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

Algorithm Specifications and Supporting Documentation – Version 1.1

Carlos Aguilar Melchor Thibauld Feneuil Nicolas Gama

Shay Gueron James Howe David Joseph Antoine Joux

Edoardo Persichetti Tovohery H. Randrianarisoa

Matthieu Rivain Dongze Yue

November 3, 2023

CISPA
CryptoExperts
Florida Atlantic University
Meta
SandboxAQ
Sapienza University
Umeå University
University of Haifa

Contents

Changelog

2023-11-03: Version 1.0 → Version 1.1

The original syndrome decoding (SD) parameters proposed in Version 1.0 did not achieve the claimed security levels because of the non-uniqueness of the SD solution (while uniqueness was assumed to estimate the security). There was a small number of solutions with the original parameters (between 30 and 1000 solutions for all proposed parameter sets) which decreased the claimed security by a few bits [CT23]. In Version 1.1, we derive new parameters for our SD instances to avoid this issue: for a given code length m and dimension k, the weight parameter w is defined such that the number of expected solutions is below 1.01. We have also updated our script estimating the hardness of the information-set decoding (ISD) algorithm to be more conservative – see explanation added in the last paragraph of Section 7.1. Besides the new parameters depicted in Section 4, the resulting sizes and running times have also been updated in Section 4 and Section 5.

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 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 MPCitH framework has recently been improved in a series of works which makes it an effective and versatile tool for the design of post-quantum signature schemes. The SD-in-the-Head protocol was initially proposed in [FJR22] and further improved in subsequent works [AMGH+22; FR22]. The present specification provides a detailed description of the SD-in-the-Head scheme with two variants from these follow-up works: the *hypercube variant* and the *threshold variant*.

Organization of this specification

Section 2 gives a high-level description of the SD-in-the-Head scheme from the underlying MPC protocol to the signature scheme under its two variants. Section 3 provides a detailed description of the key generation, signature and verification algorithms for the two variants. 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 two variants and the three considered security levels. Section 5 provides performance figures for our different instances. The security of the SD-in-the-Head signature scheme is analyzed in Section 6 while Section 7 further evaluates the complexity of known attacks. We finally list some advantages and limitations of the scheme in Section 8.

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 of the SD-in-the-Head signature scheme

The SD-in-the-Head signature scheme relies on an MPC protocol which efficiently checks whether a given shared input corresponds to the solution of a syndrome decoding instance [FJR22]. By applying the MPC-in-the-Head paradigm [IKO+07], this protocol is turned into a zero-knowledge proof of knowledge for the syndrome decoding problem that is then transformed into a signature scheme using the Fiat-Shamir heuristic [FS87]. The original SD-in-the-Head scheme has been optimized in two follow-up articles:

1. [AMGH+23] proposes to correlate the sharings of several parallel repetitions of the MPC protocol using a geometric structure, known as *Hypercube-MPCitH*; this technique gives rise to the *hypercube variant* of the SD-in-the-Head signature scheme;
2. [FR22] proposes to replace the traditional additive sharings by low-threshold linear secret sharings to exploit their error-correcting feature; this technique gives rise to the *threshold approach* of the SD-in-the-Head signature scheme.

The following sections outline the high-level ideas behind the SD-in-the-Head scheme and the variants obtained by applying these two approaches. The notations used in these sections are summarized in Table 1.

2.1 Overview of the SD-in-the-Head signature scheme

2.1.1 The syndrome decoding problem

Syndrome decoding (SD) is a problem that is central to many code-based cryptosystems. A syndrome is the result of multiplying a vector x with a parity-check matrix H. The "coset weights" flavor of the SD problem [BMVT78] can be expressed as follows:

- *Problem instance:* Parity-check matrix $H \in \mathbb{F}_q^{(m-k)\times m}$ and syndrome $y \in \mathbb{F}_q^{m-k}$.
- *Solution:* Vector $x \in \mathbb{F}_q^m$ with $\mathrm{wt}(x) \leq w$ such that $Hx = y$.

To generate an SD instance, H and x (with $\mathrm{wt}(x) = w$) are drawn uniformly at random and then $y = Hx$ is calculated.

2.1.2 The SD-in-the-Head MPC protocol

In this section, we describe the MPC protocol which is at the core of the SD-in-the-Head signature scheme. The so-called SD-in-the-Head MPC protocol runs a multi-party computation which verifies the correctness of a solution x to a public SD instance (H, y). Some witness derived from x is shared between N parties which, after running the protocol, either output Accept if the input sharing is believed to encode a correct SD solution or Reject otherwise.

Standard form of the parity-check matrix. For efficiency, we assume that H is in standard form $H = (H'|I_{m-k})$, where $H' \in F_q^{(m-k)\times k}$. This enables us to express

$$y = Hx = H'x_A + x_B, \tag{1}$$

where $x = (x_A|x_B)$. This representation has two benefits:

Table 1: Notations and parameters of the SD-in-the-Head scheme.

Syndrome decoding parameters:	
q	Size of the SD based field.
m	Code length.
k	Vector dimension.
w	Hamming weight bound.
d	Parameter of the d-splitting variant.
Signature Parameters:	
λ	Security parameter.
N	Number of secret parties.
τ	Number of repetitions.
t	Number of random evaluation points.
Syndrome decoding instance:	
H	Parity-check matrix.
x	Solution of the SD instance ($\mathrm{wt}(x) \leq w$).
y	Syndrome $y = Hx$.
H'	Random part of the parity-check matrix s.t. $H = (H'\|I_{m-k})$.
(x_A, x_B)	Two halves of the SD solution s.t. $y = H'x_A + x_B$.
Fields:	
$\mathbb{F}_q$	Field with q elements: base field of the SD instance.
$f_1, \ldots, f_q$	Elements of $\mathbb{F}_q$.
$\mathbb{F}_{\text{points}}$	Extension field of $\mathbb{F}_q$ (base field of the MPC elements $\alpha, \beta, v, r, \varepsilon$).
η	Field extension s.t. $\mathbb{F}_{\text{points}} = \mathbb{F}_{q^\eta}$.
Multi-Party Computation:	
S, Q, P	Polynomials of $\mathbb{F}_q[X]$, witness of the syndrome decoding proof.
F	Vanishing polynomial of the set $\{f_1, \ldots, f_m\} \subseteq \mathbb{F}_q$, i.e. $F(X) = \prod_{i \in [1:m]}(X - f_i)$.
a, b, c	Beaver triple satisfying $a_k \cdot b_k = c_k$, $\forall k \in [1:t]$.
α, β, v	Broadcast values (coordinates lying in $\mathbb{F}_{\text{points}}$).
i	Index of a party in $[1:N]$.
$[\![v]\!]$	Sharing of a value v.
$[\![v]\!]_i$	i^{th} share of a sharing $[\![v]\!]$.
p	False positive probability of the MPC protocol.
Hypercube variant:	
D	Dimension of the hypercube s.t. $N = 2^D$.
i^*	Index of challenge party, which remains hidden.
$(i_1, \ldots, i_D)$	Representation of i on dimension D hypercube with side 2.
(k, j)	Index of a *main party* in $[1:D] \times [1:2]$, where k indexes the hypercube dimension.
aux	Input secret share of leaf party $i = N$, $[\![S \,\|\, Q \,\|\, P \,\|\, a \,\|\, b \,\|\, c]\!]_N$.
$(state_i, \rho_i)$	State and commitment randomness of a leaf party. For $i \neq N$, $state_i$ is a pseudorandom seed, and $state_N = (seed_N \| aux)$
Threshold variant:	
ℓ	Privacy threshold (number of open parties).
I	Set of open parties ($I \subseteq [1:N]$, $\|I\| = \ell$).

1. One only needs to send x_A to reveal the solution, which can then be fully recovered using $x_B = y - Hx_A$. At the MPC level, this implies that we only need to share x_A as input of the protocol.

2. From a sharing of x_A, the parties can locally compute a sharing of x_B by linearity of the above relation. The recovered sharing of $x = (x_A|x_B)$ then satisfies the SD relation $y = Hx$ by definition.

Polynomial constraints. Let $f_1, \ldots, f_q$ denote the elements of $\mathbb{F}_q$. The SD-in-the-Head MPC protocol is based on three (witness-dependent) polynomials, S, Q, and P, and one public polynomial F, for which checking the correctness of the SD solution amounts to verifying the relation:

$$S \cdot Q = P \cdot F. \tag{2}$$

These four polynomials are defined as follows:

- The polynomial $S \in \mathbb{F}_q[X]$ is obtained by Lagrange interpolation of the coordinates of x, such that $S(f_i) = x_i$ for $i \in [1:m]$. This polynomial is of degree $\deg(S) \leq m - 1$.
- The polynomial $Q \in \mathbb{F}_q[X]$ is defined as $Q(X) = \prod_{i \in E}(X - f_i)$, where E is a subset of $[1:m]$ of order $|E| = w$, such that the non-zero coordinates of x are contained in E. This polynomial is of degree $\deg(Q) = w$.
- The polynomial $F \in \mathbb{F}_q[X]$ is the "vanishing polynomial" of the set $\{f_1, \ldots, f_m\}$ which is defined as $F(X) = \prod_{i \in [1:m]}(X - f_i)$. This polynomial is of degree $\deg(F) = m$.
- The polynomial $P \in \mathbb{F}_q[X]$ is defined as $P = S \cdot Q/F$ (by definition F divides $S \cdot Q$). This polynomial is of degree $\deg(P) \leq w - 1$.

The left-hand side of Equation 2 is designed so that $S \cdot Q(f_i) = 0$ for all $f_i \in [1:m]$. This is because $S(f_i)$ is zero for every $x_i = 0$ (by construction, as S is interpolated over x), and $Q(f_i)$ is zero for every $x_i \neq 0$. On the right-hand side of Equation 2, by construction the public polynomial F is zero over $f_1, f_2, \ldots, f_m$, and the polynomial P is required because F has degree m, whereas $m < \deg(S \cdot Q) \leq m + w - 1$. If the prover can convince the verifier that they know P, Q such that $S \cdot Q = P \cdot F = 0$ at all points $f_i \in [1:m]$, then at each point f_i, either $S(f_i) = x_i = 0$, or $Q(f_i) = 0$. But since Q has degree w, it can be zero in at most w points, therefore S is non-zero in at most w points, meaning that x has weight at most w.

In summary, the soundness of the MPC protocol is based on the fact that for S defined as above, we have:

$$\mathrm{wt}(x) \leq w \quad \Leftrightarrow \quad \exists\, P, Q \text{ with } \deg(P) \leq w - 1 \text{ and } \deg(Q) = w \text{ s.t. Equation 2 holds.}$$

The parties then take as input a sharing of the witness (x_A, Q, P), locally compute sharing of x (by Equation 1) and S (by Lagrange interpolation), and then run an equality test for $S \cdot Q = P \cdot F$.

Equality test. In order to verify that the polynomial relation of Equation 2 holds, the polynomial $S \cdot Q - P \cdot F$ is evaluated at a series of points to check that it evaluates to zero everywhere. By the Schwartz-Zippel lemma, it is unlikely that the relation of Equation 2 holds true at random points if the polynomial relation is not true in general. The probability that the relation is satisfied at random points $\{r_k\}_{k \in [t]}$ without Equation 2 being true is known as the *false positive*

probability of the protocol which we denote p. In order to further reduce p, the points r_i are sampled from a larger field $\mathbb{F}_{\text{points}} \supset \mathbb{F}_q$.

Evaluating shared polynomial in a public point is a linear operation. Therefore, the parties can locally compute the evaluation $S(r_k)$, $Q(r_k)$ and $P \cdot F(r_k)$ for each random point r_k. The protocol should then simply check that the obtained sharings defined valid *multiplication triples*, namely that they satisfy the relation $S(r_k) \cdot Q(r_k) = P \cdot F(r_k)$. This is done by sacrificing random multiplication triples (a.k.a. *Beaver triples*) using the protocol of [BN20].

Wrapping up: the SD-in-the-Head MPC protocol. For some variable v, a sharing of v is denoted with double square brackets as $[\![v]\!] = ([\![v]\!]_1, \ldots, [\![v]\!]_N)$ where $[\![v]\!]_i$ denotes the share distributed to (or computed by) the i^{th} party. As input to the SD-in-the-Head protocol, the parties receive sharings $[\![x_A]\!]$, $[\![P]\!]$, $[\![Q]\!]$, as well as sharings $[\![a]\!]$, $[\![b]\!]$, $[\![c]\!]$ corresponding to t Beaver triples $a, b, c \in \mathbb{F}_{\text{points}}^t$ such that $a_k \cdot b_k = c_k$ for every $k \in [1 : t]$. The SD-in-the-Head protocol assumes a *broadcast channel*: the parties can broadcast their shares of a sharing $[\![v]\!]$ and then publicly recompute the corresponding value v. It further assumes an oracle sampling random values which are publicly distributed to all the parties (see Step 1 hereafter); this corresponds to a random challenge from the verifier once the MPC protocol is compiled into a zero-knowledge proof. The SD-in-the-Head protocol runs as follows:

1. Sample $r, \varepsilon \in \mathbb{F}_{\text{points}}^t$ uniformly at random.
2. Parties locally set $[\![x_B]\!] = y - H'[\![x_A]\!]$.
3. Parties locally compute $[\![S]\!]$ via Lagrange interpolation of $[\![x]\!] = ([\![x_A]\!] \,|\, [\![x_B]\!])$.
4. Parties locally evaluate $[\![S(r_k)]\!]$, $[\![Q(r_k)]\!]$ and $[\![F \cdot P(r_k)]\!]$.
5. For all $j \in [t]$, parties verify $([\![S(r_k)]\!], [\![Q(r_k)]\!], [\![P \cdot F(r_k)]\!])$ by sacrificing $([\![a_k]\!], [\![b_k]\!], [\![c_k]\!])$:
   a) Parties locally compute $[\![\alpha_k]\!] = \varepsilon_k \cdot [\![Q(r_k)]\!] + [\![a_k]\!]$ and set $[\![\beta_k]\!] = [\![S(r_k)]\!] + [\![b_k]\!]$.
   b) Parties broadcast $[\![\alpha_k]\!]$ and $[\![\beta_k]\!]$ to publicly recompute α_k and β_k.
   c) Parties locally compute $[\![v_k]\!] = \varepsilon_k \cdot [\![F \cdot P(r_k)]\!] - [\![c_k]\!] + \alpha_k \cdot [\![b_k]\!] + \beta_k \cdot [\![a_k]\!] - \alpha_k \cdot \beta_k$.
   d) Parties broadcast $[\![v_k]\!]$ to publicly recompute v_k.
   e) Parties output Accept if $v_k = 0$ and Reject otherwise.

In its original description [FJR22], the SD-in-the-Head MPC protocol samples r uniformly at random among the vectors of $\mathbb{F}_{\text{points}}^t$ *without duplicate coordinates*. Here we remove this constraint on r to make the scheme simpler and less prone to implementation errors. While this tweak slightly increases the false positive probability p (*i.e.* the probability that the protocol outputs Accept for an invalid input witness), this increase is small enough and does not impact the security category of the scheme for the selected parameters (see Section 4). The false positive probability of this tweaked protocol (taking r uniformly at random from $\mathbb{F}_{\text{points}}^t$) is given by the following theorem.

Theorem 2.1. *Let us denote x_A the plain value of the input sharings $[\![x_A]\!]$. If x_A corresponds to a solution of the syndrome decoding instance defined by (H', y) and if the other input sharings are genuinely computed, the SD-in-the-Head protocol always outputs* Accept. *If x_A does not*

correspond to such a solution, the SD-in-the-Head protocol outputs ACCEPT *with probability at most*

$$p := \sum_{i=0}^{t} \binom{t}{i} \left(\frac{m+w-1}{|\mathbb{F}_{points}|}\right)^{i} \left(1 - \frac{m+w-1}{|\mathbb{F}_{points}|}\right)^{t-i} \left(\frac{1}{|\mathbb{F}_{points}|}\right)^{t-i},$$

over the randomness of r and ε.

Moreover, this protocol is $(N-1)$-*private* which means that one can open the internal views of $N-1$ parties without revealing any information about the witness. This ensures the zero-knowledge property of the protocol after application of the MPC-in-the-Head transformation.

2.1.3 The SD-in-the-Head signature scheme

To obtain the SD-in-the-Head signature scheme, two successive transformations are applied to the SD-in-the-Head MPC protocol described above:

1. The MPC-in-the-Head paradigm turns the SD-in-the-Head MPC protocol into a zero-knowledge proof of knowledge (ZK-POK). The obtained ZK-POK, runs as follows:
   - the prover generates the input sharings and commit to the parties' shares,
   - the verifier challenges the prover with the randomness r, ε of the MPC protocol,
   - the prover runs the MPC protocol (*in their head*) and sends the broadcast values to the verifier,
   - the verifier challenges the prover to open all the parties in $I \subsetneq [1:N]$,
   - the prover reveals the input shares of all the parties in I,
   - the verifier checks the consistency of the MPC computation for the revealed parties.
2. The Fiat-Shamir heuristic turns the latter ZK-POK into the SD-in-the-Head signature scheme. The principle is to replace the verifier challenge of the ZK-POK by the outputs of hash functions taking previous prover's communication as inputs.

The high-level architecture of the SD-in-the-Head signature scheme is depicted in Figure 1. We do not describe the intermediate ZK-POK in details here (the interested reader is referred to the original paper [FJR22]). We stress that in the hypercube variant, like in most recent MPCitH schemes, the set I of opened parties is a random subset of cardinality $N-1$ (or equivalently $I = [1:N] \setminus \{i^*\}$ for a random i^*). On the other hand, in the threshold variant, the set I is a random subset of $[1:N]$ of cardinality ℓ where ℓ is a small constant (see Section 2.3 for details).

Introduction of the message. Although this does not appear in Figure 1, one needs to introduce the message in the hash computation to obtain a message-binding signature. We choose to introduce the message in the second hash only. This enables message-independent pre-computation of the Steps 1-to-4, which are the computationally-greedy steps of the signing algorithm.

Introduction of a salt. For security reasons, we further introduce a salt of 2λ bits. The salt is passed as argument of the hash commitments and extends the seed in some pseudo-randomness generation, in both cases to avoid collision issues. The salt is added to the signature to allow its the verification.

