The Syndrome Decoding in the Head (SD-in-the-Head) Signature Scheme

Algorithm Specifications and Supporting Documentation – Version 2.0

Carlos Aguilar Melchor   Slim Bettaieb   Loïc Bidoux

Thibauld Feneuil   Philippe Gaborit   Nicolas Gama

Shay Gueron   James Howe   Andreas Hülsing   David Joseph

Antoine Joux   Mukul Kulkarni   Edoardo Persichetti

Tovohery H. Randrianarisoa   Matthieu Rivain   Dongze Yue

February 5, 2025

CISPA
CryptoExperts
Eindhoven University of Technology
Florida Atlantic University
Meta
SandboxAQ
Sapienza University
Technology Innovation Institute
University of Haifa
University of Limoges

Contents

Changelog

2025-02-05: Version 1.1 → Version 2.0

We have updated the SDitH signature scheme to align it with the current literature. The first version of the scheme utilized the MPCitH framework using linear multiparty computation, optimized through hypercube [AGH+23] and threshold approaches [FR23b]. In the second version, we have adopted the VOLEitH framework [BBD+23], which allows us to reduce the signature size by more than half.

Additionally, we have modified the syndrome decoding field. Previously, we focused on $\mathbb{F}_{256}$ and $\mathbb{F}_{251}$; now, we are using the binary field $\mathbb{F}_2$, which offers a more conservative security assumption.

Finally, we have redesigned the seed trees to incorporate AES-128 and Rijndael-256-256 as symmetric primitives, moving away from Keccak-based hashes to enhance the speed of the scheme.

1 Introduction

This specification presents the Syndrome-Decoding-in-the-Head (SD-in-the-Head) digital signature scheme. The scheme is based on the hardness of the syndrome decoding (SD) problem for random linear codes on a finite field. It consists in a zero-knowledge proof of knowledge of a low-weight vector x solution of a syndrome decoding instance $y = Hx$, which is made non-interactive using the Fiat-Shamir transform. This zero-knowledge proof relies on the principle of "multiparty computation in the head" (MPCitH) originally introduced in [IKO+07] and notably used by the Picnic signature scheme [ZCD+20], candidate to the previous NIST call for post-quantum algorithms. The first version of SD-in-the-Head was based on the initial scheme published in [FJR22] with further improvements from [AGH+22; FR22]. This second version relies on a different proof system. This proof system is based on the VOLE-in-the-Head framework [BBD+23] and its efficient application to the SD problem [OTX24; BBG+24].

For most applications of signatures, specially the ones that require certificates such as TLS, the "public key + signature size" is thus a critical metric. The SD-in-the-Head signature scheme presented in this specification achieves 3.7 KB for this metric in Category I, which is similar to ML-DSA and less than halved compared to SLH-DSA (7.8 KB).

Organization of this specification

Section 2 gives a high-level description of the SD-in-the-Head-2 scheme. Section 3 provides a detailed description of the key generation, signature and verification algorithms. This description intends to allow a non-ambiguous implementation of the scheme. The selection of the parameters is explained in Section 4 which also exhibits our proposed instances for the three considered security levels. Section 4.4 provides performance figures for our different instances. The security of the SD-in-the-Head signature scheme is analyzed in Section 5 while Section 5.2 further evaluates the complexity of known attacks. We finally list some advantages and limitations of the scheme in Section 7.

We welcome enquiries, comments, and corrections at

consortium@sdith.org

Implementations and material related to the SD-in-the-Head signature scheme will be uploaded and maintained on:

https://github.com/sdith

2 High-level description

The second version of the SD-in-the-Head signature scheme, namely SD-in-the-Head-2, relies on the VOLE-in-the-Head framework [BBD+23] to build a 7-round zero-knowledge proof of knowledge for the syndrome decoding problem, which is then transformed into a signature scheme using the Fiat-Shamir heuristic [FS87]. The modeling used to handle the syndrome decoding problem in the VOLEitH framework has been proposed independently into the two articles [OTX24] and [BBG+24].

The following sections present the VOLE-in-the-Head framework using the PIOP formalism and describe how SD-in-the-Head-2 is built from this framework and the modeling of [OTX24; BBG+24].

2.1 VOLEitH in the PIOP formalism

The MPCitH paradigm [IKO+07] is a versatile method introduced in 2007 to build zero-knowledge proof systems using techniques from secure multi-party computation (MPC). This paradigm has been drastically improved in recent years and is particularly efficient to build zero-knowledge proofs for small circuits such as those involved in (post-quantum) signature schemes. The more recent MPCitH-based frameworks are the VOLE-in-the-Head (VOLEitH) framework from [BBD+23] and the Threshold-Computation-in-the-Head (TCitH) framework from [FR23b; FR23a].

In this subsection, we will describe the general proof system which SD-in-the-Head-2 relies on. In what follows, we present this proof system using the formalism of the Polynomial Interactive Oracle Proofs (PIOP), as presented in [Fen24]. Indeed, while the TCitH and VOLEitH frameworks were originally introduced using sharing-based and VOLE-based formalisms, respectively, we present them within the PIOP formalism, which provides a unified and comprehensive ground for these techniques.[1]

Let us assume that we want to build an interactive zero-knowledge proof with a prover convincing a verifier that they know a witness $w \in \mathbb{F}_2^n$ which satisfies some public polynomial relations:

$$f_j(w) = 0, \ \forall 1 \leq j \leq m$$

where $f_1, \ldots, f_m$ are polynomials over $\mathbb{F}_2$ of total degree at most d. Let us consider a public subset $S \subset \mathbb{F}_{2^\lambda}$. The proof system we consider is the following:

1. For $1 \leq j \leq n$, the prover samples a random degree-1 polynomials P_j such that $P_j(0) = w_j$. They also sample a random degree-$(d-1)$ polynomial $P_0 \in \mathbb{F}_{2^\lambda}[X]$. They commit to those polynomials.

2. The verifier chooses random coefficients $\gamma_1, \ldots, \gamma_m$ from $\mathbb{F}_{2^\lambda}$ and sends them to the prover. The latter then reveals the degree-$(d-1)$ polynomial $Q(X)$ defined such that

$$Q(X) \cdot X = P_0(X) \cdot X + \sum_{j=1}^{m} \gamma_j \cdot f_j(P_1(X), \ldots, P_n(X)). \quad (1)$$

[1] In the TCitH framework, instead of performing operations over Shamir's secret sharings, we can directly work over their underlying polynomials. In the VOLEitH framework, instead of performing operations over VOLE gadgets, we can directly work over their underlying degree-1 polynomials.

3. The verifier samples a random evaluation point Δ from the public set $S \subset \mathbb{F}_{2^\lambda}$ and sends it to the prover. The latter then reveals the evaluations $v_i := P_i(\Delta)$, together with a proof π that the evaluations are consistent with the commitment.

4. The verifier checks that the revealed evaluations are consistent with the commitment using π and checks that we have

$$Q(\Delta) \cdot \Delta = v_0 \cdot \Delta + \sum_{j=1}^{m} \gamma_j \cdot f_j(v_1, \ldots, v_n) \ . \tag{2}$$

The above protocol assumes that the prover has a way to commit polynomials and to provably open some evaluations later (while keeping hidden the other evaluations).

Security analysis. We can observe that the coefficient in front of the degree-0 monomial (*i.e.* the constant term) in the right term of Equation (1) is

$$\sum_{j=1}^{m} \gamma_j \cdot f_j(w_1, \ldots, w_n) \ , \tag{3}$$

so the degree-$(d-1)$ polynomial Q is well-defined because this quantity is zero when the witness w is valid. Let us assume that the prover is malicious, meaning that they do not know a valid witness. It implies that there exists j^* such that $f_{j^*}(w) \neq 0$. In that case, the probability that there exists some Q such that Equation (1) holds is at most $1/2^\lambda$ over the randomness of $\gamma_1, \ldots, \gamma_m$, because the coefficient (3) is zero only with probability $1/2^\lambda$. If Equation (1) does not hold, the probability that the check in Equation (2) passes is at most $d/|S|$, since the degree-d polynomial relation

$$Q(X) - \left(P_0 + \sum_{j=1}^{m} \gamma_j \cdot f_j^{[h]}(X, P_1(X), \ldots, P_m(X)) \right) \neq 0$$

would have at most d roots (and so the random challenge Δ should be among those roots). So, the proof system is *sound*, with a soundness error of $\frac{1}{2^\lambda} + \left(1 - \frac{1}{2^\lambda}\right) \cdot \frac{d}{|S|}$. Moreover, assuming that the commitment scheme is hiding, we can observe that the interactive proof is zero-knowledge since

- revealing $Q(X)$ leaks no information about the secret thanks to the random polynomial P_0, and
- revealing one evaluation of the polynomials $P_1, \ldots, P_n$ leaks no information about the leading term thanks to the randomness used to build those polynomials.

In what follows, we describe how to commit polynomials such that we can later open some evaluations.

The TCitH-GGM approach. Thanks to ideas from [ISN89; CDI05], the TCitH framework [FR23a] shows that we can commit $\bar{n}$ random polynomials using seed trees in such a way that the committer can later open one evaluation among a set of N while keeping the others hidden. Here is the commitment process for degree-1 polynomials:

1. One uses an all-but-one vector commitment (AVC) to sample and commit N seeds $\mathsf{seed}_1, \ldots, \mathsf{seed}_N$.
2. One expands each seed_i as $w_{\mathsf{rnd},i} := \mathsf{PRG}(\mathsf{seed}_i) \in \mathbb{F}_q^{\bar{n}}$ for $i \in \{1, \ldots, N\}$, where PRG is a pseudorandom generator.
3. One computes
$$w_{\mathsf{acc}} \leftarrow \sum_{i=1}^{N} w_{\mathsf{rnd},i} \in \mathbb{F}_2^{\bar{n}}$$
$$w_{\mathsf{base}} \leftarrow -\sum_{i=1}^{N} \phi(i) \cdot w_{\mathsf{rnd},i} \in \mathbb{F}_{2^\kappa}^{\bar{n}}$$
where $\phi : \{1, \ldots, N\} \to \mathbb{F}_{2^\kappa}$ is a public one-to-one function.
4. One defines P_j as
$$P_j(X) = (w_{\mathsf{acc}})_j \cdot X + (w_{\mathsf{base}})_j$$
for all j.

This commitment procedure has the main advantage to enable the prover to reveal one evaluation $\{P_j(\phi(i^*))\}_j$ for $i^* \in [1 : N]$ while keeping *secret* the coefficients w_{acc} and w_{base}: they just need to reveal all the $\{\mathsf{seed}_i\}_i$ except seed_{i^*} (by opening the AVC scheme) and the verifier will be able to compute $P_j(\phi(i^*))$ as

$$\sum_{i=1, i\neq i^*}^{N} (\phi(i^*) - \phi(i)) \cdot (w_{\mathsf{rnd},i})_j \quad \text{with} \quad w_{\mathsf{rnd},i} := \mathsf{PRG}(\mathsf{seed}_i).$$

Indeed, we have that

$$\begin{aligned}\sum_{i=1, i\neq i^*}^{N} (\phi(i^*) - \phi(i)) \cdot (w_{\mathsf{rnd},i})_j &= \phi(i^*) \cdot \sum_{i=1}^{N} (w_{\mathsf{rnd},i})_j - \sum_{i=1}^{N} \phi(i) \cdot (w_{\mathsf{rnd},i})_j \\ &= \phi(i^*) \cdot (w_{\mathsf{acc}})_j + (w_{\mathsf{base}})_j = P_j(\phi(i^*)).\end{aligned}$$

We can use this commitment procedure to commit to the polynomials in Step 1 of the proof system. We just need to rely on auxiliary values to enforce some coefficients of P_j. Because the public set would be of size N ($S := \{\phi(1), \ldots, \phi(N)\}$), the resulting 5-round zero-knowledge proof system has a soundness error of

$$\frac{1}{2^\lambda} + \left(1 - \frac{1}{2^\lambda}\right) \cdot \frac{d}{N} \ ,$$

and one needs to rely on protocol repetitions to achieve the desired security. Indeed, the computational complexity is linear in N and so we can not take N exponentially large. To have a λ-bit security we need to repeat the protocol τ times in parallel, such that $(d/N)^\tau \leq 2^{-\lambda}$.

The VOLEitH approach. In the VOLEitH framework, the commitment scheme enables the opening of an evaluation from an exponentially-large set S, thus avoiding parallel repetitions of the PIOP. As the TCitH framework, the VOLEitH approach starts by committing τ sets of polynomials $\{P_i^{(1)}\}_i, \ldots, \{P_i^{(\tau)}\}_i$ in parallel (exactly using the same commitment procedure). However, instead of considering those sets of polynomials individually as in the TCitH framework, the VOLEitH approach consists in "merging them" into a polynomial over an extension field of size greater that N^τ.

This merge works as follows. Consider τ polynomials $P^{(1)}, \ldots, P^{(\tau)}$ encoding a witness coefficient in their leading terms: $P^{(e)} = w \cdot X + w_{\mathsf{base}}^{(e)}$, where $w \in \mathbb{F}_2$ and $w_{\mathsf{base}}^{(e)} \in \mathbb{F}_{2^\kappa}$ for every e. These can be merged into the polynomial:

$$P(X) = w \cdot X + \underbrace{\psi(w_{\mathsf{base}}^{(1)}, \ldots, w_{\mathsf{base}}^{(\tau)})}_{\in \mathbb{F}_{2^\lambda}}$$

where $\tau \cdot \kappa \leq \lambda$ and ψ is an $\mathbb{F}_2$-morphism from $(\mathbb{F}_{2^\kappa})^\tau$ to $\mathbb{F}_{2^\lambda}$. The key observation is that we can open an evaluation $P(X)$ into any point of

$$E := \left\{\psi(v_1, \ldots, v_\tau) \mid (v_1, \ldots, v_\tau) \in \{\phi(1), \ldots, \phi(N)\}^\tau\right\} \subset \mathbb{F}_{2^\lambda}$$

by opening evaluations of $P^{(1)}, \ldots, P^{(\tau)}$ on the small domain. Specifically, we compute $P(\Delta)$ where $\Delta = \psi\left(\phi(i^{(1)}), \ldots, \phi(i^{(\tau)})\right)$ for some $(i^{(1)}, \ldots, i^{(\tau)}) \in \{1, \ldots, N\}^\tau$ as

$$\psi\big(P^{(1)}(\phi(i^{(1)})),\ \ldots,\ P^{(\tau)}(\phi(i^{(\tau)}))\big)\ .$$

Indeed, we have

$$\begin{aligned}
\psi\big(P^{(1)}(\phi(i^{(1)})),\ \ldots,\ P^{(\tau)}(\phi(i^{(\tau)}))\big) &= \psi\big(w \cdot \phi(i^{(1)}) + w_{\mathsf{base}}^{(1)}, \ldots, w \cdot \phi(i^{(\tau)}) + w_{\mathsf{base}}^{(\tau)}\big) \\
&= w \cdot \psi\big(\phi(i^{(1)}), \ldots, \phi(i^{(\tau)})\big) + \psi\big(w_{\mathsf{base}}^{(1)}, \ldots, w_{\mathsf{base}}^{(\tau)}\big) \\
&= w \cdot \Delta + \psi\big(w_{\mathsf{base}}^{(1)}, \ldots, w_{\mathsf{base}}^{(\tau)}\big) = P(\Delta)\ .
\end{aligned}$$

So, instead of running the proof system τ times in parallel over the polynomials $P^{(1)}, \ldots, P^{(\tau)}$, one runs it once over the polynomial P. Since the evaluation set S is now of size N^τ, the resulting soundness error is thus

$$\frac{1}{2^\lambda} + \left(1 - \frac{1}{2^\lambda}\right) \cdot \frac{d}{N^\tau}\ .$$

Let us remark that the merge requires the polynomials $P^{(1)}, \ldots, P^{(\tau)}$ to have the same leading term. Therefore, the prover should convince the verifier that those τ polynomials have the same leading term. Let us assume that we work with vector polynomials, *i.e.* $P^{(e)}$ is of the form $P^{(e)} = w \cdot X + w_{\mathsf{base}}^{(e)}$ for some $w \in \mathbb{F}_2^{\bar{n}}$ and $w_{\mathsf{base}}^{(e)} \in \mathbb{F}_{2^\lambda}^{\bar{n}}$. To convince the verifier, the prover will run the following *consistency check*. After committing to the τ polynomials, the prover gets a random matrix $\boldsymbol{M} \in \mathbb{F}_2^{(\lambda+B)\times\bar{n}}$ from the verifier and then reveals the polynomial

$$R^{(e)}(X) \leftarrow \boldsymbol{M} \cdot P^{(e)}(X)$$

for all $e \in \{1, \ldots, \tau\}$. The verifier can then check that the leading terms of $R^{(1)}(X), \ldots, R^{(\tau)}(X)$ are indeed the same, and, after the opening of the evaluations, they can check that

$$R^{(e)}(\phi(i^{(e)})) = \boldsymbol{M} \cdot P^{(e)}(\phi(i^{(e)}))$$

for all e.

The probability that there exists e_1 and e_2 such that the leading terms of $R^{(e_1)}(X)$ and $R^{(e_2)}(X)$ are the same, while those of $P^{(e_1)}(X)$ and $P^{(e_2)}(X)$ are different is at most $\binom{\tau}{2} \cdot \varepsilon_{\text{check}}$ where $\varepsilon_{\text{check}} := \max_{v \neq u} \Pr_{\boldsymbol{M} \leftarrow \$}[\boldsymbol{M}v = \boldsymbol{M}u]$. The parameter B is chosen such that $\binom{\tau}{2} \cdot \varepsilon_{\text{check}} \leq 2^{-\lambda}$. Let us note that this check leaks information about the leading term w. To prevent leakage about secret values, the $\lambda + B$ last coefficients of w can be chosen at random and $\boldsymbol{M}$ can be defined in the form $\boldsymbol{M} = [\boldsymbol{M'} \mid \boldsymbol{I}_{\lambda+B}]$. Because of this consistency check which relies on a random challenge from the verifier, the resulting zero-knowledge protocol has 7 rounds.

Witness as constant term. In the proof system described at the beginning of this section, we use some polynomials $P_1, \ldots, P_n$ such that $P_j(0) = w_j$ for all j, meaning the witness is encoded as the *constant term* of those polynomials. On the other hand, the VOLEitH commitment procedure merges those polynomials assuming that the witness is encoded as their *leading term*. In its original description, the TCitH framework relies on the former encoding (as for original Shamir's secret sharing) but it can easily support the latter encoding. On the other hand, the VOLEitH approach is constrained to use the leading-term encoding because ψ is $\mathbb{F}_2$-linear and so the merging strategy requires that the leading term lives in $\mathbb{F}_2$ (and not in $\mathbb{F}_{2^\lambda}$).

One advantage of the constant-term encoding is to be less expensive in terms of field multiplications. Because of the relative heaviness of the considered modelling for SD-in-the-Head-2 we propose the following tweak to support VOLEitH merging procedure while still encoding the witness in the constant term. We use the VOLEitH approach to commit to $\hat{P}(X) := X \cdot P(1/X)$. If the witness is encoded as the constant term of P, then it is encoded as the leading term of $\hat{P}$. To open the evaluation $P(\Delta)$, the prover simply opens the evaluation $\hat{P}(\Delta_{\text{inv}})$ with $\Delta_{\text{inv}} := \Delta^{-1}$, which allows the verifier to retrieve $P(\Delta)$ as $P(\Delta) = \Delta \cdot \hat{P}(\Delta_{\text{inv}})$.

2.2 Overview of SD-in-the-Head-2

The SD-in-the-Head-2 scheme relies on the proof system described in Section 2.1, using the VOLE-in-the-Head approach, which enables proving knowledge of a witness satisfying a set of degree-d polynomial constraints. In what follows, we explain how we express a syndrome decoding instance as such a system of degree-d constraints, in a way that aims for small witness size (and hence small signature size). We then describe the optimlized all-but-one vector commitment our scheme relies on. We finally address the transformation of the obtained 7-round interactive proof into a signature scheme using the standard Fiat-Shamir transformation.

SD modeling. We need to express a syndrome decoding instance as a system of degree-d constraints. To proceed, we will rely on the syndrome decoding modeling proposed by [OTX24; BBG+24]. The high-level idea is to consider a *structured* syndrome decoding problem, more precisely the *regular syndrome decoding problem* (RSD). However, as suggested in [BBG+24], *we use RSD parameters for which we have a provable security reduction to a secure unstructured syndrome decoding instance*, using the reduction from [FJR22]. Therefore, the security of SD-in-the-Head-2 still inherits from the conservative security of the oldest hard problem of code-based cryptography, namely the syndrome decoding problem for (unstructured) random linear codes.

Given a matrix $\boldsymbol{H} \in \mathbb{F}_2^{(n-k)\times n}$ and a syndrome vector $\boldsymbol{y} \in \mathbb{F}_2^{n-k}$, the RSD problem consists in finding a vector $\boldsymbol{x} \in \mathbb{F}_2^n$ such that

- $\boldsymbol{x}$ satisfies the linear relation $\boldsymbol{y} = \boldsymbol{H}\boldsymbol{x}$, and

- $\boldsymbol{x}$ is regular, meaning that it is the concatenation of w elementary vectors $e_1, \ldots, e_w$ of size $\frac{n}{w}$ (*i.e.* vector with $\frac{n}{w} - 1$ coefficients 0 and one coefficient 1).

