PERK

20/12/2025

Submitters (alphabetical order):

- Najwa Aaraj (Technology Innovation Institute, UAE)
- Slim Bettaieb (Technology Innovation Institute, UAE)
- Loïc Bidoux (Technology Innovation Institute, UAE)
- Alessandro Budroni (Technology Innovation Institute, UAE)
- Victor Dyseryn (XLIM, University of Limoges, FR)
- Andre Esser (Technology Innovation Institute, UAE)
- Thibauld Feneuil (Cryptoexperts, FR)
- Philippe Gaborit (XLIM, University of Limoges, FR)
- Mukul Kulkarni (Technology Innovation Institute, UAE)
- Victor Mateu (Technology Innovation Institute, UAE)
- Marco Palumbi (Technology Innovation Institute, UAE)
- Lucas Perin (Technology Innovation Institute, UAE)
- Matthieu Rivain (Cryptoexperts, FR)
- Jean-Pierre Tillich (INRIA, FR)
- Keita Xagawa (Technology Innovation Institute, UAE)

Contact: team@pqc-perk.org

Version: 2.2.0

Changelog

Version 2.2.0 (20/12/2025)

- Fix several bugs in the implementation.

Version 2.1.1 (07/10/2025)

- Fix typographical errors.

Version 2.1.0 (23/09/2025)

- The design of PERK has been improved and now relies on the modeling from [BBGK24] along with the VOLEitH framework [BBD+23c]. As a result, PERK signature sizes have been significantly reduced.

Version 2.0.0 (05/02/2025)

- Thibauld Feneuil, Matthieu Rivain and Keita Xagawa have joined the PERK team.

Version 1.1.0 (16/10/2023)

- Reduce signature sizes for short parameters set by approximately 5% using a ranking algorithm for permutation encoding ;

- Improve the implementation (reduced stack-memory usage and bug fixing).

Table of Contents

1 Introduction

PERK is a post-quantum digital signature scheme based on the hardness of the PERmuted Kernel Problem (PKP) [Sha90]. The scheme builds on a Zero-Knowledge Proof of Knowledge (ZK PoK) of a PKP solution computed using the modeling from [BBGK24] along with the Multi-Party Computation in the Head (MPCitH) paradigm [IKOS07]. More precisely, PERK relies on the VOLE in the Head (VOLEitH) framework [BBD+23c]. The ZK PoK is then converted into a signature scheme using the Fiat-Shamir transform [FS87].

Organization. We present in Section 2 some background and the notations we will use. Then, Section 3 and Section 4 respectively provide an overview and a detailed algorithmic description of PERK. Section 5 and Section 6 are dedicated to the parameters and the performances of PERK. A security analysis of the scheme is provided in Section 7. Finally, in Section 8, we summarize the main advantages and limitations of the scheme.

2 Preliminaries

2.1 Notations

Let λ denote the security parameter. For integers a, b we denote $[a, b]$ the set of integers i such that $a \leq i \leq b$. We write $[n]$ as a shorthand for $[0, n-1]$. We denote $\mathcal{S}_n$ the group of permutations of the set $[n]$. Let $\mathbb{F}_q$ denote the finite field of q elements where q is the power of a prime. If S is a finite set, we denote by $x \xleftarrow{\$} S$ that x is chosen uniformly at random from S. Similarly, we write $x \xleftarrow{\$,\theta} S$, if x is sampled pseudo-randomly from the set S, based on the seed θ. We use x to denote input and denote its length by $|x|$. Vectors are denoted by bold lower-case letters and matrices by bold capital letters (e.g., $\boldsymbol{v} = (v_1, \ldots, v_n) \in \mathbb{F}_q^n$ and $\boldsymbol{M} = (m_{ij})_{1 \leqslant i \leqslant k, 1 \leqslant j \leqslant n} \in \mathbb{F}_q^{k \times n}$). We denote by $\ker(\boldsymbol{M})$ the right kernel of the matrix $\boldsymbol{M}$.

We call a function $f : \mathbb{N} \to \mathbb{R}^+$ *negligible*, if for all $c \in \mathbb{N}$ there exists a $N_0 \in \mathbb{N}$ such that $f(n) < 1/n^c$ for all $n > N_0$. We write $\mathsf{negl}(\lambda)$ to denote an arbitrary negligible function. We use $\mathsf{poly}(\lambda)$ for function which is *polynomially bounded* in λ, that is there exists $c, \lambda_0 \in \mathbb{N}$ such that $\mathsf{poly}(\lambda) \leq \lambda^c$ for all $\lambda \geq \lambda_0$. We also abbreviate *probabilistic polynomial-time* as PPT. Let X and Y be two discrete random variables defined over a finite support D. The *statistical distance* between the two distributions is defined as

$$\Delta(X, Y) := \frac{1}{2} \sum_{d \in D} |\Pr[X = d] - \Pr[Y = d]|.$$

We say two ensembles of random variables $\{X_\lambda\}_{\lambda \in \mathbb{N}}$, $\{Y_\lambda\}_{\lambda \in \mathbb{N}}$ are *statistically close* if there exists a negligible function $\mathsf{negl} : \mathbb{N} \to \mathbb{R}^+$ such that $\Delta(X_\lambda, Y_\lambda) \leq \mathsf{negl}(\lambda)$ for all $\lambda \in \mathbb{N}$. We say two ensembles of random variables $\{X_x\}_{x \in \{0,1\}^*}$, $\{Y_x\}_{x \in \{0,1\}^*}$ are *statistically close* if there exists a negligible function $\mathsf{negl} : \mathbb{N} \to \mathbb{R}^+$ such that $\Delta(X_x, Y_x) \leq \mathsf{negl}(|x|)$ for all $x \in \{0,1\}^*$.

2.2 Standard cryptographic primitives

Definition 2.1 (Salt based PRG (adapted from [BBD+23b])). *Let* $\mathsf{PRG} : \{0,1\}^{2\lambda} \times \{0,1\}^\lambda \to \{0,1\}^{\hat{\ell}}$ *be a deterministic polynomial-time algorithm. Let* $\mathcal{A}$ *be a* q*-query,* N*-batch adversary. We define an experiment* $\mathsf{Expt}^{\text{mt-prg}}_{\mathsf{PRG},\mathcal{A}}(q, N, \lambda)$ *as in Figure 1. Let*

$$\mathsf{AdvPRG}^{\mathsf{PRG}}[q, N, \lambda] := \left| \Pr\left[\mathsf{out}_{\mathcal{A}} = 1 \,:\, \mathsf{out}_{\mathcal{A}} \leftarrow \mathsf{Expt}^{\text{mt-prg}}_{\mathsf{PRG},\mathcal{A}}(q, N, \lambda)\right] - \frac{1}{2} \right|.$$

Then we call PRG *as* $((q, N, \lambda), (t, \varepsilon))$*-secure if for every adversary* $\mathcal{A}$ *running in time* t*, its advantage* $\mathsf{AdvPRG}^{\mathsf{PRG}}[q, N, \lambda]$ *is at most* ε*. If* λ *is obvious in the context, we will drop* λ*.*

$\mathsf{Expt}^{\text{mt-prg}}_{\mathsf{PRG},\mathcal{A}}(q, N, \lambda)$

$b \xleftarrow{\$} \{0,1\}$
for $i \in [q]$: $\mathsf{salt}_i \xleftarrow{\$} \{0,1\}^{2\lambda}$
  for $j \in [N]$:
    if $b = 0$:
      $\mathsf{seed}_j \xleftarrow{\$} \{0,1\}^{\lambda}$
      $r_{i,j} \coloneqq \mathsf{PRG}(\mathsf{salt}_i, \mathsf{seed}_j)$
    else : $r_{i,j} \xleftarrow{\$} \{0,1\}^{\hat{\ell}}$
$b' \leftarrow \mathcal{A}\left(\{\mathsf{salt}_i\}_{i\in[q]}, \{r_{i,j}\}_{(i,j)\in[q]\times[N]}\right)$
if $b = b'$: **return** 1
else : **return** 0

Fig. 1: Multi-challenge security of salt-based PRG

Note that, any $((1,1,\lambda),(t,\varepsilon_{\mathsf{PRG}}))$-secure salt based PRG is also $((q,N,\lambda),(t,\varepsilon'_{\mathsf{PRG}}))$-secure with $\varepsilon'_{\mathsf{PRG}} \leq q \cdot N \cdot \varepsilon_{\mathsf{PRG}}$.

Definition 2.2 (Collision Resistance). *Let $H : \{0,1\}^{\ell_{\mathsf{in}}} \to \{0,1\}^{\ell_{\mathsf{out}}}$ be a deterministic function. For any adversary making at most q queries to H, we denote its advantage in finding a collision for H by $\mathsf{AdvColl}^{H}[q]$. We say that H is $(q,(t,\varepsilon))$-secure if for every adversary $\mathcal{A}$ running in time t, its advantage $\mathsf{AdvColl}^{H}[q]$ is at most ε.*

Definition 2.3 (Multi-Target Non-Invertibility). *Let $\mathsf{Com} : \{0,1\}^{\ell_{\mathsf{in}}} \to \{0,1\}^{\ell_{\mathsf{out}}}$ be a deterministic function. For a q-query adversary, we define its advantage as*

$$\mathsf{AdvNI}^{\mathsf{Com}}[q] \coloneqq \Pr\left[\mathsf{Com}(\mathsf{inp}) = \mathsf{com}_i : \begin{array}{l}\mathsf{com}_0,\ldots,\mathsf{com}_{q-1} \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{out}}} \\ (i,\mathsf{inp}) \leftarrow \mathcal{A}(\mathsf{com}_0,\ldots,\mathsf{com}_{q-1})\end{array}\right].$$

We say that Com is $(q,(t,\varepsilon))$-secure if for every adversary $\mathcal{A}$ running in time t, its advantage $\mathsf{AdvNI}^{\mathsf{Com}}[q]$ is at most ε.

Definition 2.4 (One-time PRF). *Let $F : \{0,1\}^{2\lambda} \times \{0,1\}^{*} \to \{0,1\}^{3\lambda}$ be a deterministic polynomial-time algorithm. Let an adversary $\mathcal{A}$, we define its advantage as*

$$\mathsf{AdvPRF}^{F} \coloneqq \left|\Pr\left[b = b' : \begin{array}{l} b \xleftarrow{\$} \{0,1\}; \mathsf{rand} \xleftarrow{\$} \{0,1\}^{2\lambda}; (\mathsf{state},\mathsf{inp}) \leftarrow \mathcal{A}() \\ r_0 \coloneqq F(\mathsf{rand},\mathsf{inp}); r_1 \xleftarrow{\$} \{0,1\}^{3\lambda}; b' \leftarrow \mathcal{A}(r_b,\mathsf{state})\end{array}\right] - \frac{1}{2}\right|.$$

We say that F is (t,ε)-secure if for every adversary $\mathcal{A}$ running in time t, its advantage AdvPRF^{F} is at most ε.

Definition 2.5 (Joint PRF Security). *Let* $\mathsf{PRG} : \{0,1\}^{2\lambda} \times \{0,1\}^{\lambda} \to \{0,1\}^{\hat{\ell}}$ *and* $\mathsf{Com} : \{0,1\}^{*} \times \{0,1\}^{\lambda} \to \{0,1\}^{2\lambda}$ *be deterministic polynomial-time algorithms. Let* $\mathcal{A}$ *be a* q*-query,* N*-batch adversary in the experiment from Figure 2. Let*

$$\mathsf{AdvJPRF}^{\mathsf{PRG},\mathsf{Com}}[q,N,\lambda] := \left|\Pr\left[\mathsf{out}_{\mathcal{A}} = 1 \,:\, \mathsf{out}_{\mathcal{A}} \leftarrow \mathsf{Expt}^{\text{mt-jprf}}_{\mathsf{PRG},\mathsf{Com},\mathcal{A}}(q,N,\lambda)\right] - \frac{1}{2}\right|.$$

Then we call a pair of PRG *and* Com *as* $((q,N,\lambda),(t,\varepsilon))$*-secure if for every adversary* $\mathcal{A}$ *running in time* t*, its advantage* $\mathsf{AdvJPRF}^{\mathsf{PRG},\mathsf{Com}}[q,N,\lambda]$ *is at most* ε*. If* λ *is obvious in the context, we will drop* λ*.*

$\mathsf{Expt}^{\text{mt-jprf}}_{\mathsf{PRG},\mathsf{Com},\mathcal{A}}(q,N,\lambda)$

$b \xleftarrow{\$} \{0,1\}$
for $i \in [q]$: $\mathsf{salt}_i \xleftarrow{\$} \{0,1\}^{2\lambda}$
$\left(\left\{\mathsf{inp}_{i,j}\right\}_{(i,j)\in[q]\times[N]}, \mathsf{state}\right) \leftarrow \mathcal{A}\left(\left\{\mathsf{salt}_i\right\}_{i\in[q]}\right)$
for $i \in [q]$:
  for $j \in [N]$:
    if $b = 0$:
      $\mathsf{seed}_j \xleftarrow{\$} \{0,1\}^{\lambda}$
      $r_{i,j} := \mathsf{PRG}(\mathsf{salt}_i, \mathsf{seed}_j)$
      $\mathsf{com}_{i,j} := \mathsf{Com}(\mathsf{inp}_{i,j}, \mathsf{seed}_j)$
    else :
      $r_{i,j} \xleftarrow{\$} \{0,1\}^{\hat{\ell}}$
      $\mathsf{com}_{i,j} \xleftarrow{\$} \{0,1\}^{2\lambda}$
$b' \leftarrow \mathcal{A}\left(\mathsf{state}, \{(r_{i,j}, \mathsf{com}_{i,j}\}_{(i,j)\in[q]\times[N]}\right)$
if $b = b'$: **return** 1
else : **return** 0

Fig. 2: Multi-challenge joint PRF security of commitment and PRG

2.3 Digital signature schemes

Definition 2.6 (Signature Scheme). *A signature scheme consists of three* PPT *algorithms* $\mathsf{SIG} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *which work as follows:*

- $\mathsf{KeyGen}(1^{\lambda})$*: The* key generation *algorithm takes a security parameter as input and outputs a pair of keys* $(\mathsf{pk}, \mathsf{sk})$*. The key* sk *is the private (secret)* signing key *and* pk *is the* public key *used for verification.*

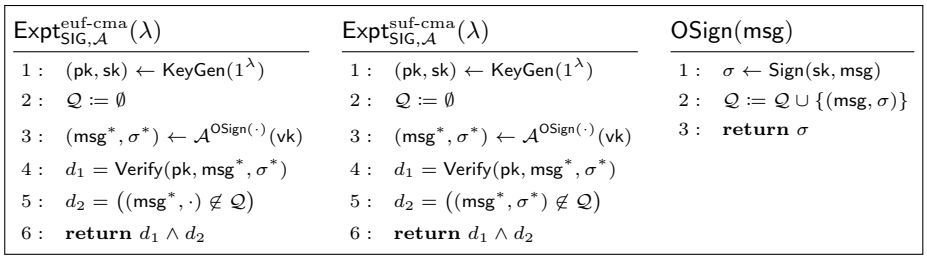

Fig. 3: EUF-CMA and SUF-CMA games.

- $\mathsf{Sign}(\mathsf{sk},\mathsf{msg})$*: The* signing *algorithm takes as input a secret signing key* sk *and a message* msg *from some message space (that may depend on* pk*). It outputs a signature* $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk},\mathsf{msg})$.
- $\mathsf{Verify}(\mathsf{pk},\mathsf{msg},\sigma)$*: The deterministic* verification *algorithm takes as input a public key* pk*, a message* msg*, and a signature* σ*. It outputs a bit* $b := \mathsf{Verify}(\mathsf{pk},\mathsf{msg},\sigma)$*, with* $b = 1$ *meaning the signature-message pair is valid and* $b = 0$ *meaning it is invalid.*

Definition 2.7 (EUF-CMA Security and EUF-NMA Security). *A signature scheme* $\mathsf{SIG} = (\mathsf{KeyGen},\mathsf{Sign},\mathsf{Verify})$ *is* existentially unforgeable under chosen-message attacks (EUF-CMA secure) *if, for all* PPT *adversaries* $\mathcal{A}$ *there is a negligible function* $\mathsf{negl}(\cdot)$ *such that,*

$$\Pr[\mathsf{Expt}^{euf\text{-}cma}_{\mathsf{SIG},\mathcal{A}}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{Expt}^{euf\text{-}cma}_{\mathsf{SIG},\mathcal{A}}(\lambda)$ *is the security game defined in Figure 3. In addition, if we consider the game where the signing oracle* OSign *is removed, then the signature scheme is said to be* existentially unforgeable under no-message attacks (EUF-NMA secure).

Definition 2.8 (SUF-CMA Security). *A signature scheme* $\mathsf{SIG} = (\mathsf{KeyGen},\mathsf{Sign},\mathsf{Verify})$ *is* strongly existentially unforgeable under chosen-message attacks (SUF-CMA secure) *if, for all* PPT *adversaries* $\mathcal{A}$ *there is a negligible function* $\mathsf{negl}(\cdot)$ *such that,*

$$\Pr[\mathsf{Expt}^{suf\text{-}cma}_{\mathsf{SIG},\mathcal{A}}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{Expt}^{suf\text{-}cma}_{\mathsf{SIG},\mathcal{A}}(\lambda)$ *is the security game defined in Figure 3*

2.4 Permuted Kernel Problem

PERK's security relies on the permuted kernel problem (PKP) introduced by Shamir in [Sha90].

Definition 2.9 (Permuted Kernel Problem (PKP)). *Let (q, m, n) be positive integers such that $m < n$, $\boldsymbol{H} \in \mathbb{F}_q^{m \times n}$, $\boldsymbol{x} \in \mathbb{F}_q^n$ and $\pi \in \mathcal{S}_n$ be a permutation such that $\boldsymbol{H}\big(\pi[\boldsymbol{x}]\big) = \boldsymbol{0}$. Given $(\boldsymbol{H}, \boldsymbol{x})$, the Permuted Kernel Problem $\mathsf{PKP}(q, m, n)$ asks to find $\tilde{\pi} \in \mathcal{S}_n$ such that $\boldsymbol{H}\big(\tilde{\pi}[\boldsymbol{x}]\big) = \boldsymbol{0}$.*

By convention, we call this problem the PKP problem. Hereafter, we interpret the PKP problem in matrix form namely the secret permutation π is seen as a permutation matrix $\boldsymbol{P} \in \mathbb{F}_q^{n \times n}$ such that $\boldsymbol{HPx} = 0$. In addition, our working field $\mathbb{F}_q$ is an extension field of $\mathbb{F}_2$.

The formal definition of the PKP assumption requires an instance distribution. To shorten the instance, we will employ ExpandMatrixM, a deterministic function $\{0,1\}^\lambda \to \mathbb{F}_q^{m \times (n-m)}$.

Definition 2.10 (Advantage against PKP). *Let (q, m, n) be positive integers such that $m < n$ and q is a power of a prime. For an adversary $\mathcal{A}$, we define its advantage $\mathsf{AdvOW}_{\mathcal{A}}$ against the PKP problem as follows:*

$$\mathsf{AdvOW}_{\mathcal{A}} \coloneqq \Pr\left[\boldsymbol{H}\big(\tilde{\pi}[\boldsymbol{x}]\big) = \boldsymbol{0} : \begin{array}{l} \boldsymbol{H}_{\mathsf{seed}} \xleftarrow{\$} \{0,1\}^\lambda; \boldsymbol{M} \coloneqq \mathsf{ExpandMatrixM}(\boldsymbol{H}_{\mathsf{seed}}); \\ \boldsymbol{H} \coloneqq [\boldsymbol{I}_m \, \boldsymbol{M}]; \boldsymbol{x}' \xleftarrow{\$} \ker(\boldsymbol{H}); \pi \xleftarrow{\$} \mathcal{S}_n; \\ \boldsymbol{x} \coloneqq \pi^{-1}[\boldsymbol{x}']; \tilde{\pi} \leftarrow \mathcal{A}(\boldsymbol{H}_{\mathsf{seed}}, \boldsymbol{x}) \end{array}\right].$$

We say that the PKP assumption holds if for any polynomial-time adversary $\mathcal{A}$, its advantage $\mathsf{AdvOW}_{\mathcal{A}}$ is negligible in λ.

3 Overview of PERK

3.1 VOLE-in-the-Head framework

VOLE correlations. A VOLE (Vector Oblivious Linear Evaluation) correlation of length $\hat{\ell}$ over $\mathbb{F}_{2^\rho}$ is defined by random values $(\boldsymbol{u}, \boldsymbol{v}) \in \mathbb{F}_2^{\hat{\ell}} \times \mathbb{F}_{2^\rho}^{\hat{\ell}}$, and $(\boldsymbol{q}, \Delta) \in \mathbb{F}_{2^\rho}^{\hat{\ell}} \times \mathbb{F}_{2^\rho}$, such that

$$q_i = u_i \Delta + v_i \qquad\qquad i \in [0, 1, \ldots, \hat{\ell} - 1].$$

The VOLE correlation serves as an information theoretically secure commitment to prover's random value $\boldsymbol{u}$. The mask $\boldsymbol{v}$ is unknown to the verifier, this provide the hiding property, while the prover needs to guess Δ in order to open the commitment to some $\boldsymbol{u}' \neq \boldsymbol{u}$, which provides the binding property. Moreover, owing to the linearity of VOLE correlations, these commitments are linearly homomorphic. Therefore, such VOLE correlations can be used to build efficient zero-knowledge proofs of knowledge, where the prover can commit to its secret witness with help of VOLE correlations and convince the verifier by computing some public function on the witness (and other public values) which can be verified using only the verifier's VOLE correlation inputs $(\boldsymbol{q}, \Delta)$.

VOLE-in-the-Head. Following the approach of [BBD+23b, BBD+23c], in order to achieve the public verifiability for our zero-knowledge proofs (and signatures) we use the *VOLE-in-the-Head* (VOLEitH) technique. In this approach the prover generates the values $\boldsymbol{u}, \boldsymbol{v}$ and commits to these values. The prover then computes the desired public relation with the help of committed VOLE correlation inputs $(\boldsymbol{u}, \boldsymbol{v})$. At this point the verifier can send Δ to the prover, and prover can send opening to the commitments to $(\boldsymbol{u}, \boldsymbol{v})$, from which the verifier can compute $\boldsymbol{q}$ without learning any extra information. Note that it is important that the prover learns the value of Δ (required to provide openings) only after it has committed to the VOLE correlation inputs, and to the computations of the zero-knowledge protocol (so it cannot change these after learning Δ), since after the prover knows Δ, the binding property of the linear homomorphic commitments does not hold.

In practice the VOLE correlation values are computed from uniform random strings of length $\hat{\ell}$. In order to create a single instance of VOLE correlation inputs $(\boldsymbol{u}, \boldsymbol{v}) \in \mathbb{F}_2^{\hat{\ell}} \times \mathbb{F}_{2^\rho}^{\hat{\ell}}$ the prover (signer) essentially needs to perform $O(2^\rho)$ additions and multiplications. Similarly, after receiving the opening the verifier also needs perform similar computation to acquire $\boldsymbol{q}$. The soundness error of the zero-knowledge proof of knowledge (relying on the binding property of linear homomorphic commitment based on VOLE correlations) is $2^{-\rho}$. Therefore, to achieve the desired security level we need to set $\rho \geq \lambda$, however this means that the prover and verifier will need to perform infeasible computations in order to even get started by creating the VOLE correlation inputs. This is mitigated by creating several parallel instances of VOLE correlations in a smaller field $\mathbb{F}_{2^\mu}$ and

concatenating them together to produce a single VOLE correlation instance in exponentially large field $\mathbb{F}_{2^\rho}$. This allows us to compute the VOLE correlations required to achieve desired security level efficiently. For each parameter set, we choose a repetition parameter, $\tau \in \mathbb{N}$ along with a VOLE field parameter, $\mu \in \mathbb{N}$ such that $\rho = \tau\mu$.

Committing to VOLE correlations. An important step in our signature scheme is to commit a vector of pseudo-random seeds, and be able to later open all-but-one of those seeds. Looking ahead these seeds will be used to generate the aforementionned VOLE correlations. The standard approach to build such an efficeint commitment scheme is to derive the seeds from a tree of length-doubling PRGs. Such a construction is called an *all-but-one vector commitment scheme*, relying on a GGM tree [GGM84], as suggested in [KKW18]. Suppose a party needs to generate N seeds and then to reveal only $N-1$ of those seeds (without knowing in advance which seed should not be revealed). The principle is to build a binary tree of depth $\lceil \log_2(N) \rceil$. The root of the tree is labeled with a master seed θ. The rest of the tree is labeled inductively by using a PRG of double extension on each parent node and splitting the output on the left and right children. To reveal all seeds except seed number i for $0 \leq i \leq N-1$, the principle is to reveal the labels on the siblings of the paths from the root of the tree to leave i. It allows to reconstruct all seeds but seed number i at the cost of communicating $\lfloor \log_2(N) \rfloor$ labels, which is more effective than communicating $N-1$ seeds. Instead of building one GGM-tree for each of the τ repetitions of the proof of knowledge, we adopted the approach of using a unified tree, as explained in [BBM+25], to save communication costs.

Computing VOLE correlations from seeds. The seeds obtained from the GGM tree can be used to generate VOLE correlations as follows: Let $\boldsymbol{r}_i = \mathsf{PRG}(\mathsf{seed}_i)$ be $\hat{\ell}$-bit pseudorandom strings for an integer $i \in [0, N-1]$ with $N \coloneqq 2^\kappa$ as the number of leaves in the GGM tree. The prover computes $\boldsymbol{u}, \boldsymbol{v}$ as

$$\boldsymbol{u} = \sum_{i=0}^{N-1} \boldsymbol{r}_i, \qquad \boldsymbol{v} = \sum_{i=0}^{N-1} i \cdot \boldsymbol{r}_i.$$

with i (and thus $\boldsymbol{v}$) as an element in $\mathbb{F}_{2^\mu}$. The verifier chooses $\Delta \in [0, N-1] \subseteq \mathbb{F}_{2^\mu}$ uniformly at random and receives all the seeds seed_i for $i \in [0, N-1] \setminus \Delta$ from the prover. The verifier can then compute

$$\boldsymbol{q} = \sum_{i=0}^{N-1} (\Delta - i) \cdot \boldsymbol{r}_i \in \mathbb{F}_{2^\mu}$$

We use the same approach as [BBD+23b] to compute the vole correlations from seeds which is detailed in Section 4.4 and algorithm 4.4. In order to achieve the desired security level, we need to repeat the above procedure τ times. However, this results in τ independent instances of VOLE correlations

$$\boldsymbol{q}_e = \boldsymbol{u}_e \Delta_e + \boldsymbol{v}_e \qquad e \in [0, 1, \ldots \tau - 1]$$

with $(\boldsymbol{u}_e, \boldsymbol{v}_e) \in \mathbb{F}_2^{\hat{\ell}} \times \mathbb{F}_{2^\mu}^{\hat{\ell}}$ and $(\boldsymbol{q}_e, \Delta_e) \in \mathbb{F}_{2^\mu}^{\hat{\ell}} \times \mathbb{F}_{2^\mu}$. Let $\boldsymbol{u}_e \in \mathbb{F}_2^{\hat{\ell}}$ be represented as a vector of length $\hat{\ell}$ over $\mathbb{F}_2$. Similarly elements in $\mathbb{F}_{2^\mu}$ such Δ_e can be represented by a vectors of length μ over $\mathbb{F}_2$. Also, $\boldsymbol{v}_e$ and $\boldsymbol{q}_e$ are vectors with elements in $\mathbb{F}_{2^\mu}$ and therefore can be represented by matrices of dimensions $\hat{\ell} \times \mu$ over $\mathbb{F}_2$. We can then write the VOLE correlation equation as

$$\boldsymbol{Q}_e = \left[\delta_{e,0}\boldsymbol{u}_e \; \delta_{e,1}\boldsymbol{u}_e \cdots \delta_{e,\mu-1}\boldsymbol{u}_e\right] + \boldsymbol{v}_e$$

where $(\delta_{e,0}, \delta_{e,1}, \ldots, \delta_{e,\mu-1})$ is the bit decomposition of $\Delta_e \in \mathbb{F}_{2^\mu}$. If the prover can somehow modify these correlations such that all τ instances use the same $\boldsymbol{u}$ value (say $\boldsymbol{u}_0$), then the prover combine (concatenate) $\boldsymbol{v}_e$ and $\boldsymbol{Q}_e$ matrices to build VOLE correlation values $\boldsymbol{v}$ and $\boldsymbol{q}$ in $\mathbb{F}_{2^\rho}^{\hat{\ell}}$. Similarly, prover computes $\Delta \in \mathbb{F}_{2^\rho}$ by concatenating all bits $\{\delta_{e,i}\}_{(e,i)\in[0,\ldots,\tau-1]\times[0,\ldots,\mu-1]}$ from individual Δ_e values. This gives us a desired VOLE correlation

$$\boldsymbol{Q} = \boldsymbol{u}_0\Delta + \boldsymbol{v}$$

with $(\boldsymbol{u}_0, \boldsymbol{v}) \in \mathbb{F}_2^{\hat{\ell}} \times \mathbb{F}_{2^\rho}^{\hat{\ell}}$ and $(\boldsymbol{Q}, \Delta) \in \mathbb{F}_{2^\rho}^{\hat{\ell}} \times \mathbb{F}_{2^\rho}$. The prover achieves this by sending the correction values $\boldsymbol{c}_e \coloneqq \boldsymbol{u}_0 - \boldsymbol{u}_e$ for $e \in [1, \ldots, \tau-1]$. These $\boldsymbol{c}_e$ values can be used by the verifier to adjust its correlation inputs such that all τ VOLE correlations in $\mathbb{F}_{2^\mu}$ hold with respect to $\boldsymbol{u}_0$.

Ensuring consistency of VOLE correlations. Note that if any of the correction values $\boldsymbol{c}_e$ is inconsistent (i.e. $\boldsymbol{c}_e \neq \boldsymbol{u}_0 - \boldsymbol{u}_e$) then the correctness of VOLE correlations does not hold and therefore the zero-knowledge proof built using such VOLE correlations cannot guarantee its correctness either. Therefore, the verifier must check that $\boldsymbol{c}_e$ values are consistent. The verifier can ensure this by asking the prover compute a random linear universal hash function of $\boldsymbol{u}_0$ and $\boldsymbol{v}$, and send the hash values (say $\tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}}$). The verifier can then compute the same function on $\boldsymbol{Q}$ and then check if the VOLE correlation $\tilde{\boldsymbol{Q}} = \tilde{\boldsymbol{u}}\Delta + \tilde{\boldsymbol{v}}$ holds true. This consistency check was used earlier in [Roy22, BBD+23b]. For more details kindly refer Section 4.4 and algorithm 4.8.

3.2 Proof of Knowledge for PKP

The zero-knowledge proof of knowledge underlying PERK is based on [BBGK24]. Recall that, the prover (signer) wants to prove the knowledge of a secret permutation matrix $\boldsymbol{P} \in \mathbb{F}_q^{n\times n}$ such that for some given public matrix $\boldsymbol{H} \in \mathbb{F}_q^{m\times n}$ (for $m < n$) and a public vector $\boldsymbol{x} \in \mathbb{F}_q^n$, the equation $\boldsymbol{HPx} = \boldsymbol{0}$ holds true. In PERK, we achieve this goal in two steps, first the prover aims to convince the verifier that it knows a permutation matrix. Then it shows that the equation $\boldsymbol{HPx} = \boldsymbol{0}$ holds true for this permutation matrix. We will therefore first focus on proving that the prover knows some permutation matrix.

Elementary vectors as building blocks. An elementary row vector $\boldsymbol{e}_i$ of length n for $0 \leq i \leq (n-1)$ is the $(n-1-i)^{\text{th}}$ row of an $n \times n$ identity matrix. [1] Note that, for any $n \times n$ permutation matrix its rows are also elementary vectors $\boldsymbol{e}_i$ (but in an arbitrary order). Therefore, as a first step towards proving knowledge of a specific secret permutation matrix, we can begin by trying to prove that we know a certain elementary vector. The key observation is that an elementary vector of length n, can be constructed by taking tensor product of elementary vectors of smaller size. In particular, PERK uses elementary vectors of lengths 4 and 2 to prove the knowledge of rows of secret permutation matrices of sizes $64, 92$, and 118. The prover proves the elementary structure of vector $\boldsymbol{e} \coloneqq [e_0, e_1, e_2, e_3] \in \mathbb{F}_2^4$ by showing that the product $e_0 \cdot e_1 = 0$ and $e_2 \cdot e_3 = 0$ simultaneously. In case of elementary vector $\boldsymbol{e} \coloneqq [e_0, e_1] \in \mathbb{F}_2^2$ this is achieved simply by showing $e_0 \cdot e_1 = 0$. Additionally, the prover should also prove that $e_0 \oplus e_1 \oplus e_2 \oplus e_3 = 1$ (resp. $e_0 \oplus e_1 = 1$). Proving these constraints simultaneously, allows the prover to demonstrate knowledge of elementary vectors, the prover achieves this by constructing equivalent low degree polynomials which have leading coefficient equal to 0 when the constraint is satisfied.

Knowledge of permutation and PKP solution. The prover computes the proof for each row of the secret permutation vector as a tensor product of d elementary vectors (for $d \in \{3, 4\}$) by constructing equivalent degree-d polynomials with leading coefficient equal to 0 if and only if the corresponding row is an elementary vector. At this stage, the prover is able to prove to the verifier the permutation structure of the matrix. The prover then shows that these degree-d polynomials when seen as entries of $n \times n$ matrix $\boldsymbol{P}$ satisfy the PKP equation $\boldsymbol{HPx} = \boldsymbol{0}$, where $(\boldsymbol{H}, \boldsymbol{x})$ corresponds to the public key of PKP. In order to prove that all these polynomials have leading coefficients equal to 0, the prover combines them all by taking a random linear combination of all the polynomials (where the coefficients of the random linear combination are provided by the verifier), adds secret masking polynomial of degree $d-1$ to the linear combination and sends it to the verifier. The verifier checks if the received polynomial has leading coefficient equal to 0 by evaluating it at a random point.

[1] We assume the indexes start from 0, that is the elementary vector $\boldsymbol{e}_0$ denotes the last that is $(n-1)^{\text{th}}$ row of the identity matrix.

4 Algorithmic Description

4.1 Object representation

Finite fields. Elements of $\mathbb{F}_q$ are stored in 16 bit unsigned integers. We also use finite field arithmetic over $\mathbb{F}_{2^{11}}$, $\mathbb{F}_{2^{64}}$, $\mathbb{F}_{2^{128}}$, $\mathbb{F}_{2^{132}}$, $\mathbb{F}_{2^{198}}$, $\mathbb{F}_{2^{264}}$. These fields are defined as polynomials over $\mathbb{F}_2$ modulo an irreducible polynomial F.

$$
\begin{aligned}
F_{11}(\gamma) &= 1 + \gamma^2 + \gamma^{11} \\
F_{64}(\gamma) &= 1 + \gamma^1 + \gamma^3 + \gamma^4 + \gamma^{64} \\
F_{128}(\gamma) &= 1 + \gamma^1 + \gamma^2 + \gamma^7 + \gamma^{128} \\
F_{132}(\gamma) &= 1 + \gamma^3 + \gamma^{12} \\
F_{198}(\gamma) &= 1 + \gamma^7 + \gamma^{18} \\
F_{264}(\gamma) &= 1 + \gamma^1 + \gamma^{11} + \gamma^{17} + \gamma^{24}
\end{aligned}
$$

We use following functions to convert bit-strings into field elements (or positive numbers) and vice versa:

- ToField converts a bit-string into a corresponding field element ;
- ToBits converts a field element into a corresponding bit-string ;
- BitDec converts a positive number into its binary decomposition ;
- NumRec takes a bit-string as the binary decomposition of a positive number and reconstructs the number.

Algorithm 4.1: ToField(bits, k)

Public information and inputs

Public information: Maps an input bitstring bits $\in \{0,1\}^{nk}$, for a positive integer $n \geq 1$ into a field element (or vector of n fields elements) $\boldsymbol{x} \in \mathbb{F}_{2^k}^n$.

---

let $\gamma_k \in \mathbb{F}_{2^k}$ // The γ element of $\mathbb{F}_{2^k}$.
if bits $\in \{0,1\}^k$:
  return $x := \sum_{i=0}^{k-1} \mathsf{bits}[i] \cdot \gamma_k^i$
elseif bits $\in \{0,1\}^{nk}$:
  for $i \in [n]$
    $\boldsymbol{x}[i] := \sum_{j=0}^{k-1} \mathsf{bits}[ni+j] \cdot \gamma_k^j$
  endfor
  return $\boldsymbol{x}$
else :
  return $\perp$
endif

Algorithm 4.2: ToBits($\boldsymbol{x}, k, n$)

Public information and inputs

Public information: Maps an input field element (or vector of n fields elements) $\boldsymbol{x} \in \mathbb{F}_{2^k}^n$, for a positive integer $n \geq 1$ into a bitstring bits $\in \{0,1\}^{nk}$.

1 : Initialize bits $\leftarrow \varepsilon$ // Empty string.
2 : Initialize bitslice $\leftarrow \varepsilon$ // Empty string.
3 : **for** $i \in [n]$
4 : Parse $\boldsymbol{x}[i]$ as $\boldsymbol{x}[i] = x_0 + x_1\gamma_k + \cdots + x_{k-1}\gamma_k^{k-1}$ with $x_0, x_1, \ldots, x_{k-1} \in \{0,1\}$
5 : bitslice $\coloneqq x_0||x_1||\cdots||x_{k-1}$ // bitslice $\in \{0,1\}^k$
6 : bits $\coloneqq$ bits||bitslice
7 : **endfor**
8 : **return** bits // bits $\in \{0,1\}^{nk}$.

Integer and bits conversions

Algorithm 4.3: BitDec(i, d)

Public information and inputs

Public information: Decomposes an integer i into bits.

1 : **for** $j \in [d]$:
2 : $b_j \coloneqq i \mod 2$
3 : $i \coloneqq (i - b_j)/2$
4 : **endfor**
5 : **return** $(b_0, b_1, \ldots, b_{d-1})$.

Algorithm 4.4: NumRec(d, bits)

Public information and inputs

Public information: Reconstructs an integer i from a bitstring.