Signature:

1. Generate random sharing $[\![x_A]\!]$, $[\![P]\!]$, $[\![Q]\!]$, $[\![a]\!]$, $[\![b]\!]$, $[\![c]\!]$
2. Commit the parties' shares:

$$[\![x_A]\!]_i, [\![P]\!]_i, [\![Q]\!]_i, [\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i \xrightarrow{\text{Commit}} \mathsf{com}_i$$

3. Derive the first challenge (randomness of MPC protocol):

$$\mathsf{com}_1, \ldots, \mathsf{com}_N \xrightarrow{\text{Hash}} h_1 \to r, \varepsilon$$

4. Simulate the MPC protocol:

$$[\![x_A]\!], [\![P]\!], [\![Q]\!], [\![a]\!], [\![b]\!], [\![c]\!], r, \varepsilon \xrightarrow{\text{MPC}} [\![\alpha]\!], [\![\beta]\!], [\![v]\!]$$

5. Derive the second challenge (index of non-opened party):

$$h_1, [\![\alpha]\!], [\![\beta]\!], [\![v]\!] \xrightarrow{\text{Hash}} h_2 \to I$$

6. Build the signature from

$$h_1,\ h_2,\ \left\{[\![x_A]\!]_i, [\![P]\!]_i, [\![Q]\!]_i, [\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i\right\}_{i\in I},\ \{\mathsf{com}_i, [\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i\}_{i\notin I}$$

Verification:

1. Recompute the commitments, for parties $i \in I$ (with I obtained from h_2):

$$[\![x_A]\!]_i, [\![P]\!]_i, [\![Q]\!]_i, [\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i \xrightarrow{\text{Commit}} \mathsf{com}_i$$

2. Recompute the first challenge (randomness of MPC protocol):

$$\mathsf{com}_1, \ldots, \mathsf{com}_N \xrightarrow{\text{Hash}} h_1 \to r, \varepsilon$$

3. Simulate the MPC protocol, for parties $i \in I$:

$$[\![x_A]\!]_i, [\![P]\!]_i, [\![Q]\!]_i, [\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i, r, \varepsilon \xrightarrow{\text{MPC}} [\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i$$

4. Recompute the second challenge (index of non-opened party):

$$h_1, [\![\alpha]\!], [\![\beta]\!], [\![v]\!] \xrightarrow{\text{Hash}} h_2$$

5. Check that recomputed h_1, h_2 match the signature.

Figure 1: Architecture of the SD-in-the-Head signature scheme.

Omitting one hash in the signature. Although the description of Figure 1 includes both hashes h_1 and h_2 to the signature, it is actually possible to omit one of them, either h_1 or h_2, but at least one of them must be included. Omitting h_1 for instance, the verification algorithm would simply used the recomputed h_1 to derive the MPC challenge and to compute h_2. A correct verification of h_2 then ensure the validity of the recomputed h_1 by the collision resistance of the hash function. On the other hand, one can omit h_2 and recompute it from h_1 in order to derive the view-opening challenge which is necessary to interpret the shares in the signature. Since h_2 deterministically depends on other elements in the signature, replacing its verification by its recomputation does not change the soundness of the scheme.

Our two variants of the SD-in-the-Head scheme makes two different choices regarding this matter: the hypercube variant omit h_1 while the threshold variant omit h_2.

Parallel repetition. As shown in [FJR22], the SD-in-the-Head ZK-POK has soundness error $\varepsilon = \frac{1}{N} + p\left(1 - \frac{1}{N}\right)$. In order to scale this to $2^{-\lambda}$ for a target security level λ, we use parallel repetition. This means that the ZK-POK is repeated $\tau := \lceil \log_2(\varepsilon)\lambda \rceil$ times in parallel to reach a global soundness error of $\varepsilon^\tau \leq 2^{-\lambda}$. (NB: This does not translate to an ε^τ security against forgery attacks in the non-interactive setting where we need to take a greater τ – see Section 7.2.) While applying the Fiat-Shamir heuristic, the N commitments of the τ executions are hashed to derive h_1, which then generates the τ independent MPC random samples r, ε, and the broadcast messages of the τ executions are hashed to derive h_2, which then generates τ independent sets I for the non-opened party of each execution.

In practice, this means that all the elements appearing in Figure 1 (all shares, commitments, MPC randomness), except h_1 and h_2, are τ-dimensional vectors of elements of the original ZK-POK.

Splitting syndrome decoding. Some instances of the SD-in-the-Head scheme rely on a variant of the syndrome decoding problem, which is called the *d-split syndrome decoding problem*. In the latter problem, the solution x is split into d blocks $x_1, \ldots, x_d \in \mathbb{F}_q^{m/d}$, each satisfying a Hamming weight constraint:

$$x = (x_1 \mid \ldots \mid x_d) \quad \text{s.t.} \quad \mathrm{wt}(x_j) = \frac{w}{d} \ \ \forall j \in [1:d] \ .$$

When using this variant, the MPC protocol is run d times in parallel to prove the above weight constraint on each block of x. This means that the polynomial P and Q in the witness are replaced by vectors of polynomials $\boldsymbol{P} = (P[1], \ldots, P[d])$ and $\boldsymbol{Q} = (Q[1], \ldots, Q[d])$ such that $\deg(P[j]) \leq w/d$ and $Q[j]$ is unary of degree w/d for every $j \in [1:d]$. The sample challenges r, ε are also d-vectorized (with one slot for each of the d protocol executions) as well as the broadcast sharings $[\![\alpha]\!], [\![\beta]\!], [\![v]\!]$.

We note that using the d-split variant implies a loss of security compared to the standard SD instance with same parameters (q, m, k, w). However, we can compensate this loss by increasing the security of the standard SD instance by a few bits (see Section 6 for details).

Avoiding interpolations. The SD-in-the-Head MPC protocol needs to compute a Lagrange interpolation to build the polynomial S from the secret x. We can avoid this interpolation by tweaking the definition of y in the public key. Instead of defining $y := Hx$, we can define:

$$y := HVx$$

where V is the matrix satisfying:

$$S = \text{Lagrange Interpolation}(x) \quad \Leftrightarrow \quad s = Vx \tag{3}$$

for s the vector of coefficients of S. Let us remark that, for a uniformly random linear code $\mathcal{C}_H$ represented by the matrix H as parity-check matrix, the linear code $\mathcal{C}_{HV}$ represented by HV is also uniformly random. This is because V is an invertible $m \times m$ matrix. By denoting $s = (s_A \mid s_B) := Vx$, we can give $[\![s_A]\!]$ instead of $[\![x_A]\!]$ as input to the MPC protocol. Thus the parties can deduce $[\![s]\!]$ from $[\![s_A]\!]$ and from $[\![s_B]\!] := y - H'[\![s_A]\!]$. Now, the sharing of the polynomial $[\![S]\!]$ is directly obtained from the sharing $[\![s]\!]$ of its coefficients, and so we do not need to perform Lagrange interpolations in the MPC protocol anymore.

Two variants of the SD-in-the-Head signature scheme. We propose two variants of the SD-in-the-Head signature scheme relying on two follow-up improvements of the scheme [AMGH+22; FR22]. These two variants differ in the underlying secret sharing scheme, the format of the commitments and the way the MPC computation is performed. We outline the specificities of the two variants in the following sections.

2.2 Principle of hypercube variant

In the MPCitH setup used in the original SD-in-the-Head scheme [FJR22], the commitment boils down to PRG expansion from seeds for the first $N-1$ input shares, subtraction to the plain witness for the last share, and commitments. Using this initial commitment, the prover then simulates the MPC computation on each of these N parties to be able to produce the relevant communications. Once the $N-1$ commitments are opened, the verifier also needs to replay those $N-1$ computations for the consistency check. Instead of following this approach, [AMGH+22] proposes a geometric method, and arranges the N shares on a D-dimensional hypercube such that $N = 2^D$. Using the *same initial commitment*, the prover and the verifier only need to simulate the computation of $\log_2(N) + 1$ parties, for the exact same soundness error as the original protocol. This hence lowers the amount of expensive MPC computations and the cost of increasing the number of cheap hash calculations.

Secret Sharing. The hypercube variant uses additive secret sharing, $[\![s]\!] = ([\![s]\!]_1, \ldots, [\![s]\!]_N)$ where the plain value is $s = \sum_{i=1}^{N} [\![s]\!]_i$. Input shares of a given plain value s are usually constructed by drawing $[\![s]\!]_1, \ldots, [\![s]\!]_{N-1}$ uniformly at random or deriving them from a seed, and set $[\![s]\!]_N$ as the remainder, often called the *auxiliary* share. A share of a public constant c implicitly corresponds to the trivial share $[\![c]\!] = (0, \ldots, 0, c)$. Broadcasting a share $[\![s]\!]$ means that each party broadcasts its local share $[\![s]\!]_i$. This operation reveals in particular the plain value s, which becomes public. Finally, given any public description of a linear function $\varphi(x_1, \ldots, x_p)$ with p variables, and p secret-shares $[\![s_1]\!], \ldots, [\![s_p]\!]$, parties can compute a secret-share of $y = \varphi(s_1, \ldots, s_p)$ by locally setting $[\![y]\!]_i = \varphi([\![s_1]\!]_i, \ldots, [\![s_p]\!]_i)$. The description of the function φ may depend on previously revealed plaintexts, even non-linearly.

Commitments. The hypercube variant uses a TreePRG construction as suggested in [KKW18], which derives 2^D leaf seeds from a root master seed. The leaf seeds are expanded into the input leaf shares. The last leaf share, which can not be solely derived from a seed, uses an auxiliary, as explained in the previous paragraph. This is illustrated in Figure 2.

The state of a leaf share is the minimal information needed to reconstruct it: for $i \in [1, 2^D - 1]$, it is the leaf seed, and for $i = 2^D$, the last state is the tuple $(\mathsf{leaf\_seed}, \mathsf{aux})$. Each state is

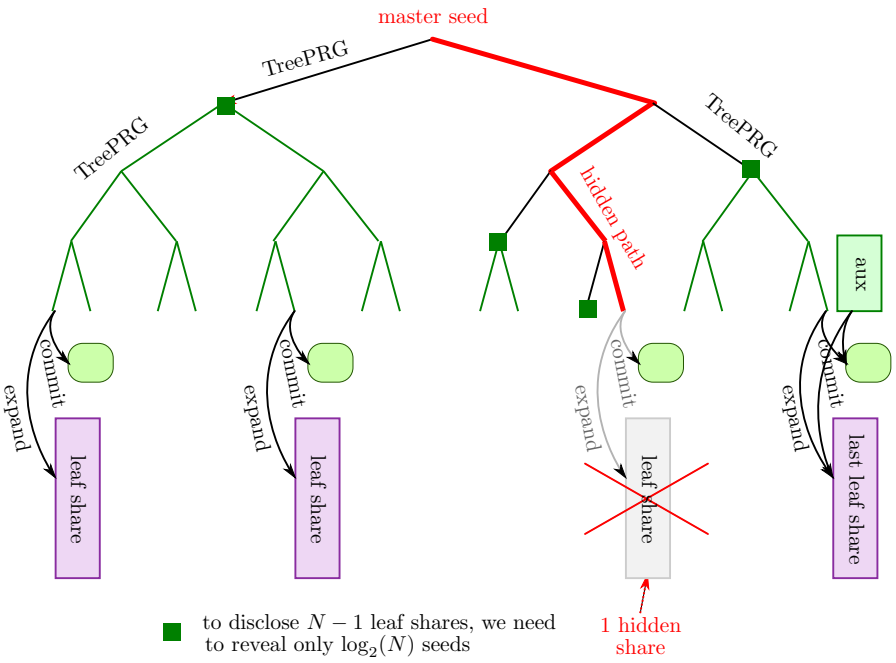

Figure 2: TreePRG construction.

committed as a leaf commitment, and the 2^D leaf commitments are hashed together to form h_1. This commitment ensures that to open all the leaf shares except one, and all the leaf commitments, one only needs to publish a sibling path of D seeds and 1 leaf commitment.

MPC simulation. An MPCitH computation based on an additive secret sharing relies on shares of the MPC parties adding up to the witness for which we want a zero-knowledge proof. Additive secret sharing correctness does not depend on how these shares are sampled: they can be uniform samples, additions of uniform samples, etc. As long as the shares add up to the witness, the result of the computation is correct. The hypercube approach proposes a way to re-express one instance of the protocol over $N = 2^D$ parties into D instances of 2 parties. For each of instance, the 2 parties add up to the original witness, thus each of these instances will be correct no matter the additive scheme or the functionality computed.

Let us explain the principle on a 2-dimensional toy example. Suppose we consider a traditional 4-party protocol with shares s_1, s_2, s_3, and aux that sum up to the witness. If we distribute them in a 2-dimensional hypercube of side 2 (i.e. a two-by-two square) we obtain what is shown in Figure 3.

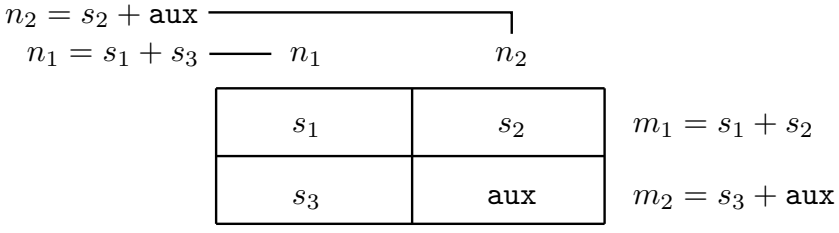

Figure 3: A simple 2-dimensional example of the hypercube construction.

Per construction we have $s_1 + s_2 + s_3 + \mathsf{aux} = \mathsf{witness}$. The hypercube approach leads to an MPC execution for two parties holding $m_1 = (s_1 + s_2)$ and $m_2 = (s_3 + \mathsf{aux})$ on one side, and an MPC execution for two parties holding $n_1 = (s_1 + s_3)$ and $n_2 = (s_2 + \mathsf{aux})$ on the other side. By associativity and commutativity, in both cases the sum of the shares is equal to the witness, and both MPCitH executions will lead to a correct result. Just as the traditional 4-party protocol would have. The non-trivial part is to prove that by doing this, the soundness error in the presence of a dishonest prover is the same in the hypercube splitting as it is in the original protocol; we refer the reader to [AMGH+22] for the explanations.

From a performance standpoint, using a 2-dimensional hypercube of side 2 provides no advantage. In the traditional approach one would: generate 4 states, commit to 4 states, and compute with 4 MPC parties. In the hypercube approach one also generates a state, commit, and do an MPC computation $2 + 2 = 4$ times. But when the dimension D increases, we see the advantage appearing. For instance, if an MPCitH protocol does a 256 party protocol, as in the original SD-in-the-Head, it requires 256 state generations and commitments. By using an 8-dimensional hypercube of side 2, one will then only perform 16 MPC computations, instead of 256 originally. The exact same information is revealed: one opens 255 initial states and give the communications that would have resulted from the unopened state, so one gets the same proof size, the MPC cost is reduced by a factor of more than 10.

An additional benefit of Hypercube-MPCitH is that it can avoid, for most of the D executions, running the MPC protocol for all the parties. Indeed, each of the D executions corresponds to a given aggregation of the same hypercube shares. Thus each secret shared variable that occurs throughout a run of the MPC algorithm corresponds to the same plain value when the shares are summed up. Therefore the prover only needs to compute these plain values once, for instance by evaluating the first 2 parties, and then, for the remaining $D-1$ runs, the last share is simply deduced by the difference to the plaintext value. Consequently, only $2-1$ parties need to be evaluated instead of 2 per run, which makes $2 + (2-1)(D-1) = D+1$ in total. For a 256-party protocol, the prover needs only to simulate the computation of 9 parties instead of 16 in the above paragraph, and 256 in the original protocol.

The $N = 2^D$ original parties (also called *leaf-parties*) are indexed on the D dimensions by coordinates $(i_1, \ldots, i_D) \in [1, 2]^D$. For each dimension $k \in [1, D]$, we have one MPC run between 2 *main parties*, and by convention, for each index $j \in [1, 2]$, the main party of index (k, j) regroups the contributions of the leaf-party shares whose k-th coordinate is j. Hence, for each axis $k \in [1, D]$, the main parties $(k, 1), \ldots, (k, 2)$ form a partition of the leaf parties. With this partitioning, whenever we disclose the values of $2^D - 1$ leaf shares and keep a single one hidden, it automatically discloses the value of exactly $2 - 1$ out of 2 main-parties shares on each of the D axes.

Signature format. Because of the specificities of the threshold variant discussed above, the format of the signature depicted in Figure 1 is tweaked as follows:

- the input of the second hash h_2 is made of the plain broadcast values as well as the broadcast shares of D main parties ($D + 1$ broadcast values in total) instead of the 2^D broadcast shares of the leaf parties,
- for each repetition, the set I of open parties is defined as $I = [1 : N] \setminus \{i^*\}$,
- thanks to the TreePRG, the revealed input shares $\left\{[\![x_A]\!]_i, [\![P]\!]_i, [\![Q]\!]_i, [\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i\right\}_{i \in I}$ simply consist of the sibling path in the seed tree as well as the auxiliary share (which is omitted if $i^* = N$).

A per-signature salt is also included (see Section 3 for details). Wrapping up, an hypercube SD-in-the-Head signature has the following format:

$$\sigma = \big(\mathsf{salt} \mid h_2 \mid (\mathsf{view}[e], \mathsf{broad\_plain}[e], \mathsf{com}[e]\,[i^*[e]]\,)_{e\in[1:\tau]}\big),$$

where i indexes the 2^D leaves of the hypercube, seed_i (contained within $\mathsf{view}[e]$) enables the verifier to derive $\{[\![x_A]\!]_i, [\![P]\!]_i, [\![Q]\!]_i, [\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i\}$, except for the case $i = 2^D$ when they are sent from directly via $\mathsf{aux} = \{[\![x_A]\!]_{2^D}, [\![P]\!]_{2^D}, [\![Q]\!]_{2^D}, [\![c]\!]_{2^D}\}$. As a caveat, in the unlikely (with $1/2^D$ chance) case $i^* = 2^D$, aux is not sent.

Trade-off offered by the hypercube variant. Compared to the original SD-in-the-Head scheme, the main benefit of the hypercube variant is to require much fewer costly MPC computations, leading to significant speed-up for equivalent parameters (and thus the same signature sizes) or to comparable running times with much smaller signature (by increasing the number of parties N). The computational complexity of the hypercube verification algorithm is essentially the same as the signature algorithm. Also, it is worth noting that with parameters beyond $N = 2^{16}$, the trade-off of increasing computations to reduce the signature size plateaus (as shown in Figure 4 in [AMGH+23]).

2.3 Principle of threshold variant

The threshold variant relies on a low-threshold linear secret sharing scheme [FR22]. Besides changing the definition of the underlying sharing scheme, the format of the commitments and the way the MPC computation is performed are further impacted.