To express an RSD instance as a contraint system, [OTX24; BBG+24] proposes to rely on a compression version of $x := (e_1 \parallel \ldots \parallel e_w)$: their idea consists in deriving each vector e_i as a tensor product of $\log_2\left(\frac{n}{w}\right)$ elementary vectors of size 2:

$$e_i \leftarrow \begin{pmatrix} b_{i,1} \\ 1 - b_{i,1} \end{pmatrix} \otimes \begin{pmatrix} b_{i,2} \\ 1 - b_{i,2} \end{pmatrix} \otimes \ldots \otimes \begin{pmatrix} b_{i,\log_2(n/w)} \\ 1 - b_{i,\log_2(n/w)} \end{pmatrix} \in \mathbb{F}^{n/w} ,$$

where $b_{i,1}, \ldots, b_{i,\log_2(n/w)}$ is the binary decomposition of the non-zero positions of e_i. In that case, the j^{th} coordinate of e_i is derived as

$$(e_i)_j \leftarrow \prod_{k=1}^{\log_2(n/w)} \left(b_{i,k} \oplus \overline{\mathsf{bit}_k(j)}\right) ,$$

where $\mathsf{bit}_k(j)$ is the k^{th} bit of the integer j and $\overline{\cdot}$ is the negation function. By doing this, we do not need to check that the e_i's are elementary since this property is guarantee by design (*i.e.* a vector e_i satisfying the above equation is always an elementary vector, whatever are the values $b_{i,1}, \ldots, b_{i,\log_2(n/w)}$). This construction leads to a witness of $w \cdot \log_2\left(\frac{n}{w}\right)$ bits which is verified using a system of degree-d constraints, with $d = \log_2\left(\frac{n}{w}\right)$. We thus obtain a small witness, but constraints of relatively high degree. We can relax [OTX24; BBG+24]'s idea with the following tweak. We build each elementary vector e_i as a tensor product of d elementary vectors of size $\mu_1, \ldots, \mu_d$ (with $\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_d = n/w$). We obtain a witness of bit-size

$$\sum_{i=1}^{d} (\mu_i - 1) ,$$

and a system of degree-d constraints. By taking μ_i a bit larger than 2 (for example, 4), we only slightly increase the witness size, but significantly we reduce the constraints' degree. On the other hand, by using elementary vectors of size $\mu_i > 2$, we need to add further (degree-2) constraints to prove that the e_i's are elementary vectors.

Specifically, the signer shall prove that they know bits $\{b_{i,j,k}\}_{1 \le i \le w, 1 \le j \le d, 1 \le k \le \mu_j - 1}$ such that

- we have $\boldsymbol{y} = \boldsymbol{H}\boldsymbol{x}$ with $x := (e_1 \parallel \ldots \parallel e_w)$, where e_i is defined as

$$e_i \leftarrow \begin{pmatrix} b_{i,1,1} \\ b_{i,1,2} \\ \vdots \\ b_{i,1,\mu_1-1} \\ 1 - \sum_k b_{i,1,k} \end{pmatrix} \otimes \begin{pmatrix} b_{i,2,1} \\ b_{i,2,2} \\ \vdots \\ b_{i,2,\mu_2-1} \\ 1 - \sum_k b_{i,2,k} \end{pmatrix} \otimes \ldots \otimes \begin{pmatrix} b_{i,d,1} \\ b_{i,d,2} \\ \vdots \\ b_{i,d,\mu_d-1} \\ 1 - \sum_k b_{i,d,k} \end{pmatrix} \in \mathbb{F}^{n/w}$$

  for all $1 \le i \le w$;

- for all pairs (i, j), $(b_{i,j,1}, \ldots, b_{i,j,\mu_1})$ has at most one non-zero coordinate, implying that

$$\begin{pmatrix} b_{i,j,1} \\ b_{i,j,2} \\ \vdots \\ b_{i,j,\mu_j-1} \\ 1 - \sum_k b_{i,j,k} \end{pmatrix}$$

is an elementary vector.

AVC Optimizations. As explained in Section 2.1, the proof system starts by committing independently to τ vector polynomials $P^{(1)}, \ldots, P^{(\tau)}$. It implies that we use τ all-but-one vector commitments, where each of them uses of GGM tree of N leaves. Therefore, we can optimize those all-but-one vector commitments by using a so-called batched all-but-one vector commitment (BAVC) scheme, which aims to be more efficient than τ independent AVC schemes. In SD-in-the-Head-2, we use the "one-tree" BAVC scheme described in [BBM+24]. Instead of considering τ independent GGM trees of N leaves in parallel, this scheme relies on a unique large GGM tree of $\tau \cdot N$ leaves where the i^{th} seed of the e^{th} parallel repetition is associated to the $(e \cdot N + i)^{\text{th}}$ leaf of the large GGM tree. As explained in [BBM+24], "opening all but τ leaves of the big tree is more efficient than opening all but one leaf in each of the τ smaller trees, because with high probability some of the active paths in the tree will merge relatively close to the leaves, which reduces the number of internal nodes that need to be revealed." Moreover, the authors of [BBM+24] suggest improving the previous approach by rejecting certain bad challenges for which the paths did not sufficiently merged. When the BAVC opening is such that the number of revealed nodes in the sibling paths exceeds a fixed threshold T_{open}, the opening is considered as a failure (*i.e.* it returns $\perp$), forcing the prover/signer to recompute another opening challenge by re-hashing with an incremented counter. This process is done until the number of revealed nodes is less than T_{open}. For example, if we consider $N = 256$ and $\tau = 16$, the number of revealed nodes is smaller than (or equal to) $T_{\text{open}} := 110$ with probability ≈ 0.2 (which is to be compared to $\tau \cdot \log_2(N) = 128$ nodes for the sibling paths of separated trees). The selected value of T_{open} induces a rejection probability $p_{\text{rej}} = 1 - 1/\theta$, for some $\theta \in (0, \infty)$, hence the signer needs to perform an average of θ hash computations for the opening challenge (instead of 1). While this strategy decreases the challenge space by a factor θ, it does not change the average number of hashes that must be computed in a forgery attempt against the signature scheme (since the latter is multiplied by θ). As noticed by the authors of [BBM+24], this strategy can be thought of as loosing $\log_2 \theta$ bit of security (because of a smaller challenge space) which are regained thanks to the implicit proof-of-work for finding a good challenge.

Fiat-Shamir transformation & grinding. To obtain the SD-in-the-Head-2 signature scheme from the SD-in-the-Head-2 zero-knowledge proof of knowledge, we rely on the Fiat-Shamir transformation [FS87] to remove the prover-verifier interactions. We further proceed with adding a random salt to enforce domain separation between signatures (and avoid seed collision issues). Each verifier challenge is computed as the output of an extendable-output function (XOF) which takes as input the data that the prover would send before receiving that challenge in an interactive protocol. The SD-in-the-Head-2 protocol is a 7-round proof system, so there are three challenges: the random matrix $\boldsymbol{M}$ of the consistency check, the random coefficients $\gamma_1, \ldots, \gamma_m$ to batch the polynomial constraints, and the evaluation point Δ for the opened evaluations. In SD-in-the-Head-2 scheme, we use a *grinding* proof-of-work in the Fiat-Shamir hash computation of the last challenge, as proposed in [BBM+24]. Together with the opening challenge, the signer samples a w_{pow}-bit value v_{pow} and keeps the opening challenge only if this additional value is zero, with w_{pow} a parameter of the scheme. If this additional value is non-zero, then the signer increments a counter and recompute another opening challenge with another w_{pow}-bit value. This process is repeated until obtaining a grinding value equal to zero. Let us remark that we can use the same counter for this grinding process and the rejection process due to the fact that the [BBM+24]'s BAVC scheme might return $\perp$ when the number of revealed nodes is larger than the chosen threshold T_{open}. This grinding tweak increases the cost of hashing the last challenge by a factor $2^{w_{\text{pow}}}$ and hence increases the soundness security of w_{pow} bits. As a result, we can take smaller parameters (N, τ) for the large tree, namely parameters achieving $\lambda - w_{\text{pow}}$ bits of security instead of λ. More precisely, the parameters N, τ and w_{pow} must be chosen such that $(d/N^\tau) \cdot 2^{-w_{\text{pow}}} \leq 2^{-\lambda}$ to achieve a λ-bit security.

3 Detailed algorithmic description

3.1 Main algorithms

3.1.1 Key generation

The key generation consists in sampling a regular syndrome decoding instance. The public key is the instance $(\boldsymbol{H}, \boldsymbol{y})$ while the secret key is composed of the instance and the associated solution $(\boldsymbol{H}, \boldsymbol{y}, \boldsymbol{x})$. Moreover, we consider that H is in standard form, *i.e.* $\boldsymbol{H} := (\boldsymbol{H'} \mid \boldsymbol{I}_{n-k})$.

We describe in Algorithm 1 the subroutine ExpandWitness which expands a λ-bit seed into a regular syndrome decoding solution $\boldsymbol{x}$ and builds the SD-in-the-Head-2 witness wit, where wit encodes the positions of the non-zero coefficients. To proceed, we first sample the position of the non-zero coefficients in each solution chunk of size m (Line 2) from which we derive the regular SD solution $\boldsymbol{x}$. Then we build the witness by encoding the positions. As explained in Section 2.2, we write each chunk as the tensor product of d elementary vectors of size $\mu_1, \ldots, \mu_d$. We omit the last coordinate of all the subvectors (Line 11), since it can be deduced from the others.

Algorithm 1 ExpandWitness

Input: a seed seed

▷ Expand the RSD solution

1: $\mathsf{prg} \leftarrow \mathsf{PRG.Init}(\mathsf{seed})$
2: $\mathsf{pos} \leftarrow \mathsf{SampleIntegers}(\mathsf{prg}, \{0, \ldots, m-1\}, w)$ ▷ $\mathsf{pos} \in \{0, \ldots, m-1\}^w$
3: **for** i from 0 to $w-1$ **do**
4: $\quad \mathsf{chunk}[i] := \mathsf{ElementaryVector}(m, \mathsf{pos}[i])$
5: $\boldsymbol{x} \leftarrow (\mathsf{chunk}[0] \parallel \ldots \parallel \mathsf{chunk}[w-1])$ ▷ $x \in \mathbb{F}_q^n$

▷ Build the witness

6: $\mathsf{wit} \leftarrow \emptyset$
7: **for** i from 0 to $w-1$ **do**
8: $\quad$ **for** j from 0 to $d-1$ **do**
9: $\quad\quad (\mathsf{pos}[i], p) \leftarrow (\mathsf{pos}[i] \ // \ \mu_j, \mathsf{pos}[i] \ \% \ \mu_j)$
10: $\quad\quad \boldsymbol{e} := (e_0, \ldots, e_{\mu_j - 1}) \leftarrow \mathsf{ElementaryVector}(\mu_j, p)$
11: $\quad\quad \mathsf{wit} \leftarrow \mathsf{wit} \parallel (e_0, \ldots, e_{\mu_j - 2})$
12: **return** $(\boldsymbol{x}, \mathsf{wit})$

The key generation is described in Algorithm 2. It first samples two seeds $\mathsf{seed_{sk}}$ and $\mathsf{seed_{pk}}$. The first seed $\mathsf{seed_{sk}}$ is used to expand the syndrome decoding solution $\boldsymbol{x}$ and the SD-in-the-Head witness wit through the subroutine ExpandWitness. The second seed $\mathsf{seed_{pk}}$ is used to generate the matrix $\boldsymbol{H'}$ which defines the parity check matrix $\boldsymbol{H} := (\boldsymbol{H'} \mid \boldsymbol{I}_{n-k})$. The algorithm then builds the public key by packing the seed $\mathsf{seed_{pk}}$ which encodes $\boldsymbol{H}$ and the vector $\boldsymbol{y} := \boldsymbol{H}\boldsymbol{x}$. It also builds the secret key by packing the seed $\mathsf{seed_{pk}}$, the vector $\boldsymbol{y}$ and the SD-in-the-Head-2 witness wit.

Algorithm 2 SD-in-the-Head-2 – Key Generation

$\mathsf{seed}_{\mathsf{sk}} \leftarrow \{0,1\}^\lambda$
$\mathsf{seed}_{\mathsf{pk}} \leftarrow \{0,1\}^\lambda$
$(\boldsymbol{x}, \mathsf{wit}) \leftarrow \mathsf{ExpandWitness}(\mathsf{seed}_{\mathsf{sk}})$ $\triangleright$ $x \in \mathbb{F}_q^n$, $|\mathsf{wit}|_2 = w \cdot ((\mu_1 - 1) + \ldots + (\mu_d - 1))$
$\boldsymbol{H'} \leftarrow \mathsf{ExpandH}(\mathsf{seed}_{\mathsf{pk}})$ $\triangleright$ $\boldsymbol{H'} \in \mathbb{F}_q^{(n-k)\times k}$
$\boldsymbol{y} = [\boldsymbol{H'} \mid \boldsymbol{I}_{n-k}] \cdot \boldsymbol{x_A}$ $\triangleright$ $\boldsymbol{y} \in \mathbb{F}_q^{n-k}$
$\mathsf{pk} = \mathsf{Serialize}(\mathsf{seed}_{\mathsf{pk}}, \boldsymbol{y})$
$\mathsf{sk} = \mathsf{Serialize}(\mathsf{seed}_{\mathsf{pk}}, \boldsymbol{y}, \mathsf{wit}, \mathsf{seed}_{\mathsf{sk}})$
return $(\mathsf{pk}, \mathsf{sk})$

3.1.2 Signing

The signing algorithm is described in Algorithm 3. After expanding the secret key as $(\boldsymbol{H'}, \boldsymbol{y}, \mathsf{wit})$, it samples a λ-bit salt which provides domain separation between signatures and a λ-bit root seed from which all the pseudo-randomness of the scheme will be derived. Then, using the routine BLC.Commit, it commits to random degree-1 vector polynomials $\boldsymbol{P}_{\mathsf{wit}}(X) \in (\mathbb{F}_{2^\lambda}[X])^{|\mathsf{wit}|}$ and $\boldsymbol{P}_{\mathsf{rnd}}(X) \in (\mathbb{F}_{2^\lambda}[X])^{(d-1)\lambda}$ such that $\boldsymbol{P}_{\mathsf{wit}}(0) = \mathsf{wit}$ and $\boldsymbol{P}_{\mathsf{rnd}}(0)$ is uniformly sampled from $\mathbb{F}_2^{(d-1)\lambda}$. While $\boldsymbol{P}_{\mathsf{wit}}$ encodes the witness in the PIOP protocol, $\boldsymbol{P}_{\mathsf{rnd}}$ aims to prevent witness leakage in the protocol. The hash digest h_{lines} is the BLC commitment to these polynomials. After the commitment phase, it runs the PIOP protocol using the routine PIOP.Prover.Run, namely it computes the degree-d polynomial P_α such that

$$P_\alpha(X) = P_0(X) \cdot X + \sum_{j=1}^{m} \gamma_j \cdot f_j(P_1(X), \ldots, P_{|\mathsf{wit}|}(X)).$$

where

- $P_1, \ldots, P_{|\mathsf{wit}|}$ are the witness polynomial, *i.e.* $\boldsymbol{P}_{\mathsf{wit}} := \left(P_1, \ldots, P_{|\mathsf{wit}|}\right)$;
- P_0 is the degree-$(d-1)$ masking polynomial built as

$$P_0(X) := \sum_{i=1}^{d-1} \left(\sum_{j=1}^{\lambda} \delta^{j-1} \cdot P_{\mathsf{rnd},i,j}(X) \right) \cdot X^{i-1}$$

  with $\boldsymbol{P}_{\mathsf{rnd}} := (P_{\mathsf{rnd},1,1}, \ldots, P_{\mathsf{rnd},1,\lambda}, \ldots, P_{\mathsf{rnd},d-1,1}, \ldots, P_{\mathsf{rnd},d-1,\lambda})$ and $(1, \delta, \delta^2 \ldots)$ is a $\mathbb{F}_2$-basis of $\mathbb{F}_{2^\lambda}$;
- $\{f_j\}_j$ are the degree-d polynomial contraints that the SD-in-the-Head-2 witness wit should satisfy (*c.f.* Section 2.2).

By design, P_α satisfies $P_\alpha(0) = 0$, so we can write P_α as $\sum_{i=1}^{d} \alpha_i \cdot X^i$ (*i.e.* without constant term). Then, the signing algorithm hashes P_α (by hashing its coefficients), samples a random evaluation point $\Delta \in \mathbb{F}_{2^\lambda}$ and opens the polynomial evaluations $\boldsymbol{P}_{\mathsf{wit}}(\Delta)$ and $\boldsymbol{P}_{\mathsf{rnd}}(\Delta)$ using the routine BLC.OpenRandomEvaluation. By "opening", we mean that the signer provides an opening data pdecom that enables the verifier to recover $\boldsymbol{p}_{\mathsf{wit}} := \boldsymbol{P}_{\mathsf{wit}}(\Delta)$ and $\boldsymbol{p}_{\mathsf{rnd}} := \boldsymbol{P}_{\mathsf{rnd}}(\Delta)$. The verifier can then check that those evaluations are consistent with the BLC commitment h_{lines}.

Algorithm 3 SD-in-the-Head-2 – Signing Algorithm

Input: a secret key sk and a message $\mathsf{msg} \in \{0,1\}^*$

$\triangleright$ Phase 0: Initialization.

$(\mathsf{seed_{pk}}, \boldsymbol{y}, \mathsf{wit}, \mathsf{seed_{sk}}) \leftarrow \mathsf{Deserialize}(\mathsf{sk}, \{0,1\}^\lambda, \mathbb{F}_q^{n-k}, \mathbb{F}_2^{|\mathsf{wit}|})$

$\boldsymbol{H'} \leftarrow \mathsf{ExpandH}(\mathsf{seed_{pk}})$ $\triangleright$ $\boldsymbol{H'} \in \mathbb{F}_q^{(n-k)\times k}$

$\mathsf{salt} \leftarrow \{0,1\}^\lambda$

$\mathsf{rseed} \leftarrow \{0,1\}^\lambda$

$\triangleright$ Phase 1: Build & Commit Witness/Masking Polynomials.

$(\boldsymbol{P}_{\mathsf{wit}}, \boldsymbol{P}_{\mathsf{rnd}}, h_{\text{lines}}, \mathsf{key}, \mathsf{aux}) \leftarrow \mathsf{BLC.Commit}(\mathsf{salt}, \mathsf{rseed}, \mathsf{wit})$ $\triangleright$ $\boldsymbol{P}_{\mathsf{wit}}(0) = \mathsf{wit}$

$\triangleright$ Phase 2: PIOP Protocol (prover side).

$P_\alpha \leftarrow \mathsf{PIOP.Prover.Run}(\boldsymbol{P}_{\mathsf{wit}}, \boldsymbol{P}_{\mathsf{rnd}}, h_{\text{lines}}, (\boldsymbol{H'}, \boldsymbol{y}))$ $\triangleright$ $P_\alpha(X) \in \mathbb{F}_{2^\lambda}^{(\leq d)}[X]$

$h_{\text{piop}} = \mathsf{Hash}_{\text{piop}}(\mathsf{pk}, h_{\text{lines}}, \alpha_1, \ldots, \alpha_d, \mathsf{msg})$ $\triangleright$ $P_\alpha(X) = \sum_{0<i\leq d} \alpha_i \cdot X^i$

$\triangleright$ Phase 3: Open random evaluations.

$\mathsf{pdecom} \leftarrow \mathsf{BLC.OpenRandomEvaluation}(\mathsf{key}, h_{\text{piop}})$

$\sigma = \mathsf{Serialize}\,(\mathsf{salt} \parallel h_{\text{piop}} \parallel \mathsf{aux} \parallel \mathsf{pdecom} \parallel (\alpha_1, \alpha_2, \ldots, \alpha_d))$

return σ

3.1.3 Verification

The verification algorithm is described in Algorithm 4. After expanding the public key as $(\boldsymbol{H'}, \boldsymbol{y})$ and parsing the signature, it recovers the evaluations $\boldsymbol{p}_{\mathsf{wit}} := \boldsymbol{P}_{\mathsf{wit}}(\Delta)$ and $\boldsymbol{p}_{\mathsf{rnd}} := \boldsymbol{P}_{\mathsf{rnd}}(\Delta)$ from pdecom as well as h_{hlines} using the routine BLC.RecomputeEvaluation. If the recovered h_{hlines} does not match the value of the signing algorithm, the later verification $h_{\text{piop}}{=}h'_{\text{piop}}$ will fail with overwhelming probability, since h_{hlines} is input of the hash computation of h'_{piop}. At the same time, BLC.RecomputeEvaluation outputs the random evaluation point Δ. The verification algorithm then deduces the evaluation $p_\alpha = P_\alpha(\Delta)$ using the routine PIOP.ComputeOutput and check that it is consistent with the polynomial P_α included in the signature. Finally, it checks that the polynomial P_α is the same as in the signing algorithm by comparing the hash digests.

Algorithm 4 SD-in-the-Head-2 – Verification Algorithm

Input: a public key pk, a signature σ and a message $\mathsf{msg} \in \{0,1\}^*$

▷ Phase 0: Initialization.

$(\mathsf{salt} \parallel h_{\text{piop}} \parallel \mathsf{aux} \parallel \mathsf{pdecom} \parallel (\alpha_1, \alpha_2, \dots, \alpha_d)) \leftarrow \mathsf{Deserialize}(\sigma)$
$(\mathsf{seed}_{\mathsf{pk}}, \boldsymbol{y}) \leftarrow \mathsf{Deserialize}(\mathsf{pk})$
$\boldsymbol{H'} \leftarrow \mathsf{ExpandH}(\mathsf{seed}_{\mathsf{pk}})$ ▷ $\boldsymbol{H'} \in \mathbb{F}_q^{(n-k)\times k}$

▷ Phase 1: Recomputing Evaluation.