1 : Parse bits as bits $\coloneqq b_0||\cdots||b_{d-1}$
2 : **return** $\sum_{j=0}^{d-1} b_j \cdot 2^j$

Vectors and Matrices. Vectors of $\mathbb{F}_q^n$ (respectively $\mathbb{F}_q^m$) are represented as arrays of length n (respectively of length m) of $\mathbb{F}_q$ elements. Matrices $\boldsymbol{H} \in \mathbb{F}_q^{m \times n}$ are represented as two dimensional arrays of $\mathbb{F}_q$ elements i.e. arrays of length m of arrays of length n.

Permutations and Witness. We use following auxiliary functions during signing process to encode positive numbers corresponding to the index of non-zero entries of the secret permutation matrix, and represent these secret indices as witness for the proof system.

- EncodeNum-64 encodes a number between 0 to 63 in base-4 notation. This function is used for NIST Level-I parameter set. Given an input pos $\in [64]$,

it outputs a unique tuple of three numbers $(i, j, k) \in [4]^3$ such that $\mathsf{pos} = 16k + 4j + i$;

- EncodeNum-128 encodes a number between 0 to 127 using hybrid base-4 and base-2 notation. This function is used for NIST Level-III and NIST Level-V parameter sets. On input $\mathsf{pos} \in [127]$, it outputs a unique tuple of four numbers $(i, j, k, b) \in [4]^3 \times \{0, 1\}$ such that $\mathsf{pos} = 64b + 16k + 4j + i$;
- EncodeNum is a wrapper function that internally calls either EncodeNum-64 or EncodeNum-128 depending on the security level ;
- LeftShift shifts the bits of an input bit-string to left by a specified amount given as input.

Algorithm 4.5: EncodeNum-64(pos)

Public information and inputs

Public information: Encodes input number $\mathsf{pos} \in [64]$ in base-4. order.

Output

Array encPosArray of length 3, encoding the input position pos.

1 : Initialize encPosArray $\leftarrow$ [**null**, **null**, **null**]
2 : $k \leftarrow \lfloor \frac{\mathsf{pos}}{16} \rfloor$
3 : $j \leftarrow \left\lfloor \frac{\mathsf{pos} - (k*16)}{4} \right\rfloor$
4 : $i \leftarrow \mathsf{pos} - (k * 16) - (j * 4)$
5 : encPosArray $\leftarrow [i, j, k]$
6 : **return** encPosArray

Algorithm 4.6: EncodeNum-128(pos)

Public information and inputs

Public information: Encodes input number $\mathsf{pos} \in [128]$ in hybrid base-4/base-2.

Output

Array encPosArray of length 4, encoding the input position pos.

1 : Initialize encPosArray $\leftarrow$ [**null**, **null**, **null**, **null**]
2 : $b \leftarrow \lfloor \frac{\mathsf{pos}}{64} \rfloor$ // $b \in \{0, 1\}$
3 : temp $\leftarrow$ EncodeNum-64 (pos $- (b * 64)$)
// Parse temp as temp $:= [i, j, k]$ where $i, j, k \in [0, 3]$
4 : encPosArray $\leftarrow [i, j, k, b]$
5 : **return** encPosArray

Algorithm 4.7: EncodeNum(pos)

Public information and inputs

Public information: Wrapper function for selecting encoding function for input number pos based on desired security level.

Output

Array encPosArray encoding the input position pos.

Security Level 1

if $\lambda = 128$:
  return EncodeNum-64 (pos)

Security Levels 3 and 5

elseif $\lambda \in \{192, 256\}$:
  return EncodeNum-128 (pos)
else :
  return $\perp$
endif

Algorithm 4.8: LeftShift(bits, shift)

Public information and inputs

Public information: Shifts input bitstring bits $\in \{0,1\}^*$ to left by shift positions if shift is smaller than length of bits.

---

if shift $\in$ [len(bits)]:
  return bits << shift
else :
  return $\perp$
endif

4.2 Sampling functions

The `randombytes` function provided by the NIST is used to sample uniformly at random the salt and various seeds (e.g., $\boldsymbol{H}_{\mathsf{seed}}, \mathsf{ker}_{\mathsf{seed}}, \mathsf{perm}_{\mathsf{seed}}$). The PRG function is instantiated using `SHAKE-128` for $\lambda = 128$ and `SHAKE-256` otherwise, along with domain separators.

Random elements of $\mathbb{F}_q$ are obtained by sampling $\log_2(q) = 11$ random bits from the PRG. Random vectors in $\mathbb{F}_q^n$ (respectively matrices in $\mathbb{F}_q^{m \times n}$) are sampled uniformly by sampling in order n (respectively $m \times n$) elements in $\mathbb{F}_q$.

$\mathsf{ExpandMatrixM}(\mathsf{H}_{\mathsf{seed}}) \to \mathbb{F}_q^{m\times(n-m)}$: Samples a matrix $\boldsymbol{M} \in \mathbb{F}_q^{m\times(n-m)}$ uniformly at random using the PRG with seed $\boldsymbol{H}_{\mathsf{seed}}$ and domain separator $\mathtt{dom} = \mathsf{0x00}$.

$\mathsf{ExpandKernelVector}(\mathsf{ker}_{\mathsf{seed}}, \mathsf{H}) \to \ker(\mathsf{H})$: Samples a vector $\boldsymbol{x'} \in \mathbb{F}_q^n$ in $\ker(\boldsymbol{H})$ uniformly at random. Specifically, we derive a basis $\boldsymbol{k}_1, \ldots, \boldsymbol{k}_{n-m} \in \mathbb{F}_q^n$ for $\ker(\boldsymbol{H})$ by taking the rows rows of the matrix $(\boldsymbol{M}^\top | \boldsymbol{I}_{n-m})$. Then we sample $n-m$ random scalars $c_1, \ldots, c_m \in \mathbb{F}_q$ using the PRG with seed $\mathsf{ker}_{\mathsf{seed}}$ and domain separator $\mathtt{dom} = \mathsf{0x10}$, and compute the resulting vector in $\ker(\boldsymbol{H})$ as $\boldsymbol{x'} = \sum_{i=0}^{n-m} c_i \cdot \boldsymbol{k}_i$. For an adversary, its advantage in distinguishing the output of ExpandKernelVector from a random kernel vector is denoted by $\mathsf{AdvPR}^{\mathsf{ExpandKernelVector}}$.

$\mathsf{ExpandPermutation}(\mathsf{perm}_{\mathsf{seed}}) \to \mathcal{S}_{\mathsf{n}}$: Samples permutation π of length n uniformly at random. Specifically, we first construct a vector $\boldsymbol{v} = (v_0, \cdots, v_{n-1}) = (0, 1, \cdots, n-1)$. Then, we use the PRG with seed $\mathsf{perm}_{\mathsf{seed}}$ and domain separator $\mathtt{dom} = \mathsf{0x20}$ for sampling a random vector $\boldsymbol{e} = (e_0, \cdots, e_{n-1}) \in (\mathbb{F}_2^{16})^n$. We construct the vector $\boldsymbol{p} = (p_0, \cdots, p_{n-1})$, where the high-order and low-order bits of p_i corresponds e_i and v_i, respectively. Finally, we sort this integer sequence in constant time using `djbsort` [Ber19], and extract the permutation π from the lower-order bits $\boldsymbol{p}$. If there are any duplicate values in the vector $\boldsymbol{e}$, we discard it and restart the procedure. For an adversary, its advantage in distinguishing the output of ExpandPermutation from a random permutation is denoted by $\mathsf{AdvPR}^{\mathsf{ExpandPermutation}}$.

4.3 Hash functions and commitments

In the following, we instantiate functions from the `SHA3` family, choosing the output length according to the security parameter λ: for $\lambda = 128$ we use `SHA3-256`, for $\lambda = 192$ we use `SHA3-384`, and for $\lambda = 256$ we use `SHA3-512`. Throughout, we denote a binary string as $x \in \{0,1\}^*$.

Pseudorandom Generators. We make use of two distinct pseudorandom generators, denoted by PRG_1 and PRG_2. Specifically, PRG_1 is instantiated from either the `SHA3` family or `Rijndael`, while PRG_2 is instantiated from either `SHAKE` or `Rijndael`.

- $\mathsf{PRG}_1(\mathsf{salt} \parallel \mathsf{index} \parallel \mathsf{seed})$ expands a seed into a binary tree according to the GGM construction, producing outputs of length 2λ. The length of index depends on the instantiation: 2 bytes for `SHA3` and 4 bytes for `AES/Rijndael`. Let $\mathsf{salt} = (\mathsf{salt}_0 \parallel \mathsf{salt}_1)$.

$$\mathsf{PRG}_1(\mathsf{salt} \parallel \mathsf{index} \parallel \mathsf{seed}) := \mathtt{SHA3}\text{-}\lambda(\mathsf{salt} \parallel \mathsf{index} \parallel \mathsf{seed} \parallel \mathtt{dom})$$

$$\mathsf{PRG}_1(\mathsf{salt} \parallel \mathsf{index} \parallel \mathsf{seed}) := (\mathsf{high} \parallel \mathsf{low})$$

where,

$$\mathsf{high} = \begin{cases} \texttt{AES-128}(\mathsf{k} = \mathsf{seed}, \mathsf{msg} = \mathsf{salt}_0 \oplus (\mathsf{0x00} \parallel \mathsf{index} \parallel \texttt{dom})) & \text{if } \lambda = 128 \\ \texttt{Rijndael-256}(\mathsf{k} = \mathsf{seed}, \mathsf{msg} = \mathsf{salt}_0 \oplus (\mathsf{0x00} \parallel \mathsf{index} \parallel \texttt{dom})) & \text{otherwise.} \end{cases}$$

$$\mathsf{low} = \begin{cases} \texttt{AES-128}(\mathsf{k} = \mathsf{seed}, \mathsf{msg} = \mathsf{salt}_0 \oplus (\mathsf{0x01} \parallel \mathsf{index} \parallel \texttt{dom})) & \text{if } \lambda = 128 \\ \texttt{Rijndael-256}(\mathsf{k} = \mathsf{seed}, \mathsf{msg} = \mathsf{salt}_0 \oplus (\mathsf{0x01} \parallel \mathsf{index} \parallel \texttt{dom})) & \text{otherwise.} \end{cases}$$

- $\mathsf{PRG}_2(\mathsf{salt} \parallel \mathsf{seed})$ converts seeds into instances for the Vole protocol, with output length $\hat{\ell}$.

  $$\mathsf{PRG}_2(\mathsf{salt} \parallel \mathsf{seed}) := \begin{cases} \texttt{SHAKE-128}(\mathsf{salt} \parallel \mathsf{seed} \parallel \texttt{dom}) & \text{if } \lambda = 128, \\ \texttt{SHAKE-256}(\mathsf{salt} \parallel \mathsf{seed} \parallel \texttt{dom}) & \text{if } \lambda \in \{192, 256\}. \end{cases}$$

  $$\mathsf{PRG}_2(\mathsf{salt} \parallel \mathsf{seed}) := \begin{cases} \texttt{AES-128}(\mathsf{k} = \mathsf{seed},\ \mathsf{msg} = \mathsf{salt}_0 \oplus (\mathsf{ctr} \parallel \mathsf{z} \parallel \texttt{dom}) & \text{if } \lambda = 128, \\ \texttt{Rijndael-256}(\mathsf{k} = \mathsf{seed},\ \mathsf{msg} = \mathsf{salt}_0 \oplus (\mathsf{ctr} \parallel \mathsf{z} \parallel \texttt{dom}) & \text{otherwise,} \end{cases}$$

  where ctr is a counter byte that starts from $\mathsf{0x00}$ and gets updated for every block produced, and $\mathsf{z} = (\mathsf{0x00} \parallel \mathsf{0x00} \parallel \mathsf{0x00} \parallel \mathsf{0x00})$.

In both cases, domain separation is enforced via an explicit tag $\texttt{dom} \in \mathsf{0x05}, \mathsf{0x06}$, ensuring independence between PRG_1 and PRG_2. For the security level parameter $\lambda = 192$, we extend salt_0 to 256 bits by appending $\mathsf{0x00}$s, and we truncate the output of $\texttt{Rijndael-256}$ to 192 bits.

Hash functions.

- $\mathsf{H}_1 : \{0,1\}^* \to \{0,1\}^{2\lambda}$ defined as

  $$\mathsf{H}_1(x) := \texttt{SHA3-}\lambda(x \parallel \texttt{dom}), \text{ where } \texttt{dom} = \mathsf{0x01}.$$

- $\mathsf{H}_2^1 : \{0,1\}^* \to \{0,1\}^{5\lambda+8}$ defined as

  $$\mathsf{H}_2^1(x) := \begin{cases} \texttt{SHAKE-128}(x \parallel \texttt{dom}) & \text{if } \lambda = 128,\ \texttt{dom} = \mathsf{0x21}, \\ \texttt{SHAKE-256}(x \parallel \texttt{dom}) & \text{if } \lambda \in \{192, 256\},\ \texttt{dom} = \mathsf{0x21}. \end{cases}$$

- $\mathsf{H}_2^2 : \{0,1\}^* \to \{0,1\}^{2\lambda}$ defined as

  $$\mathsf{H}_2^2(x) := \texttt{SHA3-}\lambda(x \parallel \texttt{dom}), \text{ where } \texttt{dom} = \mathsf{0x22}.$$

- $\mathsf{H}_2^3 : \{0,1\}^* \to \{0,1\}^{\tau_0\kappa_0+\tau_1\kappa_1+w}$ defined as

  $$\mathsf{H}_2^3(x) := \begin{cases} \texttt{SHAKE-128}(x \parallel \texttt{dom}) & \text{if } \lambda = 128,\ \texttt{dom} = \mathsf{0x23}, \\ \texttt{SHAKE-256}(x \parallel \texttt{dom}) & \text{if } \lambda \in \{192, 256\},\ \texttt{dom} = \mathsf{0x23}. \end{cases}$$

- $\mathsf{H}_3 : \{0,1\}^* \to \{0,1\}^{3\lambda}$ defined as

$$\mathsf{H}_3(x) := \begin{cases} \texttt{SHAKE-128}(x \parallel \texttt{dom}) & \text{if } \lambda = 128,\ \texttt{dom} = \texttt{0x03}, \\ \texttt{SHAKE-256}(x \parallel \texttt{dom}) & \text{if } \lambda \in \{192, 256\},\ \texttt{dom} = \texttt{0x03}. \end{cases}$$

- $\mathsf{H}_4 : \{0,1\}^* \to \{0,1\}^{\rho(cn+n+m)}$ defined as

$$\mathsf{H}_4(x) := \begin{cases} \texttt{SHAKE-128}(x \parallel \texttt{dom}) & \text{if } \lambda = 128,\ \texttt{dom} = \texttt{0x04}, \\ \texttt{SHAKE-256}(x \parallel \texttt{dom}) & \text{if } \lambda \in \{192, 256\},\ \texttt{dom} = \texttt{0x04}. \end{cases}$$

Commitments. For the commitments, we consider the two following approaches.

Commitment Com_1. This scheme is instantiated from either the `SHA3` family or `Rijndael`. Let τ denote the subtree and n the index of the leaf in the GGM tree array.

- `SHA3` instantiation: we absorb τ as a single byte and n as two bytes.
- `Rijndael` instantiation: we concatenate τ and n into a 32-bit string (4 bytes).

We denote this combined value as index. Let $\mathsf{salt} = (\mathsf{salt}_0 \parallel \mathsf{salt}_1)$.

Option 1: `SHA3` based commitment.

$\mathsf{Com}_1(\mathsf{salt} \parallel \mathsf{index} \parallel \mathsf{seed}) := \texttt{SHA3-}\lambda(\mathsf{salt} \parallel \mathsf{index} \parallel \mathsf{seed} \parallel \texttt{dom})$, where $\texttt{dom} = \texttt{0x07}$.

Option 2: `Rijndael`-based commitment.

$$\mathsf{Com}_1(\mathsf{salt} \parallel \mathsf{index} \parallel \mathsf{seed}) := (\mathsf{high} \parallel \mathsf{low}), \text{ where}$$

$$\mathsf{high} = \begin{cases} \texttt{AES-128}(\mathsf{k} = \mathsf{seed}, \mathsf{msg} = \mathsf{salt}_0 \oplus (\texttt{0x00} \parallel \mathsf{index} \parallel \texttt{dom})) & \text{if } \lambda = 128 \\ \texttt{Rijndael-256}(\mathsf{k} = \mathsf{seed}, \mathsf{msg} = \mathsf{salt}_0 \oplus (\texttt{0x00} \parallel \mathsf{index} \parallel \texttt{dom})) & \text{otherwise.} \end{cases}$$

$$\mathsf{low} = \begin{cases} \texttt{AES-128}(\mathsf{k} = \mathsf{seed}, \mathsf{msg} = \mathsf{salt}_0 \oplus (\texttt{0x01} \parallel \mathsf{index} \parallel \texttt{dom})) & \text{if } \lambda = 128 \\ \texttt{Rijndael-256}(\mathsf{k} = \mathsf{seed}, \mathsf{msg} = \mathsf{salt}_0 \oplus (\texttt{0x01} \parallel \mathsf{index} \parallel \texttt{dom})) & \text{otherwise.} \end{cases}$$

For the security level parameter $\lambda = 192$, we extend salt_0 to 256 bits by appending 0x00s, and we truncate the output of `Rijndael-256` to 192 bits. In all three cases, we take $\texttt{dom} = \texttt{0x07}$.

Commitment Com_2. This scheme is derived from the `SHA3` family

$$\mathsf{Com}_2(x) := \texttt{SHA3-}\lambda(x \parallel \texttt{dom}), \text{ where } \texttt{dom} = \texttt{0x08}.$$

4.4 VOLE-in-the-Head functions

The VOLE correlations form one of the foundational building block of our scheme. In this section, we describe how to perform basic operations on VOLE correlations and how to construct them from the batch all-but-one vector commitments construction. We also explain how, within the scheme, the prover commits to these VOLE correlations and checks their consistency.

VOLE correlations. Let $\boldsymbol{q} = u\Delta \oplus \boldsymbol{v}$ be a VOLE correlation with $(u, \boldsymbol{v}) \in \mathbb{F}_2 \times \mathbb{F}_{2^\mu}$, and $(\boldsymbol{q}, \Delta) \in \mathbb{F}_{2^\mu} \times \mathbb{F}_{2^\mu}$.[2] Such a generic VOLE correlation corresponds to a linear (degree-1) commitment to u denoted as $f_u(X) = uX + v$, with the evaluation $q = f_u(\Delta)$ given to the verifier. From here onward in this document, we denote degree-1 commitment to a bit u (or a bit-string $\boldsymbol{u} \in \mathbb{F}_2^{\hat{\ell}}$) by $[\![u]\!]$ (or $[\![\boldsymbol{u}]\!]$ respectively). This notion can be extended to polynomial-based commitments with higher degree polynomials. We write $[\![s]\!]^{(d)}$ to denote a degree-d commitment to a secret value $s \in \mathbb{F}_2$ where the prover holds $f_s(X) = \sum_{i=0}^{d} a_i X^i$ with coefficients $a_i \in \mathbb{F}_{2^\mu}$ and a_d equal to s lifted to $\mathbb{F}_{2^\mu}$ while the verifier holds $q_s = f_s(\Delta) \in \mathbb{F}_{2^\mu}$.

It is possible to compute arbitrary linear combinations of a given set of input VOLE correlations, due to their linear homomorphic property. Algorithm 4.9 LinearCombination from [BBD+23b] given below shows how VOLE correlations for linear functions of secret values $u_1, \ldots, u_n \in \mathbb{F}_2$ can be computed. In fact, it is also possible to combine k VOLE correlations $\boldsymbol{q}_i = u_i\Delta \oplus \boldsymbol{v}_i, i \in [k]$ with $(u_i, \boldsymbol{v}_i) \in \mathbb{F}_2 \times \mathbb{F}_{2^\mu}$ and $(\boldsymbol{q}_i, \Delta) \in \mathbb{F}_{2^\mu} \times \mathbb{F}_{2^\mu}$ for $i \in [k]$, to obtain a single VOLE correlation $\boldsymbol{q} = \boldsymbol{u}\Delta \oplus \boldsymbol{v}$ with $(\boldsymbol{u}, \boldsymbol{v}) \in \mathbb{F}_{2^k} \times \mathbb{F}_{2^\mu}$ and $(\boldsymbol{q}_i, \Delta) \in \mathbb{F}_{2^\mu} \times \mathbb{F}_{2^\mu}$ for an arbitrary $\mathbb{F}_{2^k} \subseteq \mathbb{F}_{2^\mu}$. To achieve this, the prover computes $\boldsymbol{u} \coloneqq \sum_{i=0}^{k-1} u_i X^i$ where $\{1, X, \ldots, X^{k-1}\}$ is the power-basis of $\mathbb{F}_{2^k}$ over $\mathbb{F}_2$. Prover also computes $\boldsymbol{v}$ as $\boldsymbol{v} \coloneqq \sum_{i=0}^{k-1} \boldsymbol{v}_i X^i$. While the verifier can compute $\boldsymbol{q}$ as $\boldsymbol{q} \coloneqq \sum_{i=0}^{k-1} \boldsymbol{q}_i \Delta^i$ (and Δ remains unchanged). We can perform homomorphic operations on the degree-d-commitments locally by the prover and the verifier with the help of the Algorithm 4.10 Add and Algorithm 4.11 Multiply given below. In the following, let $d_1 \geq d_2$ without loss of generality, also let $d = d_1 + d_2$.

Given a VOLE correlation $q = u\Delta + v$ for a random $u \in \mathbb{F}_2$, it is possible to embed arbitrary value $w \in \mathbb{F}_2$. To do so, the prover computes $t = w \oplus u$ and sends t to the verifier. Since u is uniform random and unknown to the verifier, t does not leak any information about w. The verifier then computes $q_w = q + t\Delta$. Note that the prover and verifier now possess their respective parts for the VOLE correlation $q_w = w\Delta + v$. After embedding the witness w, one can use the VOLE correlations to establish PoK of the witness for relations which can be modeled as polynomial functions of the commitment to the witness.

[2] Here we consider $\hat{\ell} = 1$ for readability and easy to read notation. Therefore, u and Δ can be seen as scalars represented by 1 and μ-bits respectively. Whereas, $\boldsymbol{v}, \boldsymbol{q}$ are vectors of length μ over $\mathbb{F}_2$. All of the discussion in Section 4.4 naturally extends to VOLE correlations with $\hat{\ell} > 1$, where we view Δ as a scalar represented by μ-bits, $\boldsymbol{u}$ is a vector of length $\hat{\ell}$ over $\mathbb{F}_2$, and $\boldsymbol{V}, \boldsymbol{Q}$ are seen as $\hat{\ell} \times \mu$ matrices over $\mathbb{F}_2$.

Algorithm 4.9: LinearCombination$\left(c_0, c_1, \ldots, c_n, (\llbracket u_i \rrbracket)_{i\in[1,n]}\right)$

Prover's computation: P.LinearCombination$\left(c_0, c_1, \ldots, c_n, (\llbracket u_i \rrbracket)_{i\in[1,n]}\right)$

Prover's input: Coefficients of linear combination $c_0, c_1, \ldots, c_n \in \mathbb{F}_2$, VOLE correlation inputs $(u_1, v_1), \ldots, (u_n, v_n) \in (\mathbb{F}_2 \times \mathbb{F}_{2^\rho})^n$.

Prover's output: VOLE correlations (u, v) for the linear combination of secret inputs.

Computes $u \coloneqq c_0 + \sum_{i=1}^{n} c_i u_i$ and $v \coloneqq \sum_{i=1}^{n} c_i v_i$.

Verifier's computation: V.LinearCombination$(c_0, c_1, \ldots, c_n, \Delta, q_1, \ldots, q_n)$

Verifier's input: Coefficients of linear combination $c_0, c_1, \ldots, c_n \in \mathbb{F}_2$, VOLE correlation inputs $\Delta, q_1, \ldots q_n \in \mathbb{F}_{2^\rho}^{n+1}$

Verifier's output: VOLE correlation q for linear combination of secret inputs.

Computes $q \coloneqq c_0\Delta + \sum_{i=1}^{n} c_i q_i$.

Algorithm 4.10: Add$\left(\llbracket s_1 \rrbracket^{(d_1)}, \llbracket s_2 \rrbracket^{(d_2)}\right)$

Public information:

Degrees of input VOLE correlations d_1, d_2.

Prover's computation: P.Add$\left(\llbracket s_1 \rrbracket^{(d_1)}, \llbracket s_2 \rrbracket^{(d_2)}\right)$

Prover's input: VOLE correlations represented as polynomials $f_{s_1}(X)$ and $f_{s_2}(X)$.

Prover's output: VOLE correlation $\llbracket s \rrbracket^{(d_1)}$ for addition of secret inputs.

Computes $\llbracket s \rrbracket^{(d_1)} = f_s(X) \coloneqq f_{s_1}(X) + f_{s_2}(X)X^{d_1-d_2}$ where $s = s_1 + s_2$.

Verifier's computation: V.Add$\left(\Delta, q_{s_1}, q_{s_2}\right)$

Verifier's input: Δ, $q_{s_1} = f_{s_1}(\Delta)$, and $q_{s_2} = f_{s_2}(\Delta)$

Verifier's output: VOLE correlation q_s for addition of secret inputs.

Computes $q_s \coloneqq q_{s_1} + q_{s_2}\Delta^{d_1-d_2}$.

Algorithm 4.11: Multiply$\left(\llbracket s_1 \rrbracket^{(d_1)}, \llbracket s_2 \rrbracket^{(d_2)}\right)$

Public information:

Degrees of input VOLE correlations d_1, d_2.

Prover's computation: P.Multiply$\left(\llbracket s_1 \rrbracket^{(d_1)}, \llbracket s_2 \rrbracket^{(d_2)}\right)$

Prover's input: VOLE correlations represented as polynomials $f_{s_1}(X)$ and $f_{s_2}(X)$.

Prover's output: VOLE correlation $\llbracket s \rrbracket^{(d_1)}$ for multiplication of secret inputs.

Computes $\llbracket s \rrbracket^{(d)} = f_s(X) \coloneqq f_{s_1}(X) f_{s_2}(X)$ where $s = s_1 s_2$.

Verifier's computation: V.Multiply(q_{s_1}, q_{s_2})

Verifier's input: $q_{s_1} = f_{s_1}(\Delta)$ and $q_{s_2} = f_{s_2}(\Delta)$

Verifier's output: VOLE correlation q_s for multiplication of secret inputs.

Computes $q_s \coloneqq q_{s_1} q_{s_2}$.

Committing to VOLE correlations. Recall that in order to achieve the desired security level (soundness), the atomic zero-knowledge protocol based on VOLE correlations should be repeated τ times. This means that the prover needs to generate τ GGM trees instances (or a forest of τ GGM trees). Let N_e be number of leaves (or seeds to be committed to) in each of such τ trees for $e \in [\tau]$ and let $N \coloneqq \sum_{e=0}^{\tau-1} N_e$. Recently authors of [BBM+25] showed how the communication cost of such commitments can be further reduced by using *batch all-but-one vector commitment (BAVC) schemes*. The idea is to generate a single big GGM tree with N leaves instead of τ individual trees per instance with N_e leaves each. Note that in both cases (either with a forest of τ smaller trees or with single unified tree) the prover needs to hide a total of τ leaves. Intuitively, opening all-but-τ leaves of the unified tree is more efficient than opening all-but-one leaves of τ smaller trees, if the leaves to be opened in the big tree are relatively close to each other (or share some ancestor node in the tree). To achieve this, following the authors of [BBM+25] we interleave the leaves of the τ instances. That is, the first τ leaves of the big tree correspond to the first entry of the individual τ vector commitments, the next τ leaves correspond to the second entries, and so on. We also use a fixed threshold value T_{open} to ensure that the revealed path is not too long (thus avoiding long signature sizes). The opening algorithm aborts if the number of nodes exceeds T_{open}. This results in rejection sampling during the opening, which reduces the entropy of the challenge space. Fortunately, in [BBM+25] the authors showed that security is actually unaffected: since each rejection sampling step results in the prover computing a hash function, which can be considered as a proof of work done during each signing operation. We refer the interested readers to [BBM+25] for further details.

The batch BAVC consists of the following algorithms.

- Algorithm 4.1 called as $\mathsf{VC.Commit}$ generates a vector commitment using a master seed mseed and a salt value salt as inputs with the help of length-doubling salt based PRG PRG_1. It outputs the leaves of the tree as seeds $\{\mathsf{seed}_{e,i}\}_{\tau,N}$, a commitment h_{com} to all the seeds and decommitment information decom which consists of commitments $\{\mathsf{com}_{e,i}\}_{\tau,N}$ to individual seeds and all intermediate nodes required to construct the unified GGM tree.

- Algorithm 4.2 $\mathsf{VC.Open}$ takes the decommitment information decom and the index set $\boldsymbol{i}^*$ of size τ corresponding to the indexes of the hidden leaves. It outputs a partial decommitment pdecom which can be used to recompute all-but-τ seeds in the GGM tree, along with τ commitments $\{\mathsf{com}_{e,\boldsymbol{i}^*}\}_{\tau,\tau}$ to the hidden seeds.

- Algorithm 4.3 $\mathsf{VC.Reconstruct}$ on inputs the index set $\boldsymbol{i}^*$ of size τ (same as in $\mathsf{VC.Open}$), pdecom, and salt value salt outputs all-but-τ seeds (hidden by the indexes in $\boldsymbol{i}^*$) by reconstructing the GGM tree, and also outputs the commitment h_{com} to all the seeds.

In the following, we assume that $N_0 = \ldots = N_{\tau_0 - 1} \geq N_{\tau_0} = \ldots = N_{\tau-1}$ for some τ_0 and define N as $N := \sum_{e=0}^{\tau-1} N_e$. We define ψ as

$$\psi(e,i) = \begin{cases} i \cdot \tau + e & \text{if } i < N_{\tau_0} \\ N_{\tau_0} \cdot \tau + (i - N_{\tau_0}) \cdot \tau_0 + e & \text{otherwise} \end{cases} \tag{1}$$

We also use $\{\mathsf{seed}_{e,i}\}_{\tau,N} := \left(\mathsf{seed}_{0,0}, \mathsf{seed}_{0,1}, \ldots, \mathsf{seed}_{(\tau-1),(N_{\tau-1}-1)}\right)$ to denote an ordered tuple of N seeds where $e \in [\tau]$ and $i \in [N_e]$. Similarly, we use $\{\mathsf{com}_{e,i}\}_{\tau,N} := \left(\mathsf{com}_{0,0}, \mathsf{com}_{0,1}, \ldots, \mathsf{com}_{(\tau-1),(N_{\tau-1}-1)}\right)$ to denote an ordered tuple of N commitments where $e \in [\tau]$ and $i \in [N_e]$. We also use $\{\mathsf{com}_{e,\boldsymbol{i}^*}\}_{\tau,\tau} := \left(\mathsf{com}_{0,\boldsymbol{i}^*[0]}, \mathsf{com}_{1,\boldsymbol{i}^*[1]}, \ldots, \mathsf{com}_{(\tau-1),\boldsymbol{i}^*[\tau-1]}\right)$ to denote an ordered tuple of τ commitments where $e \in [\tau]$, and $\boldsymbol{i}^* \in [N_0] \times [N_1] \times \cdots \times [N_{\tau-1}]$ is an ordered list of τ indexes corresponding to hidden leaves.

Algorithm 4.1: VC.Commit(mseed, salt)

Public information and inputs

Public information: A number of iterations τ, a number of parties $N = \sum_{e=0}^{\tau-1} N_e$.
Prover's input: A master seed $\mathsf{mseed} \in \{0,1\}^{\lambda}$ and a $\mathsf{salt} \in \{0,1\}^{2\lambda}$

Output

N seeds $\{\mathsf{seed}_{e,i}\}_{\tau,N} \in \left(\{0,1\}^{\lambda}\right)^{N}$, a commitment $h_{\mathsf{com}} \in \{0,1\}^{2\lambda}$ and a decommitment auxilary variable decom.

---

1 : $\mathsf{nodes}[0] \coloneqq \mathsf{mseed}$
2 : **for** $i \in [N-1]$:
3 : $\quad(\mathsf{nodes}[2i+1], \mathsf{nodes}[2i+2]) \coloneqq \mathsf{PRG}_1\,(\mathsf{salt}, \mathsf{nodes}[i] :: 2\lambda)$
4 : **endfor**
5 : **for** $e \in [\tau]$:
6 : $\quad$**for** $i \in [N_e]$:
7 : $\quad\quad\mathsf{seed}_{e,i} \coloneqq \mathsf{nodes}[N-1+\psi(e,i)]$
8 : $\quad\quad\mathsf{com}_{e,i} \coloneqq \mathsf{Com}_1\,(\mathsf{salt}, \mathsf{seed}_{e,i} :: 2\lambda)$
9 : $\quad$**endfor**
10 : **endfor**
11 : $h_{\mathsf{com}} \coloneqq \mathsf{Com}_2\left(\mathsf{salt}, (\mathsf{cmt}_{0,0}, \ldots, \mathsf{cmt}_{(\tau-1),(N_{\tau-1}-1)}) :: 2\lambda\right)$
12 : **return** $\left(\{\mathsf{seed}_{e,i}\}_{\tau,N}, h_{\mathsf{com}}, \mathsf{decom} \coloneqq (\mathsf{nodes}, \{\mathsf{com}_{e,i}\}_{\tau,N})\right)$

Algorithm 4.2: VC.Open(decom, $\boldsymbol{i}^*$)

Public information and inputs

Public information: A number of iterations τ, a number of parties $N = \sum_{e=0}^{\tau-1} N_e$, a rejection parameter T_{open}

Prover's input: A decommitment auxilary variable $\mathsf{decom} := (\mathsf{nodes}, \{\mathsf{com}_{e,i}\}_{\tau,N})$ and $\boldsymbol{i}^* \in [N_0] \times [N_1] \times \cdots \times [N_{\tau-1}]$ is an ordered list of τ indexes corresponding to hidden leaves.

Output

A sibling path pdecom and unopened commitments $\{\mathsf{com}_{e,\boldsymbol{i}^*}\}_{\tau,\tau}$

// Selecting hidden leaves from interleaved tree

hidden $:= \{N - 1 + \psi(e, i^*[e]) : e \in [\tau]\}$

revealed $:= \{N-1, \ldots, 2N-2\} \backslash$ hidden

for i from $N-2$ downto 0 :

  // Adding parent node to revealed if both children nodes are in revealed.

  if $(2i+1) \in$ revealed and $(2i+2) \in$ revealed :

    revealed $:=$ (revealed$\backslash\{2i+1, 2i+2\}) \cup \{i\}$

endfor

if len(revealed) $> T_{\text{open}}$:

  return $\perp$

pdecom $:= \emptyset$

for $i \in [2N-1]$:

  if $i \in$ revealed :

    pdecom $:=$ (pdecom $\|$ nodes$[i]$)

endfor

return (pdecom, $\{\mathsf{com}_{e,\boldsymbol{i}^*}\}_{\tau,\tau}$)

Algorithm 4.3: VC.Reconstruct$\left(\boldsymbol{i}^*, \mathsf{pdecom}, \{\mathsf{com}_{e,\boldsymbol{i}^*}\}_{\tau,\tau}, \mathsf{salt}\right)$

Public information and inputs

Public information: A number of iterations τ, a number of parties $N = \sum_{e=0}^{\tau-1} N_e$.
Verifier's input: An ordered list $\boldsymbol{i}^* \in [N_0] \times [N_1] \times \cdots \times [N_{\tau-1}]$ of τ indexes corresponding to hidden leaves, a sibling path pdecom and unopened commitments $\{\mathsf{com}_{e,\boldsymbol{i}^*}\}_{\tau,\tau}$

Output

A commitment $h_{\mathsf{com}} \in \{0,1\}^{2\lambda}$, and $\{\mathsf{seed}_{e,i}\}_{\tau,N} \coloneqq \left(\mathsf{seed}_{0,0}, \mathsf{seed}_{0,1}, \ldots, \mathsf{seed}_{(\tau-1),(N-1)}\right)$ to denote an ordered tuple of N seeds where $e \in [\tau]$ and $i \in [N_e]$. Note that the τ seeds corresponding to the hidden leaves denoted by an ordered tuple $\{\mathsf{seed}_{e,\boldsymbol{i}^*}\}_{\tau,\tau} \coloneqq \left(\mathsf{seed}_{0,\boldsymbol{i}^*[0]}, \mathsf{seed}_{1,\boldsymbol{i}^*[1]}, \ldots, \mathsf{seed}_{(\tau-1),\boldsymbol{i}^*[\tau-1]}\right)$ are set to $\perp$.

// Selecting hidden leaves from interleaved tree
hidden $\coloneqq \{N-1+\psi(e,\boldsymbol{i}^*[e]) : e \in [\tau]\}$
revealed $\coloneqq \{N-1,\ldots,2N-2\} \backslash$ hidden
for i from $N-2$ downto 0 :
  // Adding parent node to revealed if both children nodes are in revealed.
  if $(2i+1) \in$ revealed and $(2i+2) \in$ revealed :
    revealed $\coloneqq$ (revealed$\backslash\{2i+1, 2i+2\}) \cup \{i\}$
endfor
// Check if pdecom is well formed by checking that number of nodes
// in revealed matches with pdecom and is $\leq T_{\mathsf{open}}$.
if len(revealed) $\neq$ len(pdecom) **or** len(revealed) $> T_{\mathsf{open}}$:
  return $\perp$
endif
nodes$[0], \ldots,$ nodes$[2N-2] \coloneqq \emptyset, \ldots, \emptyset$
for $i \in [N-1]$:
  if $i \in$ revealed :
    (nodes$[i]$, pdecom) $\coloneqq$ pdecom
  if nodes$[i] \neq \emptyset$:
    nodes$[2i+1]$, nodes$[2i+2] \coloneqq \mathsf{PRG}_1$ (salt, nodes$[i] :: 2\lambda$)
endfor
for $e \in [\tau]$:
  for $i \in [N_e]$:
    if $i \neq \boldsymbol{i}^*[e]$:
      $\mathsf{seed}_{e,i} \coloneqq$ nodes$[N-1+\psi(e,i)]$
      $\mathsf{com}_{e,i} \coloneqq \mathsf{Com}_1$ (salt, $\mathsf{seed}_{e,i} :: 2\lambda$)
    else :
      $\mathsf{seed}_{e,i} \coloneqq \perp$
      $\mathsf{com}_{e,i} \coloneqq \mathsf{com}_{e,\boldsymbol{i}^*[e]}$
  endfor
endfor
$h_{\mathsf{com}} \coloneqq \mathsf{Com}_2\left(\mathsf{salt}, (\mathsf{cmt}_{0,0}, \ldots, \mathsf{cmt}_{(\tau-1),(N_{\tau-1}-1)}) :: 2\lambda\right)$
return $(h_{\mathsf{com}}, \{\mathsf{seed}_{e,i}\}_{\tau,N})$

Computing VOLE from seed. The N seeds committed via the vector commitments are then converted to VOLE correlations using the algorithms specified below. Note that we use the exact same algorithm Algorithm 4.4 ConvertToVOLE as [Roy22, BBD+23b] for this conversion which uses divide-and-conquer approach to compute the VOLE correlations iteratively. The only difference is sometimes we lift the elements from $\mathbb{F}_{2^\kappa}$ to $\mathbb{F}_{2^\mu}$ to ensure that the finite field arithmetic between objects is compatible.