Sharing scheme. Given a privacy threshold ℓ, a *linear secret sharing scheme* (LSSS) shares a secret $s \in \mathbb{F}_q$ into a sharing $[\![s]\!] := ([\![s]\!]_1, \ldots, [\![s]\!]_N)$ such that s can be reconstructed from any $\ell+1$ shares while no information is revealed about s from the knowledge of ℓ shares. The linear feature of the sharing scheme further implies that linear operations can be computed locally by the parties which make any LSSS compatible with the SD-in-the-Head MPC protocol described in Section 2.1.

For the threshold variant of the SD-in-the-Head scheme, we use Shamir's secret sharing (SSS) [Sha79]. A sharing $[\![s]\!] = ([\![s]\!]_1, \ldots, [\![s]\!]_N)$ of a value $s \in \mathbb{F}_q$ is generated by sampling ℓ random elements $(s_1, \ldots, s_\ell) \leftarrow \mathbb{F}_q^\ell$, letting $s_0 := s$, and defining the shares as follows:

$$\begin{cases} [\![s]\!]_1 = \sum_{i=0}^{\ell} s_i \cdot f_1^i \\ \quad\vdots \\ [\![s]\!]_N = \sum_{i=0}^{\ell} s_i \cdot f_N^i \end{cases}$$

where $f_1, \ldots, f_N$ denotes N fixed non-zero elements of the field $\mathbb{F}_q$. For an $\mathbb{F}_q$-tuple $s \in \mathbb{F}_q^{|s|}$, the sharing $[\![s]\!]$ is defined by applying the above process to each coordinate of s.

The Shamir's secret sharing has a polynomial interpretation: let P_s be the polynomial with coefficients $s_0, \ldots, s_\ell$ (with $s_0 := s$ the secret and $s_1, \ldots, s_\ell$ random coefficients), then the i^{th} share $[\![s]\!]_i$ is defined as the evaluation of P_s in the point f_i, that is $[\![s]\!]_i := P_s(f_i)$. Equivalently, the sharing $[\![s]\!] = ([\![s]\!]_1, \ldots, [\![s]\!]_N)$ is a Reed-Solomon (RS) codeword for the message $(s_0, \ldots, s_\ell)$ where the underlying RS code is of length N and dimension $\ell + 1$. This notably implies that (for a small ℓ), the sharing $[\![s]\!]$ is highly redundant. In particular, the entire sharing can be derived from any set of $\ell + 1$ shares.

When applying the MPCitH paradigm to the SD-in-the-Head MPC protocol with Shamir's secret sharing, the prover only reveals ℓ party views instead of $N-1$. This is because the latter sharing is ℓ-private and not $(N-1)$-private as the additive sharing. For the SD-in-the-Head signature scheme overview in Figure 1, this notably means that the set $I \subsetneq [1:N]$ of opened parties derived from the second hash h_2 is a random subset of cardinality ℓ.

Commitments. Since the shares in a (low-threshold) Shamir's secret sharing are highly redundant, it is not possible to expand them from random seeds. This means that the threshold variant cannot benefit of seed trees as proposed in [KKW18] and as used in the original SD-in-the-Head scheme and in the hypercube variant. Since only a low number ℓ of committed views are revealed to the verifier, the threshold variant uses a Merkle tree as commitment scheme. For the signature scheme (see Figure 1), this means that the commitments of the shares $\mathsf{com}_1, \ldots, \mathsf{com}_N$ are aggregated into a global Merkle commitment:

$$\mathsf{com} := \mathrm{MerkleTree}(\mathsf{com}_1, \ldots, \mathsf{com}_N) \ .$$

This global commitment is then used as the input of the first hash h_1. As a consequence of this tweak, the commitments com_i of the non-opened parties $i \notin I$ do not need to be included in the signature. On the other hand, the commitments of the opened parties $i \in I$ must come with their authentication paths to the global Merkle commitment com. In practice, we include the authentication path to each com_i such that $i \in I$ in the signature while excluding the Merkle root com. The latter is recomputed from the paths by the verification algorithm and then checked by recomputing the first hash h_1.

MPC simulation. The goal of the MPC simulation in the signature scheme (see Figure 1) is to derive the broadcast messages $[\![\alpha]\!]$, $[\![\beta]\!]$, $[\![v]\!]$. By definition of the SD-in-the-Head MPC protocol, there exist linear functions

$$\begin{aligned} \varphi^1_{r,\varepsilon} &: (x_A, P, Q, a, b, c) \mapsto (\alpha, \beta) \\ \varphi^2_{r,\varepsilon,\alpha,\beta} &: (x_A, P, Q, a, b, c) \mapsto v \end{aligned}$$

such that each party locally evaluates φ^1 and φ^2 to the input shares $[\![x_A]\!]_i, [\![P]\!]_i, [\![Q]\!]_i, [\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i$ to get the broadcast shares $[\![\alpha]\!]_i$, $[\![\beta]\!]_i$, $[\![v]\!]_i$. The obtained sharings $[\![\alpha]\!]$, $[\![\beta]\!]$, $[\![v]\!]$ are Shamir's secret sharing of the values α, β, v (outputs of φ^1 and φ^2 on the plain inputs). As a consequence, we only need to commit $\ell+1$ shares of the broadcast sharings $[\![\alpha]\!], [\![\beta]\!], [\![v]\!]$ to commit them entirely, and hence we only need to perform the MPC computation for a subset of $\ell+1$ shares.

Alternatively, we can directly run the MPC computation (*i.e.* evaluate the functions φ^1 and φ^2) *on the $\ell+1$ coefficients of the polynomials involved in the Shamir's secret sharing.* We denote $\mathsf{input\_plain}$ as the $\mathbb{F}_q$-tuple containing the witness and the Beaver triples:

$$\mathsf{input\_plain} := (x_A, P, Q, a, b, c)$$

namely the plain input of the protocol. The initial sharing of this plain input is done by sampling ℓ vectors $\mathsf{input\_coef}_1, \ldots, \mathsf{input\_coef}_\ell$ uniformly at random from $\mathbb{F}_q^{|\mathsf{input\_plain}|}$, and defining the i^{th} share of $\mathsf{input\_plain}$ as

$$[\![\mathsf{input\_plain}]\!]_i := \mathsf{input\_plain} + \sum_{j=1}^{\ell} f_i^j \cdot \mathsf{input\_coef}_j \ .$$

The MPC computation is then run on $\mathsf{input\_plain}$, $\mathsf{input\_coef}_1$, ..., $\mathsf{input\_coef}_\ell$ to get the (vectorized) coefficients of the polynomials corresponding to the broadcast sharings:

$$\begin{aligned}
\mathsf{input_plain} &\xrightarrow{(\varphi^1,\varphi^2)} \mathsf{broad_plain},\\
\mathsf{input_coef}_1 &\xrightarrow{(\varphi^1,\varphi^2)} \mathsf{broad_coef}_1,\\
&\vdots\\
\mathsf{input_coef}_\ell &\xrightarrow{(\varphi^1,\varphi^2)} \mathsf{broad_coef}_\ell.
\end{aligned}$$

From those $\ell+1$ vectorized coefficients, one can then recover the broadcast shares associated to any party i by:

$$([\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i) = \mathsf{broad\_plain} + \sum_{j=1}^{\ell} f_i^j \cdot \mathsf{broad\_coef}_j \ . \tag{4}$$

We use this approach to perform the MPC computation in the threshold variant. Thanks to an additional tweak explained hereafter, we only need to perform the MPC computation for the plain values $\mathsf{input\_plain} \mapsto \mathsf{broad\_plain}$ *once* for the τ executions, whereas the ℓ other vectorized coefficients $\mathsf{broad\_coef}_1$, ..., $\mathsf{broad\_coef}_\ell$ are evaluated for each execution (as they rely on different randomness).

Tweak in the equality test. As shown in [FR22], the soundness error obtained while applying the threshold approach is slightly degraded compared to the standard case with additive sharing. More precisely, the false positive probability p has a greater impact on the soundness error, which is further amplified while turning to the non-interactive setting. To compensate the security loss, one hence needs to decrease p by increasing the number of random points in the challenge r derived from h_1.

While getting closer to a negligible value for p, we can use a trick proposed in the Limbo proof system [dOT21] and which consists in using the same MPC challenge across the τ protocol executions. Using this tweak in our context implies that we sample a single pair (ε, r) from h_1 which is used in the τ parallel protocol executions. We can then also use the same Beaver triples across the τ executions. We thus get the same plain values α, β, v (a.k.a. $\mathsf{broad\_plain}$) for the broadcast messages in the τ executions. As a result, we only need to perform the computation of $\mathsf{broad\_plain}$ a single time (instead of τ times) and the obtained value is included a single time in the signature (instead of τ times).[1]

Avoiding interpolations. As depicted in Figure 1, the broadcast shares are used to derive the second hash h_2 in the signature. In the threshold variant, the full set of broadcast shares can be deduced from any set of $\ell+1$ broadcast shares because of the sharing redundancy (*i.e.* the full sharing is a Reed-Solomon codeword). One can then hash any predetermined set of shares $\{[\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i\}_{i\in E}$, with $|E| = \ell+1$, to derive h_2. The signature then includes the open shares $\left\{[\![x_A]\!]_i, [\![P]\!]_i, [\![Q]\!]_i, [\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i\right\}_{i\in I}$ from which the verifier can recompute $\{[\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i\}_{i\in I}$ which, together with a single additional share $([\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i)$ for $i \notin I$ (or the plain value of the broadcast), allows the verifier to recompute $\{[\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i\}_{i\in E}$ (and hence verify h_2).

[1] We note that this tweak is not interesting in the standard case of additive sharing or for the hypercube variant since having the same Beaver triples across the τ executions would require the introduction of auxiliary values for a and b in at least $\tau - 1$ out of τ executions.

However such a process implies that the verifier must interpolate the polynomial coefficients $\mathsf{broad\_plain}, \mathsf{broad\_coef}_1, \ldots, \mathsf{broad\_coef}_\ell$ from $\{[\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i\}_{i\in I}$ (plus the additional share or plain value) to then recover the evaluations on the set E. We use a different approach to avoid such inefficient interpolations.

Let us first remark that since the broadcast shares are fully defined from their polynomial coefficients $\mathsf{broad\_plain}, \mathsf{broad\_coef}_1, \ldots, \mathsf{broad\_coef}_\ell$, the latter can be directly used as input to the second hash h_2 instead of the evaluations for a predetermined set E. This way and given the previous tweak, the signer avoids computing these evaluations. These polynomial coefficients are further included to the signature thereby allowing the verifier to compute h_2 and to evaluate $[\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i$ for every $i \in I$, hence avoiding interpolations.

While one adds the above polynomial coefficients to the signature, one can in return remove the Beaver shares of the open parties $\{[\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i\}_{i\in I}$. Indeed, by definition of the SD-in-the-Head MPC protocol, for every party $i \in [1 : N]$ and given the shares $[\![x_A]\!]_i, [\![P]\!]_i, [\![Q]\!]_i$, there is a one-to-one (linear) relation between $([\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i)$ and $([\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i)$. Therefore, the Beaver shares $\{[\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i\}_{i\in I}$ (which are necessary to verify the consistency of the opened parties) can be derived from the witness shares $\{[\![x_A]\!]_i, [\![P]\!]_i, [\![Q]\!]_i\}_{i\in I}$ and the broadcast shares $\{[\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i\}_{i\in I}$. This process is called "reversed multiparty computation" in Section 3.

Signature format. Because of the specificities of the threshold variant discussed above, the format of the signature depicted in Figure 1 is tweaked as follows:

- the hash h_2 is omitted from the signature (see explanation in Section 2.1.3),
- the authentication paths of the commitments com_i for $i \in I$ are included in the signature,
- the commitments com_i for $i \notin I$ are removed from the signature,
- the shares $\{[\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i\}_{i\notin I}$ are replaced by the plain values α, β, v (common to all the τ executions) and the vectorized coefficients $\mathsf{broad\_coef}_1, \ldots, \mathsf{broad\_coef}_\ell$ of the polynomials defining the sharings $[\![\alpha]\!], [\![\beta]\!], [\![v]\!]$ (ℓ vectorized coefficients for each of the τ executions),
- the shares $\{[\![a]\!]_i, [\![b]\!]_i, [\![c]\!]_i\}_{i\in I}$ are removed from the signature.

To sum-up, the obtained signature has the following format:

$$\Big(\mathsf{salt} \mid h_1 \mid \mathsf{broad\_plain} \mid \mathsf{broad\_share} \mid ((\mathsf{wit\_share}\,[e]\,[i])_{i\in I[e]} \mid \mathsf{auth}\,[e])_{e\in[1:\tau]}\Big)$$

where $\mathsf{broad\_plain}$ contains the serialized plain broadcast values (for all the repetitions), $\mathsf{broad\_share}$ contains the serialized shares $\{[\![\alpha]\!]_i, [\![\beta]\!]_i, [\![v]\!]_i\}_{i\in[1:\ell]}$ (for all the repetitions), $\mathsf{wit\_share}\,[e]\,[i]$ contains the ith share of the witness $\{[\![x_A]\!]_i, [\![P]\!]_i, [\![Q]\!]_i$ for repetition e, *i.e.* $(\mathsf{wit\_share}\,[e]\,[i])_{i\in I[e]}$ are the open witness shares for repetition e, and $\mathsf{auth}\,[e]$ contains the Merkle authentication paths for repetition e.ù

Trade-off offered by the threshold variant. The main advantage of the threshold variant is to reduce the computation performed by the signer and the verifier. On the signer side, only $\ell + 1$ parties must be simulated instead of N in the standard case. The gain is even more significant for the verifier: they only need to perform ℓ party computations and verify Merkle paths for the commitment of these ℓ parties (while the signer still needs to compute and commit the shares of the N parties).

As in the hypercube variant, the computational bottleneck of the threshold variant is the generation and commitment of the input shares, while in comparison the MPC simulation is light. Here, we need to generate and commit N shares for each of the τ protocol executions. To reduce this computation cost, we can use the privacy threshold ℓ. When targeting a specific security level, slightly increasing ℓ will decrease τ. For example, for the 128-bit security level, we need to have at least $\tau = 16$ executions when $\ell = 1$, while this reduces to $\tau = 6$ if we take $\ell = 3$. However, taking a larger ℓ will increase the communication cost, thus the choice of ℓ offers an additional size vs. speed trade-off.

The main limitation of the threshold variant is to induce a larger signature size than in the standard case and hypercube variant. This is due to the use of Merkle trees instead of seed trees, where the former roughly requires twice as much space as the latter. A second limitation of the threshold variant is that the number N of parties is limited by the size of the field $\mathbb{F}_q$, *i.e.* $N \leq q$, which is due to the MDS conjecture [MS10] (see [FR22] for details).

3 Detailed algorithmic description

3.1 Notations

The elements manipulated by the signature and verification algorithms are vectors of field elements. In the following, we denote $\mathbb{F}$ to mean a field which might either be the SD base field $\mathbb{F}_q$ or the extension field $\mathbb{F}_{\text{points}} = \mathbb{F}_{q^\eta}$.

Vectors. For a vector v over $\mathbb{F}$, we denote its length $|v|$, namely $v \in \mathbb{F}^\ell \Leftrightarrow |v| = \ell$. We further denote $v\,[i]$ the ith coordinate of v. For any two vectors v_1 and v_2, we denote $(v_1 \parallel v_2) \in \mathbb{F}^{|v_1|+|v_2|}$ their concatenation. For a vector $v \in \mathbb{F}^{|v|}$, for any $n \in \mathbb{N}$, and for any sequence of positive integers $\ell_1, \ldots, \ell_n$ such that $|v| = \ell_1 + \cdots + \ell_n$, we denote

$$(v_1, \ldots, v_n) \leftarrow \text{Parse}(v, \mathbb{F}^{\ell_1}, \ldots, \mathbb{F}^{\ell_n})$$

the operation which splits v into n vectors of field elements such that

$$v = (v_1 \parallel \ldots \parallel v_n) \quad \text{and} \quad |v_i| = \ell_i \ \forall i \in [1:n] \ .$$

We shall also manipulate two-dimensional vectors of field elements. For instance $v \in (\mathbb{F}^\ell)^d$ is a vector with d coordinates which are vectors of $\mathbb{F}^\ell$. For such a vector, we naturally extend the coordinate notation such that $v\,[i] \in \mathbb{F}^\ell$ is the ith coordinate of v, itself a vector, and $v\,[i]\,[j] \in \mathbb{F}$ is the jth coordinate of $v\,[i]$. We further extend the definition of the Parse function to handle two-dimensional vectors. For some vector $v \in \mathbb{F}^{|v|}$, if we denote

$$(v_1, \ldots, v_n) \leftarrow \text{Parse}(v, (\mathbb{F}^{\ell_1})^{d_1}, \ldots, (\mathbb{F}^{\ell_n})^{d_n})$$

then v_k is the two-dimensional vector from $(\mathbb{F}^{\ell_k})^{d_k}$ satisfying

$$v_k\,[i]\,[j] = v\,[\delta_k + i \cdot \ell_k + j] \quad \forall (i,j) \in [1:d_k] \times [1:\ell_k] \ .$$

where $\delta_1 = 0$ and $\delta_k = \ell_1 d_1 + \cdots + \ell_{k-1} d_{k-1}$ for $k > 1$.

We shall also denote Serialize the inverse of Parse. For any tuple $(v_1, \ldots, v_n)$ of two-dimensional vectors, the Serialize function "flattens" this tuple by returning the vector v defined as:

$$v = \text{Serialize}(v_1, \ldots, v_n) \in \mathbb{F}^{\ell_1 d_1 + \cdots + \ell_n d_n}$$
$$\Leftrightarrow \quad (v_1, \ldots, v_n) = \text{Parse}(v, (\mathbb{F}^{\ell_1})^{d_1}, \ldots, (\mathbb{F}^{\ell_n})^{d_n}) \ .$$

In the algorithmic description below, we sometimes perform linear operations between serialized variables, such as $\mathsf{var1} + \mathsf{var2}$ (or $\mathsf{var1} - \mathsf{var2}$). This is to be interpreted as adding (or subtracting) each coordinate of the $\mathbb{F}$-vector represented by the serialized variable.