$(\Delta, \boldsymbol{p}_{\mathsf{wit}}, \boldsymbol{p}_{\mathsf{rnd}}, h_{\text{lines}}) \leftarrow \mathsf{BLC.RecomputeEvaluation}(\mathsf{aux}, \mathsf{pdecom}, \mathsf{salt}, h_{\text{piop}})$
▷ $\boldsymbol{p}_{\mathsf{wit}} := \boldsymbol{P}_{\mathsf{wit}}(\Delta), \boldsymbol{p}_{\mathsf{rnd}} := \boldsymbol{P}_{\mathsf{rnd}}(\Delta)$
if $\Delta = \perp$ **then**
  return Reject

▷ Phase 2: PIOP Protocol (verifier side).

$p_\alpha \leftarrow \mathsf{PIOP.Verifier.Run}(\Delta, \boldsymbol{p}_{\mathsf{wit}}, \boldsymbol{p}_{\mathsf{rnd}}, h_{\text{lines}}, (\boldsymbol{H'}, \boldsymbol{y}))$ ▷ $p_\alpha := P_\alpha(\Delta)$
$h'_{\text{piop}} = \mathsf{Hash}_{\text{piop}}(\mathsf{pk}, h_{\text{lines}}, \alpha_1, \dots, \alpha_d, \mathsf{msg})$ ▷ $P_\alpha(X) = \sum_{0<i\leq d} \alpha_i \cdot X^i$

▷ Phase 3: Verification.

if $h_{\text{piop}} \neq h'_{\text{piop}}$ or $p_\alpha \neq \sum_{i=1}^{d} \alpha_i \cdot \Delta^i$ **then**
  return Reject
return Accept

3.2 Subroutines

3.2.1 Symmetric cryptography primitives

The SD-in-the-Head-2 signature scheme relies on three types of symmetric cryptography primitives: a hash function (Hash), and an extendable output function (XOF), and a block cipher (Enc). The instantiations of these primitives are summarized in Table 1.

Table 1: Symmetric cryptography primitives for NIST Security Categories I, III, and V. For Category III, the block cipher is defined as a truncated version of Rijndael-256-256.

	Category I	Category III	Category V
Enc	AES-128	Rijndael-256-256*	Rijndael-256-256
Hash	SHAKE-128	SHAKE-256	SHAKE-256
XOF	SHAKE-128	SHAKE-256	SHAKE-256

3.2.2 Pseudo-randomness generation

Several subroutines used in the SD-in-the-Head-2 signature schemes involve pseudorandomness generation from a seed. Several seeds are expanded from a master seed in the key generation and in the signature algorithm (to generate the lines). One also needs to sample sequences of field elements from a seed in the key generation, the signature and verification algorithms. Finally pseudorandomness generation is also involved to derive the challenges (consistency challenge, batching challenge, and evaluation challenge) from the Fiat-Shamir hashes h_{aux}, h_{lines} and h_{piop}.

Pseudo-random generator. Most the pseudorandomness (everything except the Fiat-Shamir challenges) in SD-in-the-Head-2 is generated through a pseudo-random generator (PRG). Such a function takes a λ-bit seed $\mathsf{seed} \in \{0,1\}^\lambda$ and produces an arbitrary-long output bit-string $y \in \{0,1\}^*$ whose length is tailored to the requirements of the application. Formally, a PRG is equipped with two routines: $\mathsf{PRG.Init}(x)$ initializes the PRG state with the input $x \in \{0,1\}^*$. Once initialized, the PRG can be queried with the routine $\mathsf{PRG.GetByte}()$ to generate the next byte of the output y associated to x. In our context, we use a block cipher Enc in counter mode as a secure pseudorandom generator (PRG). The concrete instance of the block cipher we use in the SD-in-the-Head-2 scheme is given in Section 3.2.1. The initialization function $\mathsf{PRG.Init}(x, \mathsf{salt})$ might take an additional input salt, which will correspond to the nounce for the counter mode. Regarding Category III, since we use Rijndael-256-256 as mentioned in Section 3.2.1, we pad the input seed and salt with 64 least significant zero bits, and the output of the PRG corresponds to the output blocks of the cipher (without any truncation).

Extendable output function. The pseudorandomness in SD-in-the-Head-2 for Fiat-Shamir challenges is generated through an extendable output hash function (XOF). Such a function takes an arbitrary-long input bit-string $x \in \{0,1\}^*$ and produces an arbitrary-long output bit-string $y \in \{0,1\}^*$ whose length is tailored to the requirements of the application. Formally, a XOF is equipped with two routines: $\mathsf{XOF.Init}(x)$ initializes the XOF state with the input $x \in \{0,1\}^*$. Once initialized, the XOF can be queried with the routine $\mathsf{XOF.GetByte}()$ to generate the next byte of the output y associated to x. The concrete instance of the XOF we use in the SD-in-the-Head-2 scheme is given in Section 4.3. In our context, we use the XOF as a secure pseudorandom generator (PRG) which tolerates input seeds of variable lengths.

Sampling (a vector of) field elements. The subroutine SampleFieldElements(src, $\mathbb{F}, n$) samples n pseudorandom elements from $\mathbb{F}$ using the PRG/XOF src. It assumes that the XOF/PRG has been previously initialized. The implementation of the SampleFieldElements routine use the following process. It first generates from src a stream of bytes $B_1, \ldots, B_{n'}$ for some $n' \geq (n \cdot \log_2 |\mathbb{F}|)/8$. Those bytes are converted into n field elements as follows:

- For $\mathbb{F}_q = \mathbb{F}_2$: The byte B_i is interpreted as 8 field elements $b_{i,0}, \ldots, b_{i,7}$, such that $B_i = \sum_j 2^j \cdot b_{i,j}$. Interpreted all the bytes leads to a vector $(b_{1,0}, \ldots, b_{1,7}, \ldots, b_{n',0}, \ldots, b_{n',7})$ and the procedure returned the n first coordinates as the sampled field elements.
- For $\mathbb{F}_q = \mathbb{F}_{2^\lambda}$: The i^{th} pack of $\frac{\lambda}{8}$ consecutive bytes is returned as the ith sampled field element.

Sampling integers. The subroutine SampleIntegers(src, $\{0, \ldots, m-1\}, n$) samples n pseudorandom integers from $\{0, \ldots, m-1\}$ using the XOF/PRG src, where $m \leq 2^{32}$. It assumes that the XOF/PRG has been previously initialized. The implementation of the SampleIntegers routine uses the principle of rejection sampling. While denoting t_{max} as the largest multiple of m smaller then (or equal to) 2^{32}, the procedure goes as follows:

$i = 1$
while $i \leq n$ **do**
  for $0 \leq j < 4$ **do**
    $B_j \leftarrow$ src.GetByte()
  $\mathsf{v} \leftarrow B_0 + 256 \cdot B_1 + 256^2 \cdot B_2 + 256^3 \cdot B_3$
  if $\mathsf{v} \in \{0, 1, \ldots, t_{max} - 1\}$ **then**
    $f_i = B$ % m; i ++
return $(f_1, \ldots, f_n)$

The number of generated bytes which are necessary to complete the process is non-determinisitic.

Expansion of the parity-check matrix. The subroutine ExpandH takes as input λ-bit seed seed$_H$ and returns an $(n-k) \times k$ matrix of elements of $\mathbb{F}_q$. This generated matrix is the random part $\boldsymbol{H'}$ of the parity-check matrix in standard form $\boldsymbol{H} = (\boldsymbol{H'}|\boldsymbol{I}_{n-k})$. A call to ExpandH(seed$_H$) generates $\boldsymbol{H'}$ row-wise as follows:

prg $\leftarrow$ PRG.Init(seed$_H$)
for i from 0 to $k-1$ **do**
  cols$_{\boldsymbol{H'}}[i] \leftarrow$ SampleFieldElements(prg, $\mathbb{F}_q, n-k$)
$\boldsymbol{H'} \leftarrow [\text{cols}_{\boldsymbol{H'}}[0] \mid \ldots \mid \text{cols}_{\boldsymbol{H'}}[k-1]]$ ▷ $\boldsymbol{H'} \in \mathbb{F}_2^{(n-k)\times k}$
return $\boldsymbol{H'}$

Seed expansion (GGM Tree). The subroutine ExpandSeed expands a salt, a parent seed and an index into two seeds. It is used to expand GGM trees. Specifically, a call to ExpandSeed(salt, seed, idx) runs the following procedure for Categories I and V:

$\mathsf{left} \leftarrow \mathsf{Enc}_\lambda(\mathsf{key} = \mathsf{seed}, \mathsf{ptx} = \mathsf{salt} \oplus \mathsf{MapToBits}(2 \cdot \mathsf{idx}))$
$\mathsf{right} \leftarrow \mathsf{Enc}_\lambda(\mathsf{key} = \mathsf{seed}, \mathsf{ptx} = \mathsf{salt} \oplus \mathsf{MapToBits}(2 \cdot \mathsf{idx} + 1))$
return $(\mathsf{left}, \mathsf{right})$

where MapToBits computes the bitstring that corresponds to the binary decomposition of the input integer in a little-endian order. For Category III, a call to ExpandSeed(salt, seed, idx) runs as follows:

$\mathsf{left} \leftarrow \text{Rijndael-256-256}(\mathsf{key} = (0^{64} \parallel \mathsf{seed}), \mathsf{ptx} = (0^{64} \parallel \mathsf{salt}) \oplus \mathsf{MapToBits}(2 \cdot \mathsf{idx}))$
$\mathsf{right} \leftarrow \text{Rijndael-256-256}(\mathsf{key} = (0^{64} \parallel \mathsf{seed}), \mathsf{ptx} = (0^{64} \parallel \mathsf{salt}) \oplus \mathsf{MapToBits}(2 \cdot \mathsf{idx} + 1))$
return $(\mathsf{GetMSB}_{192}(\mathsf{left}), \mathsf{GetLSB}_{192}(\mathsf{right}))$

where GetMSB_{192} and GetLSB_{192} return respectively the 192 most significant bits and the 192 less significant bits.

Expansion of consistency check challenge. The subroutine ExpandConsistencyChallenge expands the first Fiat-Shamir hash h_{aux} into the matrix $\boldsymbol{M}$ used for the consistency check. It consists of the following steps:

$\mathsf{xof} \leftarrow \mathsf{XOF.Init}(h_{\text{aux}})$
for i from 0 to $\ell - 1$ **do**
 $\mathsf{cols}_{\boldsymbol{M}}[i] \leftarrow \mathsf{SampleFieldElements}(\mathsf{xof}, \mathbb{F}_2, \lambda + B)$
$\boldsymbol{M} \leftarrow [\mathsf{cols}_{\boldsymbol{M}}[0] \mid \ldots \mid \mathsf{cols}_{\boldsymbol{M}}[\ell - 1]]$ ▷ $\boldsymbol{M} \in \mathbb{F}_2^{(\lambda+B)\times\ell}$
return $\boldsymbol{M}$

Expansion of batching challenge. The subroutine ExpandBatchingChallenge expands the second Fiat-Shamir hash h_{lines} into the batching challenges $(\boldsymbol{\gamma}', \boldsymbol{\gamma})$. It consists of the following steps:

$\mathsf{xof} \leftarrow \mathsf{XOF.Init}(h_{\text{lines}})$
$\boldsymbol{\gamma}' \leftarrow \mathsf{SampleFieldElements}(\mathsf{xof}, \mathbb{F}_{2^\lambda}, w \cdot d')$, where $d' = \#\{j : \mu_j > 2\}$
$\boldsymbol{\gamma} \leftarrow \mathsf{SampleFieldElements}(\mathsf{xof}, \mathbb{F}_{2^\lambda}, \lceil \frac{n-k}{\lambda} \rceil)$
return $(\boldsymbol{\gamma}', \boldsymbol{\gamma})$

Expansion of evaluation-opening challenge. The subroutine ExpandEvaluationChallenge expands the third Fiat-Shamir hash h_{piop} into the evaluation-opening challenges $i^*[1], \ldots, i^*[\tau]$, where $i^*[e]$ is the hidden seed which should remain hidden for execution e. It also expands a w_{pow}-bit grinding digest v_{pow}, which will lead to a challenge rejection when it is non-zero. This subroutine takes as input the hash h_{piop} and a 32-bit counter ctr. It consists of the following steps:

$\mathsf{xof} \leftarrow \mathsf{XOF.Init}(h_{\text{piop}}, \mathsf{ctr})$
$\{i^*[e]\}_{e<\tau} \leftarrow \mathsf{SampleIntegers}(\mathsf{xof}, \{0, \ldots, 2^\kappa - 1\}, \tau)$
$v_{\text{pow}} \leftarrow \mathsf{SampleFieldElements}(\mathsf{xof}, \mathbb{F}_2, w_{\text{pow}})$
return $(\{i^*[e]\}_{e<\tau}, v_{\text{pow}})$

3.2.3 Hashing and commitments

Several subroutines used in the SD-in-the-Head signature scheme involve cryptographic hashing. This is the case of the subroutines computing the Fiat-Shamir hashes and the commitments. We also use a cryptographic a hash function for the seed trees (hypercube variant) and the Merkle trees (threshold variant).

Cryptographic hash function. The different hash and commitment subroutines are all derived from a common cryptographic hash function

$$\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^{2\lambda} .$$

The concrete instance of the hash function we use in the SD-in-the-Head scheme is given in Section 4.3.

We use domain separation for the different usages of the hash function. This is simply done by prepending a fixed byte value to the data to be hashed, as specified below for the different cases.

Seed Commitments. The subroutine CommitSeed takes as input a λ-bit salt, an index idx and a λ-bit seed, and it outputs a commitment digest for this seed. For performance reason, while we used hashing to commit to seeds in the first version of SD-in-the-Head, we use a block cipher in the second version. For Categories I and V, we build the commitment digest com for the seed seed as follows:

1: $\mathsf{tweak} \leftarrow 2 \cdot (\tau \cdot N + \mathsf{idx})$
2: $\mathsf{left} \leftarrow \mathsf{Enc}_\lambda(\mathsf{key} = \mathsf{seed}, \mathsf{ptx} = \mathsf{salt} \oplus \mathsf{MapToBits}(\mathsf{tweak}))$
3: $\mathsf{right} \leftarrow \mathsf{Enc}_\lambda(\mathsf{key} = \mathsf{seed}, \mathsf{ptx} = \mathsf{salt} \oplus \mathsf{MapToBits}(\mathsf{tweak} + 1))$
4: $\mathsf{com} \leftarrow (\mathsf{left} \parallel \mathsf{right})$ ▷ $\mathsf{com} \in \{0,1\}^{2\lambda}$
5: **return** com

where MapToBits computes the bitstring that corresponds to the binary decomposition of the input integer in a little-endian order. For Category III, we build the commitment digest as follows:

1: $\mathsf{tweak} \leftarrow 2 \cdot (\tau \cdot N + \mathsf{idx})$
2: $\mathsf{left} \leftarrow \mathsf{Rijndael\text{-}256\text{-}256}(\mathsf{key} = 0^{64} \parallel \mathsf{seed}, \mathsf{ptx} = (0^{64} \parallel \mathsf{salt}) \oplus \mathsf{MapToBits}(\mathsf{tweak}))$
3: $\mathsf{right} \leftarrow \mathsf{Rijndael\text{-}256\text{-}256}(\mathsf{key} = 0^{64} \parallel \mathsf{seed}, \mathsf{ptx} = (0^{64} \parallel \mathsf{salt}) \oplus \mathsf{MapToBits}(\mathsf{tweak} + 1))$
4: $\mathsf{com} \leftarrow (\mathsf{GetMSB}_{192}(\mathsf{left}) \parallel \mathsf{GetLSB}_{192}(\mathsf{right}))$ ▷ $\mathsf{com} \in \{0,1\}^{2\lambda}$
5: **return** com

where GetMSB_{192} and GetLSB_{192} return respectively the 192 most significant bits and the 192 less significant bits.

Fiat-Shamir Hashes. The hash functions used to derive the Fiat-Shamir Hashes are defined as:

$$\begin{aligned}
\mathsf{Hash}_{\mathrm{bavc}}(\mathsf{data}) &:= \mathsf{Hash}(1 \parallel \mathsf{data}) \\
\mathsf{Hash}_{\mathrm{aux}}(\mathsf{data}) &:= \mathsf{Hash}(2 \parallel \mathsf{data}) \\
\mathsf{Hash}_{\mathrm{lines}}(\mathsf{data}) &:= \mathsf{Hash}(3 \parallel \mathsf{data}) \\
\mathsf{Hash}_{\mathrm{piop}}(\mathsf{data}) &:= \mathsf{Hash}(4 \parallel \mathsf{data})
\end{aligned}$$

where the prefixes 1, 2, 3 and 4 are encoded on one byte.

3.2.4 All-but-one vector commitments

As the first version of SD-in-the-Head, SD-in-the-Head-2 relies on an *all-but-one vector commitment* (AVC). This primitive commits N random λ-bit seeds and later opens/reveals all of them except one. In practice, since we need τ such sets of N seeds, we will rely on the *batched* variant leveraging the one-tree optimisation from [BBM+24].

Commitment routine. The commitment routine of the batched all-but-one vector commitment (BAVC) scheme is described in Algorithm 5. After expanding a large GGM tree of $\tau \cdot N$ leaves using the routine ExpandSeed, it commits each seed using the routine CommitSeed where the $(j+1)^{\text{th}}$ seed of the $(i+1)^{\text{th}}$ repetition is the $(i \cdot \tau + e + 1)^{\text{th}}$ leaf of the big tree. Then it hashes all the seed commitments and the resulting digest h_{com} forms the global commitment of all the seeds. It outputs the tree and the seed commitments as the opening key.

Algorithm 5 BAVC.Commit

Input: a salt $\mathsf{salt} \in \{0,1\}^\lambda$ and a root seed $\mathsf{rseed} \in \{0,1\}^\lambda$

▷ Expand the GGM tree

$\mathsf{tree}[1] \leftarrow \mathsf{rseed}$
for i from 1 to $(\tau \cdot N - 1)$ **do**
  $(\mathsf{tree}[2i], \mathsf{tree}[2i+1]) \leftarrow \mathsf{ExpandSeed}(\mathsf{salt}, \mathsf{tree}[i], i)$

▷ Commit the seeds

for e from 0 to $(\tau - 1)$ **do**
  for i from 0 to $(N-1)$ **do**
    $\mathsf{seeds}[e][i] \leftarrow \mathsf{tree}[\tau \cdot N + (i \cdot \tau + e)]$
    $\mathsf{commit}[e][i] \leftarrow \mathsf{CommitSeed}(\mathsf{salt}, \mathsf{seeds}[e][i], i \cdot \tau + e)$

▷ Set commitment and key

$h_{\text{com}} \leftarrow \mathsf{Hash}_{\text{bavc}}(\{\mathsf{commit}[e][i]\}_{e,i})$
$\mathsf{key} \leftarrow \mathsf{tree} \,\|\, \mathsf{commit}$
return $(\mathsf{seeds}, h_{\text{com}}, \mathsf{key})$

Opening routine. The opening routine of the BAVC scheme is described in Algorithm 6. It takes as input the opening key (the tree nodes and the seed commitments) and the list of the seeds to remain hidden. It outputs the opening BAVC proof π_{BAVC}, which will enable the verifier to recompute all the revealed leaves while checking their consistency with the BAVC commitment h_{com}. To proceed, the routine first searches the smallest set of tree nodes that enables to recompute all the leaves excluding those in the input list. This search algorithm involves a queue structure which comes with four dedicated subroutines:

- Queue.Init() returns a empty queue,
- Queue.Enqueue(v) pushes a value v at the end of the queue,
- Queue.Dequeue() pops the value which is at the top of the queue, and
- Queue.Head() returns the value which is at the top of the queue *without removing it.*

This routine follows the paths between the hidden leaves and the tree root, and at each intermediary node, decides whetever the sibling node should be in the output set. If the number of revealed nodes is larger than T_{open}, the routine aborts. At the end of the search algorithm, it pads the computed sibling paths with zeroes such that the result is always of $T_{\text{open}} \cdot \lambda$ bits. Finally, it outputs the opening BAVC proof π_{BAVC} made of the sibling paths and the commitments of all the hidden leaves.