The signing algorithm PERK.Sign uses the outputs of vector commitment and ConvertToVOLE algorithms to commit to τ instances of VOLE correlations. This is achieved by Algorithm 4.5 VOLECommit which creates τ instances of VOLE correlations by running ConvertToVOLE τ times. It then also computes the correction values $\boldsymbol{c}_e$ for $e \in [1, \ldots, \tau - 1]$ and outputs the VOLE signers correlation inputs $\boldsymbol{u}, \boldsymbol{V}$, correction vales $\boldsymbol{c}_e$ for $e \in [1, \ldots, \tau - 1]$, and commitment and decommitment information from vector commitment.

The challenge decoding algorithm Algorithm 4.6 ChallDec takes an input challenge string ch of length $\log(N)$ where N is the number of leaves in the GGM tree, and outputs an index set $\boldsymbol{i}^*$ indicating the indexes of the hidden leaves. $\boldsymbol{i}^*$ is computed by parsing τ chunks of input ch and converting them to an integer $\in [N_e]$ for $e \in [\tau]$. ChallDec is used by both signer to create opening information for the verifier and by the verifier to reconstruct the VOLE correlation inputs from the GGM tree.

Algorithm 4.7 VOLEReconstruct is used by the verifier to compute its VOLE correlation inputs $\boldsymbol{Q}$ and Δ. The values of Δ is computed from the indexes of $\boldsymbol{i}^*$ obtained by running the ChallDec procedure. And $\boldsymbol{Q}$ is computed by reconstructing the committed seeds from the GGM tree with the help of VC.Reconstruct.

Algorithm 4.4: $\mathsf{ConvertToVOLE}\left(N_{e^*}, (\mathsf{seed}_{e^*,i})_{i\in[N_{e^*}]}, \mathsf{salt}, \mu :: \hat{\ell}\right)$

Public information and inputs

Inputs: A number of parties $N_{e^*} = 2^\kappa$, a tuple of N_{e^*} seeds $\left(\mathsf{seed}_{e^*,i}\right)_{i\in[N_{e^*}]} \in (\{0,1\}^\lambda)^{N_{e^*}}$ for some fixed $e^* \in [\tau]$, a $\mathsf{salt} \in \{0,1\}^{2\lambda}$, and $\hat{\ell} \in \mathbb{N}$ denoting the number of VOLE correlations output by the function. Recall that $\hat{\ell} := \ell_{\mathsf{VOLEHashMask}} + \ell + \ell_{\mathsf{CZMask}}$.

Output

Outputs $\hat{\ell}$ VOLE correlations $(\boldsymbol{u}_k, \boldsymbol{v}_k) \in \mathbb{F}_2 \times \mathbb{F}_{2^\mu}$ for $k \in [\hat{\ell}]$, All these VOLE correlations can be seen as linear (degree-1) polynomials, $[\![u_k]\!] = f_{u_k}(X) := u_k X + v_k \in \mathbb{F}_{2^\mu}[X]$.

$\kappa := \log_2(N_{e^*})$
if $\mathsf{seed}_0 = \perp$:
  $\boldsymbol{r}_{0,0} := 0^{\hat{\ell}}$
else :
  $\boldsymbol{r}_{0,0} := \mathsf{PRG}_2\left(\mathsf{salt}, \mathsf{seed}_{e^*,0} :: \hat{\ell}\right)$
for $i \in [1, N-1]$:
  $\boldsymbol{r}_{0,i} := \mathsf{PRG}_2(\mathsf{salt}, \mathsf{seed}_{e^*,i} :: \hat{\ell})$
endfor
$\boldsymbol{v}_0 = \cdots = \boldsymbol{v}_{\kappa-1} := 0^{\hat{\ell}}$
for $j \in [\kappa]$:
  for $i \in [\frac{N_{e^*}}{2^{j+1}}]$:
    $\boldsymbol{v}_j := \boldsymbol{v}_j \oplus \boldsymbol{r}_{j,2i+1}$
    $\boldsymbol{r}_{j+1,i} := \boldsymbol{r}_{j,2i} \oplus \boldsymbol{r}_{j,2i+1}$
  endfor
endfor
$\boldsymbol{u} := \boldsymbol{r}_{\kappa,0}$
if $\mu > \kappa$:
  for $i \in [\mu - \kappa]$:
    $\boldsymbol{v}_{\kappa+i} = 0^{\hat{\ell}}$
  endfor
  return $(\boldsymbol{u}, \boldsymbol{v}_0, \ldots, \boldsymbol{v}_{\kappa-1}, \boldsymbol{v}_\kappa, \ldots, \boldsymbol{v}_{\mu-1})$
else :
  return $(\boldsymbol{u}, \boldsymbol{v}_0, \ldots, \boldsymbol{v}_{\kappa-1})$ // This will happen only if $\mu = \kappa$.

Algorithm 4.5: VOLECommit$\left(\mathsf{mseed}, \mathsf{salt} :: \hat{\ell}\right)$

Public information and inputs

Public information: A number of iterations τ, a number of parties $N := \sum_{e=0}^{\tau-1} N_e$, $k_e := \log_2(N_e) \in \{\kappa_0, \kappa_1\}$, $\rho := \mu_0\tau'_0 + \mu_1\tau'_1$
Prover's input: A master seed $\mathsf{mseed} \in \{0,1\}^\lambda$ and a $\mathsf{salt} \in \{0,1\}^{2\lambda}$, a length $\hat{\ell} \in \mathbb{N}$

Output

Prover's output: A commitment $h_{\mathsf{com}} \in \{0,1\}^{2\lambda}$, a decommitment auxilary variable decom, VOLE corrections $(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1})$, VOLE correlation secrets $\boldsymbol{u} \in \mathbb{F}_2^{\hat{\ell}}$, VOLE correlation $\boldsymbol{v}$-vectors $\boldsymbol{V} \in \mathbb{F}_2^{\hat{\ell}\times\rho}$

$\left(\{\mathsf{seed}_{e,i}\}_{\tau,N}, h_{\mathsf{com}}, \mathsf{decom} := (\mathsf{nodes}, \{\mathsf{com}_{e,i}\}_{\tau,N})\right) := \mathsf{VC.Commit}(\mathsf{mseed}, \mathsf{salt})$
for $e \in [\tau]$:
  $(\boldsymbol{u}_e, \boldsymbol{v}_{e,0}, \ldots, \boldsymbol{v}_{e,\mu_e-1}) := \mathsf{ConvertToVOLE}(N_e, (\mathsf{seed}_{e,i})_{i\in[N_e]}, \mathsf{salt}, \mu_e :: \hat{\ell})$
  $\boldsymbol{V}_e := [\boldsymbol{v}_{e,0} \cdots \boldsymbol{v}_{e,\mu_e-1}] \in \mathbb{F}_2^{\hat{\ell}\times\mu_e}$
endfor
$\boldsymbol{V} := [\boldsymbol{V}_0 \cdots \boldsymbol{V}_{\tau-1}] \in \mathbb{F}_2^{\hat{\ell}\times\rho}$
$\boldsymbol{u} := \boldsymbol{u}_0$
for $e \in [1, \tau-1]$:
  $\boldsymbol{c}_e := \boldsymbol{u} \oplus \boldsymbol{u}_e$
endfor
return $(h_{\mathsf{com}}, \mathsf{decom}, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}, \boldsymbol{u}, \boldsymbol{V})$.

Algorithm 4.6: ChallDec(ch)

Public information and inputs

Public information: A number of iterations τ, a number of parties $N = \sum_{e=1}^{\tau} N_e$, $\kappa_e = \log_2(N_e)$
Inputs: A challenge $\mathsf{ch} \in \{0,1\}^{\tau_0\kappa_0+\tau_1\kappa_1}$

Output

An ordered index set $\boldsymbol{i}^* := (i^*_0, \ldots, i^*_{\tau-1}) \in [N_0] \times [N_1] \times \cdots \times [N_{\tau-1}]$ of size τ.

$\mathtt{lo} := 0$
for $e \in [\tau]$:
  $i^*_e := \mathsf{NumRec}(\kappa_e, \mathsf{ch}[\mathtt{lo} : \mathtt{lo} + \kappa_e - 1])$
  $\mathtt{lo} := \mathtt{lo} + \kappa_e$
endfor
return $\boldsymbol{i}^* := (i^*_0, \ldots, i^*_{\tau-1})$

Algorithm 4.7: VOLEReconstruct$\left(\boldsymbol{i}^*, \mathsf{pdecom}, \{\mathsf{com}_{e,\boldsymbol{i}^*}\}_{\tau,\tau}, \mathsf{salt}\right)$

Public information and inputs

Public information: A number of iterations τ, a number of parties $N = \sum_{e=0}^{\tau-1} N_e$.
Verifier's input: An ordered list $\boldsymbol{i}^* \in [N_0] \times [N_1] \times \cdots \times [N_{\tau-1}]$ of τ indexes corresponding to hidden leaves, a sibling path pdecom, unopened commitments $\{\mathsf{com}_{e,\boldsymbol{i}^*}\}_{\tau,\tau}$, and a $\mathsf{salt} \in \{0,1\}^{2\lambda}$.

Output

A commitment $h_{\mathsf{com}} \in \{0,1\}^{2\lambda}$, and verifier's VOLE correlation inputs $\boldsymbol{Q}' := [\boldsymbol{Q}'_0 \cdots \boldsymbol{Q}'_{\tau-1}] \in \mathbb{F}_2^{\hat{\ell} \times \rho}$

1 : $\mathsf{out} := \mathsf{VC.Reconstruct}\left(\boldsymbol{i}^*, \mathsf{pdecom}, \{\mathsf{com}_{e,\boldsymbol{i}^*}\}_{\tau,\tau}, \mathsf{salt}\right)$
2 : **if** $\mathsf{out} = \perp$:
3 : **return** $\perp$
4 : **else** :
5 : Parse out as $\mathsf{out} := (h_{\mathsf{com}}, \{\mathsf{seed}_{e,i}\}_{\tau,N})$.
6 : **for** $e \in [\tau]$
7 : $\Delta_e := \boldsymbol{i}^*[e]$
8 : **for** $i \in [N_e]$: $\mathsf{seed}'_{e,i} := \mathsf{seed}_{e,i \oplus \Delta_e}$ // permute $\mathsf{seed}_{e,i}$ by using Δ_e
9 : $(\boldsymbol{u}'_e, \boldsymbol{q}_{e,0}, \ldots, \boldsymbol{q}_{e,\mu_e-1}) := \mathsf{ConvertToVOLE}(N_e, (\mathsf{seed}'_{e,i})_{i \in [N_e]}, \mathsf{salt}, \mu_e :: \hat{\ell})$
10 : $\boldsymbol{Q}'_e := [\boldsymbol{q}_{e,0} \cdots \boldsymbol{q}_{e,\mu_e-1}] \in \mathbb{F}_2^{\hat{\ell} \times \mu_e}$
11 : **endfor**
12 : **return** $(h_{\mathsf{com}}, \boldsymbol{Q}'_0, \ldots, \boldsymbol{Q}'_{\tau-1})$.

Ensuring VOLE consistency. It is crucial for the verifier to ensure that the correction values $\boldsymbol{c}_e$ sent by the prover are consistent with the committed VOLE correlation input $\boldsymbol{u}_0$. The verifier ensures this by asking the prover to compute a random linear universal hash. In this section, we explain this process in detail.[3] We begin by defining family of linear universal hash functions, since it will be used to conduct the consistency checks.

Definition 4.1 (Linear universal hash functions). *A family of linear hash functions is a family of matrices $\mathcal{H} \subseteq \mathbb{F}_q^{r \times n}$. The family is* ε-almost universal*, if*

[3] This technique is independent of the underlying PoK scheme or signature scheme since this generically helps the verifier to check the consistency of the VOLE correlations committed by the prover. Therefore, in PERK we use the exact same techniques and algorithm for these checks as those implemented in FAEST [BBD+23b]. Due to this reason we simply (re)state important definitions, propositions, and lemmas which are essentially same as those presented in [BBD+23b].

for any $\boldsymbol{x} \in \mathbb{F}_q^n \setminus \{\mathbf{0}\}$,

$$\Pr\Big[\boldsymbol{H}\boldsymbol{x} = \mathbf{0} \,:\, \boldsymbol{H} \xleftarrow{\$} \mathcal{H}\Big] \leq \varepsilon.$$

The family is ε-almost uniform, *if for any* $\boldsymbol{x} \in \mathbb{F}_q^n \setminus \{\mathbf{0}\}$ *and for any* $\boldsymbol{v} \in \mathbb{F}_q^r$,

$$\Pr\Big[\boldsymbol{H}\boldsymbol{x} = \boldsymbol{v} \,:\, \boldsymbol{H} \xleftarrow{\$} \mathcal{H}\Big] \leq \varepsilon.$$

In order to preserve the zero-knowledge property when the hash outputs are shared with the verifier, the hashes used in PERK must also satisfy the hiding property given below.

Definition 4.2. *A matrix* $\boldsymbol{H} \in \mathbb{F}_q^{r\times(h+n)}$ *is* $\mathbb{F}_q^n$*-hiding if for* $\boldsymbol{v}[0, h-1] \xleftarrow{\$} \mathbb{F}_q^h$ *the distribution of* $\boldsymbol{H}\boldsymbol{v}$ *is independent from* $\boldsymbol{v}[h, n+h-1]$. *A hash family* $\mathcal{H} \subseteq \mathbb{F}_q^{r\times(h+n)}$ *is* $\mathbb{F}_q^n$*-hiding if every* $\boldsymbol{H} \in \mathcal{H}$ *is* $\mathbb{F}_q^n$*-hiding.*

Proposition 4.1. *Let* $\mathcal{H} \subseteq \mathbb{F}_q^{r\times n}$ *be* ε*-almost uniform hash family. Let* $\mathcal{H}' \subseteq \mathbb{F}_q^{r\times(r+n)}$ *be the family* $\{[\boldsymbol{I}_r\ \boldsymbol{H}] : \boldsymbol{H} \in \mathcal{H}\}$, *where* $\boldsymbol{I}_r$ *is the* $r \times r$ *identity matrix. Then, (a)* $\mathcal{H}'$ *is* ε*-almost universal, and (b)* $\mathcal{H}'$ *is* $\mathbb{F}_q^n$*-hiding.*

Proof. Let $\boldsymbol{x} \coloneqq \begin{bmatrix}\boldsymbol{x}_0\\ \boldsymbol{x}_1\end{bmatrix}$ be non-zero, with $\boldsymbol{x}_0 \in \mathbb{F}_q^r$ and $\boldsymbol{x}_1 \in \mathbb{F}_q^n$. If $\boldsymbol{H}' \in \mathcal{H}'$ then, $\boldsymbol{H}'\boldsymbol{x} = 0$ implies that $-\boldsymbol{x}_0 = \boldsymbol{H}\boldsymbol{x}_1$. Since $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ cannot be equal to zero simultaneously (because $\boldsymbol{x}$ is non-zero), this implies that $\boldsymbol{x}_1 \neq \mathbf{0}$. Therefore from ε-almost uniform property of $\mathcal{H}$, we can conclude that $\mathcal{H}'$ is ε-almost universal. The hiding property of $\mathcal{H}'$ holds because when the first r elements ($\boldsymbol{x}_0$) of the input are chosen uniformly at random then they perfectly mask the rest of the output component $\boldsymbol{H}\boldsymbol{x}_1$. □

Standard constructions of linear universal hash families. Following [BBD+23b] we also use the matrix hash family and polynomial-based hash family as building blocks of our VOLE consistency checks [CW79,BJKS94]. The matrix hash family $\mathcal{H} = \mathbb{F}_q^{r\times n}$ is q^{-r}-almost uniform. In polynomial-based hash, the input $\boldsymbol{x} \in \mathbb{F}_q^n$ is seen as the coefficients of a polynomial with degree $\leq n-1$. Sampling a hash function is implemented by evaluating such a polynomial at a uniform random point in $\mathbb{F}_q$. Since the polynomial has at most $n-1$ roots, the polynomial hash family is $\frac{n-1}{q}$-almost universal. If the random point is chosen from a set S with cardinality $|S|$, then the polynomial hash family is $\frac{n-1}{|S|}$-almost universal.

Composition and truncation of hashes. We also recall the properties of composition and truncation of hashes originally proved by the authors of [Sti92, Roy22, BBD+23b]. These properties will be useful in proving that the prover can successfully bypass the consistency check for VOLE correlations with extremely low probability.

Proposition 4.2. *Let $\mathcal{H}$ and $\mathcal{H}'$ be two independent ε and ε'-almost universal hash families respectively. Then the concatenation $\left\{\begin{bmatrix}\boldsymbol{H}\\ \boldsymbol{H}'\end{bmatrix} : \boldsymbol{H} \in \mathcal{H}, \boldsymbol{H}' \in \mathcal{H}'\right\}$ is $\varepsilon\varepsilon'$-almost universal.*

Proof. This holds true because of the independence of $\mathcal{H}$ and $\mathcal{H}'$. □

Proposition 4.3. *Let $\mathcal{H} \subseteq \mathbb{F}_q^{r' \times n}$ be ε-almost universal and $\mathcal{H}' \subseteq \mathbb{F}_q^{r \times r'}$ be ε'-almost uniform. Then the product $\{\boldsymbol{H}'\boldsymbol{H} : \boldsymbol{H} \in \mathcal{H}, \boldsymbol{H}' \in \mathcal{H}'\}$ is $(\varepsilon + \varepsilon')$-almost uniform.*

Proof. Let $\boldsymbol{x} \in \mathbb{F}_q^n \setminus \{\boldsymbol{0}\}$. Then $\Pr[\boldsymbol{x}' \neq 0 : \boldsymbol{x}' = \boldsymbol{H}\boldsymbol{x}] \geq 1 - \varepsilon$ since $\mathcal{H}$ is ε-almost universal. For $\boldsymbol{x}' \neq 0$, $\Pr\left[\boldsymbol{H}'\boldsymbol{x}' \neq \boldsymbol{v} : \boldsymbol{v} \in \mathbb{F}_q^r\right] \geq 1 - \varepsilon'$ as $\mathcal{H}'$ is ε'-almost uniform. Therefore $\Pr[\boldsymbol{H}'\boldsymbol{H}\boldsymbol{x} \neq \boldsymbol{v}] \geq (1-\varepsilon)(1-\varepsilon') \geq 1 - \varepsilon - \varepsilon'$, which implies that the product is $(\varepsilon + \varepsilon')$-almost uniform. □

Proposition 4.4. *Let $\delta \in \mathbb{N}$ and $\mathcal{H} \subseteq \mathbb{F}_q^{r \times n}$ be ε-almost uniform hash family. Then, the truncated family $\{\boldsymbol{H}_{0,r-\delta-1} : \boldsymbol{H} \in \mathcal{H}\}$, where $\boldsymbol{H}_{0,r-\delta-1}$ denotes the first $(r-\delta)$ rows of $\boldsymbol{H}$, is εq^{δ}-almost uniform.*

Proof. For each $\boldsymbol{H} \in \mathcal{H}$, let $\boldsymbol{H} \coloneqq \begin{bmatrix}\boldsymbol{H}_0\\ \boldsymbol{H}_1\end{bmatrix}$ where $\boldsymbol{H}_0 \in \mathbb{F}_q^{(r-\delta)\times n}$ and $\boldsymbol{H}_1 \in \mathbb{F}_q^{\delta \times n}$. Let $\boldsymbol{x} \in \mathbb{F}_q^n$ be a non-zero vector, and $\boldsymbol{y} \coloneqq \begin{bmatrix}\boldsymbol{y}_0\\ \boldsymbol{y}_1\end{bmatrix} \in \mathbb{F}_q^r$. If $\boldsymbol{H} \xleftarrow{\$} \mathcal{H}$, then $\Pr[\boldsymbol{H}\boldsymbol{x} = \boldsymbol{y}] \leq \varepsilon$. We can then apply conditional probability to obtain,

$$\begin{aligned}
\Pr[\boldsymbol{H}\boldsymbol{x} = \boldsymbol{y}] &\leq \varepsilon \\
\Pr[\boldsymbol{H}_0\boldsymbol{x}_0 = \boldsymbol{y}_0 \wedge \boldsymbol{H}_1\boldsymbol{x}_1 = \boldsymbol{y}_1] &\leq \varepsilon \\
\Pr[\boldsymbol{H}_0\boldsymbol{x}_0 = \boldsymbol{y}_0] \cdot \Pr[\boldsymbol{H}_1\boldsymbol{x}_1 = \boldsymbol{y}_1 \mid \boldsymbol{H}_0\boldsymbol{x}_0 = \boldsymbol{y}_0] &\leq \varepsilon \\
\Pr[\boldsymbol{H}_0\boldsymbol{x}_0 = \boldsymbol{y}_0] &\leq \varepsilon \cdot \Pr[\boldsymbol{H}_1\boldsymbol{x}_1 = \boldsymbol{y}_1 \mid \boldsymbol{H}_0\boldsymbol{x}_0 = \boldsymbol{y}_0]^{-1} \\
&\leq \varepsilon q^{\delta}
\end{aligned}$$

where the final inequality comes from fixing a $\boldsymbol{y}_1 \in \mathbb{F}_q^{\delta}$, that maximizes $p \coloneqq \Pr[\boldsymbol{H}_1\boldsymbol{x}_1 = \boldsymbol{y}_1 \mid \boldsymbol{H}_0\boldsymbol{x}_0 = \boldsymbol{y}_0]$, which implies p is at least $q^{-\delta}$. □

VOLE universal hash. In order to verify the consistency of VOLE correlation inputs in $\mathbb{F}_2^{\hat{\ell}}$, we need a hash family that is linear over $\mathbb{F}_2$. Also recall that, $\hat{\ell} \coloneqq \ell_{\mathsf{VOLEHashMask}} + \ell + \ell_{\mathsf{CZMask}}$, where $\ell_{\mathsf{VOLEHashMask}} \coloneqq \lambda + B$, and $B = 16$ is a parameter chosen for security. [4] To compute the hash, we start by mapping the seed seed into $(r_0, r_1, r_2, r_3, s, t) \in \mathbb{F}_{2^\lambda}^5 \times \mathbb{F}_{2^{64}}$. The input $\boldsymbol{x} \in \mathbb{F}_2^{\hat{\ell}}$ is split into

[4] Similar to [BBD+23b] PERK actually calls VOLEHash on inputs represented as $\hat{\ell} \times \rho$ matrix, which is translated into computing the hash on each column separately, with the same seed.

$(\boldsymbol{x}_0, \boldsymbol{x}_1)$, where $\boldsymbol{x}_1 \in \mathbb{F}_2^{\ell+\ell_{\mathsf{CZMask}}}$, and then $\boldsymbol{x}_1$ is parsed twice, first as a vector $\hat{\boldsymbol{y}}$ of $\mathbb{F}_{2^\lambda}$ elements, and then as a vector $\overline{\boldsymbol{y}}$ of $\mathbb{F}_{2^{64}}$ elements. [5] Then compute,

$$h_0 \coloneqq \hat{y}_0 s^{\frac{\hat{\ell}}{\lambda}-1} + \hat{y}_1 s^{\frac{\hat{\ell}}{\lambda}-2} + \cdots + \hat{y}_{\frac{\hat{\ell}}{\lambda}-2} s + \hat{y}_{\frac{\hat{\ell}}{\lambda}-1} \in \mathbb{F}_{2^\lambda},$$

$$h_1 \coloneqq \overline{\boldsymbol{y}}_0 t^{\frac{\hat{\ell}}{64}-1} + \overline{\boldsymbol{y}}_1 t^{\frac{\hat{\ell}}{64}-2} + \cdots + \overline{\boldsymbol{y}}_{\frac{\hat{\ell}}{64}-2} t + \overline{\boldsymbol{y}}_{\frac{\hat{\ell}}{64}-1} \in \mathbb{F}_{2^{64}}$$

Viewing h_1 as an element of $\mathbb{F}_{2^\lambda}$ (by padding zeros), the hash is then computed in $\mathbb{F}_{2^\lambda}$ as,

$$\begin{bmatrix} h_2 \\ h_3 \end{bmatrix} \coloneqq \begin{bmatrix} r_0 & r_1 \\ r_2 & r_3 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \end{bmatrix}$$

Finally we take the first $\ell_{\mathsf{VOLEHashMask}}$ (i.e. $\lambda + B$) bits of the concatenation of the field elements h_2 and h_3, and XOR it with $\boldsymbol{x}_0$. We argue security of this construction below (same as [BBD+23b]). Like [BBD+23b], we also aim for $\varepsilon \coloneqq 2^{-\ell_{\mathsf{VOLEHashMask}}} = 2^{-\lambda-B}$, with $B = 16$, in order to compensate for $\binom{\tau}{2}$ security loss shown by [BBD+23c].

Lemma 4.1. VOLEHash *is an* ε_v*-almost universal hash family in* $\mathbb{F}_2^{\ell_{\mathsf{VOLEHashMask}} \times \hat{\ell}}$*, for* $\varepsilon_v = 2^{-\ell_{\mathsf{VOLEHashMask}}} \left(1 + 2^{B-50}\right)$*, if* $\hat{\ell} \leq 2^{13}$*. Furthermore,* VOLEHash *is* $\mathbb{F}_2^{\ell+\ell_{\mathsf{CZMask}}}$*-hiding.*

Proof. We show ε_v-almost uniform property of the hash that outputs the first $\ell_{\mathsf{VOLEHashMask}}$ (i.e. $\lambda + B$) bits of (h_2, h_3), that is without adding $\boldsymbol{x}_0$. By Proposition 4.1, this implies the hiding and ε_v-almost universal property of the final hash. The first part of the hash which computes h_0, h_1, is a concatenation of two polynomial hashes, over either $\mathbb{F}_{2^\lambda}$ or $\mathbb{F}_{2^{64}}$. These are ε-almost universal with $\varepsilon = \frac{d}{|\mathbb{F}|}$, where d is the degree of the polynomial and $\mathbb{F}$ is the field, and we have $d \leq \frac{\hat{\ell}}{64}$. Both these hashes are $\mathbb{F}_2$-linear, as the binary field multiplication is bilinear over $\mathbb{F}_2$. Thus, applying Proposition 4.2 we get that the concatenation of the two polynomial hashes is then ε_0-almost universal with $\varepsilon_0 \leq \frac{\hat{\ell}^2}{2^{\lambda+76}}$. Therefore, for $\hat{\ell} \leq 2^{13}$, we have $\varepsilon_0 \leq 2^{-\lambda-50}$.

The second part of the hash starts with a 2×2 matrix hash, which is $2^{-2\lambda}$-almost uniform. After the truncation, the result is ε_1-almost uniform for $\varepsilon_1 = 2^{-\ell_{\mathsf{VOLEHashMask}}}$ due to the Proposition 4.4. The final combined hash is a product of these two parts, so from Proposition 4.3 and summing the probabilities, we get that for all $\hat{\ell} \leq 2^{13}$, the hash is ε_v-almost uniform for $\varepsilon_v = \varepsilon_0 + \varepsilon_1 \leq 2^{-\ell_{\mathsf{VOLEHashMask}}} \left(1 + 2^{B-50}\right)$. □

Following algorithms help prove consistency of the VOLE correlations.

[5] In order to allow for better parsing in PERK we swap the order of $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ from [BBD+23b]. That is $\boldsymbol{x}_0$ serves as a mask in PERK, where as [BBD+23b] uses $\boldsymbol{x}_1$ as a mask.

Algorithm 4.8: VOLEHash(seed, $\boldsymbol{x} :: \ell_{\mathsf{VOLEHashMask}}$)

Public information and inputs

Inputs: A seed, represented as a tuple $(\boldsymbol{r}_0, \boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{s}, \boldsymbol{t}) \in \{0,1\}^{5\lambda+64}$, and a vector to be hashed $\boldsymbol{x} \in \{0,1\}^{\hat{\ell}}$, represented as a pair $(\boldsymbol{x}_0, \boldsymbol{x}_1) \in \{0,1\}^{\ell_{\mathsf{VOLEHashMask}}} \times \{0,1\}^{\ell+\ell_{\mathsf{CZMask}}}$

Output

A VOLE hash $\boldsymbol{h} \in \{0,1\}^{\ell_{\mathsf{VOLEHashMask}}}$

---

1 : $r_i \coloneqq \mathsf{ToField}(\boldsymbol{r}_i, \lambda)$ for $i \in [4]$

2 : $s \coloneqq \mathsf{ToField}(\boldsymbol{s}, \lambda)$

3 : $t \coloneqq \mathsf{ToField}(\boldsymbol{t}, 64)$

4 : $\ell' \coloneqq \lambda\lceil(\ell + \ell_{\mathsf{CZMask}})/\lambda\rceil$

5 : $\boldsymbol{x}_1 \coloneqq \boldsymbol{x}_1 \| 0^{\ell' - (\ell+\ell_{\mathsf{CZMask}})}$ (pad to multiple of λ)

6 : $\hat{\boldsymbol{y}} \coloneqq \mathsf{ToField}(\boldsymbol{x}_1, \lambda)$

7 : $\overline{\boldsymbol{y}} \coloneqq \mathsf{ToField}(\boldsymbol{x}_1, 64)$

8 : $h_0 \coloneqq \sum_{i=0}^{\ell'/\lambda-1} s^{\ell'/\lambda-1-i} \hat{\boldsymbol{y}}[i]$

9 : $h_1 \coloneqq \sum_{i=0}^{\ell'/64-1} s^{\ell'/64-1-i} \overline{\boldsymbol{y}}[i]$

10 : $h_1' \coloneqq \mathsf{ToField}(\mathsf{ToBits}(h_1) \| 0^{\lambda-64}, \lambda)$

11 : $(h_2, h_3) \coloneqq (r_0 h_0 + r_1 h_1', r_2 h_0 + r_3 h_1')$

12 : $\boldsymbol{h} \coloneqq (\mathsf{ToBits}(h_2) \| \mathsf{ToBits}(h_3)[0..\ell_{\mathsf{VOLEHashMask}} - 1]) \oplus \boldsymbol{x}_0$

13 : **return** $\boldsymbol{h}$

4.5 Proof of Knowledge functions

In this section, we present the various algorithms used by the prover and by the verifier to prove (respectively verify) the knowledge of the secret permutation which serves as solution to the PKP instance defined by the public key. This is achieved with help of VOLE correlations created as described in Section 4.4, and then using such VOLE correlations to compute linear functions over polynomials to ascertain that different constraints related to the elementary vector structure, and the satisfiability of the PKP solution are fulfilled by these polynomials.

The prover begins by proving the knowledge of elementary vectors of length 4 or 2, and then generates polynomials corresponding to the rows of the secret permutation matrix by computing the tensor products of these smaller elementary vectors. In order to prove the elementary structure of the length 4 (or length 2) vector, the prover needs to show that following constraints are obeyed by the underlying vector:

1. For a vector $\boldsymbol{e} \coloneqq [e_0, e_1, e_2, e_3] \in \mathbb{F}_2^4$ the product $e_0 \cdot e_1 = 0$ and $e_2 \cdot e_3 = 0$ simultaneously. In case of elementary vector $\boldsymbol{e} \coloneqq [e_0, e_1] \in \mathbb{F}_2^2$ this is achieved simply by showing $e_0 \cdot e_1 = 0$.

2. Additionally, the prover should also prove that $e_0 \oplus e_1 \oplus e_2 \oplus e_3 = 1$ (respectively $e_0 \oplus e_1 = 1$).

Note that proving constraint 2, actually allows for optimization leading to saving of a single bit per elementary vector. The prover simply sends 3 bits (or 1 bit) and the remaining bit of the elementary is computed by the verifier as: $e_3 \coloneqq 1 \oplus e_0 \oplus e_1 \oplus e_2$ (respectively $e_1 \coloneqq 1 \oplus e_0$). The algorithms algorithm 4.9 CompWit and algorithm 4.10 ExpWit describe this process, these algorithms are used by the prover and the verifier respectively.

Witness Compression and Expansion

Algorithm 4.9: CompWit($\boldsymbol{w}'$)

Public information and inputs

Public information: Compress input witness $\boldsymbol{w}'$ into shorter witness $\boldsymbol{w}$ by dropping bits.

1 : Initialize $\boldsymbol{w} := \varepsilon$
2 : **if** len($\boldsymbol{w}'$) = 12:
// Parse $\boldsymbol{w}'$ as $\boldsymbol{w}' := \boldsymbol{w}'_0||\boldsymbol{w}'_1||\boldsymbol{w}'_2$
// where,
// $\boldsymbol{w}'_i := w'_{i,0}||w'_{i,1}||w'_{i,2}||w'_{i,3}$.
3 : **for** $i \in [3]$:
4 : **for** $j \in [3]$:
5 : $w_{i,j} := w'_{i,j}$
6 : **endfor**
7 : **endfor**
8 : **assert** (len($\boldsymbol{w}$) = 9)
9 : **return** $\boldsymbol{w}$
10 : **elseif** len($\boldsymbol{w}'$) = 14:
// Parse $\boldsymbol{w}'$ as
// $\boldsymbol{w}' := \boldsymbol{w}'_0||\boldsymbol{w}'_1||\boldsymbol{w}'_2||\boldsymbol{w}'_3$
// with $\boldsymbol{w}'_i$ defined as in line 2
// for $i \in [3]$ and
// $\boldsymbol{w}'_3 := w'_{3,0}||w'_{3,1}$.
11 : $\boldsymbol{w} :=$ CompWit $(\boldsymbol{w}'_0||\boldsymbol{w}'_1||\boldsymbol{w}'_2)$
12 : $\boldsymbol{w} := \boldsymbol{w}||w'_{3,0}$
13 : **assert** (len($\boldsymbol{w}$) = 10)
14 : **return** $\boldsymbol{w}$
15 : **else** :
16 : **return** $\perp$.
17 : **endif**

Algorithm 4.10: ExpWit($\boldsymbol{w}$)

Public information and inputs

Public information: Expand compressed input witness $\boldsymbol{w}$ into witness $\boldsymbol{w}'$.

1 : Initialize $\boldsymbol{w}' := \varepsilon$
2 : **if** len($\boldsymbol{w}$) = 9:
// Parse $\boldsymbol{w}$ as
// $\boldsymbol{w} := w_{0,0}||\cdots||w_{i,j}||\cdots||w_{2,2}$.
// For $i \in [3]$ and $j \in [3]$.
3 : **for** $i \in [3]$:
4 : $w'_{i,3} := 1$
5 : **for** $j \in [3]$:
6 : $w'_{i,j} := w_{i,j}$
7 : $w'_{i,3} := w'_{i,3} \oplus w_{i,j}$
8 : $\boldsymbol{w}' := \boldsymbol{w}'||w'_{i,j}$
9 : **endfor**
10 : $\boldsymbol{w}' := \boldsymbol{w}'||w'_{i,3}$
11 : **endfor**
12 : **assert** (len($\boldsymbol{w}'$) = 12)
13 : **return** $\boldsymbol{w}'$
14 : **elseif** len($\boldsymbol{w}$) = 10:
// Parse $\boldsymbol{w}$ as
// $\boldsymbol{w} := \tilde{\boldsymbol{w}}||w_{3,0}$, and
// $\tilde{\boldsymbol{w}} := w_{0,0}||\cdots||w_{ij}||\cdots||w_{2,2}$.
// For $i \in [3]$ and $j \in [3]$.
15 : $\boldsymbol{w}' :=$ ExpWit($\tilde{\boldsymbol{w}}$)
16 : $\boldsymbol{w}' := \boldsymbol{w}'||w_{3,0}||(1 \oplus w_{3,0})$
17 : **assert** (len($\boldsymbol{w}'$) = 14)
18 : **return** $\boldsymbol{w}'$
19 : **else** :
20 : **return** $\perp$.
21 : **endif**

Prover. In this section we present all the algorithms that will be used by the prover (signer) to produce the proof of knowledge. The prover begins by encoding each row of the secret permutation matrix into smaller elementary vectors. Algorithms EncPosArrayToWitness and PosToWitness together take an input position corresponding to the non-zero element in a row of permutation matrix,

and output its corresponding unique witness encoded as blocks of elementary vectors. Let $\boldsymbol{e}_i$ be an elementary vector of length 4, generated by shifting the bit-string `'0001'` to left by i positions. Therefore, $\boldsymbol{e}_0 \coloneqq$ `'0001'`, $\boldsymbol{e}_1 \coloneqq$ `'0010'`, $\boldsymbol{e}_2 \coloneqq$ `'0100'`, and $\boldsymbol{e}_3 \coloneqq$ `'1000'`. Also, let $\boldsymbol{e}'_b$ be an elementary vector of length 2, generated by shifting the bit-string `'01'` to left by either b positions for $b \in \{0, 1\}$. Therefore, $\boldsymbol{e}'_0 \coloneqq$ `'01'` and $\boldsymbol{e}'_1 \coloneqq$ `'10'`.

Algorithm 4.11 EncPosArrayToWitness takes the unique encoded position array $[i, j, k]$ corresponding to some position pos $\in [64]$ generated from EncodeNum-64, and outputs its corresponding unique witness $\boldsymbol{w}' \coloneqq \boldsymbol{e}_k||\boldsymbol{e}_j||\boldsymbol{e}_i \in \mathbb{F}_2^{12}$.

