of group X") without revealing their identity or unnecessary information [CL01, CV02]. It ensures both *selective disclosure* (revealing only chosen attributes) and *unlinkability* (different uses of the same credential cannot be linked to the same user).

A general blueprint for anonymous credentials is the following:

1. Issuance: The user commits to their attributes m and requests a credential. The issuer signs these attributes $\sigma = \mathsf{Sign}_{\mathrm{sk}_I}(m)$ with their secret key sk_I, which might be done blindly. The credential is composed of the pair (m, σ).
2. Presentation: To prove possession of the credential, the user constructs a non-interactive zero-knowledge proof:
$$\pi = \mathsf{NIZK}\big(\exists\, (m, \sigma) : \mathsf{Ver}_{\mathrm{pk}_I}(m, \sigma) = 1 \;\wedge\; \Phi(m)\big)$$
where $\Phi(m)$ encodes what is disclosed. By verifying π, the verifier is ensured that the user has a valid credential with attributes satisfying Φ.

# R1CS for Φ	1	2^{15}	2^{20}
Total # R1CS	$\approx 2^{15}$	$\approx 2^{16}$	$\approx 2^{20}$
t_P (s)	6.4	13	482
t_V (s)	0.5	0.8	29
$\|\pi\|$ (KB)	151	140	236

Table 7: Proving possession of the credential using Aurora, with CAPSS signatures based on Anemoi-3 over a 256-bit field (trade-off "default"). The used internal Aurora's parameter: the RS extra dimensions is set as 2, the FRI location parameter is set as 3 and the zero-knowledge mode is set. t_P=proving time, t_V=verification time.

In our context, the credential is encoded by a vector of field elements $m \in \mathbb{F}^n$, which are hashed using an AO permutation in XOF mode and signed with a CAPSS signature. The disclosure predicate Φ is represented as an arithmetic circuit over $\mathbb{F}$, and we instantiate the NIZK with Aurora. In Table 7 we report the running times for both the prover (the user) and the verifier, as well as the size of the proof π, when considering the CAPSS signature scheme based on Anemoi-3 over a 256-bit field. All benchmarks were conducted on an AMD Ryzen Threadripper PRO 7995WX. For a fairly large circuit of 2^{15} R1CS constraints encoding Φ we obtain a presentation proof of 140 KB with verification time below 1 second. These results indicate that combining a SNARK-friendly signature such as CAPSS with a zk-SNARK, both relying exclusively on symmetric cryptography, offers a promising path toward post-quantum anonymous credentials and provides a compelling alternative to existing lattice-based constructions [BLNS23, KLN25].

Acknowledgements

This research was supported by the Ethereum Foundation, under the 2024 Academic Grants Round. The authors would like to thank Léo Perrin for insightful discussions on arithmetization-friendly permutations.

References

AAB$^+$20. Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Trans. Symm. Cryptol.*, 2020(3):1–45, 2020.

ADK24. Shahla Atapoor, Cyprien Delpech de Saint Guilhem, and Al Kindi. STARK-based signatures from the RPO permutation. Cryptology ePrint Archive, Report 2024/1553, 2024.

AdSGK24. Shahla Atapoor, Cyprien Delpech de Saint Guilhem, and Al Kindi. STARK-based signatures from the RPO permutation. Cryptology ePrint Archive, Paper 2024/1553, 2024.

AKM$^+$22. Tomer Ashur, Al Kindi, Willi Meier, Alan Szepieniec, and Bobbin Threadbare. Rescue-prime optimized. Cryptology ePrint Archive, Report 2022/1577, 2022.

BBC$^+$23. Clémence Bouvier, Pierre Briaud, Pyrros Chaidos, Léo Perrin, Robin Salen, Vesselin Velichkov, and Danny Willems. New design techniques for efficient arithmetization-oriented hash functions: Anemoi permutations and Jive compression mode. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part III*, volume 14083 of *LNCS*, pages 507–539. Springer, Cham, August 2023.

BBD$^+$23a. Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 581–615. Springer, Cham, August 2023.

BBD$^+$23b. Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. FAEST: Algorithm Specifications – Version 1.1, 2023. `https://faest.info/faest-spec-v1.1.pdf`.