Arithmetic operations. In the algorithmic description, we shall use the operator $\cdot$ to denote the product over $\mathbb{F}_q$. We shall further use this operator for the scalar product between a value $u \in \mathbb{F}_q$ and a vector $v = (v_1, \ldots, v_\ell) \in \mathbb{F}_q^\ell$, that is

$$u \cdot v = (u \cdot v\,[1], \ldots, u \cdot v\,[\ell]) \ .$$

An element of $\mathbb{F}_{\text{points}} = \mathbb{F}_{q^\eta}$ is represented as a vector of η elements of $\mathbb{F}_q$. For any $v \in \mathbb{F}_{q^\eta}$, we denote $v \equiv (v_1, \ldots, v_\eta) \in \mathbb{F}_q^\eta$ the relation between v and the corresponding $\mathbb{F}_q$-vector. We shall

use the operator $\otimes$ to denote the product over $\mathbb{F}_{q^\eta}$. For any $u, v \in \mathbb{F}_q^\eta$, with $u \equiv (u_1, \ldots, u_\eta)$ and $v \equiv (v_1, \ldots, v_\eta)$, the product $z = u \otimes v$ is defined as:

$$z \equiv (z_1, \ldots, z_\eta) \quad \text{s.t.} \quad \sum_{i=1}^{\eta} z_i X^{i-1} = \left(\sum_{i=1}^{\eta} u_i X^{i-1}\right)\left(\sum_{i=1}^{\eta} v_i X^{i-1}\right) \bmod f(X) \ , \quad (5)$$

where $f(X)$ is the degree-η irreducible polynomial of $\mathbb{F}_q[X]$ such that $\mathbb{F}_{q^\eta} \equiv \mathbb{F}_q[X]/f(X)$.

Intermediate variables. We use the mathematical notations introduced in Section 2.1 and summarized in Table 1. We use bold characters to stress that a variable is d-vectorized: it is a d-dimensional array for which each coordinate corresponds to one block of the witness in the d-split SD variant (see Section 2.1.3). The different variables are serialized in $\mathbb{F}_q$-strings which are summarized in Table 2.

Table 2: Descriptions of the low-level notations used in our scheme.

Bit strings:		
$\mathsf{seed}_{\mathsf{root}}$	$\{0,1\}^\lambda$	Root seed which is expanded into $\mathsf{seed}_{\mathsf{wit}}$ and seed_H.
$\mathsf{seed}_{\mathsf{wit}}$	$\{0,1\}^\lambda$	Seed for the generation of the witness $(s_A, \boldsymbol{Q'}, \boldsymbol{P})$.
seed_H	$\{0,1\}^\lambda$	Seed for the generation of the parity-check matrix H.
mseed	$\{0,1\}^\lambda$	Master seed for all the pseudo-randomness of the signature.
salt	$\{0,1\}^{2\lambda}$	Salt for the pseudo-randomness and commitments of the signature.
com	$\{0,1\}^{2\lambda}$	Commitments of the parties' input shares.
Indexes:		
e	$1,\dots,\tau$	Index for the current repetition.
i	$1,\dots,N$	Index for the current party.
p	$1,\dots,D$	Index for the current dimension (only for hypercube).
p	$1,\dots,\ell$	Index for the current open party (only for threshold).
j	$1,\dots,t$	Index for the current evaluation point r.
ν	$1,\dots,d$	Index for the current chunk of the d-split SD solution.
Serialized variables:		
wit_plain	$\mathbb{F}_q^{k+2k}$	Serialized plain witness $(s_A, \boldsymbol{Q'}, \boldsymbol{P})$.
input_plain	$\mathbb{F}_q^{k+2w+t(2d+1)\eta}$	Serialized plain input $(s_A, \boldsymbol{Q'}, \boldsymbol{P}, \boldsymbol{a}, \boldsymbol{b}, c)$ of the MPC protocol.
beav_ab_plain	$\mathbb{F}_q^{2dt\eta}$	Serialized plain uniformly-sampled part $(\boldsymbol{a}, \boldsymbol{b})$ of the Beaver triple.
beav_c_plain	$\mathbb{F}_q^{t\eta}$	Serialized plain correlated part c of the Beaver triple.
broad_plain	$\mathbb{F}_q^{2dt\eta}$	Serialized plain broadcast values $\boldsymbol{\alpha}, \boldsymbol{\beta}$.
wit_share	$\mathbb{F}_q^{k+2k}$	Serialized shares $[\![(s_A, \boldsymbol{Q'}, \boldsymbol{P})]\!]_i$.
input_share	$\mathbb{F}_q^{k+2w+t(2d+1)\eta}$	Serialized input shares $[\![(s_A, \boldsymbol{Q'}, \boldsymbol{P}, \boldsymbol{a}, \boldsymbol{b}, c)]\!]_i$.
beav_ab_share	$\mathbb{F}_q^{2dt\eta}$	Serialized shares $[\![(\boldsymbol{a}, \boldsymbol{b})]\!]_i$.
beav_c_share	$\mathbb{F}_q^{t\eta}$	Serialized shares $[\![c]\!]_i$.
broad_share	$\mathbb{F}_q^{(2d+1)t\eta}$	Serialized shares $[\![(\boldsymbol{\alpha}, \boldsymbol{\beta}, v)]\!]_i$ of the broadcast values $\boldsymbol{\alpha}, \boldsymbol{\beta}, v$.
chal	$\mathbb{F}_q^{(1+d)\cdot t\cdot\eta}$	The MPC challenges $(r, \boldsymbol{\varepsilon})$
input_mshare	$\mathbb{F}_q^{k+2w+t(2d+1)\eta}$	Serialized input share $[\![(s_A, \boldsymbol{Q'}, \boldsymbol{P}, \boldsymbol{a}, \boldsymbol{b}, c)]\!]_p$ of a main party (hypercube only).
aux$[e]$	$\mathbb{F}_q^{k+2w+t\eta}$	Auxiliary state of the last party for the repetition e (hypercube only).
input_coef	$\mathbb{F}_q^{k+2w+t(2d+1)\eta}$	Serialized polynomial ceofficient for SSS of $(s_A, \boldsymbol{Q'}, \boldsymbol{P}, \boldsymbol{a}, \boldsymbol{b}, c)$ (threshold only).
broad_coef	$\mathbb{F}_q^{(2d+1)t\eta}$	Serialized polynomial ceofficient for SSS of $\boldsymbol{\alpha}, \boldsymbol{\beta}, v$ (threshold only).
Notations for hypercube approach:		
rseed$[e]$	λ-bit seeds	Seeds which are the roots of the seed tree.
seed$[e]$ $[i]$	λ-bit seeds	Parties' seeds (leaves of the seed trees).
acc	$\mathbb{F}_q^{k+2w+t(2d+1)\eta}$	Accumulator which sums all the parties' shares.
state$[e]$ $[i]$	Bit string	Parties' initial state.
path$[e]$	Bit string	Seed path giving the open parties' seeds.
view$[e]$	Bit string	Components enabling to derive the parties' initial states.
Notations for threshold approach:		
com′	2λ-bit digests	Digests of parties' input shares (leaves of Merkle trees).
auth$[e]$	Bit string	Authentication paths of the open parties' views.
seed	λ-bit seed	Master seed from which all signature randomness is derived.
Misc:		
with_offset	Boolean	Current party is involved in constant addition.

3.2 Subroutines

In this subsection, we describe different subroutines which are involved in our key generation, signature, and verification algorithms. These sub-routines are related to (i) the MPC simulation, (ii) the randomness generation, (iii) the hash functions, (iv) the seed trees and (v) the Merkle trees.

3.2.1 MPC subroutines

We describe hereafter all the subroutines required for the MPC simulation following the description of Section 2.1.2.

Polynomial evaluation. We define the function Evaluate which takes as input an $\mathbb{F}_q$-vector Q representing the coefficients of polynomial of $\mathbb{F}_q[X]$ and a point $r \in \mathbb{F}_{\text{points}}$, computes the evaluation $Q(r)$. Formally, we have

$$\text{Evaluate} : \begin{cases} \bigcup_{|Q|} (\mathbb{F}_q)^{|Q|} \times \mathbb{F}_{q^\eta} & \to \mathbb{F}_{q^\eta} \\ (Q, r) & \mapsto \sum_{i=1}^{|Q|} Q[i] \cdot r^{i-1} \end{cases} \qquad \text{where } r^{i-1} = \underbrace{r \otimes r \otimes \cdots \otimes r}_{i-1 \text{ times}} .$$

Let us stress that the powers r^i lies on the extension field $\mathbb{F}_{q^\eta}$ while the polynomial coefficients $Q[i+1]$ lies on the base field $\mathbb{F}_q$.

Complete and truncate $\boldsymbol{Q}$. Each polynomial $\boldsymbol{Q}[\nu]$ is defined as $\boldsymbol{Q}[\nu](X) = \prod_{i \in E}(X - f_i)$ for some set E of cardinality w/d. By definition, this polynomial is unary and we do not need to share its leading coefficient (which always equals 1). We then define $\boldsymbol{Q'}$ as the d-vectorized polynomial such that $\boldsymbol{Q'}[\nu](X) = \boldsymbol{Q}[\nu](X) - X^{w/d}$ for every $\nu \in [1 : d]$. We thus have $\boldsymbol{Q} \in (\mathbb{F}_q^{w/d+1})^d$ while $\boldsymbol{Q'} \in (\mathbb{F}_q^{w/d})^d$. We shall denote CompleteQ the subroutine which complete $\boldsymbol{Q'}$ with the leading coefficient and TruncateQ the subroutine which truncates from its leading coefficient $\boldsymbol{Q}$, so that we have:

$$\boldsymbol{Q} = \text{CompleteQ}(\boldsymbol{Q'}, 1)$$
$$\boldsymbol{Q'} = \text{TruncateQ}(\boldsymbol{Q})$$

We shall call the routine CompleteQ$(\cdot, 0)$ on some shares of $\boldsymbol{Q'}$. Indeed, the shares of $\boldsymbol{Q'}$ must all be completed with a leading 0 but one of them which is completed by a leading 1, so that the shares of the leading coefficient of $\boldsymbol{Q}$ well sum up to 1.

Inner product. The subroutine InnerProducts, described in Algorithm 1, computes the Beaver triples to be sacrificed in the MPC protocol. It takes as input a serialized $\mathbb{F}_q$-string beav_ab_plain representing the pairs $(\boldsymbol{a}, \boldsymbol{b})$ of the Beaver triples and returns the corresponding serialized vector of inner products. Note that those inner products are simple multiplications when $d = 1$.

Algorithm 1 InnerProducts

Input: beav_ab_plain
Output: beav_c_plain

$(\boldsymbol{a}, \boldsymbol{b}) \leftarrow \text{Parse}(\mathsf{beav\_ab\_plain}, (\mathbb{F}_{q^\eta}^d)^t, (\mathbb{F}_{q^\eta}^d)^t)$
for $j \in [1:t]$ **do**
  $c[j] \leftarrow \sum_{\nu=1}^{d} \boldsymbol{a}[j][\nu] \otimes \boldsymbol{b}[j][\nu]$ ▷ $c[j] \in \mathbb{F}_{q^\eta}$
$\mathsf{beav\_c\_plain} = \text{Serialize}(c)$
return beav_c_plain

Computation of plain broadcast values. The subroutine ComputePlainBroadcast, described in Algorithm 2, computes the publicly recomputed values of the MPC protocol (*i.e.* the plain values corresponding to the broadcasted shares). It takes as input the plain input of the MPC protocol, made of the witness $(s_A, \boldsymbol{Q'}, \boldsymbol{P})$ and the Beaver triples $(\boldsymbol{a}, \boldsymbol{b}, c)$, the syndrome decoding instance (H', y), and the MPC challenge $(r, \boldsymbol{\varepsilon})$. From these inputs, it computes and returns the plain broadcast values $(\boldsymbol{\alpha}, \boldsymbol{\beta})$. Note that the subroutine does not recompute v which is always zero.

Algorithm 2 ComputePlainBroadcast

Input: $\mathsf{input\_plain} := (\mathsf{wit\_plain}, \mathsf{beav\_ab\_plain}, \mathsf{beav\_c\_plain})$, chal, (H', y)
Output: broad_plain

$(s_A, \boldsymbol{Q'}, \boldsymbol{P}) \leftarrow \text{Parse}(\mathsf{wit\_plain}, \mathbb{F}_q^k, (\mathbb{F}_q^{w/d})^d, (\mathbb{F}_q^{w/d})^d)$
$(\boldsymbol{a}, \boldsymbol{b}) \leftarrow \text{Parse}(\mathsf{beav\_ab\_plain}, (\mathbb{F}_{q^\eta}^d)^t)$
$c \leftarrow \text{Parse}(\mathsf{beav\_c\_plain}, \mathbb{F}_{q^\eta}^t)$
$(r, \boldsymbol{\varepsilon}) \leftarrow \text{Parse}(\mathsf{chal}, \mathbb{F}_{q^\eta}^t, (\mathbb{F}_{q^\eta}^d)^t)$
$s = (s_A \mid y + H' s_A)$ ▷ $s \in \mathbb{F}_q^m$
$\boldsymbol{Q} = \text{CompleteQ}(\boldsymbol{Q'}, 1)$ ▷ $\boldsymbol{Q} \in (\mathbb{F}_q^{(w/d)+1})^d$
$\boldsymbol{S} \leftarrow \text{Parse}(s, (\mathbb{F}_q^{m/d})^d)$
for $j \in [1:t]$ **do**
  for $\nu \in [1:d]$ **do**
    $\boldsymbol{\alpha}[j][\nu] = \boldsymbol{\varepsilon}[j][\nu] \otimes \text{Evaluate}(\boldsymbol{Q}[\nu], r[j]) + \boldsymbol{a}[j][\nu]$ ▷ $\boldsymbol{\alpha}[j][\nu] \in \mathbb{F}_{q^\eta}$
    $\boldsymbol{\beta}[j][\nu] = \text{Evaluate}(\boldsymbol{S}[\nu], r[j]) + \boldsymbol{b}[j][\nu]$ ▷ $\boldsymbol{\beta}[j][\nu] \in \mathbb{F}_{q^\eta}$
$\mathsf{broad\_plain} = \text{Serialize}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
return broad_plain

Party computation. The subroutine PartyComputation, described in Algorithm 3, performs the party computation, namely it computes the shares broadcast by a party. It takes the input shares of the party $[\![(s_A, \boldsymbol{Q'}, \boldsymbol{P})]\!]_i$ and $[\![(\boldsymbol{a}, \boldsymbol{b}, c)]\!]_i$, the syndrome decoding instance (H', y), the MPC challenge $(r, \boldsymbol{\varepsilon})$ and the recomputed values $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ and returns the broadcast shares $[\![(\boldsymbol{\alpha}, \boldsymbol{\beta}, v)]\!]_i$ of the party.

This subroutine further takes as input a Boolean with_offset which indicates whether the constant part of the computed affine function should be introduced or not for this party. For instance, in the case of additive sharing (hypercube variant), when the parties locally add a constant value to a sharing, the constant addition is only done by one party. The Boolean with_offset is set to `True` for this party while it is set to `False` for the other parties.

In Algorithm 3, we use $(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{\beta}})$ to denote the plain broadcast values while $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ denotes one share of these values (corresponding to the party being computed).

Algorithm 3 PartyComputation

Input: input_share := (wit_share, beav_ab_share, beav_c_share), chal, (H', y), broad_plain, with_offset

Output: broad_share

$(s_A, \boldsymbol{Q'}, \boldsymbol{P}) \leftarrow \text{Parse}(\text{wit\_share}, \mathbb{F}_q^k, (\mathbb{F}_q^{w/d})^d, (\mathbb{F}_q^{w/d})^d)$
$(\boldsymbol{a}, \boldsymbol{b}) \leftarrow \text{Parse}(\text{beav\_ab\_share}, (\mathbb{F}_{q^\eta}^d)^t)$
$c \leftarrow \text{Parse}(\text{beav\_c\_share}, \mathbb{F}_{q^\eta}^t)$
$(r, \boldsymbol{\varepsilon}) \leftarrow \text{Parse}(\text{chal}, \mathbb{F}_{q^\eta}^t \times (\mathbb{F}_{q^\eta}^d)^t)$
$(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{\beta}}) \leftarrow \text{Parse}(\text{broad\_plain}, (\mathbb{F}_{q^\eta}^d)^t, (\mathbb{F}_{q^\eta}^d)^t)$
if with_offset is `True` **then**
    $s = (s_A \mid y + H' s_A)$ ▷ $s \in \mathbb{F}_q^n$
    $\boldsymbol{Q} = (\boldsymbol{Q'}, 1)$ ▷ $\boldsymbol{Q} \in (\mathbb{F}_q^{(w/d)+1})^d$
else
    $s = (s_A \mid H' s_A)$ ▷ $s \in \mathbb{F}_q^n$
    $\boldsymbol{Q} = (\boldsymbol{Q'}, 0)$ ▷ $\boldsymbol{Q} \in (\mathbb{F}_q^{(w/d)+1})^d$
$\boldsymbol{S} \leftarrow \text{Parse}(s, (\mathbb{F}_q^{m/d})^d)$
for $j \in [1 : t]$ **do**
    $v[j] = -c[j]$ ▷ $v[j] \in \mathbb{F}_{q^\eta}$
    for $\nu \in [1 : d]$ **do**
        $\boldsymbol{\alpha}[j][\nu] = \boldsymbol{\varepsilon}[j][\nu] \otimes \text{Evaluate}(\boldsymbol{Q}[\nu], r[j]) + \boldsymbol{a}[j][\nu]$ ▷ $\boldsymbol{\alpha}[j][\nu] \in \mathbb{F}_{q^\eta}$
        $\boldsymbol{\beta}[j][\nu] = \text{Evaluate}(\boldsymbol{S}[\nu], r[j]) + \boldsymbol{b}[j][\nu]$ ▷ $\boldsymbol{\beta}[j][\nu] \in \mathbb{F}_{q^\eta}$
        $v[j]$ += $\boldsymbol{\varepsilon}[j][\nu] \otimes \text{Evaluate}(F, r[j]) \otimes \text{Evaluate}(\boldsymbol{P}[\nu], r[j])$
        $v[j]$ += $\bar{\boldsymbol{\alpha}}[j][\nu] \otimes \boldsymbol{b}[j][\nu] + \bar{\boldsymbol{\beta}}[j][\nu] \otimes \boldsymbol{a}[j][\nu]$
        if with_offset is `True` **then**
            $v[j]$ += $-\boldsymbol{\alpha}[j][\nu] \otimes \boldsymbol{\beta}[j][\nu]$
broad_share = Serialize$(\boldsymbol{\alpha}, \boldsymbol{\beta}, v)$
return broad_share

Inverse computation. The subroutine InversePartyComputation, described in Algorithm 4, computes the shares of the Beaver triples from the shares of the witness and the broadcast shares of a party. This subroutine is used by the verification in the threshold variant to avoid interpolations, as explained in Section 2.3. For any wit_share, beav_ab_share, beav_c_share and broad_share, the functionality of this subroutine is such that:

$$(\mathsf{beav\_ab\_share}, \mathsf{beav\_c\_share}) = \text{InversePartyComputation}(\mathsf{wit\_share}, \mathsf{broad\_share}, \mathsf{extra})$$
$$\iff \quad \mathsf{broad\_share} = \text{PartyComputation}(\mathsf{wit\_share}, \mathsf{beav\_ab\_share}, \mathsf{beav\_c\_share}, \mathsf{extra})$$

where extra corresponds to any $(\mathsf{chal}, (H', y), \mathsf{broad\_plain}, \mathsf{with\_offset})$.