Algorithm 4.12 PosToWitness on an input position pos, first computes the unique encoded position array encPosArray corresponding to pos by calling EncodeNum. If encPosArray contains exactly 3 elements then, PosToWitness outputs corresponding unique witness for pos as $\boldsymbol{w}' \coloneqq \boldsymbol{e}_k||\boldsymbol{e}_j||\boldsymbol{e}_i \in \mathbb{F}_2^{12}$ by subsequently calling EncPosArrayToWitness. Otherwise if encPosArray $\coloneqq [i, j, k, b]$ contains exactly 4 elements then, it computes $\boldsymbol{e}_k||\boldsymbol{e}_j||\boldsymbol{e}_i$ as described above. It then outputs corresponding unique witness for pos as $\boldsymbol{w}' \coloneqq \boldsymbol{e}'_b||\boldsymbol{e}_k||\boldsymbol{e}_j||\boldsymbol{e}_i \in \mathbb{F}_2^{14}$.

Next, the prover creates the VOLE correlations that will be used in proof generation, by embedding the witness (now represented as elementary vectors). This is achieved by Algorithm 4.13 EmbedWitnessBlock which takes the witness generated by PosToWitness as input and embeds it block-by-block inside random VOLE correlation also given as inputs. The output of this algorithm are the VOLE correlations $[\![\boldsymbol{\beta}'_i]\!]$ corresponding to the elementary blocks of witness. Algorithm 4.14 EmbedWitness aggregates the VOLE correlations embedding the witness from individual elementary vectors of lengths 4 and 2, and outputs VOLE correlations $[\![\boldsymbol{\beta}']\!]$ corresponding to the aggregation of 3 elementary vectors of lengths 4 (and in case of L3 and L5 parameters, another elementary vector of length 2).

Once the prover possesses the VOLE correlations (linear polynomials) corresponding to the elementary vector entries, it then computes the degree-d VOLE correlations (degree-d polynomials) corresponding to each individual row of the secret permutation matrix with help of Algorithm 4.16 TensorProductToElementaryVector, which internally calls Algorithm 4.15 TensorProduct to compute the tensor product between two blocks. At this stage the prover has computed n polynomials per row each of degree d, $[\![\boldsymbol{z}]\!]^{(d)} \coloneqq ([\![z_0]\!]^{(d)}, \ldots, [\![z_{n-1}]\!]^{(d)})$ which are the VOLE correlations corresponding to the rows of the secret permutation matrix, viewed as an elementary vector of length n. The Algorithm 4.17 VOLE-ElementaryVector produces the VOLE correlations $[\![\boldsymbol{z}]\!]^{(d)}$ along with intermediate values $[\![\boldsymbol{\beta}']\!]$, and masked witness $\boldsymbol{t}$ which will be sent to the verifier.

After obtaining the VOLE correlations for each of the n rows by running Algorithm 4.17 VOLE-ElementaryVector n times, the prover further computes extra n degree-d polynomials, $[\![\mathsf{ColCheck}_j]\!]^{(d)}$ which ensure that each column of secret matrix adds upto exactly 1. This is described in Algorithm 4.18 VOLE-

Permutation. The check that columns add upto exactly 1 along with the elementary structure of the individual rows is sufficient to prove the permutation structure of the secret matrix.

The prover proves the elementary structure by computing the polynomials which have leading coefficient equal to 0 if and only if constraint 1 mentioned above ($e_0 \cdot e_1 = 0$ and $e_2 \cdot e_3 = 0$) is satisfied using Algorithm 4.19 Check-ElementaryBlock, and Algorithm 4.20 Check-ElementaryVector.

So far we have seen that, the prover has generated many degree-d polynomials, which should all have leading coefficients equal to 0. In order to verify, this the prover needs to send these polynomials to the verifier which can then evaluate them at a random point of its choice and check if the all the leading coefficients are equal to zero as expected. However, sending so many degree-d polynomials is inefficient, the prover can instead send a single degree-d polynomial by taking a random linear combination of all the polynomials, where the coefficients of the random linear combination are chosen by the verifier.

Note that, in order to convince the verifier the prover still needs to send a degree-d polynomial to the verifier whose leading coefficient is supposed to be zero. However, since all these polynomials are computed by taking tensor products and other linear functions of secret witness (embedded in the VOLE correlations), each coefficient of this polynomial obtained by the prover contains information about the secret. As the prover only needs to reveal that leading coefficient is zero, it should add a masking polynomial to blind the remaining coefficients. The Algorithm 4.21 CheckZero achieves this and outputs a masked polynomial with leading coefficient equal to zero which can be used by the verifier.

Finally, the Algorithm 4.22 Check-PKP puts all of the checks for checking the elementary structure of rows (blocks), column sums equaling to 1, and satisfiability of PKP equation together by computing degree-d polynomials with zero as leading coefficients if and only if these constraints are satisfied. These polynomials are then merged together into a single polynomial by computing (verifier dictated) random linear combination, which is then masked with the help of Algorithm 4.21 CheckZero. As a result, the prover should send the masked witness $\boldsymbol{t}$ obtained from Algorithm 4.18 VOLE-Permutation along with the masked polynomial proof output by Algorithm 4.22 Check-PKP to the verifier.

Algorithm 4.11: EncPosArrayToWitness(encPosArray)

Public information and inputs

Public information: Given encoded position array encPosArray of length 3 with all input elements in $[4]$, outputs its corresponding unique witness $\boldsymbol{w}' \in \{0,1\}^{12}$ with hamming weight 3.

Initialize $\boldsymbol{w}' \coloneqq \varepsilon$ // Empty string
if len(encPosArray) $= 3$:
  // Parse encPosArray as encPosArray $\coloneqq [i, j, k]$ with $i, j, k \in [4]$, else output error.
  for index $\in [3]$:
    $\boldsymbol{w}'_{\text{index}} \coloneqq$ LeftShift ('0001', encPosArray[index])
    $\boldsymbol{w}' \coloneqq \boldsymbol{w}' || \boldsymbol{w}'_{\text{index}}$
  endfor
  assert (len$(\boldsymbol{w}') = 12$)
  return $\boldsymbol{w}'$
else :
  return $\perp$
endif

Algorithm 4.12: PosToWitness(pos)

Public information and inputs

Public information: Given secret input pos outputs its corresponding unique witness $\boldsymbol{w}'$.

---

1 : encPosArray := EncodeNum (pos)

Security Level 1

2 : **if** len(encPosArray) = 3 :

3 : $\quad \boldsymbol{w}' :=$ EncPosArrayToWitness (encPosArray)

4 : $\quad$ **return** $\boldsymbol{w}'$

Security Levels 3 and 5

5 : **elseif** len(encPosArray) = 4 :

6 : $\quad \boldsymbol{w}' :=$ EncPosArrayToWitness (encPosArray$[0:2]$)

7 : $\quad \boldsymbol{w}'_3 :=$ LeftShift ('01', encPosArray$[3]$)

8 : $\quad \boldsymbol{w}' := \boldsymbol{w}' || \boldsymbol{w}'_3$

9 : $\quad$ **assert** (len$(\boldsymbol{w}') = 14$)

10 : $\quad$ **return** $\boldsymbol{w}'$

11 : **else** :

12 : $\quad$ **return** $\perp$

13 : **endif**

Algorithm 4.13: P.EmbedWitnessBlock$\left(\boldsymbol{w}'_i, (\llbracket u_k \rrbracket)_{k\in[3]}\right)$

Public information and inputs

Public information: Length of the witness block = 4.
Prover's input: i^{th} secret witness block $\boldsymbol{w}'_i \coloneqq w'_{i,0}||w'_{i,1}||w'_{i,2}||w'_{i,3} \in \mathbb{F}_2^4$, 3 VOLE correlations $\llbracket u_k \rrbracket$ for random $(u_k, v_k) \in \mathbb{F}_2 \times \mathbb{F}_{2^\rho}$ represented as polynomials $f_{u_k}(X) = u_k X + v_k$ for $k \in [3]$.

Output

Prover's output: VOLE correlations $\llbracket \boldsymbol{\beta}'_i \rrbracket \coloneqq \left(\llbracket \beta'_{i,0} \rrbracket, \llbracket \beta'_{i,1} \rrbracket, \llbracket \beta'_{i,2} \rrbracket, \llbracket \beta'_{i,3} \rrbracket\right)$, where $\llbracket \beta'_{i,j} \rrbracket \in \mathbb{F}_2 \times \mathbb{F}_{2^\rho}$ for $j \in [4]$.

Construct VOLE correlations with witness

1 : // Parse $\boldsymbol{w}'_i$ as $\boldsymbol{w}'_i \coloneqq w'_{i,0}||w'_{i,1}||w'_{i,2}||w'_{i,3}$.
2 : **for** $j \in [3]$:
3 : $\quad \beta'_{i,j}(X) \coloneqq w'_{i,j} X + v_j$ // $\beta'_{i,j}$ are polynomials with coefficients in $\mathbb{F}_{2^\rho}$.
4 : **endfor**
5 : $\beta'_{i,3}(X) \coloneqq w'_{i,3} X + v_0 + v_1 + v_2$
6 : $\llbracket \boldsymbol{\beta}'_i \rrbracket \coloneqq \left(\llbracket \beta'_{i,0} \rrbracket, \llbracket \beta'_{i,1} \rrbracket, \llbracket \beta'_{i,2} \rrbracket, \llbracket \beta'_{i,3} \rrbracket\right)$
7 : **return** $\llbracket \boldsymbol{\beta}'_i \rrbracket$

Algorithm 4.14: P.EmbedWitness$\left(\boldsymbol{w}', (\llbracket u_k \rrbracket)_{k\in[\ell_{\mathsf{row}}]}\right)$

Public information and inputs

Public information: Length of the expanded secret witness $|\boldsymbol{w}'| \coloneqq 2\ell_{\mathsf{row}} - 6$.
Prover's input: Secret witness $\mathsf{w}' \in \mathbb{F}_2^{(2\ell_{\mathsf{row}}-6)}$, ℓ_{row} VOLE correlations $\llbracket u_k \rrbracket$ for random $(u_k, v_k) \in \mathbb{F}_2 \times \mathbb{F}_{2^\rho}$ represented as polynomials $f_{u_k}(X) = u_k X + v_k$ for $k \in [\ell_{\mathsf{row}}]$.

Output

Prover's output: $(2\ell_{\mathsf{row}} - 6)$ VOLE correlations $\llbracket \boldsymbol{\beta}' \rrbracket$ with elements in $\mathbb{F}_2 \times \mathbb{F}_{2^\rho}$.

Construct VOLE correlations with witness

1 : **if** $\mathsf{len}(\boldsymbol{w}') = 12$: // Security level 1
// Parse $\boldsymbol{w}'$ as $\boldsymbol{w}' \coloneqq \boldsymbol{w}'_0||\boldsymbol{w}'_1||\boldsymbol{w}'_2$, where, $\boldsymbol{w}'_i \coloneqq w'_{i,0}||w'_{i,1}||w'_{i,2}||w'_{i,3}$.
2 : **for** $i \in [3]$:
3 : $\llbracket \boldsymbol{\beta}'_i \rrbracket \coloneqq \mathsf{P.EmbedWitnessBlock}\left(\boldsymbol{w}'_i, (\llbracket u_k \rrbracket)_{k\in[3i,3i+2]}\right)$
4 : **endfor**
5 : $\llbracket \boldsymbol{\beta}' \rrbracket \coloneqq (\llbracket \boldsymbol{\beta}'_0 \rrbracket, \llbracket \boldsymbol{\beta}'_1 \rrbracket, \llbracket \boldsymbol{\beta}'_2 \rrbracket)$
6 : **return** $\llbracket \boldsymbol{\beta}' \rrbracket$
7 : **elseif** $\mathsf{len}(\boldsymbol{w}') = 14$: // Security levels 3 and 5
// Parse $\boldsymbol{w}'$ as $\boldsymbol{w}' \coloneqq \boldsymbol{w}'_0||\boldsymbol{w}'_1||\boldsymbol{w}'_2||w'_{3,0}||w'_{3,1}$, where, $\boldsymbol{w}'_i$ are as above (line 1)
8 : $(\llbracket \boldsymbol{\beta}'_0 \rrbracket, \llbracket \boldsymbol{\beta}'_1 \rrbracket, \llbracket \boldsymbol{\beta}'_2 \rrbracket) \coloneqq \mathsf{P.EmbedWitness}\left(\boldsymbol{w}'_0||\boldsymbol{w}'_1||\boldsymbol{w}'_2, (\llbracket u_k \rrbracket)_{k\in[9]}\right)$
9 : $\beta'_{3,0}(X) \coloneqq w'_{3,0}X + v_9$
10 : $\beta'_{3,1}(X) \coloneqq w'_{3,1}X + v_9$
11 : $\llbracket \boldsymbol{\beta}'_3 \rrbracket \coloneqq \left(\llbracket \beta'_{3,0} \rrbracket, \llbracket \beta'_{3,1} \rrbracket\right)$
12 : $\llbracket \boldsymbol{\beta}' \rrbracket \coloneqq (\llbracket \boldsymbol{\beta}'_0 \rrbracket, \llbracket \boldsymbol{\beta}'_1 \rrbracket, \llbracket \boldsymbol{\beta}'_2 \rrbracket, \llbracket \boldsymbol{\beta}'_3 \rrbracket)$
13 : **return** $\llbracket \boldsymbol{\beta}' \rrbracket$
14 : **else** :
15 : **return** $\perp$
16 : **endif**

Algorithm 4.15: P.TensorProduct$\left(\llbracket \boldsymbol{\beta}'_i \rrbracket^{(d_i)}, \llbracket \boldsymbol{\beta}'_j \rrbracket^{(d_j)}, \mathsf{num}\right)$

Public information and inputs

Public information: Sizes n_i and n_j of two blocks of VOLE correlations $\llbracket \boldsymbol{\beta}'_i \rrbracket^{(d_i)}$ and $\llbracket \boldsymbol{\beta}'_j \rrbracket^{(d_j)}$ respectively. Each element of $\llbracket \boldsymbol{\beta}'_i \rrbracket^{(d_i)}$ (resp. $\llbracket \boldsymbol{\beta}'_j \rrbracket^{(d_j)}$) is represented as degree d_i (resp. d_j) polynomial with coefficients in $\mathbb{F}_{2^\rho}$.
Prover's input: $(n_i + n_j)$ secret polynomials
$\llbracket \beta'_{i,0} \rrbracket^{(d_i)}, \ldots, \llbracket \beta'_{i,(n_i-1)} \rrbracket^{(d_i)}, \llbracket \beta'_{j,0} \rrbracket^{(d_j)}, \ldots, \llbracket \beta'_{j,(n_j-1)} \rrbracket^{(d_j)}$.

Output

Prover's output: num^* polynomials, $\llbracket \boldsymbol{\zeta} \rrbracket^{(d_i+d_j)} := (\llbracket \zeta_0 \rrbracket^{(d_i+d_j)}, \ldots, \llbracket \zeta_{\mathsf{num}^*-1} \rrbracket^{(d_i+d_j)})$ each of degree $d_i + d_j$ which can be seen as VOLE correlations, where $\mathsf{num}^* := \min(n_i n_j, \mathsf{num})$.

Computing tensor product

1 : counter $:= 0$
2 : **for** index$_j \in [n_j]$:
3 : **for** index$_i \in [n_i]$:
4 : **if** counter $\in$ [num]:
5 : $\llbracket \zeta_{\mathsf{counter}} \rrbracket^{(d_i+d_j)} :=$ P.Multiply $\left(\llbracket \beta'_{j,\mathsf{index}_j} \rrbracket^{(d_j)}, \llbracket \beta'_{i,\mathsf{index}_i} \rrbracket^{(d_i)}\right)$
6 : counter $:=$ counter $+ 1$
7 : **else** :
8 : **break**
9 : **endfor**
10 : **endfor**
11 : $\llbracket \boldsymbol{\zeta} \rrbracket^{(d_i+d_j)} := (\llbracket \zeta_0 \rrbracket^{(d_i+d_j)}, \ldots, \llbracket \zeta_{\mathsf{counter}-1} \rrbracket^{(d_i+d_j)})$
12 : **return** $\llbracket \boldsymbol{\zeta} \rrbracket^{(d_i+d_j)}$

Algorithm 4.16: P.TensorProductToElementaryVector($[\![\boldsymbol{\beta}']\!], n$)

Public information and inputs

Public information: Length of the secret elementary vector n, degree $d := \lceil \log_4(n) \rceil$.
Prover's input: $(2\ell_{\text{row}} - 6)$ secret degree-1 (linear) polynomials represented as $[\![\boldsymbol{\beta}']\!]$.

Output

Prover's output: n polynomials, $[\![\boldsymbol{z}]\!]^{(d)} := ([\![z_0]\!]^{(d)}, \ldots, [\![z_{n-1}]\!]^{(d)})$ each of degree d which can be seen as VOLE correlations corresponding to the secret elementary vector.

Compute elementary vector using tensor product

1 : **if** len($[\![\boldsymbol{\beta}']\!]$) = 12: // Security level 1
// Parse $[\![\boldsymbol{\beta}']\!]$ as $[\![\boldsymbol{\beta}']\!] := ([\![\boldsymbol{\beta}'_0]\!], [\![\boldsymbol{\beta}'_1]\!], [\![\boldsymbol{\beta}'_2]\!])$.
// Generating 16 degree-2 VOLE correlations in $\mathbb{F}_2 \times \mathbb{F}_{2^\rho}$.
2 : $[\![\boldsymbol{\zeta}_{0,1}]\!]^{(2)} :=$ P.TensorProduct $([\![\boldsymbol{\beta}'_0]\!], [\![\boldsymbol{\beta}'_1]\!], n)$
// Generating 64 degree-3 VOLE correlations in $\mathbb{F}_2 \times \mathbb{F}_{2^\rho}$.
3 : $[\![\boldsymbol{z}]\!]^{(3)} :=$ P.TensorProduct $\left([\![\boldsymbol{\zeta}_{0,1}]\!]^{(2)}, [\![\boldsymbol{\beta}'_2]\!], n\right)$
4 : **return** $[\![\boldsymbol{z}]\!]^{(3)}$
5 : **elseif** len($[\![\boldsymbol{\beta}']\!]$) = 14: // Security levels 3 and 5
// Parse $[\![\boldsymbol{\beta}']\!]$ as $[\![\boldsymbol{\beta}']\!] := ([\![\boldsymbol{\beta}'_0]\!], [\![\boldsymbol{\beta}'_1]\!], [\![\boldsymbol{\beta}'_2]\!], [\![\boldsymbol{\beta}'_3]\!])$.
// Generating 16 degree-2 VOLE correlations in $\mathbb{F}_2 \times \mathbb{F}_{2^\rho}$.
6 : $[\![\boldsymbol{\zeta}_{0,1}]\!]^{(2)} :=$ P.TensorProduct $([\![\boldsymbol{\beta}'_0]\!], [\![\boldsymbol{\beta}'_1]\!], n)$
// Generating 64 degree-3 VOLE correlations in $\mathbb{F}_2 \times \mathbb{F}_{2^\rho}$.
7 : $[\![\boldsymbol{\zeta}_{0,1,2}]\!]^{(3)} :=$ P.TensorProduct $\left([\![\boldsymbol{\zeta}_{0,1}]\!]^{(2)}, [\![\boldsymbol{\beta}'_2]\!], n\right)$
8 : // Generating n degree-4 VOLE correlations in $\mathbb{F}_2 \times \mathbb{F}_{2^\rho}$.
9 : $[\![\boldsymbol{z}]\!]^{(4)} :=$ P.TensorProduct $\left([\![\boldsymbol{\zeta}_{0,1,2}]\!]^{(3)}, [\![\boldsymbol{\beta}'_3]\!], n\right)$
10 : **return** $[\![\boldsymbol{z}]\!]^{(4)}$
11 : **else** :
12 : **return** $\perp$
13 : **endif**

Algorithm 4.17: P.VOLE-ElementaryVector$\left(\mathsf{pos}, (\llbracket u_k \rrbracket)_{k\in[\ell_{\mathsf{row}}]}\right)$

Public information and inputs

Public information: Length of the secret elementary vector n, degree $d := \lceil \log_4 n \rceil$, size of the compressed secret witness $\ell_{\mathsf{row}} := d + 6$.
Prover's input: Secret position $\mathsf{pos} \in [n]$, ℓ_{row} VOLE correlations $\llbracket u_k \rrbracket$ for random $(u_k, v_k) \in \mathbb{F}_2 \times \mathbb{F}_{2^\rho}$ represented as polynomials $f_{u_k}(X) = u_k X + v_k$ for $k \in [\ell_{\mathsf{row}}]$.

Output

Prover's output: Masked compressed secret $\boldsymbol{t} \in \mathbb{F}_2^{\ell_{\mathsf{row}}}$, n polynomials $\llbracket \boldsymbol{z} \rrbracket^{(d)} := (\llbracket z_0 \rrbracket^{(d)}, \ldots, \llbracket z_{n-1} \rrbracket^{(d)})$ each of degree d which can be seen as VOLE correlations corresponding to the secret elementary vector, along with $(2\ell_{\mathsf{row}} - 6)$ secret degree-1 (linear) polynomials represented as $\llbracket \boldsymbol{\beta}' \rrbracket$.

Compute masked secret

$\boldsymbol{w}' := \mathsf{PosToWitness}(\mathsf{pos})$

$\boldsymbol{w} := \mathsf{CompWit}(\boldsymbol{w}')$

$\boldsymbol{t} := \boldsymbol{w} \oplus \boldsymbol{u}$

Construct VOLE correlations with witness

$\llbracket \boldsymbol{\beta}' \rrbracket := \mathsf{P.EmbedWitness}\left(\boldsymbol{w}', (\llbracket u_k \rrbracket)_{k\in[\ell_{\mathsf{row}}]}\right)$

Compute elementary vector using tensor product

$\llbracket \boldsymbol{z} \rrbracket^{(d)} := \mathsf{P.TensorProductToElementaryVector}\left(\llbracket \boldsymbol{\beta}' \rrbracket, n\right)$

// Note that $\llbracket \boldsymbol{z} \rrbracket^{(d)} := (\llbracket z_0 \rrbracket^{(d)}, \ldots, \llbracket z_{n-1} \rrbracket^{(d)})$.

return $\left(\boldsymbol{t}, \llbracket \boldsymbol{\beta}' \rrbracket, \llbracket \boldsymbol{z} \rrbracket^{(d)}\right)$

Algorithm 4.18: P.VOLE-Permutation$\big(\boldsymbol{P}, (\llbracket u_{i,k} \rrbracket)_{i\in[n],k\in[\ell_{\mathsf{row}}]}\big)$

Public information and inputs

Public information: Matrix dimension n of the secret permutation matrix, degree $d := \lceil \log_4 n \rceil$, size of the compressed secret witness $\ell_{\mathsf{row}} := d + 6$.
Prover's input: Secret permutation matrix $\boldsymbol{P}$ represented as n positions $(\mathsf{pos}_0, \ldots, \mathsf{pos}_{n-1})$, $n \cdot \ell_{\mathsf{row}}$ VOLE correlations $\llbracket u_{i,k} \rrbracket$ for random $(u_{i,k}, v_{i,k}) \in \mathbb{F}_2 \times \mathbb{F}_{2^\rho}$ represented as polynomials $f_{u_{i,k}}(X) = u_{i,k}X + v_{i,k}$ for $(i,k) \in [n] \times [\ell_{\mathsf{row}}]$.

Output

Prover's output: Masked compressed secret $\boldsymbol{t} \in \mathbb{F}_2^\ell$, where $\ell := n \cdot \ell_{\mathsf{row}}$. Also, n^2 polynomials $\llbracket \boldsymbol{z} \rrbracket^{(d)} := (\llbracket z_{0,0} \rrbracket^{(d)}, \ldots, \llbracket z_{(n-1),(n-1)} \rrbracket^{(d)})$ each of degree d which can be seen as VOLE correlations corresponding to the individual entries of the secret permutation matrix $\boldsymbol{P}$. Along with $n(2\ell_{\mathsf{row}} - 6)$ secret degree-1 (linear) polynomials represented as $\llbracket \boldsymbol{\beta} \rrbracket$, and n polynomials $\llbracket \mathsf{ColCheck} \rrbracket^{(d)} := (\llbracket \mathsf{ColCheck}_0 \rrbracket^{(d)}, \ldots, \llbracket \mathsf{ColCheck}_{n-1} \rrbracket^{(d)})$ each of degree d which can be seen as VOLE correlations corresponding to the sums of the individual columns of the secret permutation matrix $\boldsymbol{P}$.

Compute $\boldsymbol{P}$ row-wise as n elementary vectors

1 : **for** $i \in [n]$:
2 : $\quad (\boldsymbol{t}_i, \llbracket \boldsymbol{\beta}_i \rrbracket, \llbracket \boldsymbol{z}_i \rrbracket^{(d)}) := \text{P.VOLE-ElementaryVector}\big(\mathsf{pos}_i, (\llbracket u_{i,k} \rrbracket)_{k\in[\ell_{\mathsf{row}}]}\big)$
3 : **endfor**

Compute $\boldsymbol{P}$ columns check

4 : **for** $j \in [n]$:
5 : $\quad \llbracket \mathsf{ColSum}_j \rrbracket^{(d)} := \sum_{i=0}^{n-1} \llbracket z_{i,j} \rrbracket^{(d)}$
6 : $\quad \llbracket \mathsf{ColCheck}_j \rrbracket^{(d)} := \llbracket \mathsf{ColSum}_j \rrbracket^{(d)} - X^d$
7 : **endfor**

8 : $\boldsymbol{t} := (\boldsymbol{t}_0, \cdots, \boldsymbol{t}_{n-1})$
9 : $\llbracket \boldsymbol{\beta} \rrbracket := (\llbracket \boldsymbol{\beta}_0 \rrbracket, \cdots, \llbracket \boldsymbol{\beta}_{n-1} \rrbracket)$
10 : $\llbracket \boldsymbol{z} \rrbracket^{(d)} := (\llbracket \boldsymbol{z}_0 \rrbracket^{(d)}, \ldots, \llbracket \boldsymbol{z}_{n-1} \rrbracket^{(d)})$
11 : $\llbracket \mathsf{ColCheck} \rrbracket^{(d)} := (\llbracket \mathsf{ColCheck}_0 \rrbracket^{(d)}, \ldots, \llbracket \mathsf{ColCheck}_{n-1} \rrbracket^{(d)})$
12 : **return** $(\boldsymbol{t}, \llbracket \boldsymbol{\beta} \rrbracket, \llbracket \boldsymbol{z} \rrbracket^{(d)}, \llbracket \mathsf{ColCheck} \rrbracket^{(d)})$

Algorithm 4.19: P.Check-ElementaryBlock($[\![\boldsymbol{\beta}'_i]\!]$)

Public information and inputs

Public information: Size $n' \in \{2, 4\}$of a secret block of VOLE correlations $[\![\boldsymbol{\beta}'_i]\!]$.
Prover's input: A block of secret VOLE correlations $[\![\boldsymbol{\beta}'_i]\!]$, where each element of $[\![\boldsymbol{\beta}'_i]\!]$ is represented as degree-1 (linear) polynomial with coefficients in $\mathbb{F}_{2^\rho}$.

Output

Prover's output: $\frac{n'}{2}$ quadratic polynomials, $[\![\boldsymbol{e}'_i]\!]^{(2)} \coloneqq \left([\![e'_{i,0,1}]\!]^{(2)}, [\![e'_{i,2,3}]\!]^{(2)}\right)$ (or $[\![e'_{i,0,1}]\!]^{(2)}$) which can be seen as VOLE correlations.

---

1 : **if** len($[\![\boldsymbol{\beta}'_i]\!]$) $= 4$:
  // Parse $[\![\boldsymbol{\beta}'_i]\!]$ as $[\![\boldsymbol{\beta}'_i]\!] \coloneqq \left([\![\beta'_{i,0}]\!], [\![\beta'_{i,1}]\!], [\![\beta'_{i,2}]\!], [\![\beta'_{i,3}]\!]\right)$.
2 : $[\![e'_{i,0,1}]\!]^{(2)} \coloneqq$ P.Multiply $\left([\![\beta'_{i,0}]\!], [\![\beta'_{i,1}]\!]\right)$
3 : $[\![e'_{i,2,3}]\!]^{(2)} \coloneqq$ P.Multiply $\left([\![\beta'_{i,2}]\!], [\![\beta'_{i,3}]\!]\right)$
4 : $[\![\boldsymbol{e}'_i]\!]^{(2)} \coloneqq \left([\![e'_{i,0,1}]\!]^{(2)}, [\![e'_{i,2,3}]\!]^{(2)}\right)$
5 : **return** $[\![\boldsymbol{e}'_i]\!]^{(2)}$
6 : **elseif** len($[\![\boldsymbol{\beta}'_i]\!]$) $= 2$:
  // Parse $[\![\boldsymbol{\beta}'_i]\!]$ as $[\![\boldsymbol{\beta}'_i]\!] \coloneqq \left([\![\beta'_{i,0}]\!], [\![\beta'_{i,1}]\!]\right)$.
7 : $[\![e'_{i,0,1}]\!]^{(2)} \coloneqq$ P.Multiply $\left([\![\beta'_{i,0}]\!], [\![\beta'_{i,1}]\!]\right)$
8 : **return** $[\![e'_{i,0,1}]\!]^{(2)}$
9 : **else** :
10 : **return** $\perp$
11 : **endif**

Algorithm 4.20: P.Check-ElementaryVector($[\![\boldsymbol{\beta}']\!]$)

Public information and inputs

Public information: Length of the secret elementary vector n, degree $d := \lceil \log_4(n) \rceil$, $\ell_{\text{row}} := d + 6$.

Prover's input: $(2\ell_{\text{row}} - 6)$ secret degree-1 (linear) polynomials represented as $[\![\boldsymbol{\beta}']\!]$.

Output

Prover's output: $(d+3)$ degree-d polynomials, $[\![\boldsymbol{e}']\!]^{(d)}$ which can be seen as VOLE correlations.

---

if len($[\![\boldsymbol{\beta}']\!]$) $= 12$: // Security level 1
  // Parse $[\![\boldsymbol{\beta}']\!]$ as $[\![\boldsymbol{\beta}']\!] := ([\![\boldsymbol{\beta}'_0]\!], [\![\boldsymbol{\beta}'_1]\!], [\![\boldsymbol{\beta}'_2]\!])$.
  for $i \in [3]$:
    $[\![\boldsymbol{e}'_i]\!]^{(2)} :=$ P.Check-ElementaryBlock $([\![\boldsymbol{\beta}'_i]\!])$
    $[\![\boldsymbol{e}'_i]\!]^{(3)} := X \cdot [\![\boldsymbol{e}'_i]\!]^{(2)}$
  endfor
  $[\![\boldsymbol{e}']\!]^{(3)} := \left([\![\boldsymbol{e}'_0]\!]^{(3)}, [\![\boldsymbol{e}'_1]\!]^{(3)}, [\![\boldsymbol{e}'_2]\!]^{(3)}\right)$
  return $[\![\boldsymbol{e}']\!]^{(3)}$
elseif len($[\![\boldsymbol{\beta}']\!]$) $= 14$: // Security levels 3 and 5
  // Parse $[\![\boldsymbol{\beta}']\!]$ as $[\![\boldsymbol{\beta}']\!] := ([\![\boldsymbol{\beta}'_0]\!], [\![\boldsymbol{\beta}'_1]\!], [\![\boldsymbol{\beta}'_2]\!], [\![\boldsymbol{\beta}'_3]\!])$.
  $\left([\![\boldsymbol{e}'_0]\!]^{(3)}, [\![\boldsymbol{e}'_1]\!]^{(3)}, [\![\boldsymbol{e}'_2]\!]^{(3)}\right) :=$ P.Check-ElementaryVector $([\![\boldsymbol{\beta}'_0]\!], [\![\boldsymbol{\beta}'_1]\!], [\![\boldsymbol{\beta}'_2]\!])$
  for $i \in [3]$:
    $[\![\boldsymbol{e}'_i]\!]^{(4)} := X \cdot [\![\boldsymbol{e}'_i]\!]^{(3)}$
  endfor
  $[\![e'_{3,0,1}]\!]^{(2)} :=$ P.Check-ElementaryBlock $([\![\boldsymbol{\beta}'_3]\!])$
  $[\![e'_{3,0,1}]\!]^{(4)} := X^2 \cdot [\![e'_{3,0,1}]\!]^{(2)}$
  $[\![\boldsymbol{e}']\!]^{(4)} := \left([\![\boldsymbol{e}'_0]\!]^{(4)}, [\![\boldsymbol{e}'_1]\!]^{(4)}, [\![\boldsymbol{e}'_2]\!]^{(4)}, [\![e'_{3,0,1}]\!]^{(4)}\right)$
  return $[\![\boldsymbol{e}']\!]^{(4)}$
else :
  return $\perp$
endif

Algorithm 4.21: $\mathsf{P.CheckZero}\big(\llbracket w \rrbracket^{(d)}, (\llbracket u_{i,k} \rrbracket)_{(i,k)\in[d-1]\times[\rho]}\big)$

Public information and inputs

Public information: Degree of input VOLE correlation $\llbracket w \rrbracket^{(d)}$ (seen as polynomial) d.
Prover's input: Degree-d VOLE correlation $\llbracket w \rrbracket^{(d)}$, $(d-1)\rho$ random VOLE correlation represented as $f_{u_{i,k}}(X) = u_{i,k}X + v_{i,k}$ where $(u_{i,k}, v_{i,k}) \in \mathbb{F}_2 \times \mathbb{F}_{2^\rho}$ for $(i,k) \in [d-1] \times [\rho]$.

Output

Prover's output: Polynomial $\llbracket a \rrbracket$. Note that for an honest prover, the leading coefficient (coefficient of X^d term will be equal to 0) and therefore $\llbracket a \rrbracket$ will consists of only d coefficients for terms X^i for $i \in [d]$.

// Generating VOLE correlations in $\mathbb{F}_{2^\rho} \times \mathbb{F}_{2^\rho}$

for $i \in [d-1]$:
  $u'_i \coloneqq \sum_{k=0}^{\rho-1} u_{i,k}\gamma_\rho^k$ // $\left\{\gamma_\rho^k\right\}_{k=0}^{\rho-1}$ is the power basis of $\mathbb{F}_{2^\rho}$ with coefficients in $\mathbb{F}_2$.
  $v'_i \coloneqq \sum_{k=0}^{\rho-1} v_{i,k}\gamma_\rho^k$
  $f_{u'_i}(X) \coloneqq u'_i X + v'_i$ // (u'_i, v'_i) in $\mathbb{F}_{2^\rho} \times \mathbb{F}_{2^\rho}$.
endfor
$f_{\mathsf{mask}}(X) \coloneqq \sum_{i=0}^{d-2} f_{u'_i}(X) \cdot X^i$
$\llbracket a \rrbracket \coloneqq f_w(X) + f_{\mathsf{mask}}(X)$
return $\llbracket a \rrbracket$

Algorithm 4.22:
P.Check-PKP$\left(\boldsymbol{P}, \mathsf{pk}, (\llbracket u_{i,k} \rrbracket)_{(i,k)\in[n]\times[\ell_{\mathsf{row}}]}, (\llbracket u_{i',k'} \rrbracket)_{(i',k')\in[d-1]\times[\rho]}, \mathsf{seed}\right)$

Public information

Public information: Matrix dimension n of the secret permutation matrix, degree $d := \lceil \log_4 n \rceil$, size of the compressed secret witness $\ell_{\mathsf{row}} := d + 6$.

Prover's input: Secret permutation matrix $\boldsymbol{P}$ represented as n positions $(\mathsf{pos}_0, \ldots, \mathsf{pos}_{n-1})$, public key $\mathsf{pk} = (\boldsymbol{H}, \boldsymbol{x})$, $n \cdot \ell_{\mathsf{row}}$ VOLE correlations $\llbracket u_{i,k} \rrbracket$ for random $(u_{i,k}, v_{i,k}) \in \mathbb{F}_2 \times \mathbb{F}_{2^\rho}$ represented as polynomials $f_{u_{i,k}}(X) = u_{i,k}X + v_{i,k}$ for $(i,k) \in [n] \times [\ell_{\mathsf{row}}]$, $(d-1)\rho$ VOLE correlations $\llbracket u_{i',k'} \rrbracket$ for random $u_{i',k'}, v_{i',k'} \in \mathbb{F}_2 \times \mathbb{F}_{2^\rho}$ represented as $f_{u_{i',k'}}(X) = u_{i',k'}X + v_{i',k'}$ for $(i',k') \in [d-1] \times [\rho]$, $\mathsf{seed} \in \{0,1\}^{2\lambda}$.

Output

Prover's output: Degree-d polynomial $\llbracket a \rrbracket$ as a proof to show that $\boldsymbol{P}$ is a solution to the PKP instance defined by pk.

Compute $\boldsymbol{P}$ in matrix form

$(\boldsymbol{t}, \llbracket \boldsymbol{\beta} \rrbracket, \llbracket \boldsymbol{z} \rrbracket^{(d)}, \llbracket \mathsf{ColCheck} \rrbracket^{(d)}) := \mathsf{P.VOLE\text{-}Permutation}\left(\boldsymbol{P}, (\llbracket u_{i,k} \rrbracket)_{(i,k)\in[n]\times[\ell_{\mathsf{row}}]}\right)$

// Parse $\llbracket \mathsf{ColCheck} \rrbracket^{(d)}$ as $(f_0(X), f_1(X), \ldots, f_{n-1}(X))$

Check elementary vectors

for $i \in [n]$

$\llbracket \mathsf{ElemVecCheck}_i \rrbracket^{(d)} := \mathsf{P.Check\text{-}ElementaryVector}\left(\llbracket \boldsymbol{\beta}_i \rrbracket\right)$

// Each $\llbracket \mathsf{ElemVecCheck}_i \rrbracket^{(d)}$ contains 6 degree-d polynomials if $\lambda = 128$,

// and 7 degree-d polynomials if $\lambda \in \{192, 256\}$.

endfor

$\llbracket \mathsf{ElemVecCheck} \rrbracket^{(d)} := \left(\llbracket \mathsf{ElemVecCheck}_0 \rrbracket^{(d)}, \ldots, \llbracket \mathsf{ElemVecCheck}_{n-1} \rrbracket^{(d)}\right)$

// Parse $\llbracket \mathsf{ElemVecCheck} \rrbracket^{(d)}$ as $(f_n(X), f_{n+1}(X), \ldots, f_{cn+n-1}(X))$,

// where, $c = 6$ if $\lambda = 128$, and $c = 7$ if $\lambda \in \{192, 256\}$.