Algorithm 6 BAVC.Open

Input: a BAVC key key and a set $\{i^*[e]\}_{e<\tau}$ of τ indexes in $\{0,\ldots,N-1\}$

$(\mathsf{tree}, \mathsf{commit}) \leftarrow \mathsf{key}$
$\mathsf{queue} \leftarrow \mathsf{Queue.Init}()$
for $\mathsf{idx} \in \{i^*[e] \cdot \tau + e : e \in \{0,\ldots,\tau-1\}\}$ in the *decreasing* order **do**
  $\mathsf{queue.Enqueue}(\tau \cdot N + \mathsf{idx})$
$\mathsf{path} \leftarrow \emptyset$
while $\mathsf{queue.Head}() \neq 1$ **do** ▷ While the queue head is not the root
  $\mathsf{node\_idx} \leftarrow \mathsf{queue.Dequeue}()$
  if $|\mathsf{queue}| \geq 2$ **then**
    $\mathsf{sibling\_idx} = \mathsf{node\_idx} \oplus 1$
    if $\mathsf{queue.Head}() = \mathsf{sibling\_idx}$ **then** ▷ Check if the queue head is the sibling node
      $\mathsf{queue.Dequeue}()$
    else if $|\mathsf{path}| < T_{\text{open}} \cdot \lambda$ **then** ▷ Check if the sibling path is not too long
      $\mathsf{path} \leftarrow \mathsf{path} \,\|\, \mathsf{tree}[\mathsf{sibling\_idx}]$ ▷ Append the sibling node to the path
    else
      return $\perp$ ▷ Return failure since the sibling path is too long
  $\mathsf{queue.Enqueue}(\lfloor \mathsf{node\_idx}/2 \rfloor)$
$\mathsf{path} \leftarrow \mathsf{PadWithZero}(\mathsf{path}, T_{\text{open}} \cdot \lambda)$ ▷ $\mathsf{path} \in \{0,1\}^{T_{\text{open}} \cdot \lambda}$
$\mathsf{commit}_{i^*} \leftarrow (\mathsf{commit}[0][i^*[0]], \ldots, \mathsf{commit}[\tau-1][i^*[\tau-1]])$ ▷ $\mathsf{commit}_{i^*} \in \{0,1\}^{\tau \cdot (2\lambda)}$
$\pi_{\mathsf{BAVC}} \leftarrow (\mathsf{path}, \mathsf{commit}_{i^*})$
return π_{BAVC}

Reconstruction routine. The reconstruction routine of the BAVC scheme is described in Algorithm 7. It takes as input the list of hidden seeds, the BAVC opening proof π_{BAVC} (containing the sibling paths and the commitments of the hidden seeds) and the salt. Using the same search algorithm as in the opening routine, it prefills the tree with the nodes in the sibling paths. It checks whetever the used padding is valid (*i.e.* the padded bits are only zeroes). The routine then expands the GGM tree using the prefilled nodes and recomputes the commitments of all the revealed seeds. Finally, it recomputes the BAVC commitment h_{com} and returns it together with the revealed seeds.

Algorithm 7 BAVC.Reconstruct

Input: a set $\{i^*[e]\}_{e<\tau}$ of τ indexes in $\{0,\dots,N-1\}$, a BAVC opening π_{BAVC} and a salt $\mathsf{salt} \in \{0,1\}^\lambda$

▷ Prefill the partial GGM tree

$(\mathsf{path}, \mathsf{commit}_{i^*}) \leftarrow \pi_{\mathsf{BAVC}}$
$\mathsf{queue} \leftarrow \mathsf{Queue.Init}()$
for $\mathsf{idx} \in \{i^*[e]\cdot\tau + e : e \in \{0,\dots,\tau-1\}\}$ in the *decreasing* order **do**
  $\mathsf{queue.Enqueue}(\tau\cdot N + \mathsf{idx})$
$\mathsf{tree}[1], \dots, \mathsf{tree}[2\cdot\tau\cdot N-1] \leftarrow \emptyset,\dots,\emptyset$
while $\mathsf{queue.Head}() \neq 1$ **do** ▷ While the queue head is not the root
  $\mathsf{node\_idx} \leftarrow \mathsf{queue.Dequeue}()$
  if $|\mathsf{queue}| \geq 2$ **then**
    $\mathsf{sibling\_idx} = \mathsf{node\_idx} \oplus 1$
    if $\mathsf{queue.Head}() = \mathsf{sibling\_idx}$ **then** ▷ Check if the queue head is the sibling node
      $\mathsf{queue.Dequeue}()$
    else if $|\mathsf{path}| < T_{\text{open}}\cdot\lambda$ **then** ▷ Check if the sibling path is not too long
      $(\mathsf{tree}[\mathsf{sibling\_idx}], \mathsf{path}) \leftarrow \mathsf{path}$ ▷ Extract the λ first bits of path
    else
      return $\perp$ ▷ Return failure since the sibling path is too long
  $\mathsf{queue.Enqueue}(\lfloor \mathsf{node\_idx}/2 \rfloor)$
if $|\mathsf{path}| > 0$ and $\mathsf{path} \neq 0$ **then** ▷ Check that the padding is correct
  return $\perp$

▷ Expand the partial GGM tree

for i from 1 to $(\tau\cdot N-1)$ **do**
  if $\mathsf{tree}[i] \neq \emptyset$ **then**
    $(\mathsf{nodes}[2i], \mathsf{nodes}[2i+1]) \leftarrow \mathsf{ExpandSeed}(\mathsf{salt}, \mathsf{nodes}[i], i)$

▷ Recompute commitment

for e from 0 to $(\tau-1)$ **do**
  for i from 0 to $(N-1)$ **do**
    if $i \neq i^*[e]$ **then**
      $\mathsf{seeds}[e][i] \leftarrow \mathsf{nodes}[\tau\cdot N + (i\cdot\tau+e)]$
      $\mathsf{com}[e][i] \leftarrow \mathsf{CommitSeed}(\mathsf{salt}, \mathsf{seeds}[e][i], i\cdot\tau+e)$
    else
      $\mathsf{seeds}[e][i] \leftarrow \emptyset$
      $(\mathsf{com}[e][i] \,\|\, \mathsf{commit}_{i^*}) \leftarrow \mathsf{commit}_{i^*}$
$h_{\text{com}} \leftarrow \mathsf{Hash}_{\text{bavc}}(\{\mathsf{com}[e][i]\}_{e,i})$
return $(h_{\text{com}}, \mathsf{seeds})$

3.2.5 Batch line commitment

As explained in Section 2.1, the prover in the PIOP protocol of SD-in-the-Head-2 needs to commit degree-1 vector polynomials $\boldsymbol{P}_{\mathsf{wit}} := (P_1, \ldots, P_{|\mathsf{wit}|}) \in \big(\mathbb{F}_{2^\lambda}[X]\big)^{|\mathsf{wit}|}$ such that $\boldsymbol{P}_{\mathsf{wit}}(0) = \mathsf{wit} \in \mathbb{F}_2^{|\mathsf{wit}|}$, and $\boldsymbol{P}_{\mathsf{rnd}} := (P_{\mathsf{rnd},1,1}, \ldots, P_{\mathsf{rnd},1,\lambda}, \ldots, P_{\mathsf{rnd},d-1,1}, \ldots, P_{\mathsf{rnd},d-1,\lambda}) \in \big(\mathbb{F}_{2^\lambda}[X]\big)(d-1)\lambda$ such that $\boldsymbol{P}_{\mathsf{rnd}}(0) \in \mathbb{F}_2^{(d-1)\lambda}$. Therefore, we propose a primitive named *batch line commitment* following the VOLEitH approach which is dedicated to commit (and later open evaluations of) these polynomials.

Gray code ϕ_{Gray}. The batch line commitment depends on a public one-to-one function $\phi : \{1, \ldots, N\} \to \mathbb{F}_2^{1\times\kappa}$. This function is used in the commitment procedure to commit polynomials while enabling the later opening of one evaluation on a point from $S := \{\phi(0), \ldots, \phi(N-1)\}$. While the definition of ϕ has no importance in the correctness and the soundness of the scheme, it might impact the performance. In the SD-in-the-Head-2 signature scheme, we use the Gray code for ϕ, namely we use

$$\phi_{\mathrm{Gray}} : i \in \{0, \ldots, N-1\} \mapsto \mathsf{bin}_\kappa(i) \oplus (\mathsf{bin}_\kappa(i){>>}1) \ ,$$

where

$$\mathsf{bin}_\kappa(i) := (b_{\kappa-1}, \ldots, b_0) \in \mathbb{F}_2^{1\times\kappa},$$
$$\mathsf{bin}_\kappa(i){>>}1 := (0, b_{\kappa-1}, \ldots, b_1) \in \mathbb{F}_2^{1\times\kappa},$$

with $i = \sum_{j=0}^{\kappa-1} b_j \cdot 2^j$. This code has the nice property that two consecutive values $\phi_{\mathrm{Gray}}(i)$ and $\phi_{\mathrm{Gray}}(i+1)$ differ on only one position.

In Algorithm 8 and Algorithm 10, ϕ_{Gray} is used to compute

$$\boldsymbol{r}_{\mathsf{acc}} \leftarrow \sum_{i=0}^{N-1} \boldsymbol{r}_{\mathsf{rnd},i} \ ,$$
$$\boldsymbol{r}_{\mathsf{base}} \leftarrow \sum_{i=0}^{N-1} \phi_{\mathrm{Gray}}(i) \cdot \boldsymbol{r}_{\mathsf{rnd},i} \ .$$

Thanks to the structure of ϕ_{Gray}, we can compute $\boldsymbol{r}_{\mathsf{acc}}$ and $\boldsymbol{r}_{\mathsf{base}}$ with only $2N$ bit operations, while it would be in $O(N \cdot \kappa)$ if we would use a natural bit-representation for ϕ. Indeed, we can compute

$$\boldsymbol{r}_{\mathsf{acc},j} \leftarrow \sum_{i=0}^{j} \boldsymbol{r}_{\mathsf{rnd},i} \tag{4}$$
$$\boldsymbol{r}_{\mathsf{base},j} \leftarrow \sum_{i=0}^{j} \boldsymbol{r}_{\mathsf{acc},i} \cdot (\phi_{\mathrm{Gray}}(i) \oplus \phi_{\mathrm{Gray}}(i+1)) \tag{5}$$

for all $0 \le j \le N-1$, assuming $\phi_{\mathrm{Gray}}(N) = 0$. Then, we can set $\boldsymbol{r}_{\mathsf{acc}}$ and $\boldsymbol{r}_{\mathsf{base}}$ respectively as

$\boldsymbol{r}_{\mathsf{acc},N-1}$ and $\boldsymbol{r}_{\mathsf{base},j}$. Indeed, it comes from

$$\begin{aligned}
\boldsymbol{r}_{\mathsf{base}} &= \sum_{j=0}^{N-1} \boldsymbol{r}_{\mathsf{acc},j} \cdot (\phi_{\mathrm{Gray}}(j) \oplus \phi_{\mathrm{Gray}}(j+1)) \\
&= \sum_{j=0}^{N-1} \left(\sum_{i=0}^{j} \boldsymbol{r}_{\mathsf{rnd},i} \right) \cdot (\phi_{\mathrm{Gray}}(j) \oplus \phi_{\mathrm{Gray}}(j+1)) \\
&= \sum_{i=0}^{N-1} \boldsymbol{r}_{\mathsf{rnd},i} \cdot \sum_{j=i}^{N-1} (\phi_{\mathrm{Gray}}(j) \oplus \phi_{\mathrm{Gray}}(j+1)) \\
&= \sum_{i=0}^{N-1} \boldsymbol{r}_{\mathsf{rnd},i} \cdot \phi_{\mathrm{Gray}}(i) \ .
\end{aligned}$$

To sum up, we can compute $\boldsymbol{r}_{\mathsf{acc}}$ using N bit operations using Equation (4), then we can compute $\boldsymbol{r}_{\mathsf{base}}$ using just N additional bit operations using Equation (5) since $\phi_{\mathrm{Gray}}(i) \oplus \phi_{\mathrm{Gray}}(i+1)$ has only one non-zero bit at a public position.

The function ψ. The batch line commitment depends on a public one-to-one function $\psi : \mathbb{F}_2^{\tau \cdot \kappa} \to \mathbb{F}_{2^\lambda}$. In practice, we define ψ as

$$\psi : \boldsymbol{v} \in \mathbb{F}_2^{\tau \cdot \kappa} \mapsto \sum_{i=0}^{\tau \cdot \kappa - 1} v_i \cdot \xi^i,$$

where ξ is defined in Table 2.

Commitment routine. The commitment routine of the batched line commitment (BLC) scheme is described in Algorithm 8. After expanding τ sets of N seeds using the BAVC scheme, it computes the polynomials $\boldsymbol{P}^{(0)}, \ldots, \boldsymbol{P}^{(\tau-1)}$ as

$$\boldsymbol{P}^{(e)} = \boldsymbol{r}_{\mathsf{acc}}^{(e)} \cdot X + \boldsymbol{r}_{\mathsf{base}}^{(e)}$$

for all $0 \leq e < \tau$, where

$$\begin{aligned}
\boldsymbol{r}_{\mathsf{acc}}^{(e)} &\leftarrow \sum_{i=1}^{N} \boldsymbol{r}_{\mathsf{rnd},i}^{(e)} \in \mathbb{F}_2^{|\mathsf{wit}|+(d-1)\lambda+(\lambda+B)} \\
\boldsymbol{r}_{\mathsf{base}}^{(e)} &\leftarrow - \sum_{i=1}^{N} \phi_{\mathrm{Gray}}(i) \cdot \boldsymbol{r}_{\mathsf{rnd},i}^{(e)} \in \mathbb{F}_{2^\kappa}^{|\mathsf{wit}|+(d-1)\lambda+(\lambda+B)}
\end{aligned}$$

with $\boldsymbol{r}_{\mathsf{rnd},i}^{(e)} := \mathsf{PRG.Init}(\mathsf{seeds}[e]\,[i])$. As explained in Section 2.1, the signer can reveal one evaluation of those polynomials among N, while keeping the others hidden. We want to merge those τ vector polynomials into a single polynomial for which the signer will be able to reveal one evaluation among N^τ. To proceed, we use the merging strategy of the VOLEitH framework.

However, since the merging strategy of the VOLEitH approach requires that $\boldsymbol{P}^{(0)}, \ldots, \boldsymbol{P}^{(\tau-1)}$ encodes the same values (*i.e.* the leading terms should be the same), the routine computes an auxiliary value $\Delta \boldsymbol{r}^{(e)} := \boldsymbol{r}_{\mathsf{acc}}^{(0)} - \boldsymbol{r}_{\mathsf{acc}}^{(e)}$ for all $e > 0$ to define the polynomials $\boldsymbol{P'}^{(0)}, \ldots, \boldsymbol{P'}^{(\tau-1)}$ as

$$\boldsymbol{P'}^{(e)} = \begin{cases} \boldsymbol{P}^{(e)} & \text{if } e = 0 \\ \boldsymbol{P}^{(e)} + (\Delta \boldsymbol{r}^{(e)}) \cdot X = \boldsymbol{r}_{\mathsf{acc}}^{(0)} \cdot X + \boldsymbol{r}_{\mathsf{base}}^{(e)} & \text{otherwise.} \end{cases}$$

From now, the signer has committed τ vector polynomials $\{\boldsymbol{P'}^{(e)}\}_{e<\tau}$ with the same leading term, but should convince the verifier that they really have the same leading term. To proceed, it runs the consistency check which consists in computing $\boldsymbol{P}^{(e)}_{\boldsymbol{\alpha'}} \in (\mathbb{F}_{2^\kappa})^{\lambda+B}$ as $[\boldsymbol{M} \mid \boldsymbol{I}_{\lambda+B}] \cdot \boldsymbol{P'}^{(e)}$ for all e. More precisely, it expands the consistency challenge $\boldsymbol{M}$ and computes

$$\boldsymbol{\alpha'}_{\mathsf{plain}} := [\boldsymbol{M} \mid \boldsymbol{I}_{\lambda+B}] \cdot \boldsymbol{r}^{(0)}_{\mathsf{acc}} \ \in \mathbb{F}_2^{\lambda+B}$$

(leading term of all $\boldsymbol{P}^{(e)}_{\boldsymbol{\alpha'}}$'s) and

$$\boldsymbol{\alpha'}^{(e)}_{\mathsf{base}} := [\boldsymbol{M} \mid \boldsymbol{I}_{\lambda+B}] \cdot \boldsymbol{r}^{(e)}_{\mathsf{base}} \ \in \mathbb{F}_{2^\kappa}^{\lambda+B}$$

(constant term of $\boldsymbol{P}^{(e)}_{\boldsymbol{\alpha'}}$) for all e.

After hashing the consistency check output $\{\boldsymbol{P}^{(e)}_{\boldsymbol{\alpha'}}\}_{e<\tau}$, the signer merges the τ vector polynomials $\boldsymbol{P'}^{(0)}, \dots, \boldsymbol{P'}^{(\tau-1)}$. The routine computes $\hat{\boldsymbol{P}} = \psi(\boldsymbol{P'}^{(0)}, \dots, \boldsymbol{P'}^{(\tau-1)})$ as:

$$\hat{\boldsymbol{P}}(X) = \boldsymbol{r}^{(0)}_{\mathsf{acc}} \cdot X + \psi(\boldsymbol{r}^{(0)}_{\mathsf{base}}, \dots, \boldsymbol{r}^{(1)}_{\mathsf{base}}) \ \in (\mathbb{F}_{2^\lambda}[X])^{|\mathsf{wit}|+(d-1)\cdot\lambda+(\lambda+B)} .$$

We then define $\hat{\boldsymbol{P}}_{\mathsf{wit}}$ as the $|\mathsf{wit}|$ first coordinates of $\hat{\boldsymbol{P}}$ and $\hat{\boldsymbol{P}}_{\mathsf{rnd}}$ as the $\lambda\cdot(d-1)$ next coordinates. Then, the routine builds $\boldsymbol{P}_{\mathsf{rnd}}$ as $X \cdot \hat{\boldsymbol{P}}_{\mathsf{rnd}}(1/X)$ (*i.e.* it swaps the leading and the constant terms). For the witness polynomials, we need to enfore the constant term to be the input wit, so the routine computes $\Delta\mathsf{wit}$ as $\mathsf{wit} - \hat{\boldsymbol{P}}(\infty)$ where $\hat{\boldsymbol{P}}(\infty)$ is the leading term of $\hat{\boldsymbol{P}}$ and build $\boldsymbol{P}_{\mathsf{wit}}$ as $X \cdot \hat{\boldsymbol{P}}_{\mathsf{wit}}(1/X) + \Delta\mathsf{wit}$.

The commitment routine outputs the committed degree-1 polynomials $\boldsymbol{P}_{\mathsf{wit}}$ and $\boldsymbol{P}_{\mathsf{rnd}}$, together with their BLC commitment h_{lines}, the BAVC key and the public auxiliary values.

Opening routine. The opening routine of the BLC scheme is described in Algorithm 9. It takes as input a BAVC opening key and a hash digest. Using the routine ExpandEvaluationChallenge, it expands a pseudo-random evaluation point Δ_{inv} defined as

$$\Delta_{\text{inv}} \leftarrow \psi([i^{*(0)}, \dots, i^{*(\tau-1)}]) ,$$

where $i^{*(e)}$'s are pseudo-random values from $\{0, \dots, N-1\}$. At the same time, it expands a random w_{pow}-bit grinding value v_{pow}. If the BAVC opening fails, the grinding value v_{pow} is not zero or if $\Delta_{\text{inv}} = 0$ (and hence fails to be invertible), the routine increments a counter and re-expands an other challenge until the three conditions are satisfied. It outputs the counter and the BAVC opening proof.

Recomputation routine. The recomputation routine ot the BLC scheme is described in Algorithm 10. It first re-expands the evaluation challenge using ExpandEvaluationChallenge and checks $v_{\text{pow}} = 0$ and $\Delta_{\text{inv}} \neq 0$. After the check, it deduces the evaluation point $\Delta := (\Delta_{\text{inv}})^{-1}$. It gets all the opened seeds using BAVC.Reconstruct, *i.e.* all the seeds except those in $\{i^*[e]\}_{e<\tau}$. It computes $\boldsymbol{P}^{(0)}(\phi_{\text{Gray}}(i^{*(0)})), \dots, \boldsymbol{P}^{(\tau-1)}(\phi_{\text{Gray}}(i^{*(\tau-1)}))$ using the relation

$$\forall e,\ \boldsymbol{P}^{(e)}(\phi_{\text{Gray}}(i^{*(e)})) = \sum_{i=1, i\neq i^{*(e)}}^{N} \Big(\phi_{\text{Gray}}(i^{*(e)}) - \phi_{\text{Gray}}(i)\Big) \cdot \boldsymbol{r}^{(e)}_{\mathsf{rnd},i} ,$$

with $\boldsymbol{r}^{(e)}_{\mathsf{rnd},i} :=$ PRG.Init(seeds[e][i]). It then deduces $\boldsymbol{P'}^{(e)}(\phi_{\text{Gray}}(i^{*(e)}))$ for all e, using the relation

$$\forall e,\ \boldsymbol{P'}^{(e)}(\phi_{\text{Gray}}(i^{*(e)})) = \begin{cases} \boldsymbol{P}^{(e)}(\phi_{\text{Gray}}(i^{*(e)})) & \text{if } e = 0 \\ \boldsymbol{P}^{(e)}(\phi_{\text{Gray}}(i^{*(e)})) + (\Delta\boldsymbol{r}^{(e)}) \cdot \phi_{\text{Gray}}(i^{*(e)}) & \text{otherwise.} \end{cases}$$

Algorithm 8 $\mathsf{BLC.Commit}$

Input: a salt $\mathsf{salt} \in \{0,1\}^\lambda$, a root seed $\mathsf{rseed} \in \{0,1\}^\lambda$, and a witness $\mathsf{wit} \in \mathbb{F}_2^{|\mathsf{wit}|}$

▷ Phase 1: Expand seeds.