Algorithm 4 InversePartyComputation

Input: wit_share, broad_share, chal, (H', y), broad_plain, with_offset
Output: (beav_ab_share, beav_c_share)

$(s_A, \boldsymbol{Q'}, \boldsymbol{P}) \leftarrow \text{Parse}(\mathsf{wit\_share}, \mathbb{F}_q^k, (\mathbb{F}_q^{w/d})^d, (\mathbb{F}_q^{w/d})^d)$
$(\boldsymbol{\alpha}, \boldsymbol{\beta}, v) \leftarrow \text{Parse}(\mathsf{broad\_share}, (\mathbb{F}_{q^\eta}^d)^t, \mathbb{F}_{q^\eta}^t)$
$(r, \boldsymbol{\varepsilon}) \leftarrow \text{Parse}(\mathsf{chal}, \mathbb{F}_{q^\eta}^t \times (\mathbb{F}_{q^\eta}^d)^t)$
$(\bar{\boldsymbol{\alpha}}, \bar{\boldsymbol{\beta}}) \leftarrow \text{Parse}(\mathsf{broad\_plain}, (\mathbb{F}_{q^\eta}^d)^t, (\mathbb{F}_{q^\eta}^d)^t)$
if with_offset is `True` **then**
    $s = (s_A \mid y + H' s_A)$ ▷ $s \in \mathbb{F}_q^n$
    $\boldsymbol{Q} = (\boldsymbol{Q'}, \mathbf{1})$ ▷ $\boldsymbol{Q} \in (\mathbb{F}_q^{(w/d)+1})^d$
else
    $s = (s_A \mid H' s_A)$ ▷ $s \in \mathbb{F}_q^n$
    $\boldsymbol{Q} = (\boldsymbol{Q'}, \mathbf{0})$ ▷ $\boldsymbol{Q} \in (\mathbb{F}_q^{(w/d)+1})^d$
$\boldsymbol{S} \leftarrow \text{Parse}(s, (\mathbb{F}_q^{m/d})^d)$
for $j \in [1 : t]$ **do**
    $c[j] = -v[j]$ ▷ $c[j] \in \mathbb{F}_{q^\eta}$
    for $\nu \in [1 : d]$ **do**
        $\boldsymbol{a}[j][\nu] = \boldsymbol{\alpha}[j][\nu] - \boldsymbol{\varepsilon}[j][\nu] \otimes \text{Evaluate}(\boldsymbol{Q}[\nu], r[j])$ ▷ $\boldsymbol{a}[j][\nu] \in \mathbb{F}_{q^\eta}$
        $\boldsymbol{b}[j][\nu] = \boldsymbol{\beta}[j][\nu] - \text{Evaluate}(\boldsymbol{S}[\nu], r[j])$ ▷ $\boldsymbol{b}[j][\nu] \in \mathbb{F}_{q^\eta}$
        $c[j]$ += $\boldsymbol{\varepsilon}[j][\nu] \otimes \text{Evaluate}(F, r[j]) \otimes \text{Evaluate}(\boldsymbol{P}[\nu], r[j])$
        $c[j]$ += $\bar{\boldsymbol{\alpha}}[j][\nu] \otimes \boldsymbol{b}[j][\nu] + \bar{\boldsymbol{\beta}}[j][\nu] \otimes \boldsymbol{a}[j][\nu]$
        if with_offset is `True` **then**
            $c[j]$ += $-\boldsymbol{\alpha}[j][\nu] \otimes \boldsymbol{\beta}[j][\nu]$
beav_ab_share $= \text{Serialize}(\boldsymbol{a}, \boldsymbol{b})$
beav_c_share $= \text{Serialize}(v)$
return (beav_ab_share, beav_c_share)

3.2.2 Pseudo-randomness generation

Several subroutines used in the SD-in-the-Head signature schemes involve pseudorandomness generation from a seed. Several seeds are expanded from a master seed in the key generation and in the hypercube variant of the signature algorithm (to generate the sharings). One also needs to sample sequences of field elements from a seed in the key generation, the signature and verification algorithms (both variants). Finally pseudorandomness generation is also involved to derive the challenges (MPC challenge and view-opening challenge) from the Fiat-Shamir hashes h_1 and h_2.

Extendable output function. The pseudorandomness in SD-in-the-Head 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: XOF.Init(x) initializes the XOF state with the input $x \in \{0,1\}^*$. Once initialized, the XOF can be queried with the routine 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 scheme is given in Section 4.5. In our context, we use the XOF as a secure pseudorandom generator (PRG) which tolerates input seeds of variable lengths.

Sampling from XOF. We shall denote by Sample, the routine generating pseudorandom element from an arbitrary set $\mathcal{V}$. A call to

$$v \leftarrow \text{XOF.Sample}(\mathcal{V})$$

outputs a uniform random element $v \in \mathcal{V}$. The Sample routine relies on calls to GetByte to generate pseudorandom bytes which are then formatted to obtain a uniform variable $v \in \mathcal{V}$, possibly using rejection sampling. The implementation of Sample depends on the target set $\mathcal{V}$. We detail the case of sampling field elements hereafter, namely when $\mathcal{V} = \mathbb{F}_q^n$ for some n.

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

- For $\mathbb{F}_q = \mathbb{F}_{256}$: The byte B_i is simply returned as the ith sampled field element. The XOF is called to generate $n' = n$ bytes.
- For $\mathbb{F}_q = \mathbb{F}_{251}$: The byte B_i is interpreted as an integer $B_i \in \{0, 1, \ldots, 255\}$. We use the principle of rejection sampling to only select integer values modulo 251, namely we reject byte values in $\{251, \ldots, 255\}$. The procedure goes as follows:

  1: $i = 1$
  2: **while** $i \leq n$ **do**
  3: $\quad B \leftarrow$ XOF.GetByte()
  4: $\quad$ **if** $B \in \{0, 1, \ldots, 250\}$ **then**
  5: $\quad\quad f_i = B$; i ++
  6: **return** $(f_1, \ldots, f_n)$

  The number of generated bytes n' which are necessary to complete the process is non-deterministic. In average on needs to generates $n' \approx (256/251)n \approx 1.02n$ bytes.

Seed expansion. The subroutine ExpandSeed expands a salt and a master seed into a given number of seeds. Specifically, a call to ExpandSeed($\mathsf{salt}, \mathsf{seed}, n$) initializes the XOF by calling XOF.Init($\mathsf{salt} \parallel \mathsf{seed}$) and then calls XOF.GetByte() to generate a stream of bytes $B_1, \ldots, B_{n\lambda/8}$ which are divided into n output λ-bit seeds $\mathsf{seed}_1, \ldots, \mathsf{seed}_n$ as follows:

$$(\underbrace{B_1, \ldots, B_{\lambda/8}}_{\mathsf{seed}_1}, \ldots, \underbrace{B_{(n-1)\lambda/8+1}, \ldots, B_{n\lambda/8}}_{\mathsf{seed}_n})$$

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

$$\begin{aligned}
&\text{XOF.Init}(\mathsf{seed}_H)\\
&(f_1, \ldots, f_{(m-k)\cdot k}) \leftarrow \text{XOF.SampleFieldElements}((m-k)\cdot k)\\
&H'\texttt{[}i\texttt{]}\,\texttt{[}j\texttt{]} := f_{(j-1)k+i} \quad \forall\ (i,j) \in [1 : m-k] \times [1 : k]\ .
\end{aligned}$$

Expansion of MPC challenge. The subroutine ExpandMPCChallenge expands the first Fiat-Shamir hash h_1 into the MPC challenges $(r, \boldsymbol{\varepsilon}) \in \mathbb{F}_{q^\eta}^t \times (\mathbb{F}_{q^\eta}^d)^t$. This subroutine takes as input the hash h_1 and the number n of pairs $(r, \boldsymbol{\varepsilon})$ to be generated. It consists of the following steps:

$$\begin{aligned}
&\text{XOF.Init}(h_1)\\
&v \leftarrow \text{XOF.SampleFieldElements}(nt\eta(d+1))\\
&(\mathsf{chal}\texttt{[}1\texttt{]}, \ldots, \mathsf{chal}\texttt{[}n\texttt{]}) = \text{Parse}(v, \mathbb{F}_q^{t\eta(d+1)}, \ldots, \mathbb{F}_q^{t\eta(d+1)})\ ,
\end{aligned}$$

where each $\mathsf{chal}\texttt{[}e\texttt{]}$ represents a serialized pair $(r, \boldsymbol{\varepsilon}) \in \mathbb{F}_{q^\eta}^t \times (\mathbb{F}_{q^\eta}^d)^t$.

For the hypercube variant we have one challenge per parallel execution, *i.e.* $n = \tau$, while for the threshold variant, we use a global challenge for all the executions, *i.e.* $n = 1$ (see Section 2.3).

Expansion of view-opening challenge. The subroutine ExpandViewChallenge, expands the second Fiat-Shamir hash h_2 into the view-opening challenge $I\texttt{[}1\texttt{]}, \ldots, I\texttt{[}\tau\texttt{]}$, where $I\texttt{[}e\texttt{]} \subset [1 : N]$ is the set of parties to be opened for execution e. This subroutine takes as input the hash h_2 and a mode character, either `hypercube` or `threshold`. It first initializes the XOF by calling

$$\text{XOF.Init}(h_2)\ .$$

For the `hypercube` mode, the generated sets are of cardinal $N-1$ and are simply represented by the indexes $i^*\texttt{[}1\texttt{]}, \ldots, i^*\texttt{[}\tau\texttt{]}$ such that $I\texttt{[}e\texttt{]} = [1 : N] \setminus i^*\texttt{[}e\texttt{]}$ for each execution e. The subroutine then calls

$$i^*[e] \leftarrow \text{XOF.Sample}([1 : N]) \quad \forall e \in [1 : \tau]\ .$$

For the `threshold` mode, the generated sets are of cardinal ℓ. The subroutine then calls

$$I[e] \leftarrow \text{XOF.Sample}(\{J \subseteq [1 : N]\ ;\ |J| = \ell\}) \quad \forall e \in [1 : \tau]\ .$$

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

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

The concrete instance of the hash function we use in the SD-in-the-Head scheme is given in Section 4.5.

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.

Commitments. The subroutine Commit takes as input a 2λ-bit salt, an execution index e, a share index i and some data $\mathsf{data} \in \{0,1\}^*$. It hashes them all together and returns the corresponding digest. Specifically, we define:

$$\mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{data}) = \mathrm{Hash}(0 \parallel \mathsf{salt} \parallel e_0 \parallel e_1 \parallel i_0 \parallel i_1 \parallel \mathsf{data}) \ ,$$

where e_0, e_1, i_0, i_1 are the byte values such that $e = e_0 + 256 \cdot e_1$ and $i = i_0 = 256 \cdot i_1$, where 0, e_0, e_1, i_0 and i_1 are encoded on one byte, and where salt is encoded on $2\lambda/8$ bytes.

Fiat-Shamir Hashes. The hash functions Hash_1 and Hash_2 used to derive the Fiat-Shamir Hashes h_1 and h_2 are defined as:

$$\mathrm{Hash}_1(\mathsf{data}) = \mathrm{Hash}(1 \parallel \mathsf{data}) \ ,$$

and

$$\mathrm{Hash}_2(\mathsf{data}) = \mathrm{Hash}(2 \parallel \mathsf{data}) \ ,$$

where the prefixes 1 and 2 are both encoded on one byte.

3.2.4 Seed trees (hypercube variant)

To save communication, the hypercube variant relies on seed trees via three subroutines:

- TreePRG: it takes a 2λ-bit salt and a λ-bit seed, and returns N λ-bit seeds which correspond to the leaves of a binary seed tree with seed as root. The nodes are numbered in hierarchical order: the root has index 1, and the left and right children of node i have indexes $2i, 2i+1$, and the seeds of the whole tree for $i \in [2, 2.2^D - 1]$ are derived via the following recursive formula:

$$(\mathsf{seed}_{2i} \parallel \mathsf{seed}_{2i+1}) \leftarrow \mathrm{Hash}(3 \parallel \mathsf{salt} \parallel e_0 \parallel e_1 \parallel i_0 \parallel i_1 \parallel \mathsf{seed}_i)$$

  Here, $e = e_0 + 256 \cdot e_1 \in [1, \tau]$ and $i = i_0 + 256 \cdot i_1 \in [1, 2 \cdot 2^D - 1]$ are 16-bit little-endian execution and node indexes.

- GetSeedSiblingPath: it takes a 2λ-bit salt, a λ-bit seed and an index i^*, and it returns the sibling path of the seed leave indexed by i^* in a binary seed tree. It returns the D seeds that are sibling of the ancestors of i^* in the tree, namely:

$$\mathsf{path}_j = \mathsf{seed}_{(i^* \gg (D-j)) \oplus 1} \text{ for } j \in [1, D]$$

  Here, $\gg$ is the logical right shift, and $\oplus 1$ flips the least significant bit. It is possible to store the $2.2^D - 1$ seeds, extract the sibling path from it, and delete the remaining seeds, or equivalently to re-derive those seeds from the root seed and the salt in D calls of the derivation formula above.

- GetLeavesFromSiblingPath: it takes an index i^*, a 2λ-bit salt and a seed path, and it returns all the leaves except the one of index i^* of the seed tree for which GetSeedSiblingPath on i^* would output this path.

3.2.5 Merkle trees (threshold variant)

The threshold variant relies on Merkle trees for the commitment of the party shares. Those Merkle commitments and decommitments are handled using the following subroutines:

- MerkleTree (Algorithm 5): This subroutine takes as input a list of N commitments $c_1, \ldots, c_N \in \{0,1\}^{2\lambda}$ (outputs of the Commit subroutine) and returns the nodes and root of the binary Merkle tree with $c_1, \ldots, c_N$ as leaves. In this tree, the hash call to compute the ith node is prefixed by $(3 \parallel i_0 \parallel i_1)$ where i_0, i_1 are the byte values such that $i = i_0 + 256 \cdot i_1$, and where 3, i_0, i_1 are encoded on one byte. We use `null` to denote a special character which specifies that a node value is undetermined.[2]
- GetMerklePath (Algorithm 6): This subroutine takes as input a list of nodes nodes representing a Merkle tree and a set $I \subset [1:N]$ indexing leaves to be authenticated. It returns the authentication paths for leaves indexed by I in the Merkle tree.
- GetMerkleRootFromAuth (Algorithm 7): This subroutine takes as input a set of indexes $I \subset [N]$, the corresponding commitments (or leaves) $\{c_i\}_{i \in I}$ to be authenticated and the corresponding authentication paths auth. It returns the root of the Merkle tree recomputed from these leaves and authentication paths or `invalid` in case it fails to compute the root. The proposed algorithm involves a queue structure. This structure 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*.

  Algorithm 7 has the advantage to require only a small memory space, since the number of elements in the queue is always less than $|I|$.

Algorithm 5 MerkleTree

Input: N commitments $c_1, \ldots, c_N \in \{0,1\}^{2\lambda}$
Output: nodes, root, the nodes and root of the Merkle tree

1: $n = \lceil \log_2(N) \rceil$
2: $\mathsf{nodes}[2^n] = c_1$, ..., $\mathsf{nodes}[2^n + N - 1] = c_N$ ▷ Leaves of the Merkle tree
3: $\mathsf{nodes}[2^n + N] = \ldots = \mathsf{nodes}[2^{n+1} - 1] = \texttt{null}$
4: **for** i from $2^n - 1$ downto 1 **do**
5: $$\mathsf{nodes}[i] = \begin{cases} \mathrm{Hash}(3 \parallel i_0 \parallel i_1 \parallel \mathsf{nodes}[2i] \parallel \mathsf{nodes}[2i+1]) & \text{if } \mathsf{nodes}[2i+1] \neq \texttt{null} \\ \mathsf{nodes}[2i] & \text{otherwise} \end{cases}$$
6: ▷ $i = i_0 + 256 \cdot i_1$
7: $\mathsf{root} = \mathsf{nodes}[1]$
8: **return** (nodes, root) ▷ Merkle tree, with its root

[2] We stress that the value of `null` does not need to be specified since it does not enter any hash computation.

Algorithm 6 GetMerklePath

Input: A Merkle tree nodes, a set $I \subset [1 : N]$ indexing leaves to be authenticated
Output: The authentication paths auth for the leaves in I

missing $\leftarrow \{2^n + i - 1, i \notin I\}$
for i from $2^n - 1$ downto 1 **do**
    if $(2i) \in$ missing and $(2i + 1) \in$ missing **then**
        missing $\leftarrow$ (missing$\backslash\{2i, 2i + 1\}) \cup \{i\}$
auth $\leftarrow \emptyset$
for h from n downto 1 **do**
    for i from 2^h to $2^{h+1} - 1$ **do**
        if $i \in$ missing **then**
            auth $\leftarrow$ (auth $\|$ nodes$[i]$)
return auth

Algorithm 7 GetMerkleRootFromAuth

Input: a set $I \subset [1 : N]$ indexing leaves to be authenticated, the commitments $\{c_i\}_{i \in I}$, the authentication paths auth
Output: root, the recomputed Merkle root

queue $\leftarrow$ Queue.Init()
for $i \in I$ in the increasing order **do**
    queue.Enqueue$((c_i, 2^n + i - 1))$ ▷ queue $\leftarrow (c_i, 2^n + i - 1)$
(height, last_index) $= (2^n, 2^n + N - 1)$
while $i \neq 1$ where $(\_\_, i) \leftarrow$ queue.Head() **do** ▷ While the queue head is not the root.
    (node, i) $\leftarrow$ queue.Dequeue() ▷ (node, i) $\leftarrow$ queue
    if $i <$ height **then** ▷ If the height changes
        (height, last_index) $= (\lfloor$height$/2\rfloor, \lfloor$last_index$/2\rfloor)$
    if i even and $i ==$ last_index **then**
        queue.Enqueue((node, $\lfloor i/2 \rfloor$))
    else
        $(\_\_, i') \leftarrow$ queue.Head() ▷ Get the index of the next node in the queue
        if i even and $i' = i + 1$ **then**
            (node$'$, i') $\leftarrow$ queue.Dequeue()
        else
            if $|$auth$| \geq 2\lambda$ **then**
                (node$'$ $\|$ auth) $\leftarrow$ auth ▷ Extract the 2λ first bits of auth
            else
                return `invalid`
            if i odd **then**
                (node, node$'$) $\leftarrow$ (node$'$, node) ▷ Swap the nodes
        pnode $\leftarrow$ Hash$(3 \| i_0 \| i_1 \|$ node $\|$ node$'$) ▷ $\lfloor i/2 \rfloor = i_0 + 256 \cdot i_1$
        queue.Enqueue((pnode, $\lfloor i/2 \rfloor$))
(root, $\_\_$) $\leftarrow$ queue.Dequeue() ▷ (root, $\_\_$) $\leftarrow$ queue
return root

3.3 Key generation

The key generation simply consists in sampling a (d-split) syndrome decoding instance. The public key is the instance (H, y) while the secret key is composed of the instance and the associated solution (H, y, x). We use the trick described in Section 2.1 to avoid interpolations in the signing and verification algorithms: the vector y is defined as $y := HVx$ (instead of $y := Hx$) where V is the interpolation matrix (see Equation 3) and we denote $s := Vx$. Moreover, we consider that H is in standard form, *i.e.* $H := (H' \mid I_{m-k})$.