Compute $\boldsymbol{x}' = \boldsymbol{P}\boldsymbol{x}$

for $i \in [n]$

$\llbracket \boldsymbol{x}'_i \rrbracket^{(d)} := \sum_{j=0}^{n-1} \llbracket z_{i,j} \rrbracket^{(d)} \cdot \boldsymbol{x}_j$

endfor

Compute $\boldsymbol{y} = \boldsymbol{H}\boldsymbol{x}'$

for $i \in [m]$

$\llbracket \boldsymbol{y}_i \rrbracket^{(d)} = f_{i+cn+n}(X) := \sum_{j=0}^{n-1} h_{i,j} \cdot \llbracket \boldsymbol{x}'_j \rrbracket^{(d)}$

endfor

Merge polynomials and run CheckZero

$\boldsymbol{\alpha} := \mathsf{H}_4(\mathsf{seed} :: \rho \cdot (cn + n + m))$ // $\boldsymbol{\alpha}$ should be parsed as $\in \mathbb{F}_{2^\rho}^{cn+n+m}$

$f(X) := \sum_{j=0}^{cn+n+m-1} \alpha_j \cdot f_j(X)$

$\llbracket a \rrbracket := \mathsf{P.CheckZero}\left(f(X), (\llbracket u_{i',k'} \rrbracket)_{(i',k')\in[d-1]\times[\rho]}\right)$

$\mathsf{proof} := \llbracket a \rrbracket$

return proof

Verifier. In this section we present all the algorithms that will be used by the verifier to verify that the prover has knowledge of the secret permutation which serves as a solution to the PKP instance corresponding to the public key. As expected in VOLE-in-the-Head (or MPC-in-the-Head) type PoK, the verifier's algorithm bear a close resemblance to those used by the prover. In the case of PERK, the main difference is that while prover's algorithms explained in Section 4.5 take polynomials as inputs and manipulate them, the verifier's algorithms described in this section perform analogous manipulations on evaluations of corresponding polynomials. The verifier possesses the VOLE correlation inputs $\boldsymbol{q}$ and Δ, and it also receives the masked (compressed) witness $\boldsymbol{t}$ and masked polynomial $[\![a]\!]$ from the prover.

The first step the verifier performs is to update the VOLE correlation inputs $\boldsymbol{q}$ with the help of the masked witness $\boldsymbol{t}$, to ensure that they satisfy the VOLE correlation with respect to witness $\boldsymbol{w}$ (instead of $\boldsymbol{u}$). This is achieved by Algorithm 4.23 EmbedMaskedWitnessBlock. The output of this algorithm are the VOLE correlations $\boldsymbol{q}'_{\boldsymbol{\beta}'_i}$ corresponding to the elementary blocks of witness. Similar to the prover's case, the Algorithm 4.24 EmbedMaskedWitness aggregates the VOLE correlations corresponding to the individual elementary vectors of lengths 4 and 2, and outputs VOLE correlations $\boldsymbol{q}'_{\boldsymbol{\beta}'}$ corresponding to the aggregation of 3 elementary vectors of lengths 4 (and in case of L3 and L5 parameters, another elementary vector of length 2). Once the verifier possesses the VOLE correlation inputs ($\boldsymbol{q}'_{\boldsymbol{\beta}'}$) corresponding to the elementary vector entries, it then computes VOLE correlations $\boldsymbol{q}_{\boldsymbol{z}}$ corresponding to each individual row of the secret permutation matrix with help of Algorithm 4.26 TensorProductToElementaryVector, which internally calls Algorithm 4.25 TensorProduct to compute the tensor product between two blocks. The Algorithm 4.27 VOLE-ElementaryVector produces the VOLE correlations $\boldsymbol{q}'_{\boldsymbol{\beta}'}$ along with $\boldsymbol{q}_{\boldsymbol{z}}$.

After obtaining the VOLE correlations for each of the n rows by running Algorithm 4.27 VOLE-ElementaryVector n times, the verifier proceeds to compute the extra n VOLE correlation values, $\boldsymbol{q}_{\mathsf{ColCheck}}$ which ensure that each column of secret matrix adds upto exactly 1. This is described in Algorithm 4.28 VOLE-Permutation. The verifier checks the elementary structure by computing the values $\boldsymbol{q}'_{\boldsymbol{e}'}$ using Algorithm 4.29 Check-ElementaryBlock, and Algorithm 4.30 Check-ElementaryVector.

In order to check that the leading coefficient of the masked polynomial $[\![a]\!]$ is zero, the verifier first checks the degree of the polynomial is equal to $d-1$, it then evaluates the masking polynomial using its inputs $\boldsymbol{q}, \Delta$ and then finally checks if the evaluation of the polynomial $[\![a]\!]$ sent by the prover at point Δ matches the addition of the $\boldsymbol{q}$ value obtained from the computations checking constraints related to the PKP problem and the evaluation of masking polynomial. The Algorithm 4.31 CheckZero achieves this and outputs 1 when all the values match and outputs 0 otherwise indicating the failure to verify the prover's claim.

Finally, as in the prover's case, the Algorithm 4.32 Check-PKP puts all of the checks for checking the elementary structure of rows (blocks), column sums equaling to 1, and satisfiability of PKP equation together by evaluating degree-d polynomials at Δ. These evaluations are then merged together into a single value by computing (verifier dictated) random linear combination, which is then checked with the help of Algorithm 4.31 CheckZero.

Algorithm 4.23: V.EmbedMaskedWitnessBlock$\left(\Delta, \boldsymbol{t}'_i, (q_k)_{k\in[3]}\right)$

Public information and inputs

Public information: Length of the masked witness block $= 4$.
Verifier's input: VOLE correlation challenge $\Delta \in \mathbb{F}_{2^\rho}$, i^{th} block $\boldsymbol{t}'_i := t'_{i,0}||t'_{i,1}||t'_{i,2}||t'_{i,3} \in \mathbb{F}_2^4$ of the masked witness $\boldsymbol{t}'$, 3 VOLE correlation inputs (q_0, q_1, q_2) with each of them in $\mathbb{F}_{2^\rho}$.

Output

Verifier's output: VOLE correlation values $\boldsymbol{q}'_{\boldsymbol{\beta}'_i} := \left(q'_{\beta'_{i,0}}, q'_{\beta'_{i,1}}, q'_{\beta'_{i,2}}, q'_{\beta'_{i,3}}\right) \in \mathbb{F}_{2^\rho}^4$

Construct VOLE correlations with witness

// Parse $\boldsymbol{t}'_i$ as $\boldsymbol{t}'_i := t'_{i,0}||t'_{i,1}||t'_{i,2}||t'_{i,3}$.
for $j \in [3]$:
  $q'_{\beta'_{i,j}} := t'_{i,j} \cdot \Delta + q_j$
endfor
$q'_{\beta'_{i,3}} := t'_{i,3} \cdot \Delta + q_0 + q_1 + q_2$
$\boldsymbol{q}'_{\boldsymbol{\beta}'_i} := \left(q'_{\beta'_{i,0}}, q'_{\beta'_{i,1}}, q'_{\beta'_{i,2}}, q'_{\beta'_{i,3}}\right)$
return $\boldsymbol{q}'_{\boldsymbol{\beta}'_i}$

Algorithm 4.24: V.EmbedMaskedWitness$(\Delta, \boldsymbol{t}', \boldsymbol{q})$

Public information and inputs

Public information: Length of the compressed masked witness ℓ_{row}, and length of the masked witness $|\boldsymbol{t}'| \coloneqq 2 \cdot \ell_{\text{row}} - 6$.

Verifier's input: VOLE correlation challenge $\Delta \in \mathbb{F}_{2^\rho}$, the masked witness $\boldsymbol{t}'$, ℓ_{row} VOLE correlation inputs $\boldsymbol{q} \coloneqq (q_0, \ldots, q_{\ell_{\text{row}}-1})$ with each of them in $\mathbb{F}_{2^\rho}$.

Output

Verifier's output: $\boldsymbol{q}'_{\boldsymbol{\beta}'}$ with elements in $\mathbb{F}_{2^\rho}$.

Construct VOLE correlations with witness

1 : **if** $\mathsf{len}(\boldsymbol{t}') = 12$: // Security level 1

// Parse $\boldsymbol{t}'$ as $\boldsymbol{t}' \coloneqq \boldsymbol{t}'_0||\boldsymbol{t}'_1||\boldsymbol{t}'_2$, where, $\boldsymbol{t}'_i \coloneqq t'_{i,0}||t'_{i,1}||t'_{i,2}||t'_{i,3}$.

2 : **for** $i \in [3]$:

3 : $\boldsymbol{q}'_{\boldsymbol{\beta}'_i} \coloneqq \mathsf{V.EmbedMaskedWitnessBlock}\left(\Delta, \boldsymbol{t}'_i, \boldsymbol{q}[3i : 3i+2]\right)$

4 : **endfor**

5 : $\boldsymbol{q}'_{\boldsymbol{\beta}'} \coloneqq \left(\boldsymbol{q}'_{\boldsymbol{\beta}'_0}, \boldsymbol{q}'_{\boldsymbol{\beta}'_1}, \boldsymbol{q}'_{\boldsymbol{\beta}'_2}\right)$

6 : **return** $\boldsymbol{q}'_{\boldsymbol{\beta}'}$

7 : **elseif** $\mathsf{len}(\boldsymbol{t}') = 14$: // Security levels 3 and 5

// Parse $\boldsymbol{t}'$ as $\boldsymbol{t}' \coloneqq \boldsymbol{t}'_0||\boldsymbol{t}'_1||\boldsymbol{t}'_2||t'_{3,0}||t'_{3,1}$, where, $\boldsymbol{t}'_i$ are as above (line 1)

8 : $\left(\boldsymbol{q}'_{\boldsymbol{\beta}'_0}, \boldsymbol{q}'_{\boldsymbol{\beta}'_1}, \boldsymbol{q}'_{\boldsymbol{\beta}'_2}\right) \coloneqq \mathsf{V.EmbedMaskedWitness}\left(\Delta, \boldsymbol{t}'_0||\boldsymbol{t}'_1||\boldsymbol{t}'_2, \boldsymbol{q}[0:8]\right)$

9 : $q'_{\beta'_{3,0}} \coloneqq t'_{3,0} \cdot \Delta + q_9$

10 : $q'_{\beta'_{3,1}} \coloneqq t'_{3,1} \cdot \Delta + q_9$

11 : $\boldsymbol{q}'_{\boldsymbol{\beta}'_3} \coloneqq \left(q'_{\beta'_{3,0}}, q'_{\beta'_{3,1}}\right)$

12 : $\boldsymbol{q}'_{\boldsymbol{\beta}'} \coloneqq \left(\boldsymbol{q}'_{\boldsymbol{\beta}'_0}, \boldsymbol{q}'_{\boldsymbol{\beta}'_1}, \boldsymbol{q}'_{\boldsymbol{\beta}'_2}, \boldsymbol{q}'_{\boldsymbol{\beta}'_3}\right)$

13 : **return** $\boldsymbol{q}'_{\boldsymbol{\beta}'}$

14 : **else** :

15 : **return** $\perp$

16 : **endif**

Algorithm 4.25: V.TensorProduct$\left(\boldsymbol{q}'_{\boldsymbol{\beta}'_i}, \boldsymbol{q}'_{\boldsymbol{\beta}'_j}, \mathsf{num}\right)$

Public information and inputs

Public information: Sizes n_i and n_j of the two blocks of verifier's VOLE correlation inputs $\boldsymbol{q}'_{\boldsymbol{\beta}'_i}$ and $\boldsymbol{q}'_{\boldsymbol{\beta}'_j}$ respectively. Each element of $\boldsymbol{q}'_{\boldsymbol{\beta}'_i}$ (resp. $\boldsymbol{q}'_{\boldsymbol{\beta}'_j}$) is in $\mathbb{F}_{2^\rho}$.

Verifier's input: $(n_i + n_j)$ values $\boldsymbol{q}'_{\boldsymbol{\beta}'_i} := \left(q'_{\beta'_{i,0}}, \ldots, q'_{\beta'_{i,(n_i-1)}}\right)$ and $\boldsymbol{q}'_{\boldsymbol{\beta}'_j} := \left(q'_{\beta'_{j,0}}, \ldots, q'_{\beta'_{j,(n_j-1)}}\right)$.

Output

Verifier's output: $\boldsymbol{q}_{\boldsymbol{\zeta}} := \left(q_{\zeta_0}, \ldots, q_{\zeta_{\mathsf{num}^*-1}}\right)$ is a block of num^* values in $\mathbb{F}_{2^\rho}$, where $\mathsf{num}^* := \min(n_i n_j, \mathsf{num})$.

Computing tensor product

1 : counter := 0
2 : **for** index$_j \in [n_j]$:
3 : **for** index$_i \in [n_i]$:
4 : **if** counter $\in$ [num]:
5 : $q_{\zeta_{\mathsf{counter}}} := q'_{\beta'_{j,\mathsf{index}_j}} \cdot q'_{\beta'_{i,\mathsf{index}_i}}$
6 : counter := counter + 1
7 : **else** :
8 : **break**
9 : **endfor**
10 : **endfor**
11 : $\boldsymbol{q}_{\boldsymbol{\zeta}} := \left(q_{\zeta_0}, \ldots, q_{\zeta_{\mathsf{num}^*-1}}\right)$
12 : **return** $\boldsymbol{q}_{\boldsymbol{\zeta}}$

Algorithm 4.26: V.TensorProductToElementaryVector$\left(\boldsymbol{q}'_{\boldsymbol{\beta}'}, \mathsf{num}\right)$

Public information and inputs

Public information: Length of the input vector $\mathsf{num} \in [128]$. Lengths of the blocks $|\boldsymbol{q}'_{\boldsymbol{\beta}'_i}| := 4$ for $i \in [3]$, and $|\boldsymbol{q}'_{\boldsymbol{\beta}'_3}| := 2$
Verifier's input: VOLE correlation input $\boldsymbol{q}'_{\boldsymbol{\beta}'}$ with elements in $\mathbb{F}_{2^\rho}$.

Output

Verifier's output: $\boldsymbol{q}_{\boldsymbol{z}}$ a block of values in $\mathbb{F}_{2^\rho}$.

Compute elementary vector using tensor product

if $\mathsf{len}(\boldsymbol{q}'_{\boldsymbol{\beta}'}) = 12$: // Security level 1
  // Parse $\boldsymbol{q}'_{\boldsymbol{\beta}'}$ as $\boldsymbol{q}'_{\boldsymbol{\beta}'} := \left(\boldsymbol{q}'_{\boldsymbol{\beta}'_0}, \boldsymbol{q}'_{\boldsymbol{\beta}'_1}, \boldsymbol{q}'_{\boldsymbol{\beta}'_2}\right)$.
  $\boldsymbol{q}_{\boldsymbol{\zeta}_{0,1}} := \mathsf{V.TensorProduct}\left(\boldsymbol{q}'_{\boldsymbol{\beta}'_0}, \boldsymbol{q}'_{\boldsymbol{\beta}'_1}, \mathsf{num}\right)$
  $\boldsymbol{q}_{\boldsymbol{z}} := \mathsf{V.TensorProduct}\left(\boldsymbol{q}_{\boldsymbol{\zeta}_{0,1}}, \boldsymbol{q}'_{\boldsymbol{\beta}'_2}, \mathsf{num}\right)$
  return $\boldsymbol{q}_{\boldsymbol{z}}$
elseif $\mathsf{len}(\boldsymbol{q}'_{\boldsymbol{\beta}'}) = 14$: // Security levels 3 and 5
  // Parse $\boldsymbol{q}'_{\boldsymbol{\beta}'}$ as $\boldsymbol{q}'_{\boldsymbol{\beta}'} := \left(\boldsymbol{q}'_{\boldsymbol{\beta}'_0}, \boldsymbol{q}'_{\boldsymbol{\beta}'_1}, \boldsymbol{q}'_{\boldsymbol{\beta}'_2}, \boldsymbol{q}'_{\boldsymbol{\beta}'_3}\right)$.
  $\boldsymbol{q}_{\boldsymbol{\zeta}_{0,1}} := \mathsf{V.TensorProduct}\left(\boldsymbol{q}'_{\boldsymbol{\beta}'_0}, \boldsymbol{q}'_{\boldsymbol{\beta}'_1}, \mathsf{num}\right)$
  $\boldsymbol{q}_{\boldsymbol{\zeta}_{0,1,2}} := \mathsf{V.TensorProduct}\left(\boldsymbol{q}_{\boldsymbol{\zeta}_{0,1}}, \boldsymbol{q}'_{\boldsymbol{\beta}'_2}, \mathsf{num}\right)$
  $\boldsymbol{q}_{\boldsymbol{z}} := \mathsf{V.TensorProduct}\left(\boldsymbol{q}_{\boldsymbol{\zeta}_{0,1,2}}, \boldsymbol{q}'_{\boldsymbol{\beta}'_3}, \mathsf{num}\right)$
  return $\boldsymbol{q}_{\boldsymbol{z}}$
else :
  return $\perp$
endif

Algorithm 4.27: V.VOLE-ElementaryVector($\Delta, \boldsymbol{t}, \boldsymbol{q}$)

Public information and inputs

Public information: Length of the elementary vector n, length of the compressed masked witness $|\boldsymbol{t}| \coloneqq \ell_{\mathsf{row}}$.
Verifier's input: VOLE correlation challenge $\Delta \in \mathbb{F}_{2^\rho}$, the compressed masked witness $\boldsymbol{t}$, ℓ_{row} VOLE correlation inputs $\boldsymbol{q} \coloneqq (q_0, \ldots, q_{\ell_{\mathsf{row}}-1})$ with each of them in $\mathbb{F}_{2^\rho}$.

Output

Verifier's output: Tuple of $(2 \cdot \ell_{\mathsf{row}} - 6)$ VOLE correlation values corresponding to the shares $\boldsymbol{\beta}'$ held by the prover, along with another tuple of n VOLE correlation values $\boldsymbol{q_z} \coloneqq \left(q_{z_0}, q_{z_1}, \ldots q_{z_{n-1}}\right)$ corresponding to the secret elementary vector of length n held by the prover. All VOLE correlation values output are in $\mathbb{F}_{2^\rho}$.

Compute masked secret

1 : $\boldsymbol{t}' \coloneqq \mathsf{ExpWit}(\boldsymbol{t})$

2 : $\boldsymbol{q}'_{\boldsymbol{\beta}'} \coloneqq \mathsf{V.EmbedMaskedWitness}\left(\Delta, \boldsymbol{t}', \boldsymbol{q}\right)$

Compute elementary vector using tensor product

3 : $\boldsymbol{q_z} \coloneqq \mathsf{V.TensorProductToElementaryVector}\left(\boldsymbol{q}'_{\boldsymbol{\beta}'}, n\right)$

4 : **return** $\left(\boldsymbol{q}'_{\boldsymbol{\beta}'}, \boldsymbol{q_z}\right)$

Algorithm 4.28: V.VOLE-Permutation($\Delta, \boldsymbol{t}, \boldsymbol{q}$)

Public information and inputs

Public information: Matrix dimension n of the secret permutation matrix, length of the compressed masked witness $|\boldsymbol{t}| \coloneqq \ell$. Note that $\ell = n\ell_{\mathsf{row}}$.
Verifier's input: VOLE correlation challenge $\Delta \in \mathbb{F}_{2^\rho}$, the compressed masked witness $\boldsymbol{t} \coloneqq (\boldsymbol{t}_0, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_{n-1})$, where each $\boldsymbol{t}_i \in \mathbb{F}_2^{\ell_{\mathsf{row}}}$, ℓ VOLE correlation inputs $\boldsymbol{q} \coloneqq (\boldsymbol{q}_0, \ldots, \boldsymbol{q}_{n-1})$ with each $\boldsymbol{q}_i$ consists of ℓ_{row} values in $\mathbb{F}_{2^\rho}$.

Output

Verifier's output: Tuple of VOLE correlation values corresponding to the shares $\boldsymbol{\beta}$ held by the prover, tuple of n VOLE correlation values $\boldsymbol{q_z} \coloneqq \left(q_{z_0}, q_{z_1}, \ldots q_{z_{n-1}}\right)$ corresponding to the secret elementary vector of length n held by the prover, along with tuple of n VOLE correlation values $\boldsymbol{q}_{\mathsf{ColCheck}} \coloneqq \left(\boldsymbol{q}_{\mathsf{ColCheck}_0}, \boldsymbol{q}_{\mathsf{ColCheck}_1}, \ldots, \boldsymbol{q}_{\mathsf{ColCheck}_{n-1}}\right)$ corresponding to (sum of column entries - 1) for each column of the secret permutation matrix held by the prover. All VOLE correlation values output are in $\mathbb{F}_{2^\rho}$.

Compute $\boldsymbol{P}$ row-wise as n elementary vectors

for $i \in [n]$:
  $(\boldsymbol{q}_{\boldsymbol{\beta}_i}, \boldsymbol{q}_{\boldsymbol{z}_i}) \coloneqq$ V.VOLE-ElementaryVector$(\Delta, \boldsymbol{t}_i, \boldsymbol{q}_i)$
endfor

Compute $\boldsymbol{P}$ columns check

for $j \in [n]$:
  $q_{\mathsf{ColSum}_j} = \sum_{i=0}^{n-1} q_{z_{i,j}}$
  $q_{\mathsf{ColCheck}_j} = q_{\mathsf{ColSum}_j} - \Delta^d$
endfor

$\boldsymbol{q}_{\boldsymbol{\beta}} \coloneqq \left(\boldsymbol{q}_{\boldsymbol{\beta}_0}, \boldsymbol{q}_{\boldsymbol{\beta}_1}, \ldots, \boldsymbol{q}_{\boldsymbol{\beta}_{n-1}}\right)$
$\boldsymbol{q}_{\boldsymbol{z}} \coloneqq \left(\boldsymbol{q}_{\boldsymbol{z}_0}, \boldsymbol{q}_{\boldsymbol{z}_1}, \ldots, \boldsymbol{q}_{\boldsymbol{z}_{n-1}}\right)$
$\boldsymbol{q}_{\mathsf{ColCheck}} \coloneqq \left(\boldsymbol{q}_{\mathsf{ColCheck}_0}, \boldsymbol{q}_{\mathsf{ColCheck}_1}, \ldots, \boldsymbol{q}_{\mathsf{ColCheck}_{n-1}}\right)$
return $(\boldsymbol{q}_{\boldsymbol{\beta}}, \boldsymbol{q}_{\boldsymbol{z}}, \boldsymbol{q}_{\mathsf{ColCheck}})$

Algorithm 4.29: V.Check-ElementaryBlock$\left(\boldsymbol{q}'_{\boldsymbol{\beta}'_i}\right)$

Public information and inputs

Public information: Length of the elementary vector block to be checked $\in \{2, 4\}$.
Verifier's input: A tuple of VOLE correlation inputs $\boldsymbol{q}'_{\boldsymbol{\beta}'_i}$ with same size as the elementary vector block to be checked, with each element of $\boldsymbol{q}'_{\boldsymbol{\beta}'_i}$ in $\mathbb{F}_{2^\rho}$.

Output

Verifier's output: VOLE correlation values $\boldsymbol{q}'_{\boldsymbol{e}'_i}$ (which can be a single element or a tuple) to help verify the elementary vector structure the secret vector held by the prover.

---

1 : **if** $\mathsf{len}(\boldsymbol{q}'_{\boldsymbol{\beta}'_i}) = 4$:

  // Parse $\boldsymbol{q}'_{\boldsymbol{\beta}'_i}$ as $\boldsymbol{q}'_{\boldsymbol{\beta}'_i} \coloneqq \left(q'_{\beta'_{i,0}}, q'_{\beta'_{i,1}}, q'_{\beta'_{i,2}}, q'_{\beta'_{i,3}}\right)$.

2 : $q'_{e'_{i,0,1}} \coloneqq q'_{\beta'_{i,0}} \cdot q'_{\beta'_{i,1}}$

3 : $q'_{e'_{i,0,2}} \coloneqq q'_{\beta'_{i,2}}, q'_{\beta'_{i,3}}$

4 : $\boldsymbol{q}'_{\boldsymbol{e}'_i} \coloneqq \left(q'_{e'_{i,0,1}}, q'_{e'_{i,0,2}}\right)$

5 : **return** $\boldsymbol{q}'_{\boldsymbol{e}'_i}$

6 : **elseif** $\mathsf{len}(\boldsymbol{q}'_{\boldsymbol{\beta}'_i}) = 2$:

  // Parse $\boldsymbol{q}'_{\boldsymbol{\beta}'_i}$ as $\boldsymbol{q}'_{\boldsymbol{\beta}'_i} \coloneqq \left(q'_{\beta'_{i,0}}, q'_{\beta'_{i,1}}\right)$.

7 : $q'_{e'_{i,0,1}} \coloneqq q'_{\beta'_{i,0}} \cdot q'_{\beta'_{i,1}}$

8 : **return** $q'_{e'_{i,0,1}}$

9 : **else** :

10 : **return** $\perp$

11 : **endif**

Algorithm 4.30: V.Check-ElementaryVector$\left(\Delta, \boldsymbol{q}'_{\boldsymbol{\beta}'}\right)$

Public information and inputs

Public information: Length of the tuple of VOLE correlation inputs corresponding to the masked witness $|\boldsymbol{q}'_{\boldsymbol{\beta}'}| := 2 \cdot \ell_{\mathsf{row}} - 6$. Lengths of the blocks $|\boldsymbol{q}'_{\boldsymbol{\beta}'_i}| := 4$ for $i \in [3]$, and $|\boldsymbol{q}'_{\boldsymbol{\beta}'_3}| := 2$

Verifier's input: VOLE correlation challenge $\Delta \in \mathbb{F}_{2^\rho}$, a tuple of VOLE correlation inputs $\boldsymbol{q}'_{\boldsymbol{\beta}'}$ with each of element in $\mathbb{F}_{2^\rho}$.

Output

Verifier's output: A tuple of VOLE correlation values $\boldsymbol{q}'_{\boldsymbol{e}'}$ to help verify the elementary vector structure the secret vector held by the prover.

---

1 : **if** $\mathsf{len}(\boldsymbol{q}'_{\boldsymbol{\beta}'}) = 12$: // Security level 1

// Parse $\boldsymbol{q}'_{\boldsymbol{\beta}'}$ as $\boldsymbol{q}'_{\boldsymbol{\beta}'} := \left(\boldsymbol{q}'_{\boldsymbol{\beta}'_0}, \boldsymbol{q}'_{\boldsymbol{\beta}'_1}, \boldsymbol{q}'_{\boldsymbol{\beta}'_2}\right)$.

2 : **for** $i \in [3]$:

3 : $\boldsymbol{q}'_{\boldsymbol{e}'_i} :=$ V.Check-ElementaryBlock $\left(\boldsymbol{q}'_{\boldsymbol{\beta}'_i}\right)$

4 : $\boldsymbol{q}'_{\boldsymbol{e}'_i} := \Delta \cdot \boldsymbol{q}'_{\boldsymbol{e}'_i}$ // Here each element of the tuple $\boldsymbol{q}'_{\boldsymbol{e}'_i}$ should be multiplied by Δ.

5 : **endfor**

6 : $\boldsymbol{q}'_{\boldsymbol{e}'} := \left(\boldsymbol{q}'_{\boldsymbol{e}'_0}, \boldsymbol{q}'_{\boldsymbol{e}'_1}, \boldsymbol{q}'_{\boldsymbol{e}'_2}\right)$

7 : **return** $\boldsymbol{q}'_{\boldsymbol{e}'}$

8 : **elseif** $\mathsf{len}(\boldsymbol{q}'_{\boldsymbol{\beta}'}) = 14$: // Security levels 3 and 5

// Parse $\boldsymbol{q}'_{\boldsymbol{\beta}'}$ as $\boldsymbol{q}'_{\boldsymbol{\beta}'} := \left(\boldsymbol{q}'_{\boldsymbol{\beta}'_0}, \boldsymbol{q}'_{\boldsymbol{\beta}'_1}, \boldsymbol{q}'_{\boldsymbol{\beta}'_2}, \boldsymbol{q}'_{\boldsymbol{\beta}'_3}\right)$.

9 : $\left(\boldsymbol{q}'_{\boldsymbol{e}'_0}, \boldsymbol{q}'_{\boldsymbol{e}'_1}, \boldsymbol{q}'_{\boldsymbol{e}'_2}\right) :=$ V.Check-ElementaryVector $\left(\boldsymbol{q}'_{\boldsymbol{\beta}'_0}, \boldsymbol{q}'_{\boldsymbol{\beta}'_1}, \boldsymbol{q}'_{\boldsymbol{\beta}'_2}\right)$

10 : **for** $i \in [3]$:

11 : $\boldsymbol{q}'_{\boldsymbol{e}'_i} := \Delta \cdot \boldsymbol{q}'_{\boldsymbol{e}'_i}$ // Here each element of the tuple $\boldsymbol{q}'_{\boldsymbol{e}'_i}$ should be multiplied by Δ.

12 : **endfor**

13 : $q'_{e'_{3,0,1}} :=$ V.Check-ElementaryBlock $\left(\boldsymbol{q}'_{\boldsymbol{\beta}'_3}\right)$

14 : $q'_{e'_{3,0,1}} := \Delta^2 \cdot q'_{e'_{3,0,1}}$

15 : $\boldsymbol{q}'_{\boldsymbol{e}'} := \left(\boldsymbol{q}'_{\boldsymbol{e}'_0}, \boldsymbol{q}'_{\boldsymbol{e}'_1}, \boldsymbol{q}'_{\boldsymbol{e}'_2}, q'_{e'_{3,0,1}}\right)$

16 : **return** $\boldsymbol{q}'_{\boldsymbol{e}'}$

17 : **else** :

18 : **return** $\perp$

19 : **endif**

Algorithm 4.31: V.CheckZero$\big(\Delta, q_f, (q_{u_{i,k}})_{(i,k)\in[d-1]\times[\rho]}, [\![a]\!]\big)$

Public information and inputs

Public information: Degree of input polynomial $[\![a]\!] \coloneqq d-1$.

Verifier's input: Polynomial $[\![a]\!]$, $q_f, q_{u_{i,k}} = f_{u_{i,k}}(\Delta)$ for $(i,k) \in [d-1]\times[\rho]$, $\Delta \in \mathbb{F}_{2^\rho}$.

Output

Verifier's output: Boolean indicating if leading coefficient (coefficient of X^d) of some degree-d polynomial $f(X)$ (for which verifier already holds $q_f \in \mathbb{F}_{2^\rho}$) is equal to zero or not.

---

if degree of $[\![a]\!] \neq d-1$:
  return $\perp$
else :
  // Generating $q'_{u_i} \in \mathbb{F}_{2^\rho}$ for $i \in [d-1]$
  for $i \in [d-1]$:
    $q'_{u_i} \coloneqq \sum_{k=0}^{\rho-1} q_{u_{i,k}} \cdot \gamma_\rho^k$ // $\left\{\gamma_\rho^k\right\}_{k=0}^{\rho-1}$ is the power basis of $\mathbb{F}_{2^\rho}$.
  endfor
  // Parse $[\![a]\!]$ as $[\![a]\!] \coloneqq a_{d-1}X^{d-1} + \cdots + a_1 X + a_0$
  Compute $\tilde{q} \coloneqq \sum_{i=0}^{d-1} a_i \cdot \Delta^i$
  $q = q_f + \sum_{i=0}^{d-2} q'_{u_i} \cdot \Delta^i$
  $b \coloneqq \left(q \stackrel{?}{=} \tilde{q}\right)$
return b

Algorithm 4.32: V.Check-PKP(seed, pk, $\boldsymbol{t}, \Delta, \boldsymbol{q}, \boldsymbol{q}_{\mathsf{cz}}, [\![a]\!]$)

Public information

Public information: Matrix dimension n of the secret permutation matrix, $d = \lceil \log_4 n \rceil$.
Verifier's input: VOLE correlation challenge $\Delta \in \mathbb{F}_{2^\rho}$, the compressed masked witness $\boldsymbol{t} := (\boldsymbol{t}_0, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_{n-1})$, where each $\boldsymbol{t}_i \in \mathbb{F}_2^{\ell_{\mathsf{row}}}$, ℓ VOLE correlation inputs $\boldsymbol{q} := (\boldsymbol{q}_0, \ldots, \boldsymbol{q}_{n-1})$ with each $\boldsymbol{q}_i$ consists of ℓ_{row} values in $\mathbb{F}_{2^\rho}$, $(d-1)$ VOLE correlations $\boldsymbol{q}_{\mathsf{cz}}$ in $\mathbb{F}_{2^\rho}$, polynomial $[\![a]\!]$, and $\mathsf{seed} \in \{0,1\}^{2\lambda}$.

Output

Verifier's output: Boolean value b indicating if the proof is accepted or not.

Compute VOLE correlations corresponding to shares of $\boldsymbol{P}$ in matrix form

$(\boldsymbol{q}_{\boldsymbol{\beta}}, \boldsymbol{q}_{\boldsymbol{z}}, \boldsymbol{q}_{\mathsf{ColCheck}}) :=$ V.VOLE-Permutation $(\Delta, \boldsymbol{t}, \boldsymbol{q})$

// Parse $\boldsymbol{q}_{\mathsf{ColCheck}}$ as $\left(q_{f_0}, q_{f_1}, \ldots, q_{f_{n-1}}\right)$

Check elementary vectors

for $i \in [0, n-1]$

  $\boldsymbol{q}_{\mathsf{ElemVecCheck}_i} :=$ V.Check-ElementaryVector $\left(\Delta, \boldsymbol{q}_{\boldsymbol{\beta}_i}\right)$

  // Each $\boldsymbol{q}_{\mathsf{ElemVecCheck}_i}$ contains 6 elements if $\lambda = 128$,

  // and 7 elements if $\lambda \in \{192, 256\}$.

endfor

$\boldsymbol{q}_{\mathsf{ElemVecCheck}} := \left(\boldsymbol{q}_{\mathsf{ElemVecCheck}_0}, \ldots, \boldsymbol{q}_{\mathsf{ElemVecCheck}_{n-1}}\right)$

// Parse $\boldsymbol{q}_{\mathsf{ElemVecCheck}}$ as $\left(q_{f_n}, q_{f_{n+1}}, \ldots, q_{f_{cn+n-1}}\right)$,

// where, $c = 6$ if $\lambda = 128$, and $c = 7$ if $\lambda \in \{192, 256\}$.

Compute $\boldsymbol{x}' = \boldsymbol{P}\boldsymbol{x}$

for $i \in [0, n-1]$

  $q_{\boldsymbol{x}'_i} = \sum_{j=0}^{n-1} q_{z_{i,j}} \cdot \boldsymbol{x}_j$

endfor

Compute $\boldsymbol{y} = \boldsymbol{H}\boldsymbol{x}'$

for $i \in [0, m-1]$

  $q_{\boldsymbol{y}_i} = q_{f_{i+cn+n}} = \sum_{j=0}^{n-1} h_{i,j} \cdot q_{\boldsymbol{x}'_j}$

endfor

Merge polynomials and run CheckZero

$\boldsymbol{\alpha} := \mathsf{H}_4(\mathsf{seed} :: \rho \cdot (cn + n + m))$ // $\boldsymbol{\alpha}$ should be parsed as $\in \mathbb{F}_{2^\rho}^{cn+n+m}$

$q_f = \sum_{j=0}^{cn+n+m-1} \alpha_j \cdot q_{f_j}$

$b :=$ V.CheckZero$(\Delta, q_f, \boldsymbol{q}_{\mathsf{cz}}, [\![a]\!])$

return b

4.6 PERK

In this section, we describes the PERK.KeyGen, PERK.Sign and PERK.Verify algorithms. The PERK key generation algorithm PERK.KeyGen takes the public parameters for the PKP $\mathsf{param}_{\mathsf{PKP}} \coloneqq (q, m, n)$ as input along with the security parameter λ. The key generation algorithm outputs a public key represented by a seed of length λ which generates the matrix $\boldsymbol{H}$ (in deterministic manner), and a vector $\boldsymbol{x} \in \mathbb{F}_q^n$; along with the secret key also represented by a separate seed of length λ which generates the secret permutation in a deterministic way.

The key generation algorithm, first samples 3 distinct seeds $\boldsymbol{H}_{\mathsf{seed}}$, $\mathsf{perm}_{\mathsf{seed}}$, and $\mathsf{ker}_{\mathsf{seed}}$ of length λ independently and at uniform random. It expands the seed $\boldsymbol{H}_{\mathsf{seed}}$ to compute a pseudorandom matrix $\boldsymbol{M} \in \mathbb{F}_q^{m \times (n-m)}$ and sets the public matrix in its canonical form as $\boldsymbol{H} \coloneqq [\boldsymbol{I}_m \, \boldsymbol{M}]$. Then it computes the basis of kernel of $\boldsymbol{H}$ and samples a random vector $\boldsymbol{x}'$ by taking a random linear combination of the basis vectors, this random linear combination is derived using $\mathsf{ker}_{\mathsf{seed}}$. The key generation algorithm also samples a permutation π using $\mathsf{perm}_{\mathsf{seed}}$ and sets $\boldsymbol{x} \coloneqq \pi^{-1}(\boldsymbol{x}')$. The algorithms outputs $\mathsf{pk} \coloneqq (\boldsymbol{H}_{\mathsf{seed}}, \boldsymbol{x})$, and $\mathsf{sk} \coloneqq \mathsf{perm}_{\mathsf{seed}}$.

Algorithm 4.33: PERK.KeyGen()

Public information and inputs

Public information: Public parameters $\mathsf{param}_{\mathsf{PKP}} \coloneqq (q, m, n)$, and security parameter λ.

Output

The public key as a seed that generates the public matrix concatenated with the public vector, and the private key as the seed that generates the secret permutation.

Sampling randomness

1 : $\boldsymbol{H}_{\mathsf{seed}} \xleftarrow{\$} \{0,1\}^\lambda$

2 : $\mathsf{ker}_{\mathsf{seed}} \xleftarrow{\$} \{0,1\}^\lambda$

3 : $\mathsf{perm}_{\mathsf{seed}} \xleftarrow{\$} \{0,1\}^\lambda$

Construct PKP instance

4 : $\boldsymbol{M} \leftarrow \mathsf{ExpandMatrixM}(\boldsymbol{H}_{\mathsf{seed}})$ in $\mathbb{F}_q^{m \times (n-m)}$

5 : $\boldsymbol{H} = [\boldsymbol{I}_m \, \boldsymbol{M}]$ in $\mathbb{F}_q^{m \times n}$

6 : $\boldsymbol{x}' \leftarrow \mathsf{ExpandKernelVector}(\mathsf{ker}_{\mathsf{seed}}, \boldsymbol{H})$ in $\ker(\boldsymbol{H})$

7 : $\pi \leftarrow \mathsf{ExpandPermutation}(\mathsf{perm}_{\mathsf{seed}})$ in $\mathcal{S}_n$

8 : $\boldsymbol{x} = \pi^{-1}(\boldsymbol{x}')$

9 : **return** $(\mathsf{pk} = (\boldsymbol{H}_{\mathsf{seed}}, \boldsymbol{x}), \mathsf{sk} = (\mathsf{perm}_{\mathsf{seed}}))$.