$(\mathsf{seeds}, h_{\text{com}}, \mathsf{key}) \leftarrow \mathsf{BAVC.Commit}(\mathsf{salt}, \mathsf{rseed})$

▷ Phase 2: Folding.

for e from 0 to $(\tau - 1)$ **do**

$\quad \boldsymbol{r}_{\mathsf{acc}}[e] = 0$ ▷ $\boldsymbol{r}_{\mathsf{acc}}[e] \in \mathbb{F}_2^{[|\mathsf{wit}|+(d-1)\lambda+(\lambda+B)]\times 1}$

$\quad \boldsymbol{r}_{\mathsf{base}}[e] = 0$ ▷ $\boldsymbol{r}_{\mathsf{base}}[e] \in \mathbb{F}_{2^\kappa}^{[|\mathsf{wit}|+(d-1)\lambda+(\lambda+B)]\times 1}$

$\quad$ **for** i from 0 to $(N-1)$ **do**

$\quad\quad \mathsf{prg} \leftarrow \mathsf{PRG.Init}(\mathsf{seeds}[e][i])$

$\quad\quad \boldsymbol{r}_{\mathsf{rnd}} \leftarrow \mathsf{PRG.SampleFieldElements}(\mathsf{prg}, \mathbb{F}_2, |\mathsf{wit}| + (d-1)\lambda + (\lambda+B))$

$\quad\quad \boldsymbol{r}_{\mathsf{acc}}[e] \mathrel{+}= \boldsymbol{r}_{\mathsf{rnd}}$

$\quad\quad \boldsymbol{r}_{\mathsf{base}}[e] \mathrel{+}= \boldsymbol{r}_{\mathsf{rnd}} \cdot \phi_{\text{Gray}}(i)$ ▷ $\phi_{\text{Gray}} : \{0, \ldots, 2^\kappa - 1\} \to \mathbb{F}_2^{1\times\kappa}$

$\quad$ **if** $e > 0$ **then**

$\quad\quad \mathsf{aux}[e] \leftarrow \boldsymbol{r}_{\mathsf{acc}}[0] \oplus \boldsymbol{r}_{\mathsf{acc}}[e]$

$\boldsymbol{u} \leftarrow \boldsymbol{r}_{\mathsf{acc}}[0]$ ▷ $\boldsymbol{u} \in \mathbb{F}_2^{[|\mathsf{wit}|+(d-1)\lambda+(\lambda+B)]\times 1}$

$\boldsymbol{V} \leftarrow [\boldsymbol{r}_{\mathsf{base}}[0], \ldots, \boldsymbol{r}_{\mathsf{base}}[\tau-1]]$ ▷ $\boldsymbol{V} \in \mathbb{F}_2^{[|\mathsf{wit}|+(d-1)\lambda+(\lambda+B)]\times[\tau\cdot\kappa]}$

$h_{\text{aux}} = \mathsf{Hash}_{\text{aux}}(h_{\text{com}}, \mathsf{aux}[1], \ldots, \mathsf{aux}[\tau-1])$

▷ Phase 3: Run consistency check.

$\boldsymbol{M} \leftarrow \mathsf{ExpandConsistencyChallenge}(h_{\text{aux}})$ ▷ $\boldsymbol{M} \in \mathbb{F}_2^{(\lambda+B)\times(|\mathsf{wit}|+(d-1)\lambda)}$

$\boldsymbol{\alpha}'_{\mathsf{plain}} = [\boldsymbol{I}_{\lambda+B} \mid \boldsymbol{M}] \cdot \boldsymbol{u}$ ▷ $\boldsymbol{\alpha}'_{\mathsf{plain}} \in \mathbb{F}_2^{[\lambda+B]\times 1}$

$\boldsymbol{\alpha}'_{\mathsf{base}} = [\boldsymbol{I}_{\lambda+B} \mid \boldsymbol{M}] \cdot \boldsymbol{V}$ ▷ $\boldsymbol{\alpha}'_{\mathsf{base}} \in \mathbb{F}_2^{(\lambda+B)\times(\tau\cdot\kappa)}$

▷ Phase 4: Build lines.

$(\ldots \,\|\, \boldsymbol{r}_{\mathsf{wit}} \,\|\, \boldsymbol{r}_{\mathsf{rnd}}) = \boldsymbol{u}$ where $|\boldsymbol{r}_{\mathsf{wit}}| = |\mathsf{wit}|$ and $|\boldsymbol{r}_{\mathsf{rnd}}| = \lambda \cdot (d-1)$

$(\boldsymbol{r}_{\mathsf{base,wit}} \,\|\, \boldsymbol{r}_{\mathsf{base,rnd}}) = (\psi(\boldsymbol{V}_{(\lambda+B)+1}) \,\|\, \ldots \,\|\, \psi(\boldsymbol{V}_{(\lambda+B)+|\mathsf{wit}|+(d-1)\lambda}))$

▷ $\psi : \mathbb{F}_2^{\tau\cdot\kappa} \to \mathbb{F}_{2^\lambda}$, $\boldsymbol{V}_i$'s are rows of $\boldsymbol{V}$

$\Delta\mathsf{wit} = \mathsf{wit} \oplus \boldsymbol{r}_{\mathsf{wit}}$

$P_{\mathsf{wit}}(X) \leftarrow \mathsf{wit} + \boldsymbol{r}_{\mathsf{base,wit}} \cdot X$

$P_{\mathsf{rnd}}(X) \leftarrow \boldsymbol{r}_{\mathsf{rnd}} + \boldsymbol{r}_{\mathsf{base,rnd}} \cdot X$

$h_{\text{lines}} = \mathsf{Hash}_{\text{lines}}(h_{\text{aux}}, \boldsymbol{\alpha}'_{\mathsf{plain}}, \boldsymbol{\alpha}'_{\mathsf{base}}, \Delta\mathsf{wit})$

$\mathsf{aux}' \leftarrow (\mathsf{aux} \,\|\, \boldsymbol{\alpha}'_{\mathsf{plain}} \,\|\, \Delta\mathsf{wit})$

return $(P_{\mathsf{wit}}, P_{\mathsf{rnd}}, h_{\text{lines}}, \mathsf{key}, \mathsf{aux}')$

Now, the verifier needs to check the consistency test. After expanding the consistency challenge $\boldsymbol{M}$ using $\mathsf{ExpandConsistencyChallenge}$, it computes $\boldsymbol{P}_{\boldsymbol{\alpha}'}^{(e)}(\phi_{\text{Gray}}(i^{*(e)}))$ as $[\boldsymbol{M} \mid \boldsymbol{I}_{\lambda+B}]\boldsymbol{P}'^{(e)}(\phi_{\text{Gray}}(i^{*(e)}))$ and deduces the constant term $\boldsymbol{\alpha}'^{(e)}_{\mathsf{base}}$ of $\boldsymbol{P}_{\boldsymbol{\alpha}'}^{(e)}$ as $\boldsymbol{\alpha}'^{(e)}_{\mathsf{base}} := \boldsymbol{P}_{\boldsymbol{\alpha}'}^{(e)}(\phi_{\text{Gray}}(i^{*(e)})) - \boldsymbol{\alpha}'^{(e)}_{\mathsf{plain}} \cdot \phi_{\text{Gray}}(i^{*(e)})$.

After hashing the consistency check output $\{\boldsymbol{P}_{\boldsymbol{\alpha}'}^{(e)}\}_{e<\tau}$, the verifier computes $\hat{\boldsymbol{P}}(\Delta_{\text{inv}})$ by

Algorithm 9 $\mathsf{BLC.OpenRandomEvaluation}$

Input: a BAVC opening key key and a hash digest $h_{\text{piop}} \in \{0,1\}^{2\lambda}$

$\mathsf{ctr} \leftarrow 0$ ▷ 32-bit counter
`retry:`
$(\{i^*[e]\}_{e<\tau}, v_{\text{pow}}) \leftarrow \mathsf{ExpandEvaluationChallenge}(h_{\text{piop}}, \mathsf{ctr})$
$\Delta_{\text{inv}} \leftarrow \psi([i^*[0], \ldots, i^*[\tau-1]])$ ▷ $\Delta_{\text{inv}} \in \mathbb{F}_{2^\lambda}$
$\pi_{\mathsf{BAVC}} \leftarrow \mathsf{BAVC.Open}(\mathsf{key}, \{i^*[e]\}_{e<\tau})$
if $\pi_{\mathsf{BAVC}} = \perp$ or $v_{\text{pow}} \neq 0$ or $\Delta_{\text{inv}} = 0$ **then**
    $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
    goto `retry`
$\mathsf{pdecom} = (\mathsf{ctr}, \pi_{\mathsf{BAVC}})$
return pdecom

merging the τ evaluations $\boldsymbol{P'}^{(0)}(\phi_{\text{Gray}}(i^{*(0)})), \ldots, \boldsymbol{P'}^{(\tau-1)}(\phi_{\text{Gray}}(i^{*(\tau-1)}))$ as:

$$\hat{\boldsymbol{P}}(\Delta_{\text{inv}}) := \psi\left(\boldsymbol{P'}^{(0)}(\phi_{\text{Gray}}(i^{*(0)})), \ldots, \boldsymbol{P'}^{(\tau-1)}(\phi_{\text{Gray}}(i^{*(\tau-1)}))\right) \in \mathbb{F}_{2^\lambda}^{|\mathsf{wit}|+(d-1)\lambda+(\lambda+B)}.$$

They get $\hat{\boldsymbol{P}}_{\mathsf{wit}}(\Delta_{\text{inv}})$ as the $|\mathsf{wit}|$ first coordinates of $\hat{\boldsymbol{P}}(\Delta_{\text{inv}})$ and $\hat{\boldsymbol{P}}_{\mathsf{rnd}}(\Delta_{\text{inv}})$ as the $\lambda \cdot (d-1)$ next coordinates. Then the routine computes $\boldsymbol{p}_{\mathsf{rnd}} := \boldsymbol{P}_{\mathsf{rnd}}(\Delta)$ as $\Delta \cdot \hat{\boldsymbol{P}}_{\mathsf{rnd}}(\Delta_{\text{inv}})$ and $\boldsymbol{p}_{\mathsf{wit}} := \boldsymbol{P}_{\mathsf{wit}}(\Delta)$ as $\Delta \cdot \hat{\boldsymbol{P}}_{\mathsf{rnd}}(\Delta_{\text{inv}}) + \Delta\mathsf{wit}$.

The reconstruction routine outputs the evaluations $\boldsymbol{p}_{\mathsf{wit}}$ and $\boldsymbol{p}_{\mathsf{rnd}}$, together with the evaluation point Δ and the BLC commitment h_{lines}.

Algorithm 10 BLC.RecomputeEvaluation

Input: a BLC auxiliary value aux', a BLC opening pdecom, a salt $\mathsf{salt} \in \{0,1\}^\lambda$ and a hash digest $h_\text{piop} \in \{0,1\}^{2\lambda}$

▷ Phase 1: Get evaluation point

$(\mathsf{ctr}, \pi_\mathsf{BAVC}) = \mathsf{pdecom}$

$(\mathsf{aux} \parallel \boldsymbol{\alpha}'_\mathsf{plain} \parallel \Delta\mathsf{wit}) \leftarrow \mathsf{aux}'$

$(\{i^*[e]\}_{e<\tau}, v_\text{pow}) \leftarrow \mathsf{ExpandEvaluationChallenge}(h_\text{piop}, \mathsf{ctr})$

$\Delta_\text{inv} \leftarrow \psi([i^*[0], \ldots, i^*[\tau-1]])$ ▷ $\Delta_\text{inv} \in \mathbb{F}_{2^\lambda}$

if $v_\text{pow} \neq 0$ or $\Delta_\text{inv} = 0$ **then**

  return $(\perp, \perp, \perp, \perp)$

$\Delta \leftarrow (\Delta_\text{inv})^{-1}$

▷ Phase 2: Expand seeds.

$h_\text{com}, \mathsf{seeds} \leftarrow \mathsf{BAVC.Reconstruct}(\{i^*[e]\}_{e<\tau}, \pi_\mathsf{BAVC}, \mathsf{salt})$

▷ Phase 3: Folding.

for e from 0 to $(\tau-1)$ **do**

  $\boldsymbol{r}_\mathsf{eval}[e] = 0$ ▷ $\boldsymbol{r}_\mathsf{eval}[e] \in \mathbb{F}_{2^\kappa}^{[|\mathsf{wit}|+(d-1)\lambda+(\lambda+B)]\times 1}$

  for $i \in \{0, \ldots, (N-1)\} \backslash i^*[e]$ **do**

    $\mathsf{prg} \leftarrow \mathsf{PRG.Init}(\mathsf{seed}[e][i])$

    $\boldsymbol{r}_\mathsf{rnd} \leftarrow \mathsf{PRG.SampleFieldElements}(\mathsf{prg}, \mathbb{F}_2, |\mathsf{wit}| + (d-1)\lambda + (\lambda+B))$

    $\boldsymbol{r}_\mathsf{share}[e] \mathrel{+}= \boldsymbol{r}_\mathsf{rnd} \cdot (\phi_\text{Gray}(i^*[e]) - \phi_\text{Gray}(i))$ ▷ $\phi_\text{Gray} : \{0, \ldots, 2^\kappa - 1\} \to \mathbb{F}_2^{1\times[\kappa]}$

  if $e > 0$ **then**

    $\boldsymbol{r}_\mathsf{share}[e] \mathrel{+}= \mathsf{aux}[e] \cdot \phi_{\mu_e}(i^*[e])$

$\boldsymbol{Q} \leftarrow [\boldsymbol{r}_\mathsf{share}[0], \ldots, \boldsymbol{r}_\mathsf{share}[\tau-1]]$ ▷ $\boldsymbol{Q} \in \mathbb{F}_2^{[|\mathsf{wit}|+(d-1)\lambda+(\lambda+B)]\times[\tau\cdot\kappa]}$

$h_\text{aux} = \mathsf{Hash}_\text{aux}(h_\text{com}, \mathsf{aux}[1], \ldots, \mathsf{aux}[\tau-1])$

▷ Phase 4: Run consistency check.

$\boldsymbol{M} \leftarrow \mathsf{ExpandConsistencyChallenge}(h_\text{aux})$ ▷ $\boldsymbol{M} \in \mathbb{F}_2^{(\lambda+B)\times(|\mathsf{wit}|+(d-1)\lambda)}$

$\boldsymbol{\alpha}'_\mathsf{eval} \leftarrow [\boldsymbol{I}_{\lambda+B} \mid \boldsymbol{M}] \cdot \boldsymbol{Q}$ ▷ $\boldsymbol{\alpha}'_\mathsf{eval} \in \mathbb{F}_2^{[\lambda+B]\times[\tau\cdot\kappa]}$

$\boldsymbol{\alpha}'_\mathsf{base} \leftarrow \boldsymbol{\alpha}'_\mathsf{eval} - \boldsymbol{\alpha}'_\mathsf{plain} \cdot \Delta_\text{inv}$ ▷ $\boldsymbol{\alpha}'_\mathsf{base} \in \mathbb{F}_2^{[\lambda+B]\times[\tau\cdot\kappa]}$

▷ Phase 5: Build line evaluation.

$(\boldsymbol{r}_\mathsf{eval,wit} \parallel \boldsymbol{r}_\mathsf{eval,rnd}) = (\psi(\boldsymbol{Q}_{\lambda+B+1}) \parallel \ldots \parallel \psi(\boldsymbol{Q}_{(\lambda+B)+|\mathsf{wit}|+(d-1)\lambda}))$

▷ $\psi : \mathbb{F}_2^{\tau\cdot\kappa} \to \mathbb{F}_{2^\lambda}$, $\boldsymbol{Q}_i$'s are rows of $\boldsymbol{Q}$

$\boldsymbol{p}_\mathsf{wit} \leftarrow \Delta\mathsf{wit} \oplus (\Delta \cdot \boldsymbol{r}_\mathsf{eval,wit})$

$\boldsymbol{p}_\mathsf{rnd} \leftarrow \Delta \cdot \boldsymbol{r}_\mathsf{eval,rnd}$

$h_\text{lines} = \mathsf{Hash}_\text{lines}(h_\text{aux}, \boldsymbol{\alpha}'_\mathsf{plain}, \boldsymbol{\alpha}'_\mathsf{base})$

return $(\Delta, \boldsymbol{p}_\mathsf{wit}, \boldsymbol{p}_\mathsf{rnd}, h_\text{lines})$

3.2.6 PIOP protocol

As explained in Section 2.1 and Section 3.1, the PIOP protocol aims to compute the degree-d polynomial P_α such that

$$P_\alpha(X) = P_0(X) \cdot X + \sum_{j=1}^{m} \gamma_j \cdot f_j(P_1(X), \ldots, P_{|\mathsf{wit}|}(X)).$$

where

- $P_1, \ldots, P_{|\mathsf{wit}|}$ are the witness polynomials, *i.e.* $\boldsymbol{P}_{\mathsf{wit}} := \big(P_1, \ldots, P_{|\mathsf{wit}|}\big)$;
- P_0 is the degree-$(d-1)$ masking polynomial built as

  $$P_0(X) := \sum_{i=0}^{d-2} \left(\sum_{j=0}^{\lambda-1} \xi^j \cdot P_{\mathsf{rnd},i,j}(X) \right) \cdot X^i$$

  with $\boldsymbol{P}_{\mathsf{rnd}} := (P_{\mathsf{rnd},0,0}, \ldots, P_{\mathsf{rnd},0,\lambda-1}, \ldots, P_{\mathsf{rnd},d-2,0}, \ldots, P_{\mathsf{rnd},d-2,\lambda-1})$ and $(1, \xi, \xi^2 \ldots)$ is a $\mathbb{F}_2$-basis of $\mathbb{F}_{2^\lambda}$;
- $\{f_j\}_j$ are the degree-d polynomial contraints that the SD-in-the-Head-2 witness wit should satisfy (see Section 2.2).

We have two types of polynomial constraints:

1. The first constraints consist of checking that the elementary vectors used to build the chunks of $\boldsymbol{x}$ by tensor products have exactly one non-zero coordinate. Since the last coordinates of those vectors is discarded from the witness (and then recovered as 1 minus the sum of the other coordinates), we actually need to check that there is at most one non-zero coordinate.
2. The other constraints consist of checking that the vector $\boldsymbol{x}$ obtained by concatenating the tensor products of the elementary vectors satisfies the linear relation $\boldsymbol{y} = \boldsymbol{H}\boldsymbol{x}$.

Parsing the input polynomials. The PIOP protocol takes the input vector polynomials $\boldsymbol{P}_{\mathsf{wit}}$ and $\boldsymbol{P}_{\mathsf{rnd}}$. The vector polynomial $\boldsymbol{P}_{\mathsf{wit}} := \big(P_1, \ldots, P_{|\mathsf{wit}|}\big)$ encodes the RSD witness wit, a bit-string made of the decomposition of the RSD solution $\boldsymbol{x}$ as elementary vectors (input of tensor products). The RSD witness wit can be parsed as bits $\{w_{i,j,k}\}_{i,j,k}$, where $w_{i,j,k}$ is the k^{th} coordinates of the j^{th} elementary vector in the tensor product of the i^{th} chunk of the RSD solution $\boldsymbol{x}$. By parsing $\boldsymbol{P}_{\mathsf{wit}}$ in the same way, we can get the degree-1 polynomials $\{P_{w_{i,j,k}}\}_{i,j,k}$ that encode the bits $\{w_{i,j,k}\}_{i,j,k}$.

The vector $\boldsymbol{P}_{\mathsf{rnd}} := (P_{\mathsf{rnd},1,1}, \ldots, P_{\mathsf{rnd},1,\lambda}, \ldots, P_{\mathsf{rnd},d-1,1}, \ldots, P_{\mathsf{rnd},d-1,\lambda})$ contains $\lambda \cdot (d-1)$ degree-1 polynomials with leading term from $\mathbb{F}_{2^\lambda}$ and constant term from $\mathbb{F}_2$. These polynomials are used to build a degree-$(d-1)$ random polynomial $P_0 \in \mathbb{F}_{2^\lambda}[X]$ as:

$$P_0(X) := \sum_{i=0}^{d-2} \left(\sum_{j=0}^{\lambda-1} \xi^j \cdot P_{\mathsf{rnd},i,j}(X) \right) \cdot X^i.$$

We describe in Algorithm 11 the routine that takes as inputs the vector polynomials $\boldsymbol{P}_{\mathsf{wit}}$ and $\boldsymbol{P}_{\mathsf{rnd}}$ and outputs the set of degree-1 polynomials $\{P_{w_{i,j,k}}\}_{i,j,k}$ that encode the bits $\{w_{i,j,k}\}_{i,j,k}$

of the decomposition of the RSD solution and the degree-$(d-1)$ polynomial P_0. We then describe in Algorithm 12 the routine that takes as inputs the vectors $p_{\mathsf{wit}} := \boldsymbol{P}_{\mathsf{wit}}(\Delta) \in \mathbb{F}_{2^\lambda}^{|\mathsf{wit}|}$ and $p_{\mathsf{rnd}} := \boldsymbol{P}_{\mathsf{rnd}}(\Delta) \in \mathbb{F}_{2^\lambda}^{|\mathsf{rnd}|}$ and outputs the set $\{P_{w_{i,j,k}}(\Delta)\}_{i,j,k}$ and the value $P_0(\Delta)$, for some field element $\Delta \in \mathbb{F}_{2^\lambda}$.