Algorithm 8 SampleWitness

Input: $\mathsf{seed}_{\mathsf{wit}} \in \{0,1\}^\lambda$
Output: $(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P})$

$(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P}) \leftarrow \mathrm{Init}((\mathbb{F}_q^{w/d+1})^d, (\mathbb{F}_q^{m/d})^d, (\mathbb{F}_q^{w/d})^d)$
$\mathrm{XOF.Init}(\mathsf{seed}_{\mathsf{wit}})$
for $\nu \in [1:d]$ **do**
  $\mathsf{pos}[\nu] \leftarrow \mathrm{XOF.Sample}(\{J \subseteq [1:m/d]\ ;\ |J| = w/d\})$
  $\mathsf{val}[\nu] \leftarrow \mathrm{XOF.Sample}((\mathbb{F}_q^*)^{w/d})$
  for $i \in [1:m/d]$ **do**
    $\boldsymbol{x}[\nu][i] = \sum_{j \in [1:w/d]} \mathsf{val}[\nu][j] \cdot (\mathsf{pos}[\nu][j] == i)$ ▷ $(\mathsf{pos}[\nu][j] == i) \in \{0,1\}$
    ▷ with `True` $= 1$, `False` $= 0$
  $\boldsymbol{Q}[\nu] = \mathrm{ComputeQ}(\mathsf{pos}[\nu])$
  $\boldsymbol{S}[\nu] = \mathrm{ComputeS}(\boldsymbol{x}[\nu])$
  $\boldsymbol{P}[\nu] = \mathrm{ComputeP}(\boldsymbol{Q}[\nu], \boldsymbol{S}[\nu])$
return $(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P})$

We describe in Algorithm 8 the subroutine SampleWitness which samples the d-split SD solution from a seed and builds the polynomials $(\boldsymbol{S}, \boldsymbol{Q}, \boldsymbol{P})$ of the SD-in-the-Head witness. Sampling a d-split SD solution consists in generating a vector

$$\boldsymbol{x} := (\boldsymbol{x}[1] \parallel \ldots \parallel \boldsymbol{x}[d]) \in \mathbb{F}_q^m \quad \text{s.t.} \quad \mathrm{wt}(\boldsymbol{x}[\nu]) = w/d \quad \forall \nu \in [1:d]\ .$$

For every $\nu \in [1:d]$, we sample a list $\mathsf{pos}[\nu]$ of the w/d positions of the non-zero coordinates of $\boldsymbol{x}[\nu]$ as well as a list $\mathsf{val}[\nu]$ of w/d non-zero field elements (to be assigned to the non-zero coordinates). The polynomials $\boldsymbol{S}[\nu]$, $\boldsymbol{Q}[\nu]$ and $\boldsymbol{P}[\nu]$ are derived from the obtained chunk $\boldsymbol{x}[\nu]$. This is done according to the definition of the SD-in-the-Head witness (see Section 2.1) using the following functions:

- The function ComputeQ maps a list of indices from $[1:m/d]$ to a polynomial Q:

$$Q = \mathrm{ComputeQ}(i_1, \ldots, i_{w/d}) \quad \Leftrightarrow \quad Q(X) = \prod_{j=1}^{w/d} (X - f_{i_j})\ .$$

  The output is interpreted as a vector of coefficients $Q \in \mathbb{F}_q^{w/d+1}$.

- The function ComputeS maps a vector of $\mathbb{F}_q^{m/d}$ to a polynomial S:

$$S = \mathrm{ComputeS}(x_1, \ldots, x_{m/d}) \quad \Leftrightarrow \quad S(X) = \sum_{i=1}^{m/d} g_i \cdot x_i \cdot \frac{F(X)}{X - f_i}$$

where

$$F(X) := \prod_{i=1}^{m/d}(X - f_i) \quad \text{and} \quad g_i := \prod_{j \in [1:m/d] \setminus \{i\}} (f_i - f_j)^{-1} .$$

Namely, S is obtained by Lagrange interpolation of the input vector. The output is interpreted as a vector of coefficients $S \in \mathbb{F}_q^{m/d}$.

- The function ComputeP maps polynomials Q and S output of the previous functions to a polynomial P:

$$P = \text{ComputeP}(Q, S) \quad \Leftrightarrow \quad P(X) = \frac{Q(X) \cdot S(X)}{F(X)}$$

where F is defined as above. The output is interpreted as a vector of coefficients $P \in \mathbb{F}_q^{w/d}$.

The key generation is described in Algorithm 9. It first sample a root seed $\mathsf{seed}_{\mathsf{root}}$ and then expands it into two subseeds $(\mathsf{seed}_{\mathsf{wit}}, \mathsf{seed}_H)$. The first subseed $\mathsf{seed}_{\mathsf{wit}}$ is used to sample the SD-in-the-Head witness $(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P})$ through the subroutine SampleWitness. The second seed seed_H is used to generate the matrix H' which defines the parity check matrix $H := (H' \mid I)$. The algorithm then builds the public key by packing the seed seed_H which encodes H and the vector $y := Hs = s_B + H's_A$, where $s := (s_A \mid s_B)$ is the serialized form of $\boldsymbol{S}$. It also builds the secret key by packing the seed seed_H, the vector y and the formatted witness $\mathsf{wit\_plain} := (s_A, \boldsymbol{Q'}, \boldsymbol{P})$. Here $\boldsymbol{Q'}$ is the truncated version of $\boldsymbol{Q}$, i.e., for which the leading coefficient –which always equals 1– has been removed (see Section 3.2.1). We recall that only s_A is necessary in the formatted witness since s_B can be recovered as $s_B = y - H's_A$.

Algorithm 9 SD-in-the-Head – Key Generation

$\mathsf{seed}_{\mathsf{root}} \leftarrow \{0,1\}^\lambda$
$(\mathsf{seed}_{\mathsf{wit}}, \mathsf{seed}_H) \leftarrow \text{ExpandSeed}(\mathsf{salt} := 0, \mathsf{seed}_{\mathsf{root}}, 2)$ ▷ $\mathsf{seed}_w, \mathsf{seed}_H \in \{0,1\}^\lambda$
$(\boldsymbol{Q}, \boldsymbol{S}, \boldsymbol{P}) \leftarrow \text{SampleWitness}(\mathsf{seed}_{\mathsf{wit}})$ ▷ $\boldsymbol{Q} \in (\mathbb{F}_q^{w/d+1})^d$, $\boldsymbol{S} \in (\mathbb{F}_q^{m/d})^d$, $\boldsymbol{P} \in (\mathbb{F}_q^{w/d})^d$
$s = \text{Serialize}(\boldsymbol{S})$ ▷ $s \in \mathbb{F}_q^m$
$(s_A, s_B) = \text{Parse}(s, \mathbb{F}_q^k, \mathbb{F}_q^{m-k})$ ▷ $s_A \in \mathbb{F}_q^k$, $s_B \in \mathbb{F}_q^{m-k}$
$H' \leftarrow \text{ExpandH}(\mathsf{seed}_H)$ ▷ $H' \in \mathbb{F}_q^{(m-k)\times k}$
$y = s_B + H's_A$ ▷ $y \in \mathbb{F}_q^{m-k}$
$\boldsymbol{Q'} = \text{TruncateQ}(\boldsymbol{Q})$
$\mathsf{wit\_plain} = \text{Serialize}(s_A, \boldsymbol{Q'}, \boldsymbol{P})$
$pk = (\mathsf{seed}_H, y)$; $sk = (\mathsf{seed}_H, y, \mathsf{wit\_plain})$
return (pk, sk)

3.4 Hypercube variant

We describe the signing and verification algorithms for the hypercube variant.

Signing algorithm. The signing algorithm is described in Algorithm 10. It consists of the following steps:

1. It starts with some initialization: it samples a 2λ-bit salt and a λ-bit seed, then it expands the matrix H'.

   $\mathsf{salt} \leftarrow \{0,1\}^{2\lambda}$
   $\mathsf{seed} \leftarrow \{0,1\}^{\lambda}$
   $H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$

2. For each parallel repetition, it generates the 2^D seeds of the leaf parties using TreePRG, expands those seeds into the 2^D leaf shares. For all the leaf parties $i \in [1, 2^D - 1]$, the leaf seeds expand to the full leaf shares, whereas for $i = 2^D$, the last seed expands to the uniform term of the Beaver triple, while the remaining fields are zero.

3. To avoid having to store the 2^D shares, they are aggregated on the fly:
   - the sum of all expanded leaf shares, named `acc` in the algorithm
   - the main party $(\wp, 0)$ in each dimension $\wp \in [1, D]$. In the algorithm, we omit the 0, so $\mathsf{input\_mshare}$ is only indexed by $\wp$.

4. We deduce the auxiliary, which is the difference between the plaintext and the accumulated value at the previous step. The auxiliary is part of the last leaf share, but does not affect any main party share $(\wp, 0)$.

5. We compute the 2^D leaf commitments of the 2^D leaf states: for $i \in [1, 2^D - 1]$ the leaf state is solely the corresponding leaf seed, and for $i = 2^D$, the last leaf state includes the last leaf seed and the auxiliary. All the leaf commitments are then hashed together to form the state commitment digest h_1

6. It then expands the obtained digest h_1 as the MPC challenge chal.

   $\mathsf{chal} \leftarrow \mathrm{ExpandMPCChallenge}(h_1, 1)$

7. For each repetition, it computes the plain broadcast values $\mathsf{broad\_plain} = (\boldsymbol{\alpha}, \boldsymbol{\beta}, v = 0)$. The latter only depends on the plain witness and plain Beaver triple, and thus, it is identical for the D dimensions of the hypercube. We just compute it once per repetition and include it in the signature.

   $\mathsf{broad\_plain} \leftarrow \mathrm{ComputePlainBroadcast}(\mathsf{input\_plain}, \mathsf{chal}, (H', y))$

8. It emulates the MPC protocol for each repetition and each dimension, for main party $(\wp, 0)$ only.

   for $(e, \wp) \in [1 : \tau] \times [1 : D]$ **do**
   $\mathsf{broad\_share}[e][j] = \mathrm{PartyComputation}(\mathsf{input\_coef}[e][j], \mathsf{chal}, (H', y), \mathsf{broad\_plain}, \mathtt{False})$

9. It hashes the broadcast messages and expand the obtained digest h_2 as the list of hidden leaf parties (one per repetition).

10. For each repetition e, it builds the sibling paths for the revealed views.

Algorithm 10 SD-in-the-Head – Hypercube Variant – Signature Algorithm

Input: a secret key $sk = (\mathsf{seed}_H, y, \mathsf{wit\_plain})$ and a message $m \in \{0,1\}^*$

$\mathsf{salt} \leftarrow \{0,1\}^{2\lambda}$
$\mathsf{mseed} \leftarrow \{0,1\}^{\lambda}$
$H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$ ▷ $H' \in \mathbb{F}_q^{(n-k)\times k}$
$\{\mathsf{rseed}\,[e]\,\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandSeed}(\mathsf{salt}, \mathsf{mseed}, \tau)$ ▷ $\mathsf{rseed}\,[e] \in \{0,1\}^{\lambda}$
for $e \in [1:\tau]$ **do**
    $(\mathsf{seed}\,[e]\,[i]\,)_{i\in[1:2^D]} \leftarrow \mathrm{TreePRG}(\mathsf{salt}, \mathsf{rseed}\,[e]\,)$
    $\mathsf{acc} = 0$ ▷ $\mathsf{acc} \in \mathbb{F}_q^{k+2w+t(2d+1)\eta}$
    $\mathsf{input\_mshare}\,[e]\,[p] = 0$ for all $(e,p) \in [1:\tau] \times [1:D]$
        ▷ $\mathsf{input\_mshare}\,[e]\,[i] \in \mathbb{F}_q^{k+2w+t(2d+1)\eta}$
    for $i \in [1:2^D]$ **do**
        if $i \neq 2^D$ **then**
            $\mathsf{input\_share}\,[e]\,[i] \leftarrow \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}\,[e]\,[i]\,, k+2w+t(2d+1)\eta)$
                ▷ $\mathsf{input\_share}\,[e]\,[i] \in \mathbb{F}_q^{k+2w+t(2d+1)\eta}$
            acc += $\mathsf{input\_share}\,[e]\,[i]$
            $\mathsf{state}\,[e]\,[i] = \mathsf{seed}\,[e]\,[i]$
            for $p \in [1:D]$: the p^{th} bit of $i-1$ is zero, **do**
                $\mathsf{input\_mshare}\,[e]\,[p]$ += $\mathsf{input\_share}\,[e]\,[i]$
        else
            $\mathsf{acc\_wit}, \mathsf{acc\_beav\_ab}, \mathsf{acc\_beav\_c} = \mathsf{acc}$
            $\mathsf{beav\_ab\_plain}\,[e] = \mathsf{acc\_beav\_ab} + \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}\,[e]\,[i]\,, 2dt\eta)$
            $\mathsf{beav\_c\_plain}\,[e] = \mathsf{beav\_c\_plain} \leftarrow \mathrm{InnerProducts}(\mathsf{beav\_ab\_plain})$ ▷ $a \cdot b = c$
            $\mathsf{aux}\,[e] = (\mathsf{wit\_plain} - \mathsf{acc\_wit}, \mathsf{beav\_c\_plain}\,[e] - \mathsf{acc\_beav\_c})$ ▷ $\mathsf{aux}\,[e] \in \mathbb{F}_q^{k+2w+t\eta}$
            $\mathsf{state}\,[e]\,[i] = (\mathsf{seed}\,[e]\,[i]\,, \mathsf{aux}\,[e]\,)$
        $\mathsf{com}\,[e]\,[i] = \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{state}\,[e]\,[i]\,)$
$h_1 = \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}\,[1]\,[1]\,, \ldots, \mathsf{com}\,[\tau]\,[2^D]\,)$
$(\mathsf{chal}\,[e]\,)_{e\in[1:\tau]} \leftarrow \mathrm{ExpandMPCChallenge}(h_1, \tau)$
for $e \in [1:\tau]$ **do**
    $\mathsf{input\_plain}\,[e] = (\mathsf{wit\_plain}, \mathsf{beav\_ab\_plain}\,[e]\,, \mathsf{beav\_c\_plain}\,[e]\,)$
    $\mathsf{broad\_plain}\,[e] \leftarrow \mathrm{ComputePlainBroadcast}(\mathsf{input\_plain}\,[e]\,, \mathsf{chal}\,[e]\,, (H', y))$
    for $p \in [1:D]$ **do**
        $\mathsf{broad\_share}\,[e]\,[p] = \mathrm{PartyComputation}(\mathsf{input\_mshare}\,[e]\,[p]\,, \mathsf{chal}\,[e]\,,$
            $(H', y), \mathsf{broad\_plain}\,[e]\,, \mathtt{False})$
            ▷ $\mathsf{broad\_share}\,[e]\,[p] \in \mathbb{F}_q^{(2d+1)t\eta}$
$h_2 = \mathrm{Hash}_2(m, \mathsf{salt}, h_1, \{\mathsf{broad\_plain}\,[e]\,, \{\mathsf{broad\_share}\,[e]\,[p]\,\}_{p\in[1:D]}\}_{e\in[1:\tau]})$.
$\{i^*\,[e]\,\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandViewChallenge}(h_2, 1)$.
for $e \in [1:\tau]$ **do**
    $\mathsf{path}\,[e] \leftarrow \mathrm{GetSeedSiblingPath}(\mathsf{rseed}\,[e]\,, i^*\,[e]\,)$.
    if $i^*\,[e] = 2^D$ **then**
        $\mathsf{view}\,[e] = \mathsf{path}\,[e]$
    else
        $\mathsf{view}\,[e] = (\mathsf{path}\,[e]\,, \mathsf{aux}\,[e]\,)$
$\sigma = \Big(\mathsf{salt} \mid h_2 \mid (\mathsf{view}\,[e]\,, \mathsf{broad\_plain}\,[e]\,, \mathsf{com}\,[e]\,[i^*\,[e]\,]\,)_{e\in[1:\tau]}\Big)$
return σ

Verification algorithm. The verification algorithm is described in Algorithm 11. It consists of the following steps:

1. First, it parses the signature.
2. Then, it expands the matrix H', the digest h_1 as the MPC challenge chal and the digest h_2 as the list of the open parties.
3. For each parallel repetition, it recomputes the $2^D - 1$ open leaf seeds from the sibling path of the TreePRG, and reconstructs the $2^D - 1$ leaf commitments of opened parties.
4. For each dimension, it aggregates the main share that is fully disclosed (the one that does not contain the hidden leaf), and computes its broadcast.
5. For each dimension, it deduces the broadcast share of the main party $(\wp, 0)$: either because $(\wp, 0)$ is the disclosed main party, or by difference with the plain broadcast if $(\wp, 1)$ is the disclosed main party.
6. It hashes together the broadcast shares of all main parties $(\wp, 0)$ for each repetition and each dimension, and match the digest with h_2.
7. The verification is valid if the reconstructed h_2 coincide with the ones parsed from the signature.