Algorithm 4.34: PERK.Sign(msg, sk, pk)

Public information and inputs

Public information: Security parameter λ, public parameters $\mathsf{param}_{\mathsf{PKP}} := (q, m, n)$, $\mathsf{param}_{\mathsf{VOLE}} := (\tau, \mu, \rho)$, $\mathsf{param}_{\mathsf{TreePRG}} := (\tau, \kappa, N, T_{\mathsf{open}}, w)$ where, $N := \tau 2^{\kappa}$ is the number of leaves of the VOLE commitment. Public key (expanded) $\mathsf{pk} := (\boldsymbol{H}, \boldsymbol{x})$.
Prover's input: Secret key sk represented as an array of positions of non-zero entries of the secret permutation matrix (row-wise), message msg to be signed.

Output

Signature σ for message msg generated using prover's secret key sk.

Initialization

$\tilde{\mu} := \mathsf{H}_1\,(\mathsf{pk}||\mathsf{msg} :: 2\lambda)$

$\mathsf{rand} \xleftarrow{\$} \{0,1\}^{2\lambda}$

$(\mathsf{mseed}, \mathsf{salt}) := \mathsf{H}_3\,(\mathsf{sk}||\tilde{\mu}||\mathsf{rand} :: 3\lambda) \in \{0,1\}^{\lambda} \times \{0,1\}^{2\lambda}$

VOLE construction, commitments and consistency checks

$(\boldsymbol{h}_{\mathsf{com}}, \mathsf{decom}, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}, \boldsymbol{u}, \boldsymbol{V}) := \mathsf{VOLECommit}(\mathsf{mseed}, \mathsf{salt} :: \hat{\ell})$

$\mathsf{ch}_1 := \mathsf{H}_2^1\,(\tilde{\mu}||h_{\mathsf{com}}||\boldsymbol{c}_1||\cdots||\boldsymbol{c}_{\tau-1}||\mathsf{salt} :: 5\lambda + 64)$

$\widetilde{\boldsymbol{u}} := \mathsf{VOLEHash}(\mathsf{ch}_1, \boldsymbol{u} :: \ell_{\mathsf{VOLEHashMask}}) \in \{0,1\}^{\ell_{\mathsf{VOLEHashMask}}}$

$\widetilde{\boldsymbol{V}} := \mathsf{VOLEHash}(\mathsf{ch}_1, \boldsymbol{V} :: \ell_{\mathsf{VOLEHashMask}}) \in \{0,1\}^{\ell_{\mathsf{VOLEHashMask}} \times \rho}$ // hash column-wise

$h_V := \mathsf{H}_1\left(\widetilde{\boldsymbol{V}} :: 2\lambda\right)$ // hash in column major order

Committing to witness and PKP proof

$(\boldsymbol{t}, [\![\boldsymbol{\beta}]\!], [\![\boldsymbol{z}]\!]^{(d)}, [\![\mathsf{ColCheck}]\!]^{(d)}) := \mathsf{P.VOLE\text{-}Permutation}\left(\mathsf{sk}, ([\![u_{i,k}]\!])_{i\in[n],k\in[\ell]}\right)$

$\mathsf{ch}_2 := \mathsf{H}_2^2\,(\mathsf{ch}_1||\widetilde{\boldsymbol{u}}||h_V||\boldsymbol{t} :: 2\lambda)$

$[\![a]\!] := \mathsf{P.Check\text{-}PKP}\left(\mathsf{sk}, \mathsf{pk}, [\![u_{i,k}]\!]_{i\in[n],k\in[\ell]}, ([\![u_{i',k'}]\!])_{(i',k')\in[d-1]\times[\rho]}, \mathsf{ch}_2\right)$

VOLE decommitments and opening VC

$\mathsf{ctr} := 0$

while True:

  $\mathsf{ch}_3 := \mathsf{H}_2^3\,(\mathsf{ch}_2||[\![a]\!]||\mathsf{ctr} :: \tau_0\kappa_0 + \tau_1\kappa_1 + w)$ // $\tau_0\kappa_0 + \tau_1\kappa_1 + w$ bits

  if $\mathsf{ch}_3[\tau_0\kappa_0 + \tau_1\kappa_1 : \tau_0\kappa_0 + \tau_1\kappa_1 + w - 1] = 0^w$:

    $\boldsymbol{i}^* := \mathsf{ChallDec}\,(\mathsf{ch}_3[0 : \tau_0\kappa_0 + \tau_1\kappa_1 - 1])$

    $(\mathsf{pdecom}, (\mathsf{com}_{e,\boldsymbol{i}^*[e]})_{e\in[\tau]}) := \mathsf{VC.Open}\,(\mathsf{decom}, \boldsymbol{i}^*)$

    if the above output is not $\perp$ **break**

  $\mathsf{ctr} = \mathsf{ctr} + 1$

return $\sigma := \left(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}, \widetilde{\boldsymbol{u}}, \boldsymbol{t}, [\![a]\!], \mathsf{pdecom}, (\mathsf{com}_{e,\boldsymbol{i}^*[e]})_{e\in[\tau]}, \mathsf{ch}_3, \mathsf{ctr}, \mathsf{salt}\right)$

Algorithm 4.35: PERK.Verify(pk, msg, σ)

Public information and inputs

Public information: Security parameter λ, public parameters $\mathsf{param}_{\mathsf{PKP}} := (q, m, n)$, $\mathsf{param}_{\mathsf{VOLE}} := (\tau, \mu, \rho)$, $\mathsf{param}_{\mathsf{TreePRG}} := (\tau, \kappa, N, T_{\mathsf{open}}, w)$ where, $N := \tau 2^{\kappa}$ is the number of leaves of the VOLE commitment. Public key (expanded) $\mathsf{pk} := (\boldsymbol{H}, \boldsymbol{x})$.
Verifier's input: Public key (expanded) $\mathsf{pk} := (\boldsymbol{H}, \boldsymbol{x})$, message msg, and a signature σ.

Output

Bit b indicating if the signature σ verifies as a valid signature for message msg with pk as the public key. If $b = 1$ the signature is accepted as valid, if $b = 0$ it is rejected.

Initialization

Parse $\sigma := \left(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}, \widetilde{\boldsymbol{u}}, \boldsymbol{t}, [\![a]\!], \mathsf{pdecom}, (\mathsf{com}_{e,\boldsymbol{i}^*[e]})_{e\in[0,\tau)}, \mathsf{ch}_3, \mathsf{ctr}, \mathsf{salt}\right)$
if $\mathsf{ch}_3[\tau_0\kappa_0 + \tau_1\kappa_1 : \tau_0\kappa_0 + \tau_1\kappa_1 + w - 1] \neq 0^w$ **then return** 0
$\mu := \mathsf{H}_1\left(\mathsf{pk}||\mathsf{msg} :: 2\lambda\right)$
$\boldsymbol{i}^* := \mathsf{ChallDec}\left(\mathsf{ch}_3[0 : \tau_0\kappa_0 + \tau_1\kappa_1 - 1]\right)$

Reconstruct VOLEs and check commitments

$\mathsf{out} := \mathsf{VOLEReconstruct}\left(\boldsymbol{i}^*, \mathsf{pdecom}, \{\mathsf{com}_{e,\boldsymbol{i}^*}\}_{\tau,\tau}, \mathsf{salt}\right)$
if $\mathsf{out} = \perp$:
  return 0
else :
  Parse out as $\mathsf{out} := (h_{\mathsf{com}}, \boldsymbol{Q}'_0, \ldots, \boldsymbol{Q}'_{\tau-1})$.
$\overline{\mathsf{ch}}_1 := \mathsf{H}_2^1\left(\tilde{\mu}||h_{\mathsf{com}}||\boldsymbol{c}_1||\cdots||\boldsymbol{c}_{\tau-1}||\mathsf{salt} :: 5\lambda + 64\right)$

Check VOLE's consistency

for $e \in [\tau]$
  $(\delta_0, \ldots, \delta_{\kappa_e-1}) := \mathsf{BitDec}(\boldsymbol{i}^*[e])$
  if $\mu_e > \kappa_e$:
    $\mathsf{zeropad} := \boldsymbol{0}^{\ell_{\mathsf{VOLEHashMask}} \times (\mu_e - \kappa_e)}$
  else :
    $\mathsf{zeropad} := \varepsilon$
  $\tilde{\boldsymbol{D}}_e := [\delta_0 \cdot \tilde{\boldsymbol{u}} \cdots \delta_{\kappa_e-1} \cdot \tilde{\boldsymbol{u}} \quad \mathsf{zeropad}] \in \{0,1\}^{\ell_{\mathsf{VOLEHashMask}} \times \mu_e}$
  if $e = 0$ **then** $\boldsymbol{Q}_0 := \boldsymbol{Q}'_0$
  if $e > 0$ **then** $\boldsymbol{Q}_e := \boldsymbol{Q}'_e \oplus [\delta_0 \cdot \boldsymbol{c}_e \cdots \delta_{k_e-1} \cdot \boldsymbol{c}_e \quad \mathsf{zeropad}]$
endfor
$\boldsymbol{Q} := [\boldsymbol{Q}_0 \cdots \boldsymbol{Q}_{\tau-1}]$
$\tilde{\boldsymbol{Q}} := \mathsf{VOLEHash}\left(\overline{\mathsf{ch}}_1, \boldsymbol{Q} :: \ell_{\mathsf{VOLEHashMask}}\right) \in \{0,1\}^{\ell_{\mathsf{VOLEHashMask}} \times \rho}$
$h_V := \mathsf{H}_1(\tilde{\boldsymbol{Q}} \oplus [\tilde{\boldsymbol{D}}_0 \cdots \tilde{\boldsymbol{D}}_{\tau-1}] :: 2\lambda)$
$\overline{\mathsf{ch}}_2 := \mathsf{H}_2^2(\overline{\mathsf{ch}}_1||\tilde{\boldsymbol{u}}||h_V||\boldsymbol{t} :: 2\lambda)$

Check PKP's consistency

$\overline{\mathsf{ch}}_3 := \mathsf{H}_2^3(\overline{\mathsf{ch}}_2||[\![a]\!]||\mathsf{ctr} :: \tau_0\kappa_0 + \tau_1\kappa_1 + w)$
$b_V := \mathsf{V.Check\text{-}PKP}(\mathsf{seed}, \mathsf{pk}, \boldsymbol{t}, \Delta, \boldsymbol{q}, \boldsymbol{q}_{\mathsf{fix}}, [\![a]\!])$
if $b_V = 1$ and $\mathsf{ch}_3 = \overline{\mathsf{ch}}_3$ **then**
  return 1
else return 0

5 Parameter Sets and Sizes

We provide several parameter sets using the nomenclature PERK-X-Y where $\mathsf{X} \in \{1, 3, 5\}$ denotes the security level and $\mathsf{Y} \in \{\mathsf{Short}, \mathsf{Fast}\}$ refers to size / performance trade-off considered for the parameter set.

5.1 PKP parameters

The PKP parameters $\mathsf{param}_{\mathsf{PKP}} \coloneqq (q, m, n)$ used in PERK are given in Table 1. Parameters were chosen to minimize the signature size while offering concrete bit-security of PKP above the NIST specified thresholds for category 1, 3 and 5. We give full details on estimating the security of PKP in Section Section 7.2.1.

Instance	q	n	m
PERK-1	2048	64	27
PERK-3	2048	92	43
PERK-5	2048	118	59

Table 1: PKP parameters used in PERK

5.2 MPC and VOLE parameters

The tree parameters $\mathsf{param}_{\mathsf{TreePRG}} \coloneqq (\tau, \kappa, N, T_{\mathsf{open}}, w)$ are given in Table 2. The number of parties and iterations is governed by the knowledge soundness of the protocol. The MPC parameters are also chosen to guarantee a soundness probability of $2^{-\lambda}$ for $\lambda \in \{128, 192, 256\}$ for category 1, 3 and 5 respectively. Following common practice we propose two different parameter sets, a *short* variant using $N' \in \{2048, 4096\}$ and a *fast* variant using $N' \in \{128, 256\}$, where N' denotes the number of leaf nodes in each individual tree. The table below lists the total number of leaves $N \coloneqq \tau 2^{\kappa}$ in all the trees together.

The VOLE related parameters $\mathsf{param}_{\mathsf{VOLE}} \coloneqq (\tau, \mu, \rho)$ are given in Table 3. The parameter ρ denotes the dimension of finite field ($\mathbb{F}_{2^\rho}$) in which the VOLE correlations (seen as polynomials) reside. We also need to compute the PKP relation $\boldsymbol{HPx}$ using the VOLE correlations therefore the fields should maintain the following tower relationship: $\mathbb{F}_2 \subset \mathbb{F}_q \subset \mathbb{F}_{2^\rho}$ and $\mathbb{F}_2 \subset \mathbb{F}_{2^\kappa} \subseteq \mathbb{F}_{2^\mu} \subset \mathbb{F}_{2^\rho}$.

The witness parameters $\mathsf{param}_{\mathsf{witness}} \coloneqq (n, d, \ell_{\mathsf{row}}, \ell, \ell_{\mathsf{CZMask}}, \ell_{\mathsf{VOLEHashMask}}, \hat{\ell})$ are given in Table 4. To prove the knowledge of the secret witness $\boldsymbol{P} \in \mathbb{F}_q^{n \times n}$, we are using $\hat{\ell}$ VOLE correlations that are computed using $\mathsf{param}_{\mathsf{witness}}$ where:

- n is the number of rows and columns in the permutation matrix ;

- $d \coloneqq \lceil \log_4(n) \rceil$ is the degree of polynomials, which will help prove the knowledge of the witness using VOLE correlations ;
- $\ell_{\mathsf{row}} \coloneqq d + 6$ is the length of (compressed) witness (in bits) for each row of the secret permutation matrix ;
- $\ell \coloneqq n \cdot \ell_{\mathsf{row}}$ is the length of the (compressed) witness (in bits) for full secret permutation matrix ; This is essentially the witness size that affects the signature sizes ;
- $\ell_{\mathsf{CZMask}} \coloneqq (d-1) \cdot \rho$ is the number of bits required to construct masking VOLE correlations in $\mathbb{F}_{2^\rho} \times \mathbb{F}_{2^\rho}$ required in P.CheckZero ;
- $\ell_{\mathsf{VOLEHashMask}} \coloneqq \lambda + B$ is the number of bits required for masking VOLEHash. We always set $B \coloneqq 16$ for all our parameter sets and instances ;
- $\hat{\ell} \coloneqq \ell_{\mathsf{VOLEHashMask}} + \ell + \ell_{\mathsf{CZMask}}$ is the number of bits that should be communicated in each round. Therefore, the signature size is affected by the value $(\tau - 1) \cdot \hat{\ell}$.

Note that in Tables 2 and 3, τ is the number of repetitions required to desired security level $\rho \geq \lambda$ (in our case $\rho > \lambda$ for all cases). Therefore, following criteria must always be satisfied:

- $\tau \coloneqq \tau_0 + \tau_1 = \tau_0' + \tau_1'$;
- $\rho \coloneqq \tau_0' \mu_0 + \tau_1' \mu_1$;
- $\rho \geq \lambda$;
- $\tau_0 \kappa_0 + \tau_1 \kappa_1 + w - \log_2(d) \geq \lambda$.

Instance	τ	τ_0	τ_1	κ_0	κ_1	N	T_{open}	w
PERK-1-Short	11	11	0	11	0	3200	106	9
PERK-1-Fast	16	9	7	8	7	22528	110	
PERK-3-Short	16	8	8	12	11	5120	166	10
PERK-3-Fast	24	16	8	8	7	49152		
PERK-5-Short	22	8	14	12	11	7424	222	8
PERK-5-Fast	32	26	6	8	7	61440	220	

Table 2: BAVC parameters used in PERK

Instance	τ	τ'_0	τ'_1	μ_0	μ_1	ρ
PERK-1-Short	11	11	0	12	0	132
PERK-1-Fast	16	4	12	9	8	
PERK-3-Short	16	6	10	13	12	198
PERK-3-Fast	24	6	18	9	8	
PERK-5-Short	22	22	0	12	0	264
PERK-5-Fast	32	8	24	9	8	

Table 3: VOLE fields parameters used in PERK

Instance	B	d	ℓ_{row}	ℓ	ℓ_{CZMask}	$\hat{\ell}$
PERK-1	16	3	9	576	264	984
PERK-3		4	10	920	594	1722
PERK-5		4	10	1180	792	2244

Table 4: VOLE correlations parameters used in PERK

5.3 Signature and key sizes

Table 5 presents the public key, secret key, and signature sizes of PERK. The size of the public key pk is $\lambda + n\lceil \log_2(q) \rceil$ bits while the size of the secret key sk is λ bits. In practice, our implementations concatenate the public key within the secret key in order to respect the API provided by the NIST.

A PERK signature consists of:

- a salt, a hash value ch_3 and a counter ctr making a subtotal of $3\lambda + 66$ bits ;
- $(\tau - 1)$ VOLE correlation $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}$ each of length $\hat{\ell}$ bits ;
- τ commitments $(\mathsf{com}_{e,\boldsymbol{i}^*[e]})_{e \in [\tau]}$ each of size 2λ bits ;
- a masked witness $\boldsymbol{t}$ of size $n \cdot \ell_{\mathsf{row}}$ bits ;
- a VOLEHash value $\widetilde{\boldsymbol{u}}$ of size $\lambda + 16$ bits ;
- some opening information pdecom of size $\lambda \cdot T_{\mathsf{open}}$ bits ;
- a polynomial $[\![a]\!]$ represented as coefficients with $\lceil \log_4(n) \rceil \cdot \rho$ bits.

Overall, for a security level λ, the signature size is given by:

$$|\sigma| = \underbrace{4\lambda + 82}_{\mathsf{salt},\mathsf{ctr},\mathsf{ch}_3,\widetilde{\boldsymbol{u}}} + \underbrace{\tau \cdot 2\lambda}_{\text{commitments}} + \underbrace{(\tau - 1) \cdot \hat{\ell}}_{\text{VOLE correlations}}$$

$$+ \underbrace{\lceil \log_4(n) \rceil \cdot \rho}_{[\![a]\!]} + \underbrace{\lambda \cdot T_{\mathsf{open}}}_{\mathsf{pdecom}} + \underbrace{n \cdot (\lceil \log_4(n) \rceil + 6)}_{\boldsymbol{t}}.$$

Instance	$\|\mathsf{sk}\|$	$\|\mathsf{pk}\|$	$\|\sigma\|$
PERK-1-Short	16 B	0.10 kB	3.48 kB
PERK-1-Fast	16 B	0.10 kB	4.32 kB
PERK-3-Short	24 B	0.15 kB	8.32 kB
PERK-3-Fast	24 B	0.15 kB	10.43 kB
PERK-5-Short	32 B	0.19 kB	14.83 kB
PERK-5-Fast	32 B	0.19 kB	18.22 kB

Table 5: Keys and signature sizes of PERK

6 Implementation and Performance Analysis

This section provides performance measurement of our PERK implementations.

Benchmark platform. The benchmarks have been done on a machine running Ubuntu 22.04.2 LTS, that has 64 GB of memory and an Intel® Core™ i9-13900K @ 3.00 GHz for which the Hyper-Threading and Turbo Boost features were disabled. For each parameter set, the results have been obtained by computing the average from 10 and 500 random instances for the reference and optimized implementation, respectively. The scheme has been compiled with `gcc` (version `11.4.0`) and the following third party libraries have been used: `XKCP` (commit `7fa59c0ec4`) and `djbsort` (version `20190516`).

Remark on the instantiation of PERK. The overall efficiency of PERK is determined to a large extent by the symmetric primitives employed in its construction. The PERK's pseudorandom generator (PRG) may be instantiated using `AES/Rijndael` or `SHA3`, while hash functions are realized through `SHA3`. For benchmarking purposes, we provide results for two instantiations of the commitment scheme: one using `AES/Rijndael` and another using `SHA3`. It is important to emphasize that the choice of instantiation can lead to substantial variations in performance.

6.1 Reference implementation

The reference implementation is written in `C` and have been compiled using the compilation flags `-O3 -funroll-loops -flto`. The performances of our reference implementation on the aforementioned benchmark platform are described in Table 6-8.

Instance	Keygen	Sign	Verify
PERK-1-Short	50 K	13615 M	13608 M
PERK-1-Fast	50 K	1947 M	1943 M
PERK-3-Short	93 K	89057 M	89184 M
PERK-3-Fast	93 K	9360 M	9323 M
PERK-5-Short	146 K	111810 M	111622 M
PERK-5-Fast	130 K	13678 M	13591 M

Table 6: Performances of the reference implementation (in CPU cycles) - `AES` instantiation for both PRG and commitments

Instance	Keygen	Sign	Verify
PERK-1-Short	47 K	11242 M	11235 M
PERK-1-Fast	45 K	1610 M	1606 M
PERK-3-Short	93 K	74494 M	74613 M
PERK-3-Fast	93 K	7842 M	7805 M
PERK-5-Short	136 K	93676 M	93507 M
PERK-5-Fast	127 K	11486 M	11402 M

Table 7: Performances of the reference implementation (in CPU cycles) - `AES` for PRG and `SHA3` for commitments

Instance	Keygen	Sign	Verify
PERK-1-Short	40 K	125 M	115 M
PERK-1-Fast	38 K	30.9 M	26.3 M
PERK-3-Short	82 K	401 M	354 M
PERK-3-Fast	83 K	121 M	82.3 M
PERK-5-Short	124 K	822 M	759 M
PERK-5-Fast	120 K	254 M	193 M

Table 8: Performances of the reference implementation (in CPU cycles) - `SHA3` instantiation for both PRG and commitments

6.2 Optimized implementation

A constant-time optimized implementation leveraging `AVX2` instructions have been provided. Its performances on the aforementioned benchmark platform are described in Tables 9-11. The following optimization flags have been used during

compilation: `-O3 -funroll-loops -march=native -mavx2 -mpclmul -msse4.2 -maes`.

Instance	Keygen	Sign	Verify
PERK-1-Short	34 K	16.3 M	12.6 M
PERK-1-Fast	33 K	5.0 M	3.4 M
PERK-3-Short	66 K	143 M	130 M
PERK-3-Fast	65 K	33.5 M	23.2 M
PERK-5-Short	102 K	191 M	172 M
PERK-5-Fast	100 K	53.2 M	37.4 M

Table 9: Performances of the reference implementation (in CPU cycles) - `AES` instantiation for both PRG and commitments

Instance	Keygen	Sign	Verify
PERK-1-Short	33 K	27.8 M	24.3 M
PERK-1-Fast	33 K	6.6 M	5.0 M
PERK-3-Short	66 K	155 M	142 M
PERK-3-Fast	65 K	34.9 M	24.5 M
PERK-5-Short	101 K	236 M	217 M
PERK-5-Fast	99 K	58.4 M	42.7 M

Table 10: Performances of the reference implementation (in CPU cycles) - `AES` for PRG and `SHA3` for commitments

Instance	Keygen	Sign	Verify
PERK-1-Short	35 K	47.4 M	43.5 M
PERK-1-Fast	34 K	9.4 M	7.9 M
PERK-3-Short	66 K	142 M	129 M
PERK-3-Fast	65 K	33.4 M	23.2 M
PERK-5-Short	103 K	274 M	255 M
PERK-5-Fast	102 K	63.1 M	47.2 M

Table 11: Performances of the reference implementation (in CPU cycles) - `SHA3` instantiation for both PRG and commitments

6.3 Known Answer Test values

Known Answer Test (KAT) values have been generated using the script provided by the NIST. They are available in the folder `KATs` and files are the same for both reference and optimized implementation. In addition, examples with intermediate values have also been provided in these folders. Notice that one can generate the aforementioned test files using respectively the `kat` and `verbose` modes of our implementation. The procedure to follow in order to do so is detailed in the technical documentation.

7 Security Analysis

7.1 Security proof

Our signature scheme is strongly existentially unforgeable under chosen-message attacks (SUF-CMA-secure) in the random oracle model (ROM)[6] under the assumption of hardness of PKP. The proof of SUF-CMA security, written below, happens in two stages:

- We first show that the slightly modified signature, which we call PERK', is existentially unforgeable under no-message attacks (EUF-NMA-secure) in the ROM assuming the hardness of the PKP problem.
- We then show that the signature scheme PERK is SUF-CMA-secure in the ROM by assuming that PERK' is EUF-NMA-secure in the ROM and some computational hardness of the functions.

We also discuss the beyond unforgeability features (BUFF) securities.

Notations. To simplify notations, we define the following variables:

- $\tilde{\mu} \coloneqq \mathsf{H}_1(\mathsf{pk}||\mathsf{msg})$;
- $a_1 \coloneqq (h_{\mathsf{com}}, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1})$;
- $\mathsf{ch}_1 \coloneqq \mathsf{H}_2^1(\tilde{\mu}||a_1||\mathsf{salt})$;
- $a_2 \coloneqq (\tilde{\boldsymbol{u}}, h_V, \boldsymbol{t})$;
- $\mathsf{ch}_2 \coloneqq \mathsf{H}_2^2(\mathsf{ch}_1||a_2)$;
- $a_3 \coloneqq [\![a]\!]$;
- $\mathsf{ch}_3 \coloneqq \mathsf{H}_2^3(\mathsf{ch}_2||a_3)$;
- $a_4 \coloneqq (\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}, \tilde{\boldsymbol{u}}, \boldsymbol{t}, [\![a]\!], \mathsf{pdecom}, (\mathsf{com}_{e,\boldsymbol{i}[e]}), \mathsf{ctr}, \mathsf{salt})$.

By these notations, we have $\sigma = (\mathsf{ch}_3, a_4)$. In what follows, we denote the internally reproduced values in the verifications $\mathsf{PERK.Verify}(\mathsf{pk}, \mathsf{msg}, \sigma)$ and $\mathsf{PERK.Verify}(\mathsf{pk}, \mathsf{msg}^*, \sigma)$ by $\bar{\cdot}$ and $\bar{\cdot}^*$, respectively.

Preliminaries. Hereafter, we show the extractable-binding and multi-hiding properties of VOLE commitments. The following lemma helps us estimate the bound for one-wayness and collision-resistant property of random oracles.

Lemma 7.1 (Random oracle graph). *Let $H\colon \mathcal{X} \to \mathcal{Y}$ be the random oracle. We consider the following random oracle graph game between a challenger and an adversary:*

1. *The challenger initializes two sets $\mathcal{V}$ and $\mathcal{E}$ by $\emptyset$ and runs the adversary.*
2. *The adversary can query the random oracle H in the following two ways:*
   - *The adversary queries $y \in \mathcal{Y}$ to the random oracle H;*
     - *If $y \notin \mathcal{V}$, then the challenger adds node y to $\mathcal{V}$.*

[6] If the Rijndael-based commitment is employed for Com_1, then we also require the ideal-cipher model (ICM).

– *The adversary queries $x \in \mathcal{X}$ to the random oracle H; Let $H(x) = y$.*
  - *If $y \in \mathcal{V}$ but there is no edge $(x', y) \in \mathcal{E}$, then the adversary wins (because it breaks onewayness).*
  - *If an edge exists $(x', y) \in \mathcal{E}$ with $x' \neq x$, then the adversary wins (because it finds a collision).*
  - *Else, the challenger adds nodes x and y to $\mathcal{V}$ and an edge $e = (x, y)$ from x to y to $\mathcal{E}$.*

If the adversary makes at most Q queries to H, then the adversary's advantage is at most $Q^2/|\mathcal{Y}|$.

Extractable-binding property. Hereafter, we start by showing the extractable binding property of VOLECommit.

Lemma 7.2 (Extractable Binding). *Let $\mathcal{A}$ be an adversary that makes $q_{\mathsf{com},1}$ and $q_{\mathsf{com},2}$ queries to Com_1 and Com_2, respectively, where Com_1 and Com_2 are modeled as random oracles.* [7] *We consider the following security game, which uses an extractor Ext defined later:*

1. $(h_{\mathsf{com}}, \mathsf{salt}) \leftarrow \mathcal{A}^{\mathsf{Com}_1, \mathsf{Com}_2}(1^\lambda, \mathtt{commit})$.
2. $(\boldsymbol{u}_e^*, \boldsymbol{V}_e^*)_{e \in [\tau]} \leftarrow \mathsf{Ext}(\mathcal{E}^1_{\mathsf{Com}}, \mathcal{E}^2_{\mathsf{Com}}, h_{\mathsf{com}}, \mathsf{salt})$, *where $\mathcal{E}^1_{\mathsf{Com}}$ and $\mathcal{E}^2_{\mathsf{Com}}$ are the lists for the random oracles Com_1 and Com_2.*
3. $(\mathsf{ch}_3, \mathsf{pdecom}, \mathsf{com}) \leftarrow \mathcal{A}^{\mathsf{Com}_1, \mathsf{Com}_2}(1^\lambda, \mathtt{open})$.
4. $\boldsymbol{i} \leftarrow \mathsf{ChallDec}(\mathsf{ch})$.
5. *For $e \in [\tau]$,* $(\delta_{e,0}, \ldots, \delta_{e,\mu_e - 1}) \coloneqq \mathsf{BitDec}(\boldsymbol{i}^*[e])$.
6. $(\bar{h}_{\mathsf{com}}, \boldsymbol{Q}_0, \ldots, \boldsymbol{Q}_{\tau-1}) \leftarrow \mathsf{VOLEReconstruct}^{\mathsf{Com}_1, \mathsf{Com}_2}(\boldsymbol{i}, \mathsf{pdecom}, \mathsf{com}, \mathsf{salt})$.
7. *Output* False *if*
   (a) $\bar{h}_{\mathsf{com}} = \bot$ *or* $\bar{h}_{\mathsf{com}} \neq h_{\mathsf{com}}$*; or*
   (b) $\boldsymbol{Q}_e = \boldsymbol{V}_e^* \oplus [\delta_{e,0} \boldsymbol{u}_e^* \cdots \delta_{e,\mu_e} \boldsymbol{u}_e^*]$ *for all $i \in [\tau]$.*
8. *Output* True *otherwise.*

Let $\mathsf{AdvExt}^{\mathsf{VOLE}} = \Pr[\mathcal{A}$ *wins*$]$ *be $\mathcal{A}$'s advantage. We have*

$$\mathsf{AdvExt}^{\mathsf{VOLE}} \leq (q_{\mathsf{com},1} + N)^2 / 2^{2\lambda} + (q_{\mathsf{com},2} + 1)^2 / 2^{2\lambda}.$$

Proof. The proof is essentially the same as that in FAEST's specification. We define the straight-line extractor Ext as follows:

1. Given h_{com}, find a preimage $\{\mathsf{com}_{e,i}\}$ under Com_2 from the list $\mathcal{E}^2_{\mathsf{Com}}$. If there is no preimage or multiple ones, then output $\bot$ and abort.
2. For each $e \in [\tau]$ and $i \in [N_e]$: find preimages $\mathsf{seed}_{e,i}$ of $\mathsf{com}_{e,i}$ from $\mathcal{E}^1_{\mathsf{Com}}$. If there is no such preimage, then set $\mathsf{seed}_{e,i} = \bot$. If there are multiple preimages, then output $\bot$ and abort.
3. For each $e \in [\tau]$, compute $(\boldsymbol{u}_e, \boldsymbol{V}_e)$ as follows:
   - Case 1: If Ext finds all preimages $\mathsf{seed}_{e,i}$ for e, then it computes $\boldsymbol{V}_e$ and $\boldsymbol{u}_e$ honestly via ConvertToVOLE.

[7] If the Rijndael-based commitment is employed for Com_1, then we also require the ideal-cipher model (ICM).

- Case 2: If a single preimage is missing, then set $\Delta_e = j^*$. It then computes $(\boldsymbol{u}_e, \boldsymbol{q}_{e,0}, \ldots, \boldsymbol{q}_{e,\mu_e-1})$ via ConvertToVOLE with permuted seed with Δ_e and sets $\boldsymbol{Q}_e = [\boldsymbol{q}_{e,0} \cdots \boldsymbol{q}_{e,\mu_e-1}]$. It sets $\boldsymbol{V}_e \coloneqq \boldsymbol{Q}_e \oplus [\delta_{e,0}\boldsymbol{u}_e \cdots \delta_{e,\mu_e-1}\boldsymbol{u}_e]$.

- Case 3: If multiple preimages are missing, then output $\perp$ and abort.

4. Output $(\boldsymbol{u}_e, \boldsymbol{V}_e)_{e\in[\tau]}$.

5. If it fails to extraction, then it adds the image that missed the preimage to $\mathcal{V}^1_{\mathsf{Com}}$ or $\mathcal{V}^2_{\mathsf{Com}}$.

Let Fail be the event that the extractor fails to output $(\boldsymbol{u}_e, \boldsymbol{V}_e)_e$. Because of the random oracle graph game, we have

$$\Pr[\mathsf{Fail}] \leq (q_{\mathsf{com},1} + N)^2/2^{2\lambda} + (q_{\mathsf{com},2} + 1)^2/2^{2\lambda},$$

where we add N for $q_{\mathsf{com},1}$ since we have at most N commitments missing preimages.

If $\neg\mathsf{Fail}$, then h_{com} uniquely determines $(\mathsf{com}_{e,i})_{e,i}$ and each $\mathsf{com}_{e,i}$ uniquely determines $\mathsf{seed}_{e,i}$ for $e \in [\tau]$ and $i \in [N_e]$. (NOTE: For each $e \in [\tau]$, at most one of $\mathsf{seed}_{e,i}$ can be $\perp$.) In both cases (case 1 or case 2), the extraction is perfect, as explained in FAEST's specification. Thus, the extractor's failing probability is at most $\left((q_{\mathsf{com},1} + N)^2 + (q_{\mathsf{com},2} + 1)^2\right) \cdot 2^{-2\lambda}$.

Multi-hiding property. We next show the multi-hiding property of VOLECommit. To do so, we first define the simulation algorithm, SimVOLECommit.

Algorithm 7.1: SimVOLECommit$\left(\boldsymbol{i}, \mathsf{salt}, \hat{\ell}\right)$

Public information and inputs

Public information: A number of iterations τ, a number of parties $N = \sum_{e=0}^{\tau-1} N_e$, $\hat{k}_e = \log_2(N_e)$, $\rho = k_0\tau_0 + k_1\tau_1$
Prover's input: $\boldsymbol{i}$ and salt and $\hat{\ell}$. We assume that $\boldsymbol{i}$ is accepted.

Output

A commitment $\boldsymbol{h}_{\mathsf{com}} \in \{0,1\}^{2\lambda}$, a sibling path pdecom, unopened commitments $(\mathsf{com}_{e,\boldsymbol{i}[e]})_e$, VOLE corrections $(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1})$, and VOLE correlation secrets $\boldsymbol{u}$

1 : hidden $\coloneqq \{N-1+\psi(e,\boldsymbol{i}[e]) : e \in [0,\tau)\}$
2 : opened $\coloneqq \{N-1,\ldots,2N-2\} \setminus$ hidden
3 : **for** i from $N-2$ downto 0 **do**
4 : **if** $2i+1 \in$ opened and $2i+2 \in$ opened **then**
5 : opened $=$ opened $\cup \{i\}$
6 : **endfor**
7 : $\mathsf{nodes}[0], \ldots, \mathsf{nodes}[2N-2] = \emptyset, \ldots, \emptyset$
8 : **for** $i \in [0, N-1)$ **do**
9 : **if** $2i+1 \in$ opened and $2i+2 \in$ opened **then**
10 : $(\mathsf{nodes}[2i+1], \mathsf{nodes}[2i+2]) \leftarrow \mathsf{PRG}(\mathsf{nodes}[i], \mathsf{salt})$
11 : **else**
12 : $(\mathsf{nodes}^{(j)}[2i+1], \mathsf{nodes}^{(j)}[2i+2]) \leftarrow \{0,1\}^{2\lambda}$
13 : **for** $e \in [0,\tau)$ **do**
14 : **for** $i \in [0, N_e)$ **do**
15 : $\mathsf{seed}_{e,i} = \mathsf{nodes}[N-i+\psi(e,i)]$
16 : **if** $i = \boldsymbol{i}[e]$ **then**
17 : $\mathsf{com}_{e,i} \leftarrow\!\!\$\ \{0,1\}^{2\lambda}$
18 : **else**
19 : $\mathsf{com}_{e,i} \leftarrow \mathsf{Com}_1(\mathsf{salt}, e, i, \mathsf{seed}_{e,i})$
20 : $h_{\mathsf{com}} \leftarrow \mathsf{Com}_2(\mathsf{salt}, \{\mathsf{com}_{e,i}\})$
21 : $\mathsf{decom} = (\mathsf{nodes}, (\mathsf{com}_{e,i}))$
22 : $(\mathsf{pdecom}, (\mathsf{com}_{e,\boldsymbol{i}[e]})_e) \leftarrow \mathsf{VC.Open}(\mathsf{decom}, \boldsymbol{i})$
23 : $(\boldsymbol{u}, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}) \leftarrow\!\!\$\ (\mathbb{F}_2^{\hat{\ell}})^\tau$
24 : **return** $(h_{\mathsf{com}}, \mathsf{pdecom}, (\mathsf{com}_{e,\boldsymbol{i}[e]})_e, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}, \boldsymbol{u})$.