Algorithm 11 PIOP.Prover.Format

Input: two degree-1 vector polynomials $\boldsymbol{P}_{\mathsf{wit}}$ and $\boldsymbol{P}_{\mathsf{rnd}}$.

```
ind ← 0
for i from 0 to w − 1 do
    for j from 0 to d − 1 do
        for k from 0 to μ_j − 2 do
            P_{w_{i,j,k}} ← (P_wit)_ind                         ▷ Degree-1 polynomial
            ind ← ind + 1
P_0(X) ← Σ_{i=0}^{d−2} ( Σ_{j=0}^{λ−1} ξ^j · (P_rnd(X))_{iλ+j} ) · X^i   ▷ Degree-(d − 1) polynomial
return {P_{w_{i,j,k}}}_{i,j,k}, P_0
```

Algorithm 12 PIOP.Verifier.Format

Input: a field element $\Delta \in \mathbb{F}_{2^\lambda}$, two vectors p_{wit} and p_{rnd} ▷ $p_{\mathsf{wit}} := \boldsymbol{P}_{\mathsf{wit}}(\Delta)$, $p_{\mathsf{rnd}} := \boldsymbol{P}_{\mathsf{rnd}}(\Delta)$

```
ind ← 0
for i from 0 to w − 1 do
    for j from 0 to d − 1 do
        for k from 0 to μ_j − 2 do
            p_{w_{i,j,k}} ← (p_wit)_ind
            ind ← ind + 1
p_0 ← Σ_{i=0}^{d−2} ( Σ_{j=0}^{λ−1} ξ^j · (p_rnd)_{iλ+j} ) · Δ^i
return {p_{w_{i,j,k}}}_{i,j,k}, p_0
```

Checking elementary vectors. To check that a binary vector $(w_{i,j,0}, \dots, w_{i,j,\mu_j-2}) \in \mathbb{F}_2^{\mu_j-1}$ has at most one non-zero coordinate, we check that for all $k \neq k'$, we have $w_{i,j,k} \cdot w_{i,j,k'} = 0$. However, to avoid performing $O(\mu_j^2)$ multiplications, we instead check that the polynomial $D_{i,j} \in \mathbb{F}_2[Y]$ defined as:

$$D_{i,j}(Y) := \left(\sum_{k=0}^{\mu_j-2} Y^k \cdot w_{i,j,k}\right) \cdot \left(\sum_{k=0}^{\mu_j-3} Y^{(\mu_j-1)k} \cdot w_{i,j,k}\right) - \left(\sum_{k=0}^{\mu_j-3} Y^{\mu_j \cdot k} \cdot w_{i,j,k}\right)$$

equals $0 \in \mathbb{F}_2[Y]$ for all $1 \leq i \leq w$ and $1 \leq j \leq d$. For our considered parameters, the degree of these polynomials always satisfies

$$\deg(D_{i,j}) = (\mu_j - 3) \cdot \mu_j + 1 < 32 \ .$$

Moreover, we always have $d \leq 4$. Those two upper bounds imply:

$$\forall\, 1 \leq j \leq d,\ D_{i,j}(Y) = 0 \in \mathbb{F}_2[Y] \quad \Leftrightarrow \quad \sum_{j=1}^{d} \xi^{32(j-1)} \cdot D_{i,j}(\xi) = 0 \in \mathbb{F}_{2^\lambda} \ .$$

The check of the elementary vectors hence reduces to checking the right-hand side of the above equivalence.

Looking ahead, these relations will be batched using random coefficients for the definition of the polynomial P_α. Let $\boldsymbol{v}' = (v'_1, \dots, v'_w) \in \mathbb{F}_{2^\lambda}^w$ the vector with coordinates defined as

$$v'_i = \sum_{j=1}^{d} \xi^{32(j-1)} \cdot D_{i,j}(\xi) \ , \tag{6}$$

and let $\boldsymbol{\gamma}' \in \mathbb{F}_{2^\lambda}^w$ be the challenge from the verifier. Checking the elementary vectors shall consist in checking $\boldsymbol{\gamma}'^\top \cdot \boldsymbol{v}' = 0$ which implies $\boldsymbol{v}' = 0$ with overwhelming probability over the randomness of $\boldsymbol{\gamma}'$.

The routine PIOP.Prover.UnitaryCheck performs the computation of $D_{i,j}(\xi)$ from a prover standpoint, where each witness bit $w_{i,j,k}$ is encoded through a degree-1 polynomial $P_{w_{i,j,k}}(X)$. The routine PIOP.Verifier.UnitaryCheck performs the computation of $D_{i,j}(\xi)$ from a verifier standpoint, where each witness bit $w_{i,j,k}$ is encoded through a polynomial evaluation $p_{w_{i,j,k}} := P_{w_{i,j,k}}(\Delta)$.

Algorithm 13 PIOP.Prover.UnitaryCheck

Input: arity μ_j, degree-1 polynomials $P_{w_{i,j,0}}(X), \dots, P_{w_{i,j,\mu_j-2}}(X) \in \mathbb{F}_{2^\lambda}[X]$ for which the constant terms are $w_{i,j,0}, \dots, w_{i,j,\mu_j-2} \in \mathbb{F}_2$.

1: $\mathsf{sum}_1 \leftarrow \sum_{k=0}^{\mu_j-2} \xi^k \cdot P_{w_{i,j,k}}(X)$
2: $\mathsf{sum}_2 \leftarrow \sum_{k=0}^{\mu_j-3} \xi^{(\mu_j-1)k} \cdot P_{w_{i,j,k}}(X)$
3: $\mathsf{sum}_3 \leftarrow \sum_{k=0}^{\mu_j-3} \xi^{\mu_j \cdot k} \cdot P_{w_{i,j,k}}(X)$
4: **return** $\mathsf{sum}_1 \cdot \mathsf{sum}_2 - \mathsf{sum}_3$ ▷ Degree-1 polynomial (since the witness is well-built)

Algorithm 14 PIOP.Verifier.UnitaryCheck

Input: arity μ_j, evaluations $p_{w_{i,j,0}}, \dots, p_{w_{i,j,\mu_j-2}} \in \mathbb{F}_{2^\lambda}$ ▷ $\forall k,\ p_{w_{i,j,k}} := P_{w_{i,j,k}}(\Delta)$

1: $\mathsf{sum}_1 \leftarrow \sum_{k=0}^{\mu_j-2} \xi^k \cdot p_{w_{i,j,k}}$
2: $\mathsf{sum}_2 \leftarrow \sum_{k=0}^{\mu_j-3} \xi^{(\mu_j-1)k} \cdot p_{w_{i,j,k}}$
3: $\mathsf{sum}_3 \leftarrow \sum_{k=0}^{\mu_j-3} \xi^{\mu_j \cdot k} \cdot p_{w_{i,j,k}}$
4: **return** $\mathsf{sum}_1 \cdot \mathsf{sum}_2 - \mathsf{sum}_3$

Checking RSD linear constraints. We need to check the relation $\boldsymbol{H}\boldsymbol{x} = \boldsymbol{y}$ where $\boldsymbol{x} = (e_0 \parallel \dots \parallel e_{w-1})$ is built as the concatenation of the elementary vectors e_i resulting from the tensor products. Namely, we need to check

$$\boldsymbol{v} := \sum_{i=0}^{w-1} \sum_{j=0}^{m-1} (e_i)_j \cdot \boldsymbol{h}_{i \cdot m + j} - \boldsymbol{y} = (0, \dots, 0) \in \mathbb{F}_2^{n-k} \ ,$$

where $[\boldsymbol{h}_0 \mid \boldsymbol{h}_1 \mid \dots \mid \boldsymbol{h}_{n-1}] = \boldsymbol{H}$.

Looking ahead, the coordinates of this vector (encoded as polynomials) will be batched using random coefficients in the computation of P_α. To make this batching more efficient, we embed blocks from $\mathbb{F}_2^\lambda$ into elements of $\mathbb{F}_{2^\lambda}$. Let ϕ be the linear $\mathbb{F}_2$-linear field-embedding isomorphism:

$$\phi : (v_1, \dots, v_\lambda) \in \mathbb{F}_2^\lambda \mapsto \sum_{i=1}^{\lambda} v_i \cdot \xi^{i-1} \ ,$$

and let Φ be its block-wise variant:

$$\Phi : \boldsymbol{v} \in \mathbb{F}_2^{n-k} \mapsto \big(\phi(v_1, \dots, v_\lambda), \dots, \phi(\dots, v_{n-k}, 0, \dots, 0)\big) \ \in \mathbb{F}_{2^\lambda}^{\lceil \frac{n-k}{\lambda} \rceil} \ , \tag{7}$$

where the last block is padded with $\lambda \cdot \lceil \frac{n-k}{\lambda} \rceil - (n-k)$ zeros. We have

$$\boldsymbol{v} = 0 \in \mathbb{F}_2^{n-k} \quad \Leftrightarrow \quad \Phi(\boldsymbol{v}) = 0 \in \mathbb{F}_{2^\lambda}^{\lceil \frac{n-k}{\lambda} \rceil} \ .$$

Now let $\boldsymbol{\gamma} \in \mathbb{F}_{2^\lambda}^{\lceil \frac{n-k}{\lambda} \rceil}$ be the challenge from the verifier. Checking the RSD linear constraints shall consist in checking $\boldsymbol{\gamma}^\top \cdot \Phi(\boldsymbol{v}) = 0$ which implies $\Phi(\boldsymbol{v}) = 0$ (and hence $\boldsymbol{v} = 0$) with overwhelming probability over the randomness of $\boldsymbol{\gamma}$. Then we observe that $\boldsymbol{\gamma}^\top \cdot \Phi(\boldsymbol{v})$ can be written as:

$$\boldsymbol{\gamma}^\top \cdot \Phi(\boldsymbol{v}) = \sum_{i=0}^{w-1} \sum_{j=0}^{m-1} (e_i)_j \cdot h_{i \cdot m + j}^{[\gamma]} - y^{[\gamma]} \ , \tag{8}$$

where

$$h_j^{[\gamma]} := \boldsymbol{\gamma}^\top \cdot \Phi(\boldsymbol{h}_j) \ \ \forall\, 0 \le j \le n-1 \quad \text{and} \quad y_j^{[\gamma]} := \boldsymbol{\gamma}^\top \cdot \Phi(\boldsymbol{y}) \ . \tag{9}$$

Mux tree for efficient evaluation. To compute $\boldsymbol{\gamma}^\top \cdot \Phi(\boldsymbol{v})$, either as polynomial encoding on the prover side (i.e., in the computation of P_α) or as evaluation encoding on the verifier side (i.e., in the computation of $P_\alpha(\Delta)$), a natural option would be to expand each elementary vector e_i from the witness bits $\{w_{i,j,k}\}_{j,k}$ and evaluate Equation 8. However, this approach would involve many field multiplications. We instead rely on a multiplexer tree (or "mux tree" for short).

For a vector $\boldsymbol{b} = (b_0, \dots, b_{\mu_j - 2}) \in \{0,1\}^{\mu_j - 1}$ such that $w_H(\boldsymbol{b}) \le 1$, and a vector $\boldsymbol{u} = (u_0, \dots, u_{\mu_j}) \in \mathbb{F}_{2^\lambda}^{\mu_j}$, we define the μ_j-to-1 multiplexer gate as follows:

$$\mathsf{Mux}_{\mu_j} : (\boldsymbol{b}, \boldsymbol{u}) \longmapsto \begin{cases} u_0 & \text{if } \boldsymbol{b} = 0 \\ u_1 & \text{if } b_0 = 1 \\ & \quad \vdots \\ u_{\mu_j - 1} & \text{if } b_{\mu_j - 2} = 1 \end{cases}$$

This gate can be efficiently implemented using $\mu_j - 1$ multiplications:

$$\mathsf{Mux}_{\mu_j}(\boldsymbol{b}, \boldsymbol{u}) = u_0 + \sum_{j=1}^{\mu_j - 1} b_{j-1} \cdot (u_j - u_0).$$

As explained in Section 2.2, the elementary vectors e_i's composing the RSD solution $\boldsymbol{x} = (e_0 \parallel \dots \parallel e_{m-1})$ are defined as:

$$e_i \leftarrow \begin{pmatrix} b_{i,1,1} \\ b_{i,1,2} \\ \vdots \\ b_{i,1,\mu_1 - 1} \\ 1 - \sum_k b_{i,1,k} \end{pmatrix} \otimes \begin{pmatrix} b_{i,2,1} \\ b_{i,2,2} \\ \vdots \\ b_{i,2,\mu_2 - 1} \\ 1 - \sum_k b_{i,2,k} \end{pmatrix} \otimes \dots \otimes \begin{pmatrix} b_{i,d,1} \\ b_{i,d,2} \\ \vdots \\ b_{i,d,\mu_d - 1} \\ 1 - \sum_k b_{i,d,k} \end{pmatrix} \in \mathbb{F}^{n/w}.$$

For all i, we can observe that computing $\sum_{j=0}^{m}(e_i)_j \cdot h_{i\cdot m+j}^{[\gamma]}$ is equivalent to computing the root $\mathsf{node}_{d,0}$ of a mux tree with arities $(\mu_1,\dots,\mu_d)$. Formally, for all $0 \le j \le d$ and $0 \le \ell < \mu_{j+1}\cdot\ldots\cdot\mu_d$, let

$$\mathsf{node}_{j,\ell} = \begin{cases} h_{i\cdot m+\ell}^{[\gamma]} & \text{if } j=0 \text{ (the tree leaves)} \\ \mathsf{Mux}_{\mu_j}\Big(\boldsymbol{b}_{i,j}, (\mathsf{node}_{j-1,\mu_j\cdot h},\dots,\mathsf{node}_{j-1,\mu_j\cdot(\ell+1)-1})\Big) & \text{if } j>0 \text{ (the tree nodes)} \end{cases}$$

where $\boldsymbol{b}_{i,j} := (b_{i,j,1},\dots,b_{i,j,\mu_j-1})$. We denote $\mathsf{MuxTree}((h_{i\cdot m}^{[\gamma]},\dots,h_{(i+1)\cdot m-1}^{[\gamma]}),\{\boldsymbol{b}_{i,j}\}_j)$ the root of the mux tree. Equation 8 now rewrites as:

$$\boldsymbol{\gamma}^\top \cdot \Phi(\boldsymbol{v}) = \sum_{i=0}^{w-1} \mathsf{MuxTree}((h_{i\cdot m}^{[\gamma]},\dots,h_{(i+1)\cdot m-1}^{[\gamma]}),\{\boldsymbol{b}_{i,j}\}_j)\ .$$

We describe in Algorithm 15 the computation of the mux-tree root from the prover standpoint, where each witness bit $w_{i,j,k}$ is encoded through a degree-1 polynomial $P_{w_{i,j,k}}(X)$. We then describe in Algorithm 16 the computation of the mux-tree root from the verifier standpoint, where each witness bit $w_{i,j,k}$ is encoded through a polynomial evaluation.

Algorithm 15 PIOP.Prover.MuxTree

Input: coefficients $(h_{i\cdot m}^{[\gamma]},\dots,h_{(i+1)\cdot m-1}^{[\gamma]}) \in \mathbb{F}_{2^\lambda}^m$,
degree-1 polynomials $\{(P_{w_{i,j,0}}(X),\dots,P_{w_{i,j,\mu_j-2}}(X))\}_j$ of $\mathbb{F}_{2^\lambda}[X]$

1: $(\mathsf{node}_{0,0},\dots,\mathsf{node}_{0,m-1}) \leftarrow (h_{i\cdot m}^{[\gamma]},\dots,h_{(i+1)\cdot m-1}^{[\gamma]})$ ▷ degree-0 polynomials
2: **for** j from 1 to d **do**
3: **for** h from 0 to $\mu_{j+1}\cdot\ldots\cdot\mu_d - 1$ **do**
4: $\mathsf{node}_{j,\ell} \leftarrow \mathsf{node}_{j-1,\mu_j\cdot h} + \sum_{k=0}^{\mu_j-2} P_{w_{i,j,k}}(X)\cdot\Big(\mathsf{node}_{j-1,\mu_j\cdot h+(1+k)} - \mathsf{node}_{j-1,\mu_j\cdot h}\Big)$ ▷ degree-j polynomials
5: **return** $\mathsf{node}_{d,0}$ ▷ degree-d polynomial

Algorithm 16 PIOP.Verifier.MuxTree

Input: coefficients $(h_{i\cdot m}^{[\gamma]},\dots,h_{(i+1)\cdot m-1}^{[\gamma]}) \in \mathbb{F}_{2^\lambda}^m$,
evaluations $\{(p_{w_{i,j,0}},\dots,p_{w_{i,j,\mu_j-2}})\}_j \in \mathbb{F}_{2^\lambda}^{\mu_1-1}\times\ldots\times\mathbb{F}_{2^\lambda}^{\mu_d-1}$ ▷ $p_{w_{i,j,k}} := P_{w_{i,j,k}}(\Delta)$

1: $(\mathsf{node}_{0,0},\dots,\mathsf{node}_{0,m-1}) \leftarrow (h_{i\cdot m}^{[\gamma]},\dots,h_{(i+1)\cdot m-1}^{[\gamma]})$
2: **for** j from 1 to d **do**
3: **for** h from 0 to $\mu_{j+1}\cdot\ldots\cdot\mu_d - 1$ **do**
4: $\mathsf{node}_{j,h} \leftarrow \mathsf{node}_{j-1,\mu_j\cdot h} + \sum_{k=0}^{\mu_j-2} p_{w_{i,j,k}}\cdot\Big(\mathsf{node}_{j-1,\mu_j\cdot h+(1+k)} - \mathsf{node}_{j-1,\mu_j\cdot h}\Big)$
5: **return** $\mathsf{node}_{d,0}$

PIOP Prototocol. We describe in Algorithm 17 the prover's computation in the PIOP protocol, namely how the prover compute the polynomial P_α from the masking polynomial P_0 and the witness $\boldsymbol{P}_{\mathsf{wit}} := \big(P_1,\dots,P_{|\mathsf{wit}|}\big)$. The first step consists in formatting the input polynomials using the routine PIOP.Prover.Format: it parses the vector polynomial $\boldsymbol{P}_{\mathsf{wit}}$ as $\{P_{w_{i,j,k}}\}_{i,j,k}$ and builds the degree-$(d-1)$ masking polynomial P_0 from the degree-1 vector polynomial P_{rnd}. Next, the verifier challenge $(\boldsymbol{\gamma}', \boldsymbol{\gamma})$, i.e., the randomness for batching the relations, is generated by hashing the BLC commitment h_{lines}. Then the batched field-embedded elements $\{h_j^{[\gamma]}\}$ and $y^{[\gamma]}$ are computed (see Equation 9). Finally, the polynomial P_α is computed as:

$$
\begin{aligned}
P_\alpha(X) &= P_0(X) \cdot X + \sum_j \gamma_j \cdot f_j(P_1(X), \ldots, P_{|\mathsf{wit}|}(X)). \\
&= P_0(X) \cdot X \\
&\quad \underbrace{+ \sum_{i=0}^{w-1} \mathsf{PIOP.Prover.MuxTree}\left((h_{i\cdot m}^{[\gamma]}, \ldots, h_{(i+1)\cdot m-1}^{[\gamma]}), \{P_{w_{i,j,k}}(X)\}_{j,k}\right)}_{\text{polynomial encoding } \boldsymbol{\gamma}^\top \cdot \Phi(\boldsymbol{v})} \\
&\quad \underbrace{+ \sum_{i=0}^{w-1} \gamma'_i \cdot \sum_{j=0}^{d-1} \xi^{32(j-1)} \cdot \mathsf{PIOP.Prover.UnitaryCheck}(\mu_j, \{P_{w_{i,j,k}}(X)\}_k)}_{\text{polynomial encoding } \boldsymbol{\gamma}'^\top \cdot \boldsymbol{v}'}
\end{aligned}
$$

Algorithm 17 PIOP.Prover.Run

Input: two vector polynomials $P_{\mathsf{wit}} \in (\mathbb{F}_{2^\lambda}[X])^{|\mathsf{wit}|}$ and $P_{\mathsf{rnd}} \in (\mathbb{F}_{2^\lambda}[X])^{|\mathsf{rnd}|}$, a hash digest $h_{\text{lines}} \in \{0,1\}^{2\lambda}$ and the regular syndrome instance $(\boldsymbol{H}', \boldsymbol{y})$.
Output: a degree-d polynomial P_α of $\mathbb{F}_{2^\lambda}[X]$

1: $\left(\{P_{w_{i,j,k}}\}_{i,j,k}, P_0\right) \leftarrow \mathsf{PIOP.Prover.Format}(P_{\mathsf{wit}}, P_{\mathsf{rnd}})$
2: $(\boldsymbol{\gamma}', \boldsymbol{\gamma}) \leftarrow \mathsf{ExpandBatchingChallenge}(h_{\text{lines}})$ ▷ $\boldsymbol{\gamma} \in \mathbb{F}_{2^\lambda}^{\lceil \frac{n-k}{\lambda} \rceil}, \boldsymbol{\gamma}' \in \mathbb{F}_{2^\lambda}^w$
3: $\left(\{h_j^{[\gamma]}\}_j, y^{[\gamma]}\right) \leftarrow \mathsf{BatchLinearEquations}(\boldsymbol{\gamma}, \boldsymbol{H}', \boldsymbol{y})$

▷ Compute the degree-α output polynomial $P_\alpha(X)$.
4: $P_\alpha(X) \leftarrow P_0(X) \cdot X - y^{[\gamma]}$
5: **for** i from 0 to $w-1$ **do**
6: $\quad P_\alpha(X) \leftarrow P_\alpha(X) + \mathsf{PIOP.Prover.MuxTree}\left((h_{i\cdot m}^{[\gamma]}, \ldots, h_{(i+1)\cdot m-1}^{[\gamma]}), \{P_{w_{i,j,k}}(X)\}_{j,k}\right)$
7: $\quad P_\alpha(X) \leftarrow P_\alpha(X) + \gamma'_i \cdot \sum_{j=1}^{d} \xi^{32(j-1)} \cdot \mathsf{PIOP.Prover.UnitaryCheck}(\mu_j, \{P_{w_{i,j,k}}(X)\}_k)$
8: **return** $P_\alpha(X)$

Algorithm 18 describes the verifier's computation in the PIOP protocol, namely how the verifier computes the evaluation $p_\alpha = P_\alpha(\Delta)$ from the evaluations $P_0(\Delta)$ and $\boldsymbol{P}_{\mathsf{wit}}(\Delta)$. The first step consists in formatting the input evaluations using the routine PIOP.Verifier.Format: it parses the evaluation vector $\boldsymbol{p}_{\mathsf{wit}} = \boldsymbol{P}_{\mathsf{wit}}(\Delta)$ as $\{p_{w_{i,j,k}}\}_{i,j,k} = \{P_{w_{i,j,k}}(\Delta)\}_{i,j,k}$ and builds the evaluation $p_0 = P_0(\Delta)$ from the evaluation vector $p_{\mathsf{rnd}} = P_{\mathsf{rnd}}(\Delta)$. Next, the verifier challenge $(\boldsymbol{\gamma}', \boldsymbol{\gamma})$, i.e., the randomness for batching the relations, is generated by hashing the BLC commitment h_{lines}. Then the batched field-embedded elements $\{h_j^{[\gamma]}\}$ and $y^{[\gamma]}$ are computed (see Equation 9). Finally, the evaluation $p_\alpha = P_\alpha(\Delta)$ is computed in the same way as the polynomial P_α on the prover side (see above equation).