Algorithm 11 SD-in-the-Head – Hypercube Variant – Verification Algorithm

Input: a public key $pk = (\mathsf{seed}_H, y)$, a signature σ and a message $m \in \{0,1\}^*$

Parse σ as $\Big(\mathsf{salt} \mid h_2 \mid (\mathsf{view}[e], \mathsf{broad\_plain}[e], \mathsf{com}[e][i^*[e]])_{e\in[1:\tau]}\Big)$
$H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$ ▷ $H' \in \mathbb{F}_q^{(n-k)\times k}$
$\{i^*[e]\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandViewChallenge}(h_2, 1)$
for $e \in [1:\tau]$ **do**
    $(\mathsf{seed}[e][i])_{i\in[1:2^D\backslash i^*[e]]} \leftarrow \mathrm{GetLeavesFromSiblingPath}(i^*[e], \mathsf{salt}, \mathsf{path}[e])$
    for $i \in \{2^D \backslash i^*[e]\}$ **do**
        if $i \neq 2^D$ **then**
            $\mathsf{state}[e][i] = \mathsf{seed}[e][i]$
        else
            $\mathsf{state}[e][i] = (\mathsf{seed}[e][i], \mathsf{aux}[e])$ ▷ $\mathsf{aux}[e]$ is in $\mathsf{view}[e]$
        $\mathsf{com}[e][i] = \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{state}[e][i])$
$h_1 = \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}[1][1], \ldots, \mathsf{com}[\tau][2^D])$
$\mathsf{chal} \leftarrow \mathrm{ExpandMPCChallenge}(h_1, \tau)$
for $e \in [1:\tau]$ **do**
    $\mathsf{input\_mshare}^*[e][p] = 0$ for all $(e,p) \in [1:\tau] \times [1:D]$ ▷ $\mathsf{input\_mshare}'$ is main party share **not** containing i^*
    for $i \in [1:2^D \backslash i^*[e]]$ **do**
        if $i \neq 2^D$ **then**
            $\mathsf{input\_share}[e][i] \leftarrow \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][i], k + 2w + t(2d+1)\eta)$ ▷ $\mathsf{input\_share}[e][i] \in \mathbb{F}_q^{k+2w+t(2d+1)\eta}$
        else
            $\mathsf{beav\_ab\_plain}[e][2^D] = \mathrm{SampleFieldElements}(\mathsf{salt}, \mathsf{seed}[e][2^D], 2dt\eta)$
            $\mathsf{input\_share}[e][2^D] = (\mathsf{aux}[e] \mid \mathsf{beav\_ab\_plain}[e][2^D])$
        for $p \in [1:D]$: the p^{th} bit of $i-1$ and $i^*[e]$ are different **do**
            $\mathsf{input\_mshare}'[e][p]$ += $\mathsf{input\_share}[e][i]$ ▷ $\mathsf{input\_mshare}'$ does not contain i^*
    for $p \in [1:D]$ **do** ▷ Deduce the broadcasts of main party 0
        if the p^{th} bit of $i^*[e]$ is 1 **then**
            $\mathsf{broad\_share}[e][p] = \mathrm{PartyComputation}(\mathsf{input\_mshare}'[e][p], \mathsf{chal}, (H', y), \mathsf{broad\_plain}, \mathtt{False})$
        else
            $\mathsf{broad\_share}[e][p] = \mathsf{broad\_plain}[e] - \mathrm{PartyComputation}(\mathsf{input\_mshare}'[e][p], \mathsf{chal}, (H', y), \mathsf{broad\_plain}, \mathtt{True})$

▷ $\mathsf{broad\_share}[e][p] \in \mathbb{F}_q^{(2d+1)t\eta}$
$h_2' = \mathrm{Hash}_2(m, \mathsf{salt}, h_1, \{\mathsf{broad\_plain}[e], \{\mathsf{broad\_share}[e][p]\}_{p\in[1:D]}\}_{e\in[1:\tau]})$.
return $h_2 \stackrel{?}{=} h_2'$

3.5 Threshold variant

We describe the signing and verification algorithms for the threshold variant.

Signature algorithm. The signature algorithm is described in Algorithm 12. It consists of the following steps:

1. It expands the random part H' of the parity-check matrix from seed_H:

$$H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$$

2. It generates the pseudo-randomness used for the Beaver triples and the shares. This starts by sampling a 2λ-bit salt and a λ-bit master seed to initialize the XOF:

$\mathsf{salt} \leftarrow \{0,1\}^{2\lambda}$
$\mathsf{mseed} \leftarrow \{0,1\}^{\lambda}$
$\mathrm{XOF.Init}(\mathsf{salt} \parallel \mathsf{mseed})$

Since we target a negligible false positive probability p in this variant (see Section 2.3), the same plain Beaver triple $(\boldsymbol{a}, \boldsymbol{b}, c)$ is used across all the executions. The serialized random part of the triple denoted $\mathsf{beav\_ab\_plain}$ is randomly sampled, then the coordinate-wise inner product is computed to obtain $\mathsf{beav\_c\_plain}$. The plain input of the MPC protocol denoted $\mathsf{input\_plain}$ is then obtained by serializing $\mathsf{wit\_plain}$, $\mathsf{beav\_ab\_plain}$, and $\mathsf{beav\_c\_plain}$.

$\mathsf{beav\_ab\_plain} \leftarrow \mathrm{SampleFieldElements}(\mathsf{mseed}, 2dt\eta)$
$\mathsf{beav\_c\_plain} \leftarrow \mathrm{InnerProducts}(\mathsf{beav\_ab\_plain})$
$\mathsf{input\_plain} = (\mathsf{wit\_plain}, \mathsf{beav\_ab\_plain}, \mathsf{beav\_c\_plain})$

Since we use Shamir's secret sharing with threshold ℓ to share the serialized plain input $\mathsf{input\_plain}$, the algorithm samples ℓ $\mathbb{F}_q$-vectors $\mathsf{input\_coef}[e][1], \ldots, \mathsf{input\_coef}[e][\ell]$, for each parallel execution $e \in [1:\tau]$, where $|\mathsf{input\_coef}[e][j]| = |\mathsf{input\_plain}| = k + 2w + t(2d+1)\eta$. Those vectors will then be used to compute the parties' shares.

for $e \in [1:\tau]$ **do**
  for $j \in [1:\ell]$ **do**
    $\mathsf{input\_coef}[e][j] \leftarrow \mathrm{XOF.SampleFieldElements}(k + 2w + t(2d+1)\eta)$

3. It computes and commits the parties' shares. For each execution $e \in [1:\tau]$, we have a Merkle tree whose leaves are the commitments of the shares $\mathsf{com'}[e][i]$ and whose root is denoted $\mathsf{com}[e]$.

for $e \in [1:\tau]$ **do**
  for $i \in [1:N]$ **do**

$$\mathsf{input\_share}[e][i] = \begin{cases} \mathsf{input\_plain} + \sum_{j=1}^{\ell} f_i^j \cdot \mathsf{input\_coef}[e][j] & \text{when } i \neq N \\ \mathsf{input\_coef}[e][\ell] & \text{otherwise} \end{cases}$$

    $\mathsf{com'}[e][i] = \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{input\_share}[e][i])$
  $\mathsf{com}[e] = \mathrm{MerkleTree}(\mathsf{com'}[e][1], \ldots, \mathsf{com'}[e][N])$

4. It hashes the Merkle roots to obtain the first Fiat-Shamir hash h_1 and expands it as the MPC challenge chal.

$h_1 = \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}[1], \ldots, \mathsf{com}[\tau])$
$\mathsf{chal} \leftarrow \mathrm{ExpandMPCChallenge}(h_1, 1)$

The MPC challenge is a single serialize pair (r, ε) for the threshold variant which is why the second argument of ExpandMPCChallenge is set to 1.

5. It performs the MPC simulation. Since the plain input and the MPC challenge remain the same, the publicly recomputed values (*i.e.* plain values of the broadcast shares) are the same across all the executions.

   $\mathsf{broad\_plain} \leftarrow \mathrm{ComputePlainBroadcast}(\mathsf{input\_plain}, \mathsf{chal}, (H', y))$

   The algorithm then simulates the party computation which is done on the sharing randomness as explained in Section 2.3.

   for $e \in [1 : \tau]$ **do**
   for $j \in [1 : \ell]$ **do**
   $\mathsf{broad\_share}[e][j] = \mathrm{PartyComputation}(\mathsf{input\_coef}[e][j], \mathsf{chal}, (H', y), \mathsf{broad\_plain}, \mathtt{False})$

6. It hashes the broadcast messages to obtain the second Fiat-Shamir hash h_2 and expands it as the view-opening challenge $\{I[e]\}_{e \in [1:\tau]}$.

   $h_2 = \mathrm{Hash}_2(m, \mathsf{salt}, h_1, \mathsf{broad\_plain}, \mathsf{broad\_share})$.
   $\{I[e]\}_{e \in [1:\tau]} \leftarrow \mathrm{ExpandViewChallenge}(h_2, \ell)$.

7. After building the authentication paths for the revealed views, it finally assembles the signature.

   for $e \in [1 : \tau]$ **do**
   $\mathsf{auth}[e] \leftarrow \mathrm{GetMerklePath}(\mathsf{com}[e], I[e])$.
   $\mathsf{wit\_share}[e][i] \leftarrow \mathrm{Truncate}_{k+2w}(\mathsf{input\_share}[e][i])$ for all $i \in I[e]$
   $\sigma = \Big(\mathsf{salt} \mid h_1 \mid \mathsf{broad\_plain} \mid \mathsf{broad\_share} \mid ((\mathsf{wit\_share}[e][i])_{i \in I[e]} \mid \mathsf{auth}[e])_{e \in [1:\tau]}\Big)$

   As explained Section 2.3, we include $\mathsf{broad\_share}$ in the signature instead of the shares of the Beaver triples which simplifies the verification algorithm.

Algorithm 12 SD-in-the-Head – Threshold Variant – Signature Algorithm

Input: a secret key $sk = (\mathsf{seed}_H, y, \mathsf{wit\_plain})$ and a message $\mu \in \{0,1\}^*$
Output: a signature $\sigma = \mathsf{sign}(sk, \mu)$

▷ *Expansion of parity-check matrix*
$H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$ ▷ $H' \in \mathbb{F}_q^{(n-k)\times k}$

▷ *Randomness generation for the Beaver triples and the shares*
$\mathsf{salt} \leftarrow \{0,1\}^{2\lambda}$
$\mathsf{mseed} \leftarrow \{0,1\}^{\lambda}$
$\mathrm{XOF.Init}(\mathsf{salt} \parallel \mathsf{mseed})$
$\mathsf{beav\_ab\_plain} \leftarrow \mathrm{XOF.SampleFieldElements}(2dt\eta)$ ▷ $\mathsf{beav\_ab\_plain} \in \mathbb{F}_q^{2dt\eta}$
$\mathsf{beav\_c\_plain} \leftarrow \mathrm{InnerProducts}(\mathsf{beav\_ab\_plain})$ ▷ $\mathsf{beav\_c\_plain} \in \mathbb{F}_q^{t\eta}$
$\mathsf{input\_plain} = \mathrm{Serialize}(\mathsf{wit\_plain}, \mathsf{beav\_ab\_plain}, \mathsf{beav\_c\_plain})$ ▷ $\mathsf{input\_plain} \in \mathbb{F}_q^{k+2w+t(2d+1)\eta}$
for $e \in [1:\tau]$ **do**
  for $j \in [1:\ell]$ **do**
    $\mathsf{input\_coef}[e][j] \leftarrow \mathrm{XOF.SampleFieldElements}(k+2w+t(2d+1)\eta)$
    ▷ $\mathsf{input\_coef}[e][j] \in \mathbb{F}_q^{k+2w+t(2d+1)\eta}$

▷ *Computation and commitment of the shares*
for $e \in [1:\tau]$ **do**
  for $i \in [1:N]$ **do**
    $\mathsf{input\_share}[e][i] = \begin{cases} \mathsf{input\_plain} + \sum_{j=1}^{\ell} f_i^j \cdot \mathsf{input\_coef}[e][j] & \text{when } i \neq N \\ \mathsf{input\_coef}[e][\ell] & \text{otherwise} \end{cases}$
    $\mathsf{com}'[e][i] \leftarrow \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{input\_share}[e][i])$
  $(\mathsf{nodes}, \mathsf{com}[e]) \leftarrow \mathrm{MerkleTree}(\mathsf{salt}, \mathsf{com}'[e][1], \ldots, \mathsf{com}'[e][N])$

▷ *First challenge (MPC challenge)*
$h_1 \leftarrow \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}[1], \ldots, \mathsf{com}[\tau])$
$\mathsf{chal} \leftarrow \mathrm{ExpandMPCChallenge}(h_1, 1)$ ▷ $\mathsf{chal} \in \mathbb{F}_q^{(1+d)\cdot t\cdot\eta}$

▷ *MPC simulation*
$\mathsf{broad\_plain} \leftarrow \mathrm{ComputePlainBroadcast}(\mathsf{input\_plain}, \mathsf{chal}, (H', y))$ ▷ $\mathsf{broad\_plain} \in \mathbb{F}_q^{2dt\eta}$
for $e \in [1:\tau]$ **do**
  for $j \in [1:\ell]$ **do**
    $\mathsf{broad\_share}[e][j] = \mathrm{PartyComputation}(\mathsf{input\_coef}[e][j], \mathsf{chal}, (H', y), \mathsf{broad\_plain}, \mathtt{False})$
    ▷ $\mathsf{broad\_share}[e][j] \in \mathbb{F}_q^{(2d+1)t\eta}$

▷ *Second challenge (view-opening challenge)*
$h_2 \leftarrow \mathrm{Hash}_2(\mu, \mathsf{salt}, h_1, \mathsf{broad\_plain}, \mathsf{broad\_share})$. ▷ $\mathsf{broad\_share} \in \mathbb{F}_q^{\tau\cdot\ell\cdot(2d+1)t\eta}$
$\{I[e]\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandViewChallenge}(h_2, \ell)$.

▷ *Signature building*
for $e \in [1:\tau]$ **do**
  $\mathsf{auth}[e] \leftarrow \mathrm{GetMerklePath}(\mathsf{nodes}, I[e])$.
  $\mathsf{wit\_share}[e][i] \leftarrow \mathrm{Truncate}_{k+2w}(\mathsf{input\_share}[e][i])$ for all $i \in I[e]$
$\sigma = \Big(\mathsf{salt} \mid h_1 \mid \mathsf{broad\_plain} \mid \mathsf{broad\_share} \mid ((\mathsf{wit\_share}[e][i])_{i\in I[e]} \mid \mathsf{auth}[e])_{e\in[1:\tau]}\Big)$
return σ

Verification algorithm. The verification algorithm is described in Algorithm 13. It consists of the following steps:

1. It expands the random part H' of the parity-check matrix from seed_H:

$$H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$$

2. It parses the input signature:

$$\Big(\mathsf{salt} \mid h_1 \mid \mathsf{broad\_plain} \mid \mathsf{broad\_share} \mid ((\mathsf{wit\_share}\,[e]\,[i]\,)_{i\in I\,[e]} \mid \mathsf{auth}\,[e]\,)_{e\in[1:\tau]}\Big) \leftarrow \sigma$$

3. It recomputes the second Fiat-Shamir hash h_2 from the message μ and the components of the signature and regenerate the second challenge (view-opening challenge).

$$h_2 \leftarrow \mathrm{Hash}_2(\mu, \mathsf{salt}, h_1, \mathsf{broad\_plain}, \mathsf{broad\_share})$$
$$\{I\,[e]\,\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandViewChallenge}(h_2, \ell).$$

4. It recomputes the broadcast shares of the open parties, deduces the corresponding Beaver triples, and regenerates the share commitments (see explanation in Section 2.3). Specifically, for each execution e and for each open party $i \in I\,[e]$,
   - It computes the broadcast shares of the party using the open values $\mathsf{broad\_plain}$ and the sharing randomness $\mathsf{broad\_share}$.
   $$\mathsf{sh\_broadcast}\,[e]\,[i] = \begin{cases} (\mathsf{broad\_plain}, 0) + \sum_{j=1}^{\ell} f_i^j \cdot \mathsf{broad\_share}\,[e]\,[j] & \text{when } i \neq N \\ \mathsf{broad\_share}\,[e]\,[\ell] & \text{otherwise} \end{cases}$$
   - It reverses the MPC protocol for the considered party to get the share of the Beaver triples from which the complete input share is recomposed.
   $\mathsf{with\_offset} = (\texttt{True}$ if $i \neq N$, $\texttt{False}$ otherwise)
   $(\mathsf{beav\_ab\_share}, \mathsf{beav\_c\_share}) \leftarrow \mathrm{PartyComputationFromBroadcast}($
   $\mathsf{wit\_share}\,[e]\,[i]\,, \mathsf{sh\_broadcast}\,[e]\,[i]\,, \mathsf{chal},$
   $(H', y), \mathsf{broad\_plain}, \mathsf{with\_offset})$
   $\mathsf{input\_share}\,[e]\,[i] = (\mathsf{wit\_share}\,[e]\,[i]\,, \mathsf{beav\_ab\_share}, \mathsf{beav\_c\_share})$
   - It recomputes the commitment for this party as done in the signature algorithm.
   $\mathsf{com}'\,[e]\,[i] = \mathrm{Hash}_0(\mathsf{salt}, e, i, \mathsf{input\_share}\,[e]\,[i]\,)$

   Then for each execution e, it computes the root of the corresponding Merkle tree using the recomputed commitments of the opened parties as well as their authentication paths.

$$\mathsf{com}\,[e] = \mathrm{GetMerkleRootFromAuth}(\mathsf{auth}\,[e]\,, \{\mathsf{com}'\,[e]\,[i]\,\}_{i\in I\,[e]}, I\,[e]\,)$$

5. It recomputes the first Fiat-Shamir hash h_1 by hashing all the roots of the Merkle trees and checks that it is consistent with the signature.

$$h_1' \leftarrow \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}\,[1]\,, \ldots, \mathsf{com}\,[\tau]\,)$$
$$\textbf{return } h_1 \stackrel{?}{=} h_1'$$

Algorithm 13 SD-in-the-Head – Threshold Variant – Verification Algorithm

Input: a public key $pk = (\mathsf{seed}_H, y)$, a signature σ and a message $\mu \in \{0,1\}^*$
Output: `True` if σ is a valid signature of μ under pk and `False` otherwise