Lemma 7.3 (Multi Hiding). *Let us consider the following Q-multi-hiding game for* VOLECommit *between an adversary $\mathcal{A}$ and a challenger. Let $b^* \in \{0,1\}$.*

1. *For $j \in [Q]$:*
   (a) *Take $(r^{(j)}, \mathsf{salt}^{(j)})$ uniformly at random.*
   (b) *Take a random challenge $\mathsf{ch}_3^{(j)} \leftarrow \{0,1\}^{\tau_0\kappa_0+\tau_1\kappa_1+w}$ until it is accepted:*

i. *If* $\mathsf{ch}_3^{(j)}[\tau_0\kappa_0+\tau_1\kappa_1 : \tau_0\kappa_0+\tau_1\kappa_1+w-1] \neq 0^w$*, then re-sample.*

ii. *Compute* $\boldsymbol{i}^{(j)}$ *by* $\mathsf{ChallDec}(\mathsf{ch}_3^{(j)}[0 : \tau_0\kappa_0+\tau_1\kappa_1-1])$*. If the indices lead to* $\mathsf{len}(\mathsf{revealed}) > T_{\mathsf{open}}$*,* [8] *then re-sample.*

(c) *If* $b^*=0$*:*

i. *Compute* $(h_{\mathsf{com}}^{(j)}, \mathsf{decom}^{(j)}, \boldsymbol{c}_1^{(j)}, \ldots, \boldsymbol{c}_{\tau-1}^{(j)}, \boldsymbol{u}^{(j)}, \boldsymbol{V}^{(j)}) \leftarrow \mathsf{VOLECommit}(r^{(j)}, \mathsf{salt}^{(j)}, \hat{\ell})$*, where* $\mathsf{decom}^{(j)} = (\mathsf{nodes}^{(j)}, (\mathsf{com}_{e,i}^{(j)}))$.

ii. *Compute* $\left(\mathsf{pdecom}^{(j)}, (\mathsf{com}_{e,\boldsymbol{i}^{(j)}[e]}^{(j)})\right) \leftarrow \mathsf{VC.Open}(\mathsf{decom}^{(j)}, \boldsymbol{i}^{(j)})$.

(d) *If* $b^*=1$*:*

i. *Compute* $(h_{\mathsf{com}}^{(j)}, \mathsf{pdecom}^{(j)}, (\mathsf{com}_{e,\boldsymbol{i}^{(j)}[e]}^{(j)})_e, \boldsymbol{c}_1^{(j)}, \ldots, \boldsymbol{c}_{\tau-1}^{(j)}, \boldsymbol{u}^{(j)}) \leftarrow \mathsf{SimVOLECommit}(\boldsymbol{i}^{(j)}, \mathsf{salt}^{(j)})$.

2. $b \leftarrow \mathcal{A}\Big((h_{\mathsf{com}}^{(j)}, \mathsf{pdecom}^{(j)}, (\mathsf{com}_{e,\boldsymbol{i}^{(j)}[e]}^{(j)}), \boldsymbol{c}_1^{(j)}, \ldots, \boldsymbol{c}_{\tau-1}^{(j)}, \boldsymbol{u}^{(j)}, \mathsf{ch}_3^{(j)})_{j\in[Q]}\Big)$.
3. *Output* True *if* $b=b^*$*; otherwise, output* False.

Then, the adversary's advantage

$$\mathsf{AdvHide}^{\mathsf{VOLE}}[Q] := |\Pr[\mathcal{A} \text{ wins} \mid b^*=0] - \Pr[\mathcal{A} \text{ wins} \mid b^*=1]|$$

is at most $\hat{k}\cdot\mathsf{AdvPRG}^{\mathsf{PRG}_1}[Q,\tau] + \mathsf{AdvPRF}^{\mathsf{PRG}_2,\mathsf{Com}_1}[Q,\tau]$*, where* $\hat{k} = \lceil \log_2(N) \rceil + 1$.

Proof. We use the standard GGM tree argument.

G_0**:** This is the original game with $b^*=0$. Thus, we have $\Pr[W_0] = \Pr[\mathcal{A} \text{ wins} \mid b^*=0]$.

G_1**:** We gradually replace GGM trees constructed in $\mathsf{VC.Commit}$ by following $\boldsymbol{i}^{(j)}$. We define $\mathsf{G}_{1,k}$ for $k=0,\ldots,\hat{k}$ as follows:

$\mathsf{G}_{1,k}$**:** We modify $\mathsf{VC.Commit}$ invoked from $\mathsf{VOLECommit}(r^{(j)}, \mathsf{salt}^{(j)}, \hat{\ell})$ as follows:

1. Define $\mathsf{opened}^{(j)} := \{N-1,\ldots,2N-2\} \setminus \{N-1+\psi(e, \boldsymbol{i}^{(j)}[e]) : e \in [\tau]\}$.
2. For i from $N-2$ downto 0: if both $2i+1, 2i+2 \in \mathsf{opened}^{(j)}$ then $\mathsf{opened}^{(j)} := \mathsf{opened}^{(j)} \cup \{i\}$.
3. For $i \in [N-1]$:
   (a) If $i \in [2^k]$ and $i \notin \mathsf{opened}^{(j)}$, then $(\mathsf{nodes}^{(j)}[2i+1], \mathsf{nodes}^{(j)}[2i+1]) \leftarrow \{0,1\}^{2\lambda}$;
   (b) else, $(\mathsf{nodes}^{(j)}[2i+1], \mathsf{nodes}^{(j)}[2i+1]) \leftarrow \mathsf{PRG}_1(\mathsf{salt}, \mathsf{nodes}[i] :: 2\lambda)$.

We note that the width of co-path is at most τ. Hence, this introduces the difference at most $\mathsf{AdvPRG}^{\mathsf{PRG}_1}[Q,\tau]$.

We note that, at the final game $\mathsf{G}_{1,\hat{k}}$, the hidden $\mathsf{seed}_{e,\boldsymbol{i}^{(j)}[e]}^{(j)} = \mathsf{nodes}^{(j)}[N-1+\psi(e,\boldsymbol{i}^{(j)}[e])]$ are chosen uniformly at random.

G_2**:** We next replace $(\mathsf{com}_{e,\boldsymbol{i}^{(j)}[e]})^{(j)}$ with random string and $\boldsymbol{r}_{0,\boldsymbol{i}^{(j)}[e]}^{(j)}$ with random vector for all $e \in [0,\tau)$ and $j \in [Q]$ in the computation of $\mathsf{ConvertToVOLE}(N_e, (\mathsf{seed}_{e,i}^{(j)})_{i\in[0,N_e)}, \mathsf{salt}^{(j}$ invoked by $\mathsf{VOLECommit}(r^{(j)}, \mathsf{salt}^{(j)}, \hat{\ell})$. This is justified by the joint PRF security of PRG_2 in $\mathsf{ConvertToVOLE}$ and Com_1 since $\mathsf{seed}_{e,\boldsymbol{i}^{(j)}[e]}^{(j)}$ are hidden from the adversary. Thus, the difference is at most $\mathsf{AdvPRF}^{\mathsf{PRG}_2,\mathsf{Com}_1}[Q,\tau]$.

[8] See $\mathsf{VC.Open}(\mathsf{decom}, \boldsymbol{i}^*)$. Note that decom is independent of the computation of $\mathsf{revealed}$.

G_3**:** Next, we replace $\boldsymbol{u}_e^{(j)}$ for all e and j with random vectors in the computation of $\mathsf{ConvertToVOLE}(N_e, (\mathsf{seed}_{e,i}^{(j)})_{i\in[0,N_e)}, \mathsf{salt}^{(j)}, \hat{\ell})$ invoked by $\mathsf{VOLECommit}(r^{(j)}, \mathsf{salt}^{(j)}, \hat{\ell})$. Since $\boldsymbol{u}_e^{(j)} = \sum_{i\in[0,N_e):i\neq \boldsymbol{i}^{(j)}[e]} \mathsf{PRG}_2(\mathsf{salt}^{(j)}, \mathsf{seed}_{e,i}^{(j)} :: \hat{\ell}) \oplus \boldsymbol{r}_{0,\boldsymbol{i}^{(j)}[e]}^{(j)}$, this does not change the distribution from the previous game and we have $\Pr[W_2] = \Pr[W_3]$. Now, in this game, every $\boldsymbol{u}_e^{(j)}$ for $e \in [\tau]$ is random. Thus, $\boldsymbol{c}_i^{(j)} = \boldsymbol{u}_0^{(j)} \oplus \boldsymbol{u}_i^{(j)}$ is also random and $(\boldsymbol{u}^{(j)}, \boldsymbol{c}_1^{(j)}, \ldots, \boldsymbol{c}_{\tau-1}^{(j)})$ is random. Therefore, this game is equivalent to the game for $b^* = 1$, and we have $\Pr[W_3] = \Pr[\mathcal{A} \text{ wins} \mid b^* = 1]$.

Wrapping up, we have $\hat{k} \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_1}[Q, \tau] + \mathsf{AdvPRF}^{\mathsf{PRG}_2,\mathsf{Com}_1}[Q, \tau]$ as the upper bound.

Security proofs for the EUF-NMA security. We here consider the security of the slightly modified signature scheme denoted by PERK'. Concretely speaking, we replace "$(\mathsf{mseed}, \mathsf{salt}) \leftarrow \mathsf{H}_3(\mathsf{sk}||\mu||\mathsf{rand})$" in line 3 of PERK.Sign with "$(\mathsf{mseed}, \mathsf{salt}) \leftarrow\!\!\$\ \{0,1\}^{\lambda+2\lambda}$". Our proof mainly follows that in FAEST's specification, but we modify several points to adopt their proof in our setting, e.g., showing the formal proof for grinding and optimizations.

Theorem 7.1 (EUF-NMA security in the ROM). *Let $\mathcal{B}$ be an adversary against the EUF-NMA security of PERK'. Let $q_{\mathsf{com},1}$, $q_{\mathsf{com},2}$, q_1, $q_{2,1}$, $q_{2,2}$, $q_{2,3}$, and q_4 be the number of queries $\mathcal{B}$ made to Com_1, Com_2, H_1, H_2^1, H_2^2, H_2^3, and H_4, respectively. Suppose that VOLEHash is an ε_v-almost universal hash family (Lemma 4.1). We have another adversary $\mathcal{C}$ against the PKP assumption (Definition 2.10) such that*

$$\begin{aligned}\mathsf{Adv}_{\mathcal{B}}^{euf\text{-}nma} \leq\ & \mathsf{AdvOW}_{\mathcal{C}} + \mathsf{AdvPR}^{\mathsf{ExpandKernelVector}} + \mathsf{AdvPR}^{\mathsf{ExpandPermutation}} \\ & + (q_{\mathsf{com},1} + q_{\mathsf{com},2} + q_1 + (N+2)q_{2,1} + 2q_{2,2} + q_{2,3})^2 \cdot 2^{-2\lambda} \\ & + q_{2,1}\binom{\tau}{2} \cdot \epsilon_v + q_{2,2} \cdot 2^{-\rho} + (q_4 + q_{2,2})^2 \cdot 2^{-(3\lambda+64)} \\ & + q_{2,3} d \cdot 2^{-(\tau_0\kappa_0 + \tau_1\kappa_1 + w)} + 2^{-\lambda+1}.\end{aligned}$$

The running time of $\mathcal{C}$ is about that of $\mathcal{B}$.

We follow the games defined in FAEST's specification and show the bounds of the differences between the games.

G_1**:** The original EUF-NMA game. We have

$$\Pr[W_1] = \mathsf{Adv}_{\mathcal{B}}^{\text{euf-nma}}.$$

We note that the random oracles and commitment are implemented by lazy sampling and the lists, $\mathcal{E}_{\mathsf{Com}}^1$, $\mathcal{E}_{\mathsf{Com}}^2$, $\mathcal{E}_1$, $\mathcal{E}_2^1$, $\mathcal{E}_2^2$, $\mathcal{E}_2^3$, and $\mathcal{E}_4$. Here, we use $\mathcal{E}$ because they will be edge sets of the random oracle graphs.

G_2: In this game, we consider the random oracle graph games. To do so, we additionally consider the vertex sets $\mathcal{V}^1_{\mathsf{Com}}$, $\mathcal{V}^2_{\mathsf{Com}}$, $\mathcal{V}_1$, $\mathcal{V}^1_2$, $\mathcal{V}^2_2$, and $\mathcal{V}^3_2$. If the adversary wins the random oracle graph game, then we abort.
Let Fail_i be the event that the adversary wins the random oracle graph game. For any G_i, we have $\Pr[W_i] = \Pr[\mathsf{Fail}_i] \cdot \Pr[W_i \mid \mathsf{Fail}_i] + \Pr[\neg\mathsf{Fail}_i] \cdot \Pr[W_i \mid \neg\mathsf{Fail}_i] \leq \Pr[\mathsf{Fail}_i] + \Pr[W_i \mid \neg\mathsf{Fail}_i]$. We note that if Fail_2 does not occur, then G_2 is equivalent to G_1. Thus, we have $\Pr[W_1] = \Pr[W_2 \mid \neg\mathsf{Fail}_2]$.

G_3: We next use *the extractor* Ext in the proof of Lemma 7.2 on every query $(\tilde{\mu}, h_{\mathsf{com}}, ..., \mathsf{salt})$ to H^1_2 to obtain $(\boldsymbol{u}_e, \boldsymbol{V}_e)_{e\in[\tau]}$. If the adversary's forgery is valid but one of the extracting test fails, then the adversary *loses*. Because of the modification introduced G_2, we have $\Pr[W_2 \mid \neg\mathsf{Fail}_2] = \Pr[W_3 \mid \neg\mathsf{Fail}_3]$. But, the number of queries blows up because of the queries the extractor made. We have

$$\bar{q}_{\mathsf{com},1} = q_{\mathsf{com},1} + Nq_{2,1} \text{ and } \bar{q}_{\mathsf{com},2} = q_{\mathsf{com},2} + q_{2,1},$$

since Ext is invoked at most $q_{2,1}$ times and each invocation adds at most N queries or one query to Com_1 or Com_2, respectively.

G_4: We next modify H^1_2; if the query is extractable, then it runs the VOLE consistency checks with extracted $(\boldsymbol{u}_e, \boldsymbol{V}_e)_{e\in[\tau]}$. While we omit the details of the check, FAEST's specification shows that the probability that the consistency check fails $\epsilon_v\binom{\tau}{2}$, where ϵ_v is universality of VOLEHash, by using [BBD+23c, Thm.2]. Since we take VOLEHash from FAEST, we have the same probability. Thus, we have

$$\Pr[W_3 \mid \neg\mathsf{Fail}_3] \leq \Pr[W_4 \mid \neg\mathsf{Fail}_4] + q_{2,1}\binom{\tau}{2}\epsilon_v,$$

where ϵ_v is universality of VOLEHash.

G_5: We next modify H^2_2 to force the chain of hash values. On query $(\mathsf{ch}_1, h_V, ...)$ to H^2_2, we do as follows:

- If ch_1 has no preimage, then query ch_1 under H^1_2; that is, if $\mathsf{ch}_1 \notin \mathcal{V}^1_2$, then add ch_1 to $\mathcal{V}^1_2$.
- If h_V has no preimage, then query h_V under H_1; that is, if $h_V \notin \mathcal{V}_1$, then add h_V to $\mathcal{V}_1$.

We have $\Pr[W_4 \mid \neg\mathsf{Fail}_4] = \Pr[W_5 \mid \neg\mathsf{Fail}_5]$. By this modification, the bound of the random oracle queries to the random oracle graph game are

$$\bar{q}_1 = q_1 + q_{2,2} \text{ and } \bar{q}_{2,1} = q_{2,1} + q_{2,2}.$$

G_6: We modify H^2_2 as follows: On each new query $(\mathsf{ch}_1, h_V, ...)$ to H^2_2, if the query related to ch_1 is extractable and VOLE-consistent, then do as follows:

1. Take a random $\mathsf{ch}_2 \leftarrow\!\!\$\ \{0,1\}^{3\lambda+64}$. If ch_2 is already queried to H_4, then it aborts.
2. If the ZK soundness check below fails, then abort.
   (a) Let $\boldsymbol{\alpha} \xleftarrow{\$,\mathsf{ch}_2} \mathbb{F}^{cn+m}_{2^\rho}$. That is, we take a random sample $\boldsymbol{\alpha} \leftarrow\!\!\$\ \mathbb{F}^{cn+m}_{2^\rho}$ and put $(\mathsf{ch}_2, \boldsymbol{\alpha})$ to $\mathcal{E}_4$ for H_4.

(b) Compute $\boldsymbol{e} \in \mathbb{F}_{2^\rho}^{cn+m}$, which is a vector consisting of the degree d coefficients of $f_j(X)$ for $j = 0, \ldots, cn+m-1$ constructed from the extracted witness.
(c) If $\boldsymbol{e} \neq \boldsymbol{0}$ but $\sum_{j=0}^{cn+m-1} \alpha_j e_j = 0$, then the output "fail". Otherwise, output "success".

3. Otherwise, look up a preimage $\tilde{\boldsymbol{V}}$ of h_V. If a preimage does not exist, then abort.
4. Otherwise, return ch_2.

On the collision test for step 1, we have a bound $(q_4 + q_{2,2})^2/2^{3\lambda+64}$. If $\boldsymbol{\alpha}$ is uniformly at random, then the probability that the check in step 2 fails is at most $\epsilon_{zk} = 1/2^\rho$.
Taking a union bound, we have

$$\Pr[W_5 \mid \neg\mathsf{Fail}_5] \leq \Pr[W_6 \mid \neg\mathsf{Fail}_6] + q_{2,2} \cdot 2^{-\rho} + (q_4 + q_{2,2})^2 \cdot 2^{-(3\lambda+64)}.$$

G_7**:** We next modify H_2^3 as follows: On query $(\mathsf{ch}_2, [\![a]\!], \mathsf{ctr})$ to H_2^3, we do as follows:

- If ch_2 has no preimage, then query ch_2 under H_2^2; that is, if $\mathsf{ch}_2 \notin \mathcal{V}_2^2$, then add ch_2 to $\mathcal{V}_2^2$.

We have $\Pr[W_6 \mid \neg\mathsf{Fail}_6] = \Pr[W_7 \mid \neg\mathsf{Fail}_7]$. The number of random oracle queries is

$$\bar{q}_{2,2} = q_{2,2} + q_{2,3}.$$

G_8**:** We modify the handling H_2^3 on a query $(\mathsf{ch}_2, [\![a]\!], \mathsf{ctr})$ as follows:

1. If ch_2 has no preimage, then abort. Otherwise, let $(\mathsf{ch}_1, h_V, ...)$ be the preimage of ch_2.
2. If h_V has no preimage, then abort.
3. Otherwise, extract $\boldsymbol{u}$ and $\boldsymbol{V}$, and define the witness $\boldsymbol{w}$.
4. Sample $\mathsf{ch}_3 \leftarrow \{0,1\}^{\tau_0\kappa_0+\tau_1\kappa_1+w}$.
5. If $\boldsymbol{w}$ doesn't satisfy the constraints and the bad event occurs, then abort the game. The bad event is the event that, for uniformly random $\mathsf{ch}_3 \in \{0,1\}^{\tau_0\kappa_0+\tau_1\kappa_1+w}$ in step 3, 1) ch_3 passes the verification test (T_{open} and 0^w check) and 2) for uniformly random $\mathsf{ch}_3 \in \{0,1\}^{\tau_0\kappa_0+\tau_1\kappa_1+w}$, we have $\Delta = \Delta'$, where Δ is computed from ch_3 and Δ' is computed from the query $[\![a]\!]$ by the adversary.

The probability 1) is $|\#\text{accepted challenges}|/2^{\tau_0\kappa_0+\tau_1\kappa_1+w}$ and the probability 2) is at most $d/|\#\text{accepted challenges}|$ since $[\![a]\!]$ is treated as $(d-1)$-degree polynomial $a_{d-1}X^{d-1} + \cdots + a_1X + a_0$ and the verification algorithm checks if $q_f + \sum_{i=0}^{d-2} q_{s_i} \cdot \Delta^i = \sum_{i=0}^{d-1} a_i\Delta^i$ in V.CheckZero invoked by V.Check-PKP.
Thus, we have the bound

$$\begin{aligned}\Pr[W_7 \mid \neg\mathsf{Fail}_7] &\leq \Pr[W_8 \mid \neg\mathsf{Fail}_8] + q_{2,3} \cdot \frac{d}{|\#\text{accepted challenges}|} \cdot \frac{|\#\text{accepted challenges}|}{2^{\tau_0\kappa_0+\tau_1\kappa_1+w}} \\ &= \Pr[W_8 \mid \neg\mathsf{Fail}_8] + q_{2,3} \cdot d \cdot 2^{-(\tau_0\kappa_0+\tau_1\kappa_1+w)}.\end{aligned}$$

G_9**:** We next modify the key-generation algorithm. In the experiment, the key-generation algorithm chooses $\boldsymbol{x}' \xleftarrow{\$} \ker(\boldsymbol{H})$ and $\pi \xleftarrow{\$} \mathcal{S}_n$ instead of computing $\boldsymbol{x}' \leftarrow \mathsf{ExpandKernelVector}(\mathsf{ker}_{\mathsf{seed}}, \boldsymbol{H})$ and $\pi \leftarrow \mathsf{ExpandPermutation}(\mathsf{perm}_{\mathsf{seed}})$.

$$\Pr[W_8 \mid \neg\mathsf{Fail}_8] \le \Pr[W_9 \mid \neg\mathsf{Fail}_9] + \mathsf{AdvPR}^{\mathsf{ExpandKernelVector}} + \mathsf{AdvPR}^{\mathsf{ExpandPermutation}}.$$

We then discuss the evaluation of $\Pr[\mathsf{Fail}_9]$ and the reduction to the PKP problem.

Evaluation of $\Pr[\mathsf{Fail}_9]$. The numbers of queries to the random oracles are now

$$\bar{q}_{\mathsf{com},1} = q_{\mathsf{com},1} + q_{2,1}N, \bar{q}_{\mathsf{com},2} = q_{\mathsf{com},2} + q_{2,1},$$
$$\bar{q}_1 = q_1 + q_{2,2}, \bar{q}_{2,1} = q_{2,1} + q_{2,2}, \text{ and } \bar{q}_{2,2} = q_{2,2} + q_{2,3}.$$

Thus, the advantage of the random oracle games is at most

$$\begin{aligned}\Pr[\mathsf{Fail}_9] &\le \bar{q}^2_{\mathsf{com},1}/2^{2\lambda} + \bar{q}^2_{\mathsf{com},2}/2^{2\lambda} + \bar{q}^2_1/2^{2\lambda} + \bar{q}^2_{2,1}/2^{5\lambda+64} + \bar{q}^2_{2,2}/2^{2\lambda} \\ &\le (q_{\mathsf{com}} + q_{\mathsf{com},2} + q_1 + (N+2)q_{2,1} + 2q_{2,2} + q_{2,3})^2/2^{2\lambda}.\end{aligned}$$

Reduction to PKP.

We construct a reduction algorithm $\mathcal{C}$ using $\mathcal{B}$ in the final game G_9 conditioned on that Fail_9 does not occur. The reduction algorithm $\mathcal{C}$ against our PKP assumption (Definition 2.10) is defined as follows:

1. It is given a random seed $\boldsymbol{H}_{\mathsf{seed}} \xleftarrow{\$} \{0,1\}^\lambda$ and $\boldsymbol{x}' \in \mathbb{F}_q^n$, where $\boldsymbol{M} \leftarrow \mathsf{ExpandMatrixM}(\boldsymbol{H}_{\mathsf{seed}})$, $\boldsymbol{H} := [\boldsymbol{I}_m\ \boldsymbol{M}] \in \mathbb{F}_q^{m\times n}$, $\boldsymbol{x}' \xleftarrow{\$} \ker(\boldsymbol{H})$, $\pi \xleftarrow{\$} \mathcal{S}_n$, and $\boldsymbol{x} := \pi^{-1}(\boldsymbol{x}')$. It wants to output $\tilde{\pi}$ such that $\boldsymbol{H}(\tilde{\pi}(\boldsymbol{x})) = \boldsymbol{0}$.
2. It sets $\mathsf{pk} = (\boldsymbol{H}_{\mathsf{seed}}, \boldsymbol{x})$ and run $\mathcal{B}$ in G_9.
3. Finally, $\mathcal{B}$ outputs (m^*, σ^*) and stops. If $\mathcal{B}$ wins and Fail_9 does not occur, then $\mathcal{C}$ extracts the witness $\boldsymbol{P}$ from the random oracle queries, and outputs $\tilde{\pi}$ corresponding to $\boldsymbol{P}$.

Since the simulation of $\mathcal{C}$ is perfect, if $\mathcal{B}$ wins the game, then $\mathcal{C}$ can extract the witness $\boldsymbol{P}$ from $\mathcal{B}$'s queries to the random oracles. Thus, we have

$$\Pr[W_9 \mid \neg\mathsf{Fail}_9] \le \mathsf{AdvOW}_{\mathcal{C}}.$$

Security proofs for the SUF-CMA security. We follow the proof in FAEST's specification while we consider an optimized version and we will repair some gaps in the original proof. Thus, we need to consider *a special form of Fiat-Shamir with aborts*, which only samples ch_3 instead of the whole. Hence, we enhanced the repaired proofs for 3-round Fiat-Shamir with aborts in Devevey *et al.* [DFPS23] and Barbosa *et al.* [BBD+23a]. To consider SUF-CMA security, we also employ the techniques in [KX24b].

Theorem 7.2 (SUF-CMA security in the ROM). *Let $\mathcal{A}$ be an adversary against the EUF-CMA security of PERK. Let $Q_{\mathsf{prg},1}$, $Q_{\mathsf{com},1}$, $Q_{\mathsf{com},2}$, Q_1, $Q_{2,1}$, $Q_{2,2}$, and $Q_{2,3}$ be the number of queries $\mathcal{A}$ made to PRG_1, Com_1, Com_2, H_1, H_2^1,*

H_2^2, *and* H_2^3, *respectively. Let* Q_{sig} *be the number of queries* $\mathcal{A}$ *made to the signing oracle. Let* $\tilde{Q}_{\mathsf{sig}}$ *be the number of queries the signing oracle made to* H_2^3*. We then have an adversary* $\mathcal{B}$ *against the EUF-NMA security of* PERK′ *satisfying*

$$\begin{aligned}
\mathsf{Adv}^{suf\text{-}cma}_{\mathcal{A}} &\le \mathsf{Adv}^{euf\text{-}nma}_{\mathcal{B}} + Q_{\mathsf{sig}} \cdot \mathsf{AdvPRF}^{\mathsf{H}_3} \\
&+ \mathsf{AdvColl}^{\mathsf{PRG}_1}[Q_{\mathsf{prg},1} + (N-1)Q_{\mathsf{sig}}] \\
&+ \mathsf{AdvColl}^{\mathsf{Com}_1}[Q_{\mathsf{com}} + NQ_{\mathsf{sig}}] + \mathsf{AdvColl}^{\mathsf{Com}_2}[Q_{\mathsf{com},2} + Q_{\mathsf{sig}}] \\
&+ \mathsf{AdvColl}^{\mathsf{H}_1}[Q_1 + 2Q_{\mathsf{sig}}] + \mathsf{AdvColl}^{\mathsf{H}_2^1}[Q_{2,1} + Q_{\mathsf{sig}}] \\
&+ \mathsf{AdvColl}^{\mathsf{H}_2^2}[Q_{2,2} + Q_{\mathsf{sig}}] + \mathsf{AdvColl}^{\mathsf{H}_2^3}[Q_{2,3} + \tilde{Q}_{\mathsf{sig}}] \\
&+ Q_{\mathsf{sig}}(Q_{2,1} + Q_{\mathsf{sig}})2^{-2\lambda} + Q_{\mathsf{sig}}(Q_{2,2} + Q_{\mathsf{sig}})2^{-(5\lambda+64)} + \tilde{Q}_{\mathsf{sig}}(Q_{2,3} + \tilde{Q}_{\mathsf{sig}})2^{-2\lambda} \\
&+ \hat{k} \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_1}[Q_{\mathsf{sig}}, \tau] + \mathsf{AdvJPRF}^{\mathsf{PRG}_2,\mathsf{Com}_1}[Q_{\mathsf{sig}}, \tau] \\
&+ \mathsf{AdvNI}^{\mathsf{Com}_1}[\tau Q_{\mathsf{sig}}].
\end{aligned}$$

Remark 7.1. Let us discuss the order of $\tilde{Q}_{\mathsf{sig}}$. Let $r := |\#\text{ accepted challenges}|/2^{\tau_0\kappa_0+\tau_1\kappa_1+w}$. We let $\tilde{Q}_{\mathsf{sig}} := (2/r) \cdot Q_{\mathsf{sig}}$. This $\tilde{Q}_{\mathsf{sig}}$ gives us $\Pr[\#\text{ success is less than } Q_{\mathsf{sig}}] \le \exp(-Q_{\mathsf{sig}}/4)$ and the right-hand side is negligible if $Q_{\mathsf{sig}} = \omega(\log(\lambda))$. (See e.g., [KX24a, Pf. of Thm.1].)

Corollary 7.1 (EUF-CMA security in the ROM). *Let the parameters be the same as Theorem 7.2. We then have an adversary* $\mathcal{B}$ *against the EUF-NMA security of* PERK′ *satisfying*

$$\begin{aligned}
\mathsf{Adv}^{euf\text{-}cma}_{\mathcal{A}} &\le \mathsf{Adv}^{euf\text{-}nma}_{\mathcal{B}} + Q_{\mathsf{sig}} \cdot \mathsf{AdvPRF}^{\mathsf{H}_3} \\
&+ \mathsf{AdvColl}^{\mathsf{H}_1}[Q_1 + 2Q_{\mathsf{sig}}] + \mathsf{AdvColl}^{\mathsf{H}_2^1}[Q_{2,1} + Q_{\mathsf{sig}}] + \mathsf{AdvColl}^{\mathsf{H}_2^2}[Q_{2,2} + Q_{\mathsf{sig}}] \\
&+ Q_{\mathsf{sig}}(Q_{2,1} + Q_{\mathsf{sig}})2^{-2\lambda} + Q_{\mathsf{sig}}(Q_{2,2} + Q_{\mathsf{sig}})2^{-(5\lambda+64)} + \tilde{Q}_{\mathsf{sig}}(Q_{2,3} + \tilde{Q}_{\mathsf{sig}})2^{-2\lambda} \\
&+ \hat{k} \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_1}[Q_{\mathsf{sig}}, \tau] + \mathsf{AdvJPRF}^{\mathsf{PRG}_2,\mathsf{Com}_1}[Q_{\mathsf{sig}}, \tau].
\end{aligned}$$

In what follows, we denote W_i the event that the adversary wins in G_i.

G_1**:** The original SUF-CMA game. We have

$$\Pr[W_1] = \mathsf{Adv}^{\text{suf-cma}}_{\mathcal{A}}.$$

G_2**:** The signing oracle chooses $(\mathsf{mseed}, \mathsf{salt}) \leftarrow\!\$\ \{0,1\}^{\lambda+2\lambda}$ instead of $(\mathsf{mseed}, \mathsf{salt}) := \mathsf{H}_3(\mathsf{sk}||\tilde{\mu}||\mathsf{rand})$.
Since mseed is kept secret in the whole procedure, we can treat mseed as a one-time secret key of PRF. Thus, the difference between G_1 and G_2 is

$$|\Pr[W_1] - \Pr[W_2]| \le Q_{\mathsf{sig}} \cdot \mathsf{AdvPRF}^{\mathsf{H}_3}.$$

Remark 7.2. If the signature scheme is deterministic, we use the argument for the ROM in [BPS16, Thm.4], which leads to the inequality

$$\Pr[W_1] \le 2 \cdot \Pr[W_2].$$

G_3**:** Next, we introduce a collision check for PRG for the GGM tree, Com_1, Com_2, H_1, H_2^1, H_2^2, and H_2^3. If there is a collision among the queries to those oracles in the whole security game, then the challenger aborts the game. We easily have

$$\begin{aligned}|\Pr[W_2] - \Pr[W_3]| \leq\ & \mathsf{AdvColl}^{\mathsf{PRG}_1}[Q_{\mathsf{prg},1} + (N-1)Q_{\mathsf{sig}}] \\ & + \mathsf{AdvColl}^{\mathsf{Com}_1}[Q_{\mathsf{com}} + NQ_{\mathsf{sig}}] + \mathsf{AdvColl}^{\mathsf{Com}_2}[Q_{\mathsf{com},2} + Q_{\mathsf{sig}}] \\ & + \mathsf{AdvColl}^{\mathsf{H}_1}[Q_1 + 2Q_{\mathsf{sig}}] + \mathsf{AdvColl}^{\mathsf{H}_2^1}[Q_{2,1} + Q_{\mathsf{sig}}] \\ & + \mathsf{AdvColl}^{\mathsf{H}_2^2}[Q_{2,2} + Q_{\mathsf{sig}}] + \mathsf{AdvColl}^{\mathsf{H}_2^3}[Q_{2,3} + \tilde{Q}_{\mathsf{sig}}].\end{aligned}$$

Remark 7.3. If we only consider the EUF-CMA security, then we do not need collision checks for PRG_1 for the GGM tree, Com_1, Com_2, and H_2^3.

G_4**:** The signing oracle programs H_2^1 by choosing $\mathsf{ch}_1 \leftarrow\!\$\ \{0,1\}^{5\lambda+64}$.
Since salt are chosen uniformly at random over $\{0,1\}^\lambda$, a_1's min-entropy is at least 2λ. The adaptive reprogramming technique shows that

$$|\Pr[W_3] - \Pr[W_4]| \leq Q_{\mathsf{sig}}(Q_{2,1} + Q_{\mathsf{sig}}) \cdot 2^{-2\lambda}.$$

NOTE: Since we use 2λ-bit salt to compute ch_1, we do not need additional games that make h_{com} random.

G_5**:** The signing oracle programs H_2^2 by choosing $\mathsf{ch}_2 \leftarrow\!\$\ \{0,1\}^{2\lambda}$.
We write a reduction as follows: The reduction algorithm first computes a_1, chooses ch_1, and a_2. It then queries (ch_1, a_2) to its oracle and obtains ch_2. We have

$$|\Pr[W_4] - \Pr[W_5]| \leq Q_{\mathsf{sig}}(Q_{2,2} + Q_{\mathsf{sig}}) \cdot 2^{-(5\lambda+64)}.$$

NOTE: We do not need to program H_4.

G_6**:** The signing oracle programs H_2^3 by choosing $\mathsf{ch}_3 \leftarrow\!\$\ \{0,1\}^{\tau_0\kappa_0+\tau_1\kappa_1+w}$.
We write a reduction as follows: The reduction algorithm first computes a_1, chooses ch_1, computes a_2, and chooses ch_2, and computes a_3. It then queries (ch_2, a_3) to its oracle and obtains ch_3. Since ch_2 has a min-entropy at least 2λ, we have

$$|\Pr[W_5] - \Pr[W_6]| \leq \tilde{Q}_{\mathsf{sig}}(Q_{2,3} + \tilde{Q}_{\mathsf{sig}}) \cdot 2^{-2\lambda},$$

G_7**:** We then modify the order of sampling;

1. Sample ch_1 and ch_2 uniformly at random.
2. Sample $\mathsf{ch}_{3,0}, \ldots, \mathsf{ch}_{3,B-1} \leftarrow \{0,1\}^{\tau_0\kappa_0+\tau_1\kappa_1+w}$ until it is accepted; define $\mathsf{ch}_3 \coloneqq \mathsf{ch}_{3,B-1}$.
3. Run the prover algorithm.
4. Reprogram $\mathsf{H}_2^1(\mu||h_{\mathsf{com}}||\boldsymbol{c}_1||\cdots||\boldsymbol{c}_{\tau-1}||\mathsf{salt}) \coloneqq \mathsf{ch}_1$.
5. Reprogram $\mathsf{H}_2^2(\mathsf{ch}_1||\tilde{\boldsymbol{u}}||h_V||\boldsymbol{t}) \coloneqq \mathsf{ch}_2$.
6. For $\mathsf{ctr} \in [B]$, reprogram $\mathsf{H}_2^3(\mathsf{ch}_2||[\![a]\!]||\mathsf{ctr}) \coloneqq \mathsf{ch}_{3,\mathsf{ctr}}$.

This is just a conceptual change of the order, and we have

$$\Pr[W_6] = \Pr[W_7].$$

G_8: Next, we make the signature uniformly at random as possible as FAEST's proof. Intuitively speaking, this corresponds to the replacement of the prover algorithms in the signing oracle with the simulator algorithms.

- $\mathsf{G}_{8,1}$: On each signing query, we use $\left(h_{\mathsf{com}}, \mathsf{pdecom}, (\mathsf{com}_{e,\boldsymbol{i}[e]}), \boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}, \boldsymbol{u}\right) \leftarrow \mathsf{SimVOLECommit}(\boldsymbol{i}, \mathsf{salt}, \hat{\ell})$ and adjust $\boldsymbol{V}$ to be consistent with Δ and $\boldsymbol{q}$ induced by ch_3. We stress that SimVOLECommit samples $\boldsymbol{u}$ and $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}$ uniformly at random. Following FAEST's proof, this modification is justified by the multi-hiding property of VOLECommit. Concretely speaking, the upper bound is

$$|\Pr[W_7] - \Pr[W_{8,1}]| \leq \mathsf{AdvHide}^{\mathsf{VOLE}}[Q_{\mathsf{sig}}].$$

  Now, the distribution of $(\boldsymbol{u}, \boldsymbol{V})$ is independent of $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}$.
- $\mathsf{G}_{8,2}$: On each signing query, we sample $\tilde{\boldsymbol{u}}$ and $\tilde{\boldsymbol{V}}$ at random instead of computing VOLEHash; after that, we adjust the last $\ell_{\mathsf{VOLEHashMask}} = \lambda + B$ rows of $\boldsymbol{u}$ and $\boldsymbol{V}$ to make them be consistent with ch_1. Since VOLEHash is $\mathbb{F}_2^{\ell+\lambda}$-hiding (Lemma 1 in FAEST's spec), the distributions are the same. We have

$$\Pr[W_{8,1}] = \Pr[W_{8,2}].$$

- $\mathsf{G}_{8,3}$: Next, on each signing query, we choose $[\![a]\!]$ uniformly at random instead of computing $f_{\mathsf{mask}}(X) + \sum_i \alpha_i f_i(X)$ and adjust the middle ℓ_{CZMask} rows of $\boldsymbol{u}$ and $\boldsymbol{V}$. Correctly speaking, we compute $f_i(X)$, choose $[\![a]\!]$ uniformly at random while it is consistent with q and Δ, then compute $f_{\mathsf{mask}}(X) = [\![a]\!] - \sum_i \alpha_i f_i(X)$; we then adjust $[\![u_{i',k'}]\!]_{i' \in [d-1], k' \in [\rho]}$, which decide the coefficients of $(d-1)$-degree polynomial $f_{\mathsf{mask}}(X)$. Since $[\![u_{i',k'}]\!]_{i' \in [d-1], k' \in [\rho]}$ are hidden from the adversary, the modification in this game does not change anything. Thus, we have

$$\Pr[W_{8,2}] = \Pr[W_{8,3}].$$

- $\mathsf{G}_{8,4}$: Finally, on each signing query, we replace $\boldsymbol{t} = \boldsymbol{w} + \boldsymbol{u}_{[\ell]}$ with random one since $[\![a]\!]$ is independent of $\boldsymbol{w}$. This modification doesn't change the distribution because $\boldsymbol{u}[\ell]$ is chosen uniformly at random and is not used elsewhere. Thus, we have

$$\Pr[W_{8,3}] = \Pr[W_{8,4}].$$

G_9: In this game, the adversary *loses* if there exists $(\mathsf{msg}, \sigma = (\mathsf{ch}_3, a_4)) \in \mathcal{Q}$ such that

- $(\bar{a}_1, \overline{\mathsf{ch}_1}) = (\bar{a}_1^*, \overline{\mathsf{ch}_1^*})$ and $\mathsf{ch}_3 \neq \mathsf{ch}_3^*$; or
- $(\bar{a}_1, \overline{\mathsf{ch}_1}, \ldots, \bar{a}_3, \mathsf{ch}_3) = (\bar{a}_1^*, \overline{\mathsf{ch}_1^*}, \ldots, \bar{a}_3^*, \mathsf{ch}_3^*)$ and $a_4 \neq a_4^*$.