Algorithm 18 PIOP.Verifier.Run

Input: a field element $\Delta \in \mathbb{F}_{2^\lambda}$, two vectors $p_{\mathsf{wit}} \in \mathbb{F}_{2^\lambda}^{|\mathsf{wit}|}$ and $p_{\mathsf{rnd}} \in \mathbb{F}_{2^\lambda}^{|\mathsf{rnd}|}$, a hash digest $h_{\text{lines}} \in \{0,1\}^{2\lambda}$ and the regular syndrome instance $(\boldsymbol{H'}, \boldsymbol{y})$.

Output: a field element $p_\alpha \in \mathbb{F}_{2^\lambda}$

$\left(\{p_{w_{i,j,k}}\}_{i,j,k}, p_0\right) \leftarrow \mathsf{PIOP.Verifier.Format}(p_{\mathsf{wit}}, p_{\mathsf{rnd}})$

$(\boldsymbol{\gamma'}, \boldsymbol{\gamma}) \leftarrow \mathsf{ExpandBatchingChallenge}(h_{\text{lines}})$ ▷ $\boldsymbol{\gamma} \in \mathbb{F}_{2^\lambda}^{\lceil \frac{n-k}{\lambda} \rceil}, \boldsymbol{\gamma'} \in \mathbb{F}_{2^\lambda}^{w}$

$\left(\{h_j^{[\gamma]}\}_j, y^{[\gamma]}\right) \leftarrow \mathsf{BatchLinearEquations}(\boldsymbol{\gamma}, \boldsymbol{H'}, \boldsymbol{y})$

▷ Compute the evaluation $p_\alpha := P_\alpha(\Delta)$.

$p_\alpha \leftarrow p_0 \cdot \Delta - y^{[\gamma]}$

for i from 0 to $w-1$ **do**

    $p_\alpha \leftarrow p_\alpha + \mathsf{PIOP.Verifier.MuxTree}\left((h_{i \cdot m}^{[\gamma]}, \ldots, h_{(i+1)\cdot m-1}^{[\gamma]}), \{p_{w_{i,j,k}}\}_{j,k}\right)$

    $p_\alpha \leftarrow p_\alpha + \gamma'_i \cdot \sum_{j=1}^{d} \xi^{32(j-1)} \cdot \mathsf{PIOP.Verifier.UnitaryCheck}(\mu_j, \{p_{w_{i,j,k}}\}_k)$

return p_α

Algorithm 19 BatchLinearEquations

Input: $\boldsymbol{\gamma}$, $\boldsymbol{H'}$, $\boldsymbol{y}$

for j from 0 to $n-1$ **do**

    $h_j^{[\gamma]} \leftarrow \boldsymbol{\gamma}^\top \cdot \Phi(\boldsymbol{h}_j)$ ▷ $h_j^{[\gamma]} \in \mathbb{F}_{2^\lambda}$, $\boldsymbol{h}_j$ is the j^{th} column of $\boldsymbol{H} = [\boldsymbol{H'} \mid I_{n-k}]$

$y^{[\gamma]} \leftarrow \boldsymbol{\gamma}^\top \cdot \Phi(\boldsymbol{y})$ ▷ $y^{[\gamma]} \in \mathbb{F}_{2^\lambda}$

return $\{h_j^{[\gamma]}\}, y^{[\gamma]}$

4 Parameters and performances

In this section, we propose several parameter sets for the SD-in-the-Head signature scheme. As explained hereafter, those parameters have been selected to meet the security categories I, III and V defined by the NIST while targeting good performances (signature size and running times).

4.1 Selection of parameters

RSD parameters. The RSD parameters include three values: n the length of the code, k_{RSD} the dimension of the code and w the weight of the secret vector. We recall that in an RSD instance, the secret vector is made of w blocks of size $\frac{n}{w}$ and weight 1.

For $\theta \in [1, \infty)$, we define $k_{\text{RSD}}^{\max}$, the largest RSD dimension leading to RSD density $\leq 1/\theta$:

$$k_{\text{RSD}}^{\max} := n - \left\lfloor w \cdot \log_2 \left(\frac{n}{w}\right) - \log_2 \theta \right\rfloor .$$

We further define k_{SD} as the largest SD dimension leading to SD density ≤ 1 (for non-regular SD with parameters n and w):

$$k_{\text{SD}} := n - \left\lfloor \log_2 \binom{n}{w} \right\rfloor .$$

According to the security reduction provided in Section 5.1, for any k_{RSD} such that

$$k_{\text{SD}} \leq k_{\text{RSD}} \leq k_{\text{RSD}}^{\max}$$

an algorithm solving an (n, k_{RSD}, w)-RSD instance in complexity λ' bits implies an algorithm solving an (n, k_{SD}, w)-SD instance with complexity

$$\lambda' + \log_2 \left(\frac{\binom{n}{w}}{(\frac{n}{w})^w} \right) + \frac{1}{\theta \ln 2} . \tag{10}$$

Parameters are chosen such that for $\lambda' \in \{143, 207, 272\}$ (corresponding to NIST Categories I, III and V), the best binary SD attacks for an (n, k_{SD}, w)-instance have a complexity greater than (10). Thanks to our reduction (Theorem 5.1), finding an attack with complexity less than λ' for an (n, k_{RSD}, w)-RSD instance would mean improving over the best known attacks for (n, k_{SD}, w)-SD instances, the currently hardest type of SD instances (as being on the Gilbert-Varshamov bound). In practice for implementation reasons k_{RSD} is chosen as the greatest value such that $n - k_{\text{RSD}}$ is a multiple of 8 and $k_{\text{SD}} \leq k_{\text{RSD}} \leq k_{\text{RSD}}^{\max}$.

Concretely, for our parameter selection, we proceeded as follows. For each w, we find the smallest n that is a multiple of w so that the (n, k_{SD}, w)-SD instance achieve security (10). We then set k_{RSD} as the largest number $\leq k_{\text{RSD}}^{\max}$ such that the codimension $n - k_{\text{RSD}}$ is an exact multiple of 8. Then for each degree d (we test them all), we consider the mux arities $\mu = (\mu_1, \ldots, \mu_d)$ that minimizes the witness size w (the greedy algorithm that selects $\mu_1 = \lceil n/w \rceil^{1/d}$ and continues recursively for $(\mu_2, \ldots, \mu_d)$ on $\lceil n/w/\mu_1 \rceil$ yields the optimum). We finally set the degree d as the one that minimizes $w + \lambda.(d-1)$, and thus, the total signature size. The results are provided in Table 3 hereafter. The best parameters are obtained for arities $\mu = (4, 4, 4, 4)$ or $\mu = (4, 4, 4, 3)$, inducing block sizes of either 256 or 192, with rather large values of n.

Proof system parameters. For each security level, we consider two variants: a "short" variant with larger GGM trees (decreasing the number of repetitions τ and hence the signature size), and a "fast" variant with smaller GGM trees (making the computation faster). For the "short" variant, we use $N = 2^{11}$ for Category I and $N = 2^{12}$ for Categories III and V. For the "fast" variant, we use $N = 2^8$ for all three categories. To achieve a soundness of λ bits, with $\lambda \in \{128, 192, 256\}$ (for Categories I, III and V), we must select the number of repetitions τ, and the grinding parameter w_{pow}, to satisfy:

$$\tau \cdot \kappa - \log_2(d) + w_{\text{pow}} \geq \lambda \ ,$$

where $\kappa = \log_2(N)$. Concretely, we choose τ such that $w_{\text{pow}} := \lambda + \log_2(d) - \tau \cdot \kappa$ is a small integer, typically lower than 5 for the "fast" variants, and up to 10 for the "short" variants. This strategy yields $w_{\text{pow}} = 2$ for the "fast" variants at each security level while we get $w_{\text{pow}} \in \{2, 6, 9\}$ for the "short" variant. Finally, we fix the value of T_{open} based on experiments to obtain good trade-offs between sizes and performances.

The proof system relies on the field on a λ-bit field, namely $\mathbb{F}_{2^{128}}$, $\mathbb{F}_{2^{192}}$ and $\mathbb{F}_{2^{256}}$ depending on the security level. Table 2 summarizes the field extensions that we use in our instances.

Table 2: Definition of field extensions.

Field ($\mathbb{F}_{2^\lambda}$)	Field extension
$\mathbb{F}_{2^{128}}$	$\mathbb{F}_2[\xi]/\langle \xi^{128} + \xi^7 + \xi^2 + \xi^1 + 1 \rangle$
$\mathbb{F}_{2^{192}}$	$\mathbb{F}_2[\xi]/\langle \xi^{192} + \xi^7 + \xi^2 + \xi^1 + 1 \rangle$
$\mathbb{F}_{2^{256}}$	$\mathbb{F}_2[\xi]/\langle \xi^{256} + \xi^{10} + \xi^5 + \xi^2 + 1 \rangle$

4.2 Keys and signature sizes

Public key. The public key has format $pk := (\mathsf{seed}_{\mathsf{pk}}, \boldsymbol{y})$; consisting of a λ-bit seed $\mathsf{seed}_{\mathsf{pk}}$ used to generate the matrix $\boldsymbol{H'}$, and a serialized vector $\boldsymbol{y} := \boldsymbol{Hx} \in \mathbb{F}_2^{n-k}$ corresponding to the syndrome. The public key has a total size (in bytes) of

$$|\mathsf{pk}| = \frac{1}{8}(\lambda + n - k) \ .$$

Secret key. The secret key has format $sk := (\mathsf{seed}_{\mathsf{pk}}, \boldsymbol{y}, \mathsf{wit}, \mathsf{seed}_{\mathsf{sk}})$; consisting of the same $\mathsf{seed}_{\mathsf{pk}}$ and $\boldsymbol{y}$ as the public key, as well as the λ-bit seed $\mathsf{seed}_{\mathsf{sk}}$ and the serialized witness wit. The latter is made of w sets of d truncated elementary vectors of size $\mu_1 - 1, \ldots, \mu_d - 1$. Thus, the size of the secret key (in bytes) is

$$|\mathsf{sk}| = \left\lceil \frac{1}{8}\big(2\lambda + (n-k) + w \cdot \sum_{i=1}^{d}(\mu_i - 1)\big) \right\rceil \ .$$

Signature size. The size (in bits) of a signature is:

$$\begin{aligned}
|\sigma| = 3\lambda & & \to \mathsf{salt}, h_{\text{piop}} \\
+ (\tau-1)\cdot(|\mathsf{wit}|_2 + (d-1)\lambda + (\lambda+B)) & & \to \mathsf{aux} \\
+ (\lambda + B) & & \to \boldsymbol{\alpha}'_{\mathsf{plain}} \\
+ |\mathsf{wit}|_2 & & \to \Delta\mathsf{wit} \\
+ d\cdot\lambda & & \to (\alpha_1,\ldots,\alpha_d) \\
+ \lambda\cdot T_{\text{open}} + \tau\cdot(2\lambda) + 32 & & \to \mathsf{pdecom}
\end{aligned}$$

with $|\mathsf{wit}|_2 := \mathsf{ceil}_8(w\cdot\sum_{i=1}^{d}(\mu_i-1))$, where $\mathsf{ceil}_8(x) = 8\cdot\lceil x/8\rceil$.

4.3 Selected parameters

The signature parameters of our proposed instances are summarized in Table 3 and in Table 4 for the different security categories. Table 3 gives the syndrome decoding parameters which are common to both trade-offs while Table 4 gives the proof system parameters and associated sizes.

Table 3: RSD parameters of SD-in-the-Head-2.

Parameter Sets	NIST Security		RSD Parameters					Modeling
	Category	Bits	q	n	$(n-k_{\text{RSD}})$	k_{RSD}	w	μ
SDitH2-L1-gf2	I	143	2	10 360	432	9 928	56	[4,4,4,3]
SDitH2-L3-gf2	III	207	2	18 396	592	17 804	73	[4,4,4,4]
SDitH2-L5-gf2	V	272	2	19 864	800	19 064	104	[4,4,4,3]

Table 4: Proof system parameters of SD-in-the-Head-2, with key and signature sizes.

Parameter Set	Proof System Parameters					Sizes (Bytes)			
	τ	κ	w_{pow}	T_{open}	B	pk	sk	Sig. Avg	Sig. Max
SDitH2-L1-gf2-short	11	11	9	107	16	70	163	3 705	3 705
SDitH2-L1-gf2-fast	16	8	2	101	16	70	163	4 484	4 484
SDitH2-L3-gf2-short	16	12	2	157	16	98	232	7 964	7 964
SDitH2-L3-gf2-fast	24	8	2	153	16	98	232	9 916	9 916
SDitH2-L5-gf2-short	21	12	6	216	16	132	307	14 121	14 121
SDitH2-L5-gf2-fast	32	8	2	207	16	132	307	17 540	17 540

4.4 Benchmarks

Table 5 and Table 6 provide benchmarks for the key generation, signature and verification algorithms of SD-in-the-Head-2 on a laptop and a cloud server resectively. The provided timings are median over 200 runs. For Category I, the signing and verification algorithms run in 2–3 ms with the "fast" variant and in 6–9 ms with the "short" one.

Table 5: Timings on a laptop with 12th Gen Intel Core i7-1260P (median after 200 runs).

	KeyGen	Sign	Verif
SDitH2-L1-gf2-short	0.63 ms	6.73 ms	6.04 ms
SDitH2-L1-gf2-fast	0.74 ms	2.01 ms	1.79 ms
SDitH2-L3-gf2-short	3.02 ms	42.26 ms	39.83 ms
SDitH2-L3-gf2-fast	1.56 ms	6.36 ms	5.75 ms
SDitH2-L5-gf2-short	1.55 ms	60.48 ms	57.23 ms
SDitH2-L5-gf2-fast	1.82 ms	9.42 ms	8.70 ms

Table 6: Timings on a cloud server with AMD EPYC 7B13 @ 2.45GHZ (median after 200 runs).

	KeyGen	Sign	Verif
SDitH2-L1-gf2-short	0.61ms	9.33ms	8.18ms
SDitH2-L1-gf2-fast	0.59ms	2.96ms	2.67ms
SDitH2-L3-gf2-short	1.71ms	44.54ms	41.38ms
SDitH2-L3-gf2-fast	1.61ms	7.86ms	7.11ms
SDitH2-L5-gf2-short	1.94ms	62.18ms	57.56ms
SDitH2-L5-gf2-fast	1.92ms	11.17ms	10.23ms

5 Security

5.1 SD to RSD security reduction

We provide hereafter a security reduction from SD to RSD. Namely, any SD instance can be solved using an RSD solver. The latter must be called a certain number of times which implies a security gap between the two instances. This gap must be compensated by increasing the RSD parameters. As explained in Section 4.1, this is precisely the approach we followed to select the RSD parameters of SD-in-the-Head-2.

Theorem 5.1. *Let $\mathbb{F}$ be a finite field. Let $\theta \in [1, \infty)$. Let n, k_{SD}, k_{RSD}, w be positive integers such that $k_{\text{SD}} \leq k_{\text{RSD}} \leq n$, $w < n$, $w \mid n$, and*

$$k_{\text{RSD}} \leq n - w \cdot \log_2 \left(\frac{n}{w}\right) - \log_2 \theta \ .$$

Let $\mathcal{A}_{\text{RSD}}$ be an algorithm solving a random $(\mathbb{F}, n, k_{\text{RSD}}, w)$-instance of the regular syndrome decoding problem in time t with success probability ε_{RSD}. Then there exists an algorithm $\mathcal{A}_{\text{SD}}$ solving a random $(\mathbb{F}, n, k_{\text{SD}}, w)$-instance of the standard syndrome decoding problem in time t with probability ε_{SD}, where

$$\varepsilon_{\text{SD}} \geq e^{-1/\theta} \cdot \frac{\left(\frac{n}{w}\right)^w}{\binom{n}{w}} \cdot \varepsilon_{\text{RSD}} \ .$$

Remark 1. *In the above theorem, for $\theta = 1$, the constraint $k_{\text{RSD}} \leq n - w \cdot \log_2 \left(\frac{n}{w}\right)$ implies that the regular SD instance $(\mathbb{F}, n, k_{\text{RSD}}, w)$ has density at most 1. Taking a greater θ implies a lower density.*

Proof. We adapt the proof of [FJR22] to our context. To prove the theorem, we build an algorithm $\mathcal{A}_{\text{SD}}$ to solve the traditional SD problem of parameters (n, k, w) using an algorithm $\mathcal{A}_{\text{RSD}}$ which solves the regular SD problem with the same parameters.

Algorithm $\mathcal{A}_{\text{SD}}$ (on input an SD instance $(\boldsymbol{H}, \boldsymbol{y})$):

1. Sample a permutation σ of $\{1, \ldots, n\}$.
2. Permute the columns of $\boldsymbol{H}$ using σ to get $\hat{\boldsymbol{H}}$.
3. Remove the $k_{\text{RSD}} - k_{\text{SD}}$ last rows of $\hat{\boldsymbol{H}}$ and the last $k_{\text{RSD}} - k_{\text{SD}}$ coordinates of $\boldsymbol{y}$ to obtain $\hat{\boldsymbol{H}}_{\text{tr}}$ and $\boldsymbol{y}_{\text{tr}}$.
3. Run $\mathcal{A}_{\text{RSD}}$ on input $(\hat{\boldsymbol{H}}_{\text{tr}}, \boldsymbol{y}_{\text{tr}})$ to get $\hat{\boldsymbol{x}}$.
4. If $\hat{\boldsymbol{x}} = \perp$, return $\perp$.
5. If $\hat{\boldsymbol{H}}\hat{\boldsymbol{x}} \neq \boldsymbol{y}$, return $\perp$.
6. Permute the coordinates of $\hat{\boldsymbol{x}}$ using σ^{-1} to get $\boldsymbol{x}$.
7. Return $\boldsymbol{x}$.

The probability to transform an SD instance into a regular SD instance in Step 2 is $\left(\frac{n}{d}\right)^w / \binom{n}{w}$.

Thus we have

$$
\begin{aligned}
\varepsilon_{\text{SD}} &:= \Pr[\mathcal{A}_{\text{SD}}(\boldsymbol{H}, \boldsymbol{y}) \neq \perp] \\
&\geq \Pr[\mathcal{A}_{\text{SD}}(\boldsymbol{H}, \boldsymbol{y}) \neq \perp \cap (\hat{\boldsymbol{H}}, \boldsymbol{y}) \text{ is an RSD instance}] \\
&= \frac{\left(\frac{n}{w}\right)^w}{\binom{n}{w}} \cdot \Pr[\mathcal{A}_{\text{SD}}(\boldsymbol{H}, \boldsymbol{y}) \neq \perp \mid (\hat{\boldsymbol{H}}, \boldsymbol{y}) \text{ is an RSD instance}] \\
&= \frac{\left(\frac{n}{w}\right)^w}{\binom{n}{w}} \cdot \Pr[\mathcal{A}_{\text{RSD}}(\hat{\boldsymbol{H}}_{\text{tr}}, \boldsymbol{y}_{\text{tr}}) \neq \perp \text{ and } \hat{\boldsymbol{H}}\hat{\boldsymbol{x}} = \boldsymbol{y} \mid (\hat{\boldsymbol{H}}, \boldsymbol{y}) \text{ is an RSD instance}] \\
&= \frac{\left(\frac{n}{w}\right)^w}{\binom{n}{w}} \cdot \Pr[\mathcal{A}_{\text{RSD}}(\hat{\boldsymbol{H}}_{\text{tr}}, \boldsymbol{y}_{\text{tr}}) \neq \perp \mid (\hat{\boldsymbol{H}}, \boldsymbol{y}) \text{ is an RSD instance}] \\
&\qquad \cdot \Pr[\hat{\boldsymbol{H}}\hat{\boldsymbol{x}} = \boldsymbol{y} \mid (\hat{\boldsymbol{H}}, \boldsymbol{y}) \text{ is an RSD instance}, \mathcal{A}_{\text{RSD}}(\hat{\boldsymbol{H}}_{\text{tr}}, \boldsymbol{y}_{\text{tr}}) \neq \perp]
\end{aligned}
$$

By definition we have:

$$\Pr[\mathcal{A}_{\text{RSD}}(\hat{\boldsymbol{H}}_{\text{tr}}, \boldsymbol{y}_{\text{tr}}) \neq \perp \mid (\hat{\boldsymbol{H}}, \boldsymbol{y}) \text{ is an RSD instance}] = \varepsilon_{\text{RSD}} \ .$$

On the other hand,

$$
\begin{aligned}
&\Pr[\hat{\boldsymbol{H}}\hat{\boldsymbol{x}} = \boldsymbol{y} \mid (\hat{\boldsymbol{H}}, \boldsymbol{y}) \text{ is an RSD instance}, \mathcal{A}_{\text{RSD}}(\hat{\boldsymbol{H}}_{\text{tr}}, \boldsymbol{y}_{\text{tr}}) \neq \perp] \\
&\qquad \geq \Pr[(\hat{\boldsymbol{H}}_{\text{tr}}, \boldsymbol{y}_{\text{tr}}) \text{ has a single RSD solution} \mid \mathcal{A}_{\text{RSD}}(\hat{\boldsymbol{H}}_{\text{tr}}, \boldsymbol{y}_{\text{tr}}) \neq \perp] \ .
\end{aligned}
$$

Indeed, if $(\hat{\boldsymbol{H}}_{\text{tr}}, \boldsymbol{y}_{\text{tr}})$ has a single RSD solution, then given that $(\hat{\boldsymbol{H}}, \boldsymbol{y})$ is an RSD instance, we have that the right solution is returned by $\mathcal{A}_{\text{RSD}}$. Now, we heuristically have:

$$\Pr[(\hat{\boldsymbol{H}}_{\text{tr}}, \boldsymbol{y}_{\text{tr}}) \text{ has a single RSD solution} \mid \mathcal{A}_{\text{RSD}}(\hat{\boldsymbol{H}}_{\text{tr}}, \boldsymbol{y}_{\text{tr}}) \neq \perp] \approx \left(1 - \frac{1}{nb_y}\right)^{nb_x} ,$$

which is the probability that no other $\hat{\boldsymbol{x}}$ lead to the same $\boldsymbol{y}_{\text{tr}}$, with $nb_x = \left(\frac{n}{w}\right)^w - 1$ and $nb_y = 2^{n-k_{\text{RSD}}} \leq \theta \cdot \left(\frac{n}{w}\right)^w$, implying

$$\left(1 - \frac{1}{nb_y}\right)^{nb_x} \approx e^{-1/\theta} \ .$$

□

Informally, the above result holds because an instance of the standard SD problem is an instance of the regular syndrome decoding problem with probability $\left(\frac{n}{w}\right)^w / \binom{n}{w}$. Moreover, a standard syndrome decoding instance can be "randomized" and input to the regular adversary as much as desired.