BBHR19. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 701–732. Springer, Cham, August 2019.

BBL$^+$24. Augustin Bariant, Aurélien Boeuf, Axel Lemoine, Irati Manterola Ayala, Morten Øygarden, Léo Perrin, and Håvard Raddum. The algebraic FreeLunch: Efficient Gröbner basis attacks against arithmetization-oriented primitives. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part IV*, volume 14923 of *LNCS*, pages 139–173. Springer, Cham, August 2024.

BCG20. Jonathan Bootle, Alessandro Chiesa, and Jens Groth. Linear-time arguments with sublinear verification from tensor codes. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 19–46. Springer, Cham, November 2020.

BCR$^+$19. Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Cham, May 2019.

BDPA11. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions, 2011.

BDPV08. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 181–197. Springer, Berlin, Heidelberg, April 2008.

BGLS03. Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 416–432. Springer, Berlin, Heidelberg, May 2003.

BLNS23. Jonathan Bootle, Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Alessandro Sorniotti. A framework for practical anonymous credentials from lattices. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part II*, volume 14082 of *LNCS*, pages 384–417. Springer, Cham, August 2023.

Cha82. David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *CRYPTO'82*, pages 199–203. Plenum Press, New York, USA, 1982.

CL01. Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 93–118. Springer, Berlin, Heidelberg, May 2001.

CV02. Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 21–30. ACM Press, November 2002.

DKKW25. Justin Drake, Dmitry Khovratovich, Mikhail A. Kudinov, and Benedikt Wagner. Hash-based multi-signatures for post-quantum ethereum. *CiC*, 2(1):13, 2025.

Fis06. Marc Fischlin. Round-optimal composable blind signatures in the common reference string model. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 60–77. Springer, Berlin, Heidelberg, August 2006.

FR23. Thibauld Feneuil and Matthieu Rivain. Threshold computation in the head: Improved framework for post-quantum signatures and zero-knowledge arguments. Cryptology ePrint Archive, Report 2023/1573, 2023.

FR25. Thibauld Feneuil and Matthieu Rivain. SmallWood: Hash-based polynomial commitments and zero-knowledge arguments for relatively small instances. Cryptology ePrint Archive, Paper 2025/1085, 2025.

GHR$^+$23. Lorenzo Grassi, Yonglin Hao, Christian Rechberger, Markus Schofnegger, Roman Walch, and Qingju Wang. Horst meets fluid-SPN: Griffin for zero-knowledge applications. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part III*, volume 14083 of *LNCS*, pages 573–606. Springer, Cham, August 2023.

GKR$^+$21. Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 519–535. USENIX Association, August 2021.

GKS23. Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. Poseidon2: A faster version of the poseidon hash function. In Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne, editors, *AFRICACRYPT 23*, volume 14064 of *LNCS*, pages 177–203. Springer, Cham, July 2023.

GLS$^+$23. Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part II*, volume 14082 of *LNCS*, pages 193–226. Springer, Cham, August 2023.

Hir18. Shoichi Hirose. Sequential hashing with minimum padding. *Cryptography*, 2:11, 06 2018.

IKOS07. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

KLN25. Victor Youdom Kemmoe, Anna Lysyanskaya, and Ngoc Khanh Nguyen. Lattice-based accumulator and application to anonymous credential revocation. Cryptology ePrint Archive, Paper 2025/1099, 2025.

Lee21. Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part II*, volume 13043 of *LNCS*, pages 1–34. Springer, Cham, November 2021.

SAD20. Alan Szepieniec, Tomer Ashur, and Siemen Dhooghe. Rescue-prime: a standard specification (SoK). Cryptology ePrint Archive, Report 2020/1143, 2020.

ZSE$^+$24. Xinyu Zhang, Ron Steinfeld, Muhammed F. Esgin, Joseph K. Liu, Dongxi Liu, and Sushmita Ruj. Loquat: A SNARK-friendly post-quantum signature based on the Legendre PRF with applications in ring and aggregate signatures. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part I*, volume 14920 of *LNCS*, pages 3–38. Springer, Cham, August 2024.