If there is a difference between G_8 and G_9, then the adversary submits new $(\mathsf{msg}^*, (\mathsf{ch}_3^*, a_4^*))$ such that there exists $(\mathsf{msg}, (\mathsf{ch}_3, a_4)) \in \mathcal{Q}$ satisfying either one of the conditions. Let $\bar{h}_{\mathsf{com}}$ and $\bar{h}_{\mathsf{com}}^*$ be the results of VC.Reconstruct from ch_3 and ch_3^* respectively.

Case 1: Suppose that $(\bar{a}_1, \overline{\mathsf{ch}}_1) = (\bar{a}_1^*, \overline{\mathsf{ch}}_1^*)$ and $\mathsf{ch}_3 \neq \mathsf{ch}_3^*$. Since both challenges are accepted, we have $\boldsymbol{i} \neq \boldsymbol{i}^*$, where $\boldsymbol{i} := \mathsf{ChallDec}(\mathsf{ch}_3[0 : \tau_0\kappa_0 + \tau_1\kappa_1 - 1])$ and $\boldsymbol{i}^* := \mathsf{ChallDec}(\mathsf{ch}_3^*[0 : \tau_0\kappa_0 + \tau_1\kappa_1 - 1])$. On the other hand, we have $\bar{h}_{\mathsf{com}} = \bar{h}^*_{\mathsf{com}}$ since $\bar{a}_1 = \bar{a}_1^*$. Since we checked the collision for Com_2, this implies $\{\overline{\mathsf{com}}_{e,i}\} = \{\overline{\mathsf{com}}^*_{e,i}\}$. This breaks the multi-target non-invertibility of Com_1 since the adversary outputs the preimage of $\mathsf{com}_{e,\boldsymbol{i}[e]}$ for some e such that $\boldsymbol{i}[e] \neq \boldsymbol{i}^*[e]$, where $\mathsf{com}_{e,\boldsymbol{i}[e]}$ is chosen uniformly at random in SimVOLECommit. Thus, the difference is at most $\mathsf{AdvNI}^{\mathsf{Com}_1}[\tau Q_{\mathsf{sig}}]$, in which the adversary receives random τQ_{sig} strings and outputs one of preimages of the strings.

Case 2: Next, suppose that $(\bar{a}_1, \overline{\mathsf{ch}}_1, \ldots, \bar{a}_3, \mathsf{ch}_3) = (\bar{a}_1^*, \overline{\mathsf{ch}}_1^*, \ldots, \bar{a}_3^*, \mathsf{ch}_3^*)$ and $a_4 \neq a_4^*$. Note that the condition $\mathsf{ch}_3 = \mathsf{ch}_3^*$ leads to $\boldsymbol{i} = \boldsymbol{i}^*$. Since $(\bar{a}_1, \bar{a}_2, \bar{a}_3) = (\bar{a}_1^*, \bar{a}_2^*, \bar{a}_3^*)$, we have $(\mathsf{pdecom}, \{\mathsf{com}_{e,\boldsymbol{i}[e]}\}_{e\in[\tau]}, \mathsf{ctr}, \mathsf{salt}) \neq (\mathsf{pdecom}^*, \{(\mathsf{com}^*_{e,\boldsymbol{i}[e]}\}_{e\in[\tau]}, \mathsf{ctr}^*, \mathsf{salt}^*)$.

We have four sub-cases:

- Suppose that $\mathsf{pdecom} \neq \mathsf{pdecom}^*$. In the computation of VC.Reconstruct, we have $\mathsf{nodes}[i^+] \neq \mathsf{nodes}^*[i^+]$ for some $i^+ \in \mathsf{revealed}$ calculated from $\boldsymbol{i} = \boldsymbol{i}^*$. We then consider the sub-tree of the GGM tree whose root is i^+. Let I_{all} be the indices of the nodes of the sub-tree and I_{leaves} be the indices of the leaf nodes of the sub-tree.
  We have two cases:
  - If $\{\mathsf{nodes}[i]\}_{i\in I_{\mathsf{leaves}}} = \{\mathsf{nodes}^*[i]\}_{i\in I_{\mathsf{leaves}}}$, we then have a collision in the sub-tree. That is, there is an index $j \in I_{\mathsf{all}}$ such that $\mathsf{nodes}[j] \neq \mathsf{nodes}^*[j]$ but $\mathsf{PRG}_1(\mathsf{nodes}[j], \mathsf{salt} :: 2\lambda) = \mathsf{PRG}_1(\mathsf{nodes}^*[j], \mathsf{salt}^* :: 2\lambda)$. However, this case is already excluded by the collision check for PRG_1 introduced in G_3.
  - If $\{\mathsf{nodes}[i]\}_{i\in I_{\mathsf{leaves}}} \neq \{\mathsf{nodes}^*[i]\}_{i\in I_{\mathsf{leaves}}}$, we have at least one index $j^+ \in I_{\mathsf{leaves}}$ satisfying $\mathsf{nodes}[j^+] \neq \mathsf{nodes}^*[j^+]$. Let (e, i) be a pair induced by j^+ (that is, $j^+ = N - 1 + \psi(e, i)$). If $\mathsf{com}_{e,i} = \mathsf{com}^*_{e,i}$, then we find a collision for Com_1. However, this case is already excluded by the collision check for Com_1 introduced in G_3. Otherwise, if $\mathsf{com}_{e,i} \neq \mathsf{com}^*_{e,i}$, then we find a collision for Com_2 since $\bar{a}_1 = \bar{a}_1^*$ implies $\bar{h}_{\mathsf{com}} = \bar{h}^*_{\mathsf{com}}$. However, this cannot happen since we already exclude this event by the collision check for Com_2 introduced in G_3.
- If $\{\mathsf{com}_{e,\boldsymbol{i}[e]}\}_e \neq \{\mathsf{com}^*_{e,\boldsymbol{i}^*[e]}\}$, then we have $\{\overline{\mathsf{com}}_{e,i}\}_{e,i} \neq \{\overline{\mathsf{com}}^*_{e,i}\}_{e,i}$. However, $\bar{a}_1 = \bar{a}_1^*$ implies $\bar{h}_{\mathsf{com}} = \bar{h}^*_{\mathsf{com}}$ and we have a collision for Com_2. However, this case is already excluded by the collision check for Com_2 introduced in G_3.
- If $\mathsf{ctr} \neq \mathsf{ctr}^*$, then we find a collision for H_2^3 since $(\overline{\mathsf{ch}}_2 || \bar{a}_3 || \mathsf{ctr}) \neq (\overline{\mathsf{ch}}_2^* || \bar{a}_3^* || \mathsf{ctr}^*)$ but $\mathsf{ch}_3 = \mathsf{H}_2^3(\overline{\mathsf{ch}}_2 || \bar{a}_3 || \mathsf{ctr}) = \mathsf{H}_2^3(\overline{\mathsf{ch}}_2^* || \bar{a}_3^* || \mathsf{ctr}^*) = \mathsf{ch}_3^*$. However, this case is already excluded by the collision check for H_2^3 introduced in G_3.
- If $\mathsf{salt} \neq \mathsf{salt}^*$, then we find a collision for H_2^1 since $\overline{\mathsf{ch}}_1 = \overline{\mathsf{ch}}_1^*$ but its corresponding inputs differs. However, this case is already excluded by the collision check for H_2^1 introduced in G_3.

Summing up, the adversary cannot output a valid forgery satisfying $(\bar{a}_1, \overline{\mathsf{ch}}_1, \ldots, \bar{a}_3, \mathsf{ch}_3) = (\bar{a}_1^*, \overline{\mathsf{ch}}_1^*, \ldots, \bar{a}_3^*, \mathsf{ch}_3^*)$ and $a_4 \neq a_4^*$ with $(\mathsf{msg}, (\mathsf{ch}_3, a_4)) \in \mathcal{Q}$.

The bound for both cases, we obtain that

$$|\Pr[W_{8,4}] - \Pr[W_9]| \leq \mathsf{AdvNI}^{\mathsf{Com}_1}[\tau Q_{\mathsf{sig}}].$$

Reduction to EUF-NMA. We prove that the adversary's forgery in G_9 never involves the reprogrammed points by contradiction. Suppose that the adversary's forgery involves some of the reprogrammed points when the signing oracle computes $\sigma = (\mathsf{ch}_3, a_4)$ on a query msg. Those points are related to $\mathsf{ch}_1 = \mathsf{H}_2^1(\tilde{\mu}||a_1||\mathsf{salt})$, $\mathsf{ch}_2 = \mathsf{H}_2^2(\mathsf{ch}_1||a_2)$, and $\mathsf{ch}_{3,i} = \mathsf{H}_2^3(\mathsf{ch}_2||a_3||i)$, where $i \in [B]$. In any case, because of the collision checks, the first point $\mathsf{ch}_1 = \mathsf{H}_2^1(\tilde{\mu}||a_1||\mathsf{salt})$ should be reprogrammed.[9] Hence, the adversary's forgery satisfies $\mathsf{ch}_1 = \overline{\mathsf{ch}}_1^*$, $\tilde{\mu} = \bar{\tilde{\mu}}^*$, $\bar{a}_1 = \bar{a}_1^*$, and $\mathsf{salt} = \mathsf{salt}^*$. Especially, $\tilde{\mu} = \bar{\tilde{\mu}}^*$ implies $\mathsf{msg} = \mathsf{msg}^*$ because of the collision check of H_1 introduced in G_3. Furthermore, $\mathsf{msg} = \mathsf{msg}^*$ implies $(\mathsf{ch}_3, a_4) \neq (\mathsf{ch}_3^*, a_4^*)$. Due to the first check in G_9, if $(\bar{a}_1, \overline{\mathsf{ch}}_1) = (\bar{a}_1^*, \overline{\mathsf{ch}}_1^*)$ then ch_3 should be equal to ch_3^*. Thus, the condition is boiled down to $(\mathsf{msg}, \bar{a}_1, \overline{\mathsf{ch}}_1, \mathsf{ch}_3) = (\mathsf{msg}^*, \bar{a}_1^*, \overline{\mathsf{ch}}_1^*, \mathsf{ch}_3^*)$ and $a_4 \neq a_4^*$. By the way, since $\mathsf{ch}_3 = \mathsf{ch}_3^*$ holds and there must not be collisions for H_2^3, we have $(\overline{\mathsf{ch}}_2, \bar{a}_3) = (\overline{\mathsf{ch}}_2, \bar{a}_3^*)$. Furthermore, $\overline{\mathsf{ch}}_2 = \overline{\mathsf{ch}}_2^*$ and the collision check for H_2^2 implies $(\overline{\mathsf{ch}}_1, \bar{a}_2) = (\overline{\mathsf{ch}}_1^*, \bar{a}_2^*)$. Thus, we have $(\bar{a}_1, \overline{\mathsf{ch}}_1, \ldots, \bar{a}_3, \mathsf{ch}_3) = (\bar{a}_1^*, \overline{\mathsf{ch}}_1^*, \ldots, \bar{a}_3^*, \mathsf{ch}_3^*)$ and $a_4 \neq a_4^*$, but if this holds the adversary loses in G_9. This is a contradiction, and now, we can conclude that the adversary's forgery does not involve the points reprogrammed by the signing oracle in G_9.
Thus, we can easily construct an adversary $\mathcal{B}$ against the EUF-NMA security of the signature scheme satisfying

$$\Pr[W_9] \leq \mathsf{Adv}_{\mathcal{B}}^{\text{euf-nma}}.$$

Remark 7.4. : If we only consider the EUF-CMA security, we can skip G_9 and we have $\Pr[W_{8,4}] \leq \mathsf{Adv}_{\mathcal{B}}^{\text{euf-nma}}$. The argument follows:
Since we consider the EUF-CMA security, the adversary's output should msg^* such that $(\mathsf{msg}^*, \sigma) \notin \mathcal{Q}$ for any σ. Suppose that the adversary's forgery involves some of the reprogrammed points when the signing oracle computes $\sigma = (\mathsf{ch}_3, a_4)$ on a query $\mathsf{msg} \neq \mathsf{msg}^*$. We note that, due to the collision check for H_1 introduced in G_3, we have $\mu = \mathsf{H}_1(\mathsf{pk}||\mathsf{msg}) \neq \mu^* = \mathsf{H}_1(\mathsf{pk}||\mathsf{msg}^*)$.

1. If the first reprogrammed point $\mathsf{ch}_1 = \mathsf{H}_2^1(\mu||a_1||\mathsf{salt})$ is involved, then we have $(\mu, a_1, \mathsf{salt}) = (\mu^*, \bar{a}_1^*, \mathsf{salt}^*)$ and $\mu = \mathsf{H}_1(\mathsf{pk}||\mathsf{msg}) = \mathsf{H}_1(\mathsf{pk}||\mathsf{msg}^*) = \mu^*$. But, this contradicts with $\mu \neq \mu^*$. Thus, the first reprogrammed point should not be involved in the forgery.
2. If the second reprogrammed point $\mathsf{ch}_2 = \mathsf{H}_2^2(\mathsf{ch}_1||a_2)$ is involved, then we have $(\mathsf{ch}_1, a_2) = (\overline{\mathsf{ch}}_1^*, \bar{a}_2)$. But, $\mu \neq \mu^*$ and the collision check for H_2^1 implies that this cannot happen. Thus, the first and second reprogrammed points should not be involved in the forgery. In addition, we have $\mathsf{ch}_1 \neq \overline{\mathsf{ch}}_1^*$.

[9] Otherwise, we can find the collision for H_2^1 or H_2^2 introduced in G_3, but such event is eliminated by the collision check.

3. If the third reprogrammed point $\mathsf{ch}_3 = \mathsf{H}_2^3(\mathsf{ch}_2||a_3||\mathsf{ctr})$ for some $\mathsf{ctr} \in [B]$ is involved, then we have $(\mathsf{ch}_2, a_3) = (\overline{\mathsf{ch}}_2^*, \bar{a}_3^*)$. But, $\mathsf{ch}_1 \neq \overline{\mathsf{ch}}_1^*$ and the collision check for H_2^1 implies that this cannot happen. Thus, all three reprogrammed points should not be involved in the forgery.

Thus, the forgery does not involve the reprogrammed point, and the reduction is easily obtained. [10]

Security proofs for the BUFF securities. Section 4.B.4 of the call for proposal lists additional desirable security properties beyond standard unforgeability. In this section, we evaluate the so-called BUFF securities (message-bound signatures, exclusive ownership, and non re-signability) of our proposal. For the definitions of BUFF securities, see [CDF+21]. The proofs in [KX24b] showed that several MPCitH signatures achieve some BUFF securities. We adopt their proofs in the context of the VOLEitH signature and contain the concrete proofs below for completeness.

Message-bounding signatures (MBS). The MBS security shows that any efficient adversary cannot output pk and σ with two different messages msg and msg' such that $(\mathsf{pk}, \mathsf{msg}, \sigma)$ and $(\mathsf{pk}, \mathsf{msg}', \sigma)$ are both valid. Let $\mathcal{A}$ be an adversary against the MBS security of PERK: $\mathcal{A}$ takes 1^λ as input and outputs pk, msg, msg', and σ with $\mathsf{msg} \neq \mathsf{msg}'$ satisfying $\mathsf{PERK.Verify}(\mathsf{pk}, \mathsf{msg}, \sigma) = \mathsf{PERK.Verify}(\mathsf{pk}, \mathsf{msg}', \sigma) = 1$. Let us denote the internally reproduced values in the verifications $\mathsf{PERK.Verify}(\mathsf{pk}, \mathsf{msg}, \sigma)$ and $\mathsf{PERK.Verify}(\mathsf{pk}, \mathsf{msg}', \sigma)$ by $\bar{\cdot}$ and $\bar{\cdot}'$, respectively.

- Due to the definition of PERK.Verify, we have $\mathsf{ch}_3 = \overline{\mathsf{ch}}_3 = \overline{\mathsf{ch}}_3'$, where $\overline{\mathsf{ch}}_3 = \mathsf{H}_2^3(\overline{\mathsf{ch}}_2||\bar{a}_3||\mathsf{ctr})$ and $\overline{\mathsf{ch}}_3' = \mathsf{H}_2^3(\overline{\mathsf{ch}}_2'||\bar{a}_3'||\mathsf{ctr})$. If $(\overline{\mathsf{ch}}_2, \bar{a}_3) \neq (\overline{\mathsf{ch}}_2', \bar{a}_3')$, then we find the collision for H_2^3.
- Otherwise, we have $\overline{\mathsf{ch}}_2 = \overline{\mathsf{ch}}_2'$, where $\overline{\mathsf{ch}}_2 = \mathsf{H}_2^2(\overline{\mathsf{ch}}_1||\bar{a}_2)$ and $\overline{\mathsf{ch}}_2' = \mathsf{H}_2^2(\overline{\mathsf{ch}}_1'||\bar{a}_2')$. If $(\overline{\mathsf{ch}}_1, \bar{a}_2) \neq (\overline{\mathsf{ch}}_1', \bar{a}_2')$, then we find the collision for H_2^2.
- Otherwise, we have $\overline{\mathsf{ch}}_1 = \overline{\mathsf{ch}}_1'$, where $\overline{\mathsf{ch}}_1 = \mathsf{H}_2^1(\bar{\tilde{\mu}}||\bar{a}_1||\mathsf{salt})$ and $\overline{\mathsf{ch}}_1' = \mathsf{H}_2^1(\bar{\tilde{\mu}}'||\bar{a}_1'||\mathsf{salt})$. If $(\bar{\tilde{\mu}}, \bar{a}_1, \mathsf{salt}) \neq (\bar{\tilde{\mu}}', \bar{a}_1', \mathsf{salt})$, then we find the collision for H_2^1.
- Otherwise, we have $\bar{\tilde{\mu}} = \bar{\tilde{\mu}}'$, where $\bar{\tilde{\mu}} = \mathsf{H}_1(\mathsf{pk}||\mathsf{msg})$ and $\bar{\tilde{\mu}}' = \mathsf{H}_1(\mathsf{pk}||\mathsf{msg}')$. Since $\mathsf{msg} \neq \mathsf{msg}'$, thus, we find the collision for H_1.

In any case, we can find a collision for either H_1, H_2^1, H_2^2, or H_2^3. Thus, if they are collision-resistant, then PERK is MBS-secure.

Malicious strong universal exclusive ownership (M-S-UEO). We consider M-S-UEO, which is the strongest form of exclusive ownership.[11] The M-S-UEO security shows that any efficient adversary cannot output two different public key

[10] We notice that FAEST's proof did not consider the collision-resistance property of H_1, H_2^1, and H_2^2.

[11] If the scheme is M-S-UEO-secure, then it also Strong destructive exclusive ownership (S-DEO-secure) and Strong conservative exclusive ownership (S-CEO-secure).

pk and pk', two (possibly different) messages msg and msg', and a signature σ such that $(\mathsf{pk}, \mathsf{msg}, \sigma)$ and $(\mathsf{pk}', \mathsf{msg}', \sigma)$ are both valid. Let $\mathcal{A}$ be an adversary against the M-S-UEO security of PERK: $\mathcal{A}$ takes 1^λ as input and outputs pk, pk', msg, msg', and σ with $\mathsf{pk} \neq \mathsf{pk}'$ such that $\mathsf{PERK.Verify}(\mathsf{pk}, \mathsf{msg}, \sigma) = \mathsf{PERK.Verify}(\mathsf{pk}', \mathsf{msg}', \sigma) = 1$. By a similar argument to the MBS security above, we have a collision for H_2^3, H_2^2, or H_2^1 by ch_3, ch_2, or ch_1, respectively. If $\tilde{\mu} = \tilde{\mu}'$, then we have $\mathsf{H}_1(\mathsf{pk}||\mathsf{msg}) = \mathsf{H}_1(\mathsf{pk}'||\mathsf{msg}')$ and obtain a collision $(\mathsf{pk}, \mathsf{msg}) \neq (\mathsf{pk}', \mathsf{msg}')$ for H_1 since $\mathsf{pk} \neq \mathsf{pk}'$. Thus, if the hash functions H_1, H_2^1, H_2^2, and H_2^3 are collision resistant, then PERK is M-S-UEO-secure.

Weak non-resignability (WNR). Roughly speaking, (weak) non-resignability shows that, given pk and σ on a hidden message msg (with some leakage)[12], any efficient adversary cannot output pk' and σ' such that $(\mathsf{pk}', \mathsf{msg}, \sigma')$ is valid. There are some generic conversion for EUF-CMA-secure signature scheme in the (Q)ROM.

Since the signature is produced on $\tilde{\mu} \coloneqq \mathsf{H}_1(\mathsf{pk}||\mathsf{msg})$ instead of msg itself, our signature scheme inherently implements the BUFF transform with a random oracle H_1. Don, Fehr, Huang, Liao, and Struck [DFH+24] showed the BUFF transform allows us to achieve $\mathsf{sNR}^{\mathsf{H}_1,\perp}$. Thus, PERK satisfies $\mathsf{sNR}^{\mathsf{H}_1,\perp}$.

There is another weakened NR, $\mathsf{NR}^{\mathsf{H}_1,\perp}$, in [DFHS24], but this requires the \$-BUFF transform that computes $y = \mathsf{H}_1(\mathsf{pk}||\mathsf{msg}||s)$ with salt s.

7.2 Known attacks against PKP

7.2.1 Overview of known attacks. The Permuted Kernel Problem (PKP) problem was introduced by Shamir in 1990 [Sha90]. Despite its long standing history in cryptographic applications [Sha90, BFK+19, Beu20, BG23] and consequently many cryptanalytic efforts [Geo92, BCCG93, PC94, JJ01, LP11, KMP19, SBC23], algorithms to solve the PKP are still rather simple adaptations of combinatorial enumeration and meet-in-the-middle techniques. Indeed, the best attack on standard PKP is a meet-in-the-middle adaptation known as the KMP algorithm by Koussa, Macario-Rat and Patarin [KMP19]. Even though there has been some recent progress on attacks [SBC23], those do not improve over the KMP algorithm in the case of standard PKP on which PERK is based.

7.2.2 KMP algorithm on PKP. In this section we briefly sketch the KMP algorithm to solve the PKP. Fully fledged descriptions, analysis and estimation scripts are given for example in [KMP19, SBC23, EVZB24]. The algorithm by Koussa, Macario-Rat and Patarin [KMP19] is a slight variant of previously known combinatorial techniques [Geo92, BCCG93, PC94, JJ01]. The algorithm was first proposed for the inhomogeneous version of PKP, where $\mathbf{H}\pi(\mathbf{x}) = \mathbf{y}$ for a given vector $\mathbf{y} \in \mathbb{F}_q^m$ [KMP19]. The algorithm was then recently extended to the multi-dimensional case [SBC23], i.e. the case where multiple $\mathbf{x}_i$ and $\mathbf{y}_i$ are provided and the the solution is a permutation π, with $\mathbf{H}\pi(\mathbf{x}_i) = \mathbf{y}_i$ for all i.

[12] The definitions vary depending on how the information of msg is leaked to the adversary.

Santini, Baldi and Chiaraluce also introduced further improvements to this generalized KMP algorithm. However, as it only improves for $i > 1$ pairs $(\mathbf{x}_i, \mathbf{y}_i)$, we do not consider it for the security analysis of PERK, which uses $i = 1$.

Initially, the matrix $\mathbf{H}$ is transformed into semi-systematic form by applying a change of basis (modelled by the invertible matrix $\mathbf{Q}$)

$$\mathbf{QH} = \begin{pmatrix} \mathbf{I}_{m-u} & \mathbf{H}_1 \\ \mathbf{0} & \mathbf{H}_2 \end{pmatrix},$$

where $\mathbf{H}_1 \in \mathbb{F}_q^{(m-u)\times(n-m+u)}$,$\mathbf{H}_2 \in \mathbb{F}_q^{u\times(n-m+u)}$ and u is an optimization parameter of the algorithm. For the inhomogeneous variant, where $\mathbf{y} \neq \mathbf{0}$, one maintains the validity of the PKP identity by multiplying the syndrome $\mathbf{y}$ by the same matrix $\mathbf{Q}$

$$\begin{aligned}\mathbf{QH}\pi(\mathbf{x}) &= \begin{pmatrix} \mathbf{I}_{m-u} & \mathbf{H}_1 \\ \mathbf{0} & \mathbf{H}_2 \end{pmatrix} \pi(\mathbf{x}) = \begin{pmatrix} \mathbf{I}_{m-u} & \mathbf{H}_1 \\ \mathbf{0} & \mathbf{H}_2 \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = (\mathbf{x}_1 + \mathbf{H}_1\mathbf{x}_2, \mathbf{H}_2\mathbf{x}_2)^\top \\ &= (\mathbf{y}_1, \mathbf{y}_2)^\top = \mathbf{Qy},\end{aligned}$$

where $\mathbf{Qy} = (\mathbf{y}_1, \mathbf{y}_2) \in \mathbb{F}_q^{m-u} \times \mathbb{F}_q^u$ and $\pi(\mathbf{x}) = (\mathbf{x}_1, \mathbf{x}_2) \in \mathbb{F}_q^{m-u} \times \mathbb{F}_q^u$. The algorithm now focuses on solving the identity $\mathbf{H}_2\mathbf{x}_2 = \mathbf{y}_2$. For any found $\mathbf{x}_2$ satisfying the identity it is than checked if $\mathbf{x}_1 = \mathbf{y}_1 - \mathbf{H}_1\mathbf{x}_2$ and $\mathbf{x}_2$ together form a permutation of $\mathbf{x}$.

Candidates for $\mathbf{x}_2$ are obtained by a meet-in-the-middle enumeration strategy. Therefore $\mathbf{x}_2$ is further split as $\mathbf{x}_2 = (\mathbf{x}_{21}, \mathbf{x}_{22})$, with $\mathbf{x}_{21}, \mathbf{x}_{22} \in \mathbb{F}_q^{u\times((n-m+u)/2)}$ to obtain the meet-in-the-middle identity

$$\mathbf{H}_2(\mathbf{x}_{21}, \mathbf{0}) = \mathbf{y}_2 - (\mathbf{0}, \mathbf{x}_{22}). \tag{2}$$

Then the algorithm enumerates all candidates for $\mathbf{x}_{21}$ and $\mathbf{x}_{22}$, that is all permutations of any selection of $(n-m+u)/2$ entries of $\mathbf{x}$. For each such vector the left (resp. right) side of Equation (2) is stored in a list L_1 (resp. L_2). In a final step the algorithm searches for matches between the lists L_1 and L_2 yielding the candidates for $\mathbf{x}_2$. From there $\mathbf{x}_1$ can be computed as $\mathbf{x}_1 = \mathbf{y}_1 - \mathbf{H}_1\mathbf{x}_2$. If $(\mathbf{x}_1, \mathbf{x}_2)$ forms a permutation of $\mathbf{x}$ this yields the solution π.

The complexity of the algorithm is (up to polynomial factors) linear in the sizes of the lists L_1, L_2 and L, where L is the list of matches. The expected sizes are

$$|L_1| = |L_2| = \binom{n}{(n-m+u)/2}\big((n-m+u)/2\big)! \quad \text{and} \quad |L| = \frac{|L_1 \times L_2|}{q^u}$$

7.2.3 Relation between PKP and the Code Equivalence Problem. In their recent work Santini et al. [SBC23] formalized the equivalence between PKP and the subcode equivalence problem. Namely, PKP asks to find a permutation that sends the one dimensional code $\mathbf{x}$ into the code with parity check matrix

H, which defines the problem. For variants of PKP using higher dimensional codes this can have some implications regarding security, depending on the concrete choice of code dimension. However, again, for standard PKP using a 1-dimensional code **x** this has no effect on security.

7.2.4 PKP over extension fields. In the context of PERK, we consider PKP over $\mathbb{F}_q$ with $q = p^r$ being a prime power. To the best of our knowledge, no algorithms are known that exploit the structure of the extension field $\mathbb{F}_q$. Especially, there are no known adaptations or enhancements of the KMP algorithm that leverage this characteristic.

Note that, this observation aligns with the evidence from the closely related syndrome decoding problem over $\mathbb{F}_q$. The most effective algorithm for this problem, when $q > 2$, relies on an enumeration routine that is conceptually similar to the KMP algorithm. Yet, even in the case of syndrome decoding over extension fields, no algorithmic improvements have been identified that exploit the extension field's structure. This was recently confirmed in [EW25].

7.2.5 Parameter selection. For parameter selection we fix $q = 2048$. We then, for any choice of n rely on a standard choice of m. That is, for any choice of n we choose m minimal such that the expected amount of solutions to a random instance of the $\mathsf{PKP}(q, m, n)$ is smaller than one. Subsequently we use the *CryptographicEstimators* library[13] [EVZB24], for the concrete complexity estimation of the KMP algorithm for any set of parameters (q, n, m). Eventually, for any security level we choose n such that the complexity estimation yields a comfortable margin to the NIST specified security levels.

Note that in addition to this margin, the estimation of the KMP algorithm via the CryptographicEstimators is already a lower bound on the algorithm's complexity by neglecting some factors. Furthermore the KMP algorithm suffers from a memory complexity that is equal to its time complexity. Therefore, realistic attacks have to resort to time-memory trade-offs further adding to this margin. The proposed parameters can therefore be seen as conservative choices with respect to security.

Overall the detailed procedure leads to the choices of parameters given in Section 5 whose estimated bit complexity is given in Table 12.

[13] https://github.com/Crypto-TII/cryptographic_estimators

Instance	q	n	m	Bit Security
PERK-1	2048	64	27	150
PERK-3	2048	92	43	220
PERK-5	2048	118	59	286

Table 12: Bit security estimates of PERK parameters

8 Advantages and Limitations

We now discuss some advantages and limitations of PERK.

8.1 Advantages

Some advantages of our design are:

+ PERK features very small public key and secret key sizes along with moderate signature sizes. Therefore, on the combined metric of pk + signature size, PERK produces sizes of approximately 3.5 kB for NIST security level 1 which compares well with other signature schemes.
+ Contrarily to many post-quantum schemes, the security of PERK is not based on a problem relying on cyclic structure or ring structure.
+ Resilience against PKP attacks: A large part of the signature size scales with the security parameter λ (due to the seed trees and commitments) and not directly with the PKP parameters. As a consequence, increasing the PKP parameters has a limited impact on the total size of the signature.
+ PERK performances are constrained by numerous calls to symmetric cryptographic primitives. Any speedup to the implementation of these primitives directly benefit PERK. In particular, hardware acceleration support for such primitives improves the performance of the scheme.

8.2 Limitations

In the following, we point out the limitations of PERK.

– While PKP was initially defined over prime fields, PERK relies on PKP defined over $\mathbb{F}_q$ where $\mathbb{F}_q$ is an extension field of $\mathbb{F}_2$. While, no algorithms exploiting the structure of the extension field are known, this variant has been less studied than the original long time standing PKP problem.

– While PERK's performance profile is comparable to other MPCitH-based constructions, those can not compete with the fastest post-quantum secure schemes, usually based on structured lattices.

References

BBD+23a. Manuel Barbosa, Gilles Barthe, Christian Doczkal, Jelle Don, Serge Fehr, Benjamin Grégoire, Yu-Hsuan Huang, Andreas Hülsing, Yi Lee, and Xiaodi Wu. Fixing and mechanizing the security proof of Fiat-Shamir with aborts and Dilithium. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 358–389. 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. Technical report, National Institute of Standards and Technology, 2023. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

BBD+23c. 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.

BBGK24. Slim Bettaieb, Loïc Bidoux, Philippe Gaborit, and Mukul Kulkarni. Modelings for generic PoK and applications: Shorter SD and PKP based signatures. Cryptology ePrint Archive, Report 2024/1668, 2024.

BBM+25. Carsten Baum, Ward Beullens, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. One tree to rule them all: Optimizing ggm trees and owfs for post-quantum signatures. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 463–493. Springer, 2025.

BCCG93. Thierry Baritaud, Mireille Campana, Pascal Chauvaud, and Henri Gilbert. On the security of the permuted kernel identification scheme. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 305–311. Springer, Berlin, Heidelberg, August 1993.

Ber19. Daniel J. Bernstein. djbsort. https://sorting.cr.yp.to/, 2019. [Online; accessed 20-June-2023].

Beu20. Ward Beullens. Sigma protocols for MQ, PKP and SIS, and Fishy signature schemes. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 183–211. Springer, Cham, May 2020.

BFK+19. Ward Beullens, Jean-Charles Faugère, Eliane Koussa, Gilles Macario-Rat, Jacques Patarin, and Ludovic Perret. PKP-based signature scheme. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 3–22. Springer, Cham, December 2019.

BG23. Loïc Bidoux and Philippe Gaborit. Compact post-quantum signatures from proofs of knowledge leveraging structure for the PKP, SD and RSD problems. In *Codes, Cryptology and Information Security (C2SI)*, pages 10–42. Springer, 2023.

BJKS94. Jürgen Bierbrauer, Thomas Johansson, Gregory Kabatianskii, and Ben Smeets. On families of hash functions via geometric codes and concatenation. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 331–342. Springer, Berlin, Heidelberg, August 1994.

BPS16. Mihir Bellare, Bertram Poettering, and Douglas Stebila. From identification to signatures, tightly: A framework and generic transforms. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 435–464. Springer, Berlin, Heidelberg, December 2016.

CDF+21. Cas Cremers, Samed Düzlü, Rune Fiedler, Marc Fischlin, and Christian Janson. BUFFing signature schemes beyond unforgeability and the case of post-quantum signatures. In *2021 IEEE Symposium on Security and Privacy*, pages 1696–1714. IEEE Computer Society Press, May 2021.

CW79. J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

DFH+24. Jelle Don, Serge Fehr, Yu-Hsuan Huang, Jyun-Jie Liao, and Patrick Struck. Hide-and-seek and the non-resignability of the BUFF transform. In Elette Boyle and Mohammad Mahmoody, editors, *TCC 2024, Part III*, volume 15366 of *LNCS*, pages 347–370. Springer, Cham, December 2024.

DFHS24. Jelle Don, Serge Fehr, Yu-Hsuan Huang, and Patrick Struck. On the (in)security of the BUFF transform. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part I*, volume 14920 of *LNCS*, pages 246–275. Springer, Cham, August 2024.

DFPS23. Julien Devevey, Pouria Fallahpour, Alain Passelègue, and Damien Stehlé. A detailed analysis of Fiat-Shamir with aborts. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 327–357. Springer, Cham, August 2023.

EVZB24. Andre Esser, Javier A. Verbel, Floyd Zweydinger, and Emanuele Bellini. SoK: CryptographicEstimators - a software library for cryptographic hardness estimation. In Jianying Zhou, Tony Q. S. Quek, Debin Gao, and Alvaro A. Cárdenas, editors, *ASIACCS 24*. ACM Press, July 2024.

EW25. Freja Elbro and Violetta Weger. Can we speed up information set decoding by using extension field structure? Cryptology ePrint Archive, Paper 2025/1402, 2025.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Berlin, Heidelberg, August 1987.

Geo92. Jean Georgiades. Some remarks on the security of the identification scheme based on permuted kernels. *Journal of Cryptology*, 5(2):133–137, January 1992.

GGM84. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the cryptographic applications of random functions. In G. R. Blakley and David Chaum, editors, *CRYPTO'84*, volume 196 of *LNCS*, pages 276–288. Springer, Berlin, Heidelberg, August 1984.

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.

JJ01. Éliane Jaulmes and Antoine Joux. Cryptanalysis of PKP: A new approach. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 165–172. Springer, Berlin, Heidelberg, February 2001.

KKW18. Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

KMP19. Eliane Koussa, Gilles Macario-Rat, and Jacques Patarin. On the complexity of the permuted kernel problem. Cryptology ePrint Archive, Report 2019/412, 2019.

KX24a. Haruhisa Kosuge and Keita Xagawa. Probabilistic hash-and-sign with retry in the quantum random oracle model. In Qiang Tang and Vanessa Teague, editors, *PKC 2024, Part I*, volume 14601 of *LNCS*, pages 259–288. Springer, Cham, April 2024.

KX24b. Mukul Kulkarni and Keita Xagawa. Strong existential unforgeability and more of MPC-in-the-head signatures. Cryptology ePrint Archive, Report 2024/1069, 2024.

LP11. Rodolphe Lampe and Jacques Patarin. Analysis of some natural variants of the pkp algorithm. *Cryptology ePrint Archive*, 2011.

PC94. Jacques Patarin and Pascal Chauvaud. Improved algorithms for the permuted kernel problem. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 391–402. Springer, Berlin, Heidelberg, August 1994.

Roy22. Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 657–687. Springer, Cham, August 2022.

SBC23. Paolo Santini, Marco Baldi, and Franco Chiaraluce. Computational hardness of the permuted kernel and subcode equivalence problems. *IEEE Transactions on Information Theory*, 2023.

Sha90. Adi Shamir. An efficient identification scheme based on permuted kernels (extended abstract) (rump session). In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 606–609. Springer, New York, August 1990.

Sti92. Douglas R. Stinson. Universal hashing and authentication codes. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 74–85. Springer, Berlin, Heidelberg, August 1992.