All the regular SD instances used in SD-in-the-Head are chosen such that the corresponding standard SD instance achieves a security level which compensates the degradation. We stress that this might be overly conservative.

5.2 Attacks against the SD problem

The binary SD problem has been studied for many years: the first attack was proposed by Prange in the 60's. Later further attacks were proposed by Stern and Dumer at the turning of the 80's and the 90's. Many new attacks came out since 2010: Becker, Joux, May and

Meurer [BJM+12] in 2012, then May and Ozerov [MO15] in 2015 and Both and May [BM18] in 2018. Eventually a cryptographic estimator was proposed by Esser and Bellini in 2022 [EB22].

While original attacks did not use much memory, recent attacks make extensive use of memory. In this context, the cost of memory accesses (as a function of the memory size) is an essential parameter [EB22]. This cost can be considered to be constant, logarithmic, cubic root or square root. While a constant-time memory-access cost might seem very optimistic from the attacker standpoint (and hence over conservative), a square root access might seem too pessimistic (hence risky in terms of security). Unfortunately, there is no wide consensus on which cost is the most practically relevant.[2] It is interesting to notice that considering a cubic or square root memory-access cost (which is theoretically meaningful) significantly limit the impact of recent attacks whose efficiency relies on the availability of a large memory.

For our SD security estimates, we made the conservative choices of considering a logarithmic memory-access cost and a memory size limited to 2^{143} for Category I, and 2^{160} for Categories III and V (which roughly corresponds to the number of atoms on earth). To derive our concrete parameters, we used the "Syndrome Decoding Estimator", an open-source tool available at: https://github.com/Crypto-TII/syndrome_decoding_estimator.

5.3 Unforgeability

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

Our security statement is based on the following assumptions:

- **SD hardness.** Solving the considered SD instance is $(\epsilon_{\text{SD}}, t)$-*hard* for some $(\epsilon_{\text{SD}}, t)$ which are implicit functions of the security parameter λ. Formally, any adversary $\mathcal{A}$ on input a random SD instance and running in time at most t has probability at most ϵ_{SD} to output the solution of the input instance.

- **Random Oracle Modem (ROM).** Our security statement holds in the ROM where the (extendable-output) hash function Hash is modelled as a random oracle.

- **Ideal Cipher Model (ICM).** Our security statement holds in the ICM where the block cipher Enc is modelled as an ideal cipher.

Based on the ROM and the ICM, the EUF-CMA security of SD-in-the-Head holds from the soundness and zero-knowledge properties of the underlying ZK-PoK (which are overviewed in Section 2). The formal EUF-CMA security proof of SD-in-the-Head will be added to a future version of the specification. It will heavily rely on usual techniques for MPC-in-the-Head signature schemes with GGM trees.

[2] See, e.g., https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/EiwxGnfQgec/m/xBky_FKFDgAJ

6 Variants

In this section, we outline two possible variants of SD-in-the-Head-2, namely a variant using the base field $\mathbb{F}_{256}$ and a variant using the Threshold-Computation-in-the-Head framework. We might consider including these variants as formal instances of the scheme in the future.

6.1 The $\mathbb{F}_{256}$ variant

In the previous sections, we focused on the RSD problem over the binary field $\mathbb{F}_2$. We might consider larger fields $\mathbb{F}_q$, with $q > 2$. Given a matrix $\boldsymbol{H} \in \mathbb{F}_q^{(n-k)\times n}$ and a syndrome vector $\boldsymbol{y} \in \mathbb{F}_q^{n-k}$, the RSD problem over $\mathbb{F}_q$ consists in finding a vector $\boldsymbol{x} \in \mathbb{F}_q^n$ such that

- $\boldsymbol{x}$ satisfies the linear relation $\boldsymbol{y} = \boldsymbol{H}\boldsymbol{x}$, and
- $\boldsymbol{x}$ is regular, meaning that it is the concatenation of w scaled elementary vectors $e_1, \ldots, e_w$ of size $\frac{n}{w}$ (*i.e.* vector with $\frac{n}{w} - 1$ coefficients 0 and one non-zero random coefficient).

The main difference when working on a larger field is that the non-zero coordinates of the solution $\boldsymbol{x}$ are not necessary 1, they can be other non-zero values. In this context, the witness wit should contain the value of those coordinates, together with their positions. To be able to embed the tensor-product decomposition of the RSD solution over $\mathbb{F}_2$ while working over $\mathbb{F}_q$, we need to consider fields of characteristic 2, *i.e.* binary field extensions. The size of the secret key (in bytes) is then

$$|\mathsf{sk}| = \left\lceil \frac{1}{8}\left(2\lambda + (n-k)\cdot \log_2(q) + w \cdot \sum_{i=1}^{d}(\mu_i - 1) + \underbrace{w \cdot \log_2(q)}_{\substack{\text{Values of the}\\ \text{non-zero coordinates}}}\right)\right\rceil ,$$

while the public key has a total size (in bytes) of

$$|\mathsf{pk}| = \left\lceil \frac{1}{8}(\lambda + (n-k)\cdot \log_2(q))\right\rceil .$$

The polynomial constraints that the witness should satisfy are very similar to those of the $\mathbb{F}_2$ case. The only difference comes from the fact that we need to scale each chunk e_i by the non-zero value. In practice, this can be done just after applying the mux tree at each chunk. The size (in bits) of a signature is:

$$\begin{aligned}
|\sigma| = 3\lambda & \quad\quad \to \mathsf{salt}, h_{\text{piop}} \\
+ (\tau - 1)\cdot(|\mathsf{wit}|_2 + (d-1)\lambda + (\lambda + B)) & \quad\quad \to \mathsf{aux} \\
+ (\lambda + B) & \quad\quad \to \boldsymbol{\alpha}'_{\mathsf{plain}} \\
+ |\mathsf{wit}|_2 & \quad\quad \to \Delta\mathsf{wit} \\
+ d\cdot\lambda & \quad\quad \to (\alpha_1, \ldots, \alpha_d) \\
+ \lambda \cdot T_{\text{open}} + \tau\cdot(2\lambda) + 32 & \quad\quad \to \mathsf{pdecom}
\end{aligned}$$

with $|\mathsf{wit}|_2 := \mathsf{ceil}_8(w\cdot\log_2(q) + w\cdot\sum_{i=1}^{d}(\mu_i - 1))$, where $\mathsf{ceil}_8(x) = 8\cdot\lceil x/8\rceil$.

We exhibit some instances over $\mathbb{F}_{256}$ in Table 3 and in Table 4 for the different security categories. Table 3 gives the syndrome decoding parameters which were obtained following the approach described in Section 4.1 adapted to $\mathbb{F}_{256}$. Table 4 gives the proof system parameters (which are the same as for the $\mathbb{F}_2$ instances) and the associated sizes. We observe that we get similar sizes than over $\mathbb{F}_2$.

While working on $\mathbb{F}_2$ can be considered as a more conservative choice than working on $\mathbb{F}_{256}$, the latter has the main advantage to lead to a lighter scheme. In particular, the PIOP computation is expected to be significantly faster over $\mathbb{F}_{256}$ as requiring much fewer large-field multiplications. This is because the manipulated vectors are shorter while, on the other hand, the large field remains the same (i.e. $\mathbb{F}_{2^\lambda}$). Moreover, the uncompressed matrix $\boldsymbol{H'}$ is smaller over $\mathbb{F}_{256}$. For instance, for Category I, we have 560 KB for $\boldsymbol{H'}$ on $\mathbb{F}_2$ versus 140 KB on $\mathbb{F}_{256}$.

Table 7: RSD parameters of SD-in-the-Head-2 for $\mathbb{F}_{256}$.

Parameter Sets	**NIST Security**		**RSD Parameters**					**Modeling**
	Category	Bits	q	n	$(n-k_{\mathrm{RSD}})$	k_{RSD}	w	μ
SDitH2-L1-gf256	I	143	256	2 176	64	2 112	34	[4,4,4]
SDitH2-L3-gf256	III	207	256	3 200	96	3 104	50	[4,4,4]
SDitH2-L5-gf256	V	272	256	4 224	120	4 104	66	[4,4,4]

Table 8: Proof system parameters of SD-in-the-Head-2 over $\mathbb{F}_{256}$, with key and signature sizes.

Parameter Set	**Proof System Parameters**					**Sizes (Bytes)**			
	τ	κ	w	T_{open}	B	pk	sk	Sig.	Baseline ($\mathbb{F}_2$)
SDitH2-L1-gf256-short	11	11	9	107	16	80	169	3 661	3 705
SDitH2-L1-gf256-fast	16	8	2	101	16	80	169	4 420	4 484
SDitH2-L3-gf256-short	16	12	2	157	16	120	251	7 916	7 964
SDitH2-L3-gf256-fast	24	8	2	153	16	120	251	9 844	9 916
SDitH2-L5-gf256-short	21	12	6	216	16	152	325	14 079	14 121
SDitH2-L5-gf256-fast	32	8	2	207	16	152	325	17 476	17 540

6.2 The TCitH variant

As explained in Section 2.1, an alternative choice to the VOLE-in-the-Head framework [BBD+23] is the Trheshold-Computation-in-the-Head (TCitH) framework [FR23a]. The TCitH-based commitment scheme enables the prover/signer to open one evaluation among only N, while the computational complexity of the commitment procedure is linear in N. While the PIOP soundness error is d/N^τ for VOLEitH, it is $(d/N)^\tau$ for TCitH with τ parallel repetitions, which means that we must ensure $(d/N)^\tau \cdot 2^{-w_{\mathrm{pow}}} \leq 2^{-\lambda}$ to achieve a λ-bit security. The size (in bits) of a TCitH-based SD-in-the-Head-2 signature is:

$$\begin{aligned}
|\sigma| = 3\lambda & \quad \rightarrow \mathsf{salt}, h_{\mathrm{piop}} \\
+ \tau \cdot |\mathsf{wit}|_2 & \quad \rightarrow \Delta\mathsf{wit} \\
+ \tau \cdot (d-1) \cdot \mathsf{ceil}_\kappa(\lambda) & \quad \rightarrow (\alpha_2, \ldots, \alpha_d) \\
+ \lambda \cdot T_{\mathrm{open}} + \tau \cdot (2\lambda) + 32 & \quad \rightarrow \mathsf{pdecom}
\end{aligned}$$

with $|\mathsf{wit}|_2 := \mathsf{ceil}_8(w \cdot \sum_{i=1}^{d}(\mu_i - 1))$, where $\mathsf{ceil}_n(x) = n \cdot \lceil x/n \rceil$.

Table 9 gives the TCitH-based proof system parameters and the associated signature sizes. Let us stress that the key generation is the same in both approaches and so the key sizes are those in Table 4. We can observe that the signature sizes are larger when using the TCitH framework. This comes from the required increase of τ to maintain an equivalent security. This is particularly true for SD-in-the-Head-2, for which the RSD modeling leads to degree-4 (*i.e.* $d = 4$) constraints, while most of the schemes in the literature consider only degree-2. This size increasing is partly compensated by the fact that TCitH does not include the communication cost induced by the VOLEitH consistency check.

While it leads to larger signatures, the main advantage of the TCitH variant is to be structurally simpler: it does not have consistency check, it is derived from a 5-round interactive protocol (while the VOLEitH framework gives 7-round protocol), and it does not require to work in a large field extension.

Table 9: Proof system parameters of the TCitH-based variant of SD-in-the-Head-2, with signature sizes. Key sizes are similar to our main VOLEitH-based instances.

Parameter Set	**Proof System Parameters**				**Sizes (Bytes)**	
	τ	κ	w_{pow}	T_{open}	Sig.	Baseline (VOLEitH)
SDitH2-TCitH-L1-gf2-short	14	11	2	125	4 271	3 705 (-13%)
SDitH2-TCitH-L1-gf2-fast	21	8	2	135	5 509	4 484 (-19%)
SDitH2-TCitH-L3-gf2-short	19	12	2	185	8 426	7 964 (-5%)
SDitH2-TCitH-L3-gf2-fast	31	8	6	212	11 374	9 916 (-13%)
SDitH2-TCitH-L5-gf2-short	25	12	6	265	15 618	14 121 (-10%)
SDitH2-TCitH-L5-gf2-fast	42	8	4	280	19 968	17 540 (-12%)

7 Advantages and limitations

In this section we describe some advantages and limitations of the SD-in-the-Head signature scheme. The bottom line is that it provides both conservative security *and* relatively small signatures compared to current PQC standards.

7.1 Advantages of SD-in-the-Head

Conservative hardness assumption. Our signature scheme is based on the presumably hardest problem in code-based cryptography: the *unstructured* binary Syndrome Decoding (SD) problem for random linear codes. This problem is known to be NP-hard and the cryptanalysis state-of-the-art has been stable and well-established for decades.

Adaptive and tunable parameters. MPCitH enables us to tailor parameters, in particular the size of GGM trees (i.e., the size of the small evaluation domain), meaning that we can provide a variety of parameter sets tailored to different use cases similarly to SPHINCS$^+$. This is illustrated by our 'fast' and 'short' parameter sets that provide two different signature sizes/performance trade-offs. In addition, the size of the signature is composed of two parts: a part related to GGM trees and a part related to the SD instance. The latter part is not the bottleneck so that increasing the size of the SD parameters (and hence the associated SD security) only has a moderate impact on the global signature size.

Small code-based signatures. The SD-in-the-Head signature scheme achieves among the smallest code-based signatures to-date, which does not come at the cost of a large public key.

Small key sizes. Both the secret key and public key sizes are much smaller in comparison to the lattice-based signature standards, and compete with SHL-DSA. In particular, the public key, which is often transported with the signature (e.g., certificates in TLS), is between 120-240 bytes across all security levels for both variants.

Size of public key and signature. SD-in-the-Head offers competitive signature sizes along with very small public keys, which yields a competitive signature + public key size. For NIST security level I, the sum of the signature and public key sizes of SD-in-the-Head gives 3.8 kB, which is comparable or smaller than the post-quantum NIST standards ML-DSA (Dillitium) and SLH-DSA (SPHINCS+) with 3.7 kB and 7.8 kB respectively.

7.2 Limitations of SD-in-the-Head

Quadratic growth w.r.t. the security level. As other MPCitH schemes, or, more generally, as other schemes applying the Fiat-Shamir transform to a parallelly repeated ZK-PoK with non-negligible soundness error, SD-in-the-Head suffers a quadratic growth of its signature size.

Efficiency. MPCitH-like schemes require the generation of lots of pseudorandom objects, which makes them slow in comparison to other schemes such as the NIST post-quantum standard ML-DSA. Nonetheless, the efficiency of SD-in-the-Head is competitive when compared with many other post-quantum signature schemes.

Low-cost devices and embedded systems. SD-in-the-Head might be particularly heavy for low-cost devices such as smart cards or embedded systems, although it has the potential to perform well on hardware as being highly parallelizable.

References

[AGH+22] C. Aguilar-Melchor, N. Gama, J. Howe, A. Hülsing, D. Joseph, and D. Yue. The return of the SDitH. Cryptology ePrint Archive, Report 2022/1645, 2022 (cited on page 2).

[AGH+23] C. Aguilar-Melchor, N. Gama, J. Howe, A. Hülsing, D. Joseph, and D. Yue. The return of the SDitH. In C. Hazay and M. Stam, editors, *EUROCRYPT 2023, Part V*, pages 564–596. Springer, Cham, 2023 (cited on page 1).

[BBD+23] C. Baum, L. Braun, C. Delpech de Saint Guilhem, M. Klooß, E. Orsini, L. Roy, and P. Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head. In H. Handschuh and A. Lysyanskaya, editors, *CRYPTO 2023, Part V*, pages 581–615. Springer, Cham, 2023 (cited on pages 1–3, 43).

[BBG+24] S. Bettaieb, L. Bidoux, P. Gaborit, and M. Kulkarni. Modelings for generic pok and applications: shorter SD and PKP based signatures. Cryptology ePrint Archive, Report 2024/1668, 2024 (cited on pages 2, 3, 7, 8).

[BBM+24] C. Baum, W. Beullens, S. Mukherjee, E. Orsini, S. Ramacher, C. Rechberger, L. Roy, and P. Scholl. One tree to rule them all: optimizing GGM trees and OWFs for post-quantum signatures. Cryptology ePrint Archive, Report 2024/490, 2024 (cited on pages 9, 19).

[BJM+12] A. Becker, A. Joux, A. May, and A. Meurer. Decoding random binary linear codes in $2^{n/20}$: how $1 + 1 = 0$ improves information set decoding. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT 2012*, pages 520–536. Springer, Berlin, Heidelberg, 2012 (cited on page 41).

[BM18] L. Both and A. May. Decoding linear codes with high error rate and its impact for LPN security. In T. Lange and R. Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, pages 25–46. Springer, Cham, 2018 (cited on page 41).

[CDI05] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In J. Kilian, editor, *TCC 2005*, pages 342–362. Springer, Berlin, Heidelberg, 2005 (cited on page 4).

[EB22] A. Esser and E. Bellini. Syndrome decoding estimator. In G. Hanaoka, J. Shikata, and Y. Watanabe, editors, *PKC 2022, Part I*, pages 112–141. Springer, Cham, 2022 (cited on page 41).

[Fen24] T. Feneuil. The Polynomial-IOP Vision of the Latest MPCitH Frameworks for Signature Schemes. Post-Quantum Algebraic Cryptography - Workshop 2, Institut Henri Poincaré, Paris, France, 2024 (cited on page 3).

[FJR22] T. Feneuil, A. Joux, and M. Rivain. Syndrome decoding in the head: shorter signatures from zero-knowledge proofs. In Y. Dodis and T. Shrimpton, editors, *CRYPTO 2022, Part II*, pages 541–572. Springer, Cham, 2022 (cited on pages 2, 7, 39).

[FR22] T. Feneuil and M. Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. Cryptology ePrint Archive, Report 2022/1407, 2022 (cited on page 2).

[FR23a] T. Feneuil and M. Rivain. Threshold computation in the head: improved framework for post-quantum signatures and zero-knowledge arguments. Cryptology ePrint Archive, Report 2023/1573, 2023 (cited on pages 3, 4, 43).

[FR23b] T. Feneuil and M. Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. In J. Guo and R. Steinfeld, editors, *ASIACRYPT 2023, Part I*, pages 441–473. Springer, Singapore, 2023 (cited on pages 1, 3).

[FS87] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *CRYPTO'86*, pages 186–194. Springer, Berlin, Heidelberg, 1987 (cited on pages 3, 9).

[IKO+07] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge from secure multiparty computation. In D. S. Johnson and U. Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, 2007 (cited on pages 2, 3).

[ISN89] M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, (9):56–64, 1989. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/ecjc.4430720906 (cited on page 4).

[MO15] A. May and I. Ozerov. On computing nearest neighbors with applications to decoding of binary linear codes. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part I*, pages 203–228. Springer, Berlin, Heidelberg, 2015 (cited on page 41).

[OTX24] Y. Ouyang, D. Tang, and Y. Xu. Code-based zero-knowledge from VOLE-in-the-head and their applications: simpler, faster, and smaller. In K.-M. Chung and Y. Sasaki, editors, *ASIACRYPT 2024, Part V*, pages 436–470. Springer, Singapore, 2024 (cited on pages 2, 3, 7, 8).

[ZCD+20] G. Zaverucha, M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, J. Katz, X. Wang, V. Kolesnikov, and D. Kales. Picnic. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions (cited on page 2).