▷ *Expansion of parity-check matrix*
$H' \leftarrow \mathrm{ExpandH}(\mathsf{seed}_H)$ ▷ $H' \in \mathbb{F}_q^{(n-k)\times k}$
▷ *Signature parsing*
$\Big(\mathsf{salt} \mid h_1 \mid \mathsf{broad\_plain} \mid \mathsf{broad\_share} \mid ((\mathsf{wit\_share}[e][i])_{i\in I[e]} \mid \mathsf{auth}[e])_{e\in[1:\tau]}\Big) \leftarrow \sigma$
▷ *First challenge (MPC challenge)*
$\mathsf{chal} \leftarrow \mathrm{ExpandMPCChallenge}(h_1, 1)$ ▷ $\mathsf{chal} \in \mathbb{F}_q^{(1+d)\cdot t\cdot \eta}$
▷ *Second challenge (view-opening challenge)*
$h_2 \leftarrow \mathrm{Hash}_2(\mu, \mathsf{salt}, h_1, \mathsf{broad\_plain}, \mathsf{broad\_share})$
$\{I[e]\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandViewChallenge}(h_2, \ell)$.
▷ *Party computation and regeneration of Merkle commitments*
for $e \in [1:\tau]$ **do**
  for $i \in I[e]$ **do**
    $\mathsf{sh\_broadcast}[e][i] = \begin{cases} (\mathsf{broad\_plain}, 0) + \sum_{j=1}^{\ell} f_i^j \cdot \mathsf{broad\_share}[e][j] & \text{when } i \neq N \\ \mathsf{broad\_share}[e][\ell] & \text{otherwise} \end{cases}$
    $\mathsf{with\_offset} =$ (`True` if $i \neq N$, `False` otherwise)
    $(\mathsf{beav\_ab\_share}, \mathsf{beav\_c\_share}) \leftarrow \mathrm{PartyComputationFromBroadcast}($
      $\mathsf{wit\_share}[e][i], \mathsf{sh\_broadcast}[e][i], \mathsf{chal},$
      $(H', y), \mathsf{broad\_plain}, \mathsf{with\_offset})$
    $\mathsf{input\_share}[e][i] = (\mathsf{wit\_share}[e][i], \mathsf{beav\_ab\_share}, \mathsf{beav\_c\_share})$
    $\mathsf{com}'[e][i] = \mathrm{Hash}_0(\mathsf{salt}, e, i, \mathsf{input\_share}[e][i])$
  $\mathsf{com}[e] = \mathrm{GetMerkleRootFromAuth}(\mathsf{auth}[e], \{\mathsf{com}'[e][i]\}_{i\in I[e]}, I[e])$
  if $\mathsf{com}[e] =$ `invalid` **then**
    return `False`
▷ *Regeneration and verification of* h_1
$h_1' \leftarrow \mathrm{Hash}_1(\mathsf{seed}_H, y, \mathsf{salt}, \mathsf{com}[1], \ldots, \mathsf{com}[\tau])$
return $h_1 \stackrel{?}{=} h_1'$

4 Signature parameters

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 the SD parameters

To select the parameters relative to the syndrome decoding problem, we estimate the cost of the best-known algorithms to solve this problem. There exist two main families of such algorithms: the *Information Set Decoding* (ISD) algorithms and the *Generalized Birthday Algorithms* (GBA) [TS16; BBC+19]. The SD parameters are chosen such that both types of SD-solving algorithms have complexity of at least 2^κ corresponding to the complexity of breaking AES by exhaustive search (in the gate-count metric). In practice, we take κ equal to 143, 207 and 272 respectively for categories I (AES-128), III (AES-192) and V (AES-256) in accordance to [NIS22].

We chose to focus on syndrome decoding instances relying on the fields $\mathbb{F}_{251}$ and $\mathbb{F}_{256}$. Fields with up to 256 elements yield signature sizes close to the optimal. Moreover, for those fields, an element can be stored on a byte. The field $\mathbb{F}_{256}$ is particularly convenient to sample field elements and its arithmetic can be efficiently implemented on the platforms supporting carry-less multiplications. The field $\mathbb{F}_{251}$ is more convenient on platforms without carry-less multiplications and it might be less sensitive to future attacks exploiting the structure of the syndrome decoding problem on an extension field such as $\mathbb{F}_{256}$.

The remaining SD parameters (the code length m, the code dimension k and the weight w) are chosen to meet the desired security category while minimizing the signature size. For a given code length m and dimension k, the weight parameter is defined such that the number of expected solutions is below 1.01. Moreover, since the SD-in-the-Head protocol requires to have $m \leq q$ (to enable interpolation of a $(m-1)$-degree polynomial on $\mathbb{F}_q$), we use the d-split variant of the SD problem whenever necessary: we split the SD solution x (or s) into d chunks which have independent weight constraints. This relaxes the constraint between m and q as $\frac{m}{d} \leq q$.[3] In practice, we can rely on standard SD instances for Category I, and we need to rely on 2-split SD instances for Categories III and V.

The analysis of the existing attacks against the syndrome decoding problem is described in Section 7. The SD parameters we proposed are common for both variants (hypercube and threshold) and are detailed hereafter in Table 4.

4.2 Selection of the MPC parameters

For the hypercube variant, we take the hypercube dimension D equal to 8 (*i.e.* $N = 2^8$) to achieve running times around a few milliseconds while keeping short signatures. For the threshold variant, we take the maximal number N of parties allowed by the base field, namely $N = q$ (recalling that the number of parties is at most the size of the field for this variant). The signature size is then minimized for privacy threshold $\ell = 1$. However, this choice of ℓ induces an important computational overhead for commitments, thus we choose $\ell = 3$ which provides a good trade-off between signature size and running times.

[3] Another option would be to consider field extension of $\mathbb{F}_q$ for the polynomial, but this approach results in worst performances and further prevents using the tweak to avoid interpolations (see the end of Section 2.1).

It remains to select the number t of evaluation points in the MPC protocol, the field extension $\mathbb{F}_{\text{points}} = \mathbb{F}_{q^\eta}$ for the evaluation points and the number τ of repetitions. The two first parameters impact the false positive probability of the MPC protocol. We chose to have a common extension field for $\mathbb{F}_{\text{points}}$ for the two variants (hypercube and threshold) and the three security categories in order to allow a unique (optimized) implementation of the underlying extension field arithmetic. In practice, we select the degree-4 extension field $\mathbb{F}_{\text{points}} = \mathbb{F}_{q^4}$ which represents a good compromise between the different settings. Then the parameters t and τ are chosen such that the signature size is minimal while having a forgery cost larger than λ bits, when λ is 128, 192 and 256 respectively for the categories I, III and V. Currently, the best forgery attack is obtained by applying the approach of [KZ20b] and its cost is given by:

$$\text{cost}_{\text{forge}} := \min_{\tau_1,\tau_2:\tau_1+\tau_2=\tau} \left\{ \frac{1}{\sum_{i=\tau_1}^{\tau} \binom{\tau}{i} p^i (1-p)^{\tau-i}} + \left(\frac{1}{p'}\right)^{\tau_2} \right\},$$

where p is the false positive probability of the SD-in-the-Head MPC protocol (see Theorem 2.1) and p' is the probability to guess the open parties for a repetition, namely

$$p' := \begin{cases} \frac{1}{N} & \text{for the hypercube variant,} \\ \frac{1}{\binom{N}{\ell}} & \text{for the threshold variant.} \end{cases}$$

Field representations. For $q = 256$, we use the following field representation:

$$\mathbb{F}_q \equiv \mathbb{F}_2[X]/f_0(X) \text{ with } f_0(X) = X^8 + X^4 + X^3 + X + 1$$

The elements of $\mathbb{F}_{256}$ are stored on bytes in integer form. Namely, $\sum_{i=0}^{7} a_i X^i \in \mathbb{F}_{256}$ is represented by the integer $a = \sum_{i=0}^{7} a_i 2^i$, which is denoted $(a) \in \mathbb{F}_{256}$. For instance $(2) = X \in \mathbb{F}_{256}$. On the other hand, the elements of $\mathbb{F}_{251}$ are naturally represented as integers in the interval $[0 : 250]$ which are also stored on bytes.

As explained above, the extension field $\mathbb{F}_{\text{points}}$ used for the evaluation points in the MPC protocol is always defined as an extension of degree $\eta = 4$ of the base field, *i.e.* $\mathbb{F}_{\text{points}} = \mathbb{F}_{q^4}$. For $q = 256$, this extension is defined as:

$$\mathbb{F}_{q^2} \equiv \mathbb{F}_q[Y]/f_1(Y) \text{ with } f_1(Y) = Y^2 + Y + (32)$$
$$\mathbb{F}_{q^4} \equiv \mathbb{F}_{q^2}[Z]/f_2(Z) \text{ with } f_2(Z) = Z^2 + Z + (32)X$$

For $q = 251$, the field extension is defined as:

$$\mathbb{F}_{q^2} \equiv \mathbb{F}_q[Y]/f_1(Y) \text{ with } f_1(Y) = Y^2 - 2$$
$$\mathbb{F}_{q^4} \equiv \mathbb{F}_{q^2}[Z]/f_2(Z) \text{ with } f_2(Z) = Z^2 - (X + 1)$$

4.3 Symmetric cryptography primitives

The SD-in-the-Head signature scheme relies on two types of symmetric cryptography primitives: a hash function (Hash) which we instantiate with SHA3 [Dwo15], and an extendable output function (XOF) which we instantiate with SHAKE [Dwo15]. Table 3 summarizes the instances for each security category.

We recall here the usage of these symmetric primitives in the SD-in-the-Head signature scheme (see Section 3 for details):

Table 3: Symmetric cryptography primitives for NIST Security Categories I, III, and V.

	Category I	Category III	Category V
Hash	SHA3-256	SHA3-384	SHA3-512
XOF	SHAKE-128	SHAKE-256	SHAKE-256

- Hash is used for
  - the commitments,
  - the Fiat-Shamir hashes h_1 and h_2,
  - the nodes of the seed trees (hypercube variant only),
  - the nodes of the Merkle trees (threshold variant only).
- XOF is used for
  - the expansion of the parity-check matrix H,
  - the expansion of the hashes h_1 and h_2 into MPC challenges,
  - the expansion of the seeds (key generation and hypercube variant),
  - the expansion of the shares from the seeds (hypercube variant only).

4.4 Keys and signature sizes

The hypercube and threshold approaches share the same key generation procedure, originally from [FJR22], described in Algorithm 9. However, their signature and verification algorithms differ and thus their signature format (and sizes) will also be different.

Public key. The public key has format $pk := (\mathsf{seed}_H, y)$; consisting of a λ-bit seed seed_H representing the linear code of the SD instance, and a vector $y \in \mathbb{F}_q^{m-k}$ corresponding to the syndrome. Since we represent a field element by a byte, the public key has a total size of $\lambda/8 + (n-k)$ bytes.

Secret key. The secret key has format $sk := (\mathsf{seed}_H, y, \mathsf{wit\_plain})$; consisting of the same seed_H and y as the public key, as well as the witness $\mathsf{wit\_plain} = (s_A, \boldsymbol{Q}, \boldsymbol{P})$. The latter is made up of a vector $s_A \in \mathbb{F}_q^k$ and two polynomials $\boldsymbol{Q}, \boldsymbol{P} \in \mathbb{F}_q^w$. Thus, the size of the secret key is $|pk| + k + 2w := \lambda/8 + n + 2w$ bytes.

As all the existing public-key schemes, let us remark that we have an alternative definition of the key generation in which the secret key would be $\mathsf{seed}_{\mathsf{root}}$, the seed from which $(\mathsf{seed}_H, y, \mathsf{wit\_plain})$ are derived. In that case, the size of the secret key would be of $\lambda/8$ bytes, but the signer would need to recompute $\mathsf{wit\_plain}$ at each signature, increasing the running time of the signing process. Moreover, the signature algorithm would be more sensitive to side-channel attacks. We recommend to use this alternative *only when the size of the secret key is critical.*

Signature size for the hypercube variant. The size (in bits) of a signature is:

$$\begin{aligned}
\text{Total Size} = {} & 2\lambda && \rightarrow \text{size of the } \mathsf{salt} \\
& + 2\lambda && \rightarrow \text{size of } h_2 \\
& + \tau \cdot (k \cdot \log_2(|\mathbb{F}_q|) + 2w \cdot \log_2(|\mathbb{F}_q|)) && \rightarrow \text{size of } \mathsf{aux}[e] \text{ in } \mathsf{view}[e] \\
& + \tau \cdot (2d+1) \cdot t \cdot \log_2(|\mathbb{F}_{\text{points}}|) && \rightarrow \text{size of } \mathsf{broad_plain}[e] \\
& + \tau \cdot \lambda \cdot \log_2(N) && \rightarrow \text{size of } \mathsf{path}[e] \text{ in } \mathsf{view}[e] \\
& + \tau \cdot 2\lambda && \rightarrow \text{size of } \mathsf{com}[e]\,[i^*[e]]
\end{aligned}$$

Assuming that the field element in $\mathbb{F}_q$ is represented by a byte, the signature size (in bytes) is given by:

$$|\sigma| = \frac{\lambda}{2} + \tau \cdot \left(k + 2w + (2d+1) \cdot t \cdot \eta + \frac{\lambda}{8} \cdot \log_2(N) + \frac{\lambda}{4}\right)$$

We present the hypercube signature parameters, with associated key and signature sizes, in Table 5.

Signature size for the threshold variant. The size (in bits) of a signature is:

$$\begin{aligned}
\text{Total Size} = {} & 2\lambda && \rightarrow \text{size of the } \mathsf{salt} \\
& + 2\lambda && \rightarrow \text{size of } h_1 \\
& + 2d \cdot t \cdot \log_2(|\mathbb{F}_{\text{points}}|) && \rightarrow \text{size of } \mathsf{broad_plain} \\
& + \tau \cdot \ell \cdot (k \cdot \log_2(|\mathbb{F}_q|) + 2w \cdot \log_2(|\mathbb{F}_q|)) && \rightarrow \text{size of } (\mathsf{wit_share}[e]\,[i])_{i \in I[e]} \\
& + \tau \cdot \ell \cdot (2d+1) \cdot t \cdot \log_2(|\mathbb{F}_{\text{points}}|) && \rightarrow \text{size of } \mathsf{broad_share} \\
& + \tau \cdot \ell \cdot 2\lambda \cdot \log_2(N/\ell) && \rightarrow \text{size of } \mathsf{auth}[e]
\end{aligned}$$

Assuming that the field element in $\mathbb{F}_q$ is represented by a byte, the signature size (in bytes) is given by:

$$|\sigma| = \frac{\lambda}{2} + 2d \cdot t \cdot \eta + \tau \cdot \ell \cdot \left(k + 2w + (2d+1) \cdot t \cdot \eta + \frac{\lambda}{4} \cdot \log_2(N/\ell)\right)$$

We present the threshold signature parameters, with associated key and signature sizes, in Table 5.

4.5 Proposed instances

The signature parameters of our proposed instances are summarized in Table 4 and in Table 5 for the different security categories, the two base fields and the two variants (hypercube and threshold). Table 4 gives the syndrome decoding parameters which are common to both variants while Table 5 gives the MPCitH parameters and obtained sizes. We assume that a field element is represented on one byte and hence make Table 5 field-agnostic.

Table 4: Syndrome decoding parameters for both SD-in-the-Head variants for NIST Security Categories I, III, and V.

Parameter Sets	**NIST Security**		**SD Parameters**				
	Category	Bits	q	m	k	w	d
SDitH-L1-gf256	I	143	256	242	126	87	1
SDitH-L1-gf251	I	143	251	242	126	87	1
SDitH-L3-gf256	III	207	256	376	220	114	2
SDitH-L3-gf251	III	207	251	376	220	114	2
SDitH-L5-gf256	V	272	256	494	282	156	2
SDitH-L5-gf251	V	272	251	494	282	156	2

Table 5: The hypercube parameters and the threshold parameters, with key and signature sizes in bytes. The sizes are the same for both fields ($\mathbb{F}_{251}$ and $\mathbb{F}_{256}$), assuming each field element is represented on one byte.

Parameter Set	**MPCitH Parameters**						**Sizes (in bytes)**			
	N	ℓ	τ	η	t	p	pk	sk	Sig. Avg	Sig. Max
SDitH-L1-hyp	2^8	–	17	4	3	$2^{-70.6}$	132	432	8 476	8 496
SDitH-L3-hyp	2^8	–	26	4	3	$2^{-71.8}$	180	628	19 498	19 544
SDitH-L5-hyp	2^8	–	34	4	4	$2^{-94.2}$	244	838	33 843	33 924
SDitH-L1-thr	q	3	6	4	7	$2^{-164.7}$	132	432	10 382	10 684
SDitH-L3-thr	q	3	9	4	10	$2^{-239.5}$	180	628	25 277	25 964
SDitH-L5-thr	q	3	12	4	13	$2^{-306.0}$	244	838	44 460	45 676

5 Performances

Benchmark platform. Intel Xeon E-2378 at 2.6GHz. All the measurements were performed with Turbo Boost disabled. The scheme has been compiled with Clang compiler (version 15.0.7).

5.1 Benchmarks for the hypercube variant

Benchmarks for an optimized implementation of the hypercube variant on an AVX2 machine are given in Table 6.

Table 6: Benchmark of variants based on the hypercube approach on an AVX2 machine. Timings and cycles were collected on an Intel Xeon E-2378 at 2.6GHz while disabling Intel Turbo Boost.

Instance	KeyGen		Sign		Verify		RAM
	ms	cycles	sign ms	cycles	verify ms	cycles	
SDitH-gf256-L1-hyp	5.47	14.2M	4.18	10.8M	3.74	9.7M	370KB
SDitH-gf256-L3-hyp	6.41	16.6M	10.13	26.2M	8.83	22.9M	859KB
SDitH-gf256-L5-hyp	11.06	28.7M	19.25	49.9M	16.98	44.0M	1.5MB
SDitH-gf251-L1-hyp	3.05	7.9M	8.17	21.2M	7.83	20.3M	371KB
SDitH-gf251-L3-hyp	3.67	9.5M	17.98	46.6M	17.08	44.3M	861KB
SDitH-gf251-L5-hyp	6.36	16.5M	32.73	84.8M	31.26	81.0M	1.5MB

5.2 Benchmarks for the threshold variant

Benchmarks for an optimized implementation of the threshold variant on an AVX2 machine are given in Table 7.

Table 7: Benchmark of variants based on the threshold approach on an AVX2 machine. Timings and cycles were collected on an Intel Xeon E-2378 at 2.6GHz while disabling Intel Turbo Boost.

Instance	KeyGen		Sign			Verify		
	ms	cycles	sign ms	cycles	RAM	verify ms	cycles	RAM
SDitH-gf256-L1-thr	1.67	4.3M	2.47	6.4M	199KB	0.84	2.2M	50KB
SDitH-gf256-L3-thr	1.99	5.2M	6.25	16.2M	395KB	2.20	5.7M	96KB
SDitH-gf256-L5-thr	3.48	9.0M	12.61	32.7M	670KB	4.46	11.6M	173KB
SDitH-gf251-L1-thr	0.62	1.6M	1.78	4.6M	197KB	0.25	0.6M	50KB
SDitH-gf251-L3-thr	0.76	2.0M	4.29	11.1M	392KB	0.59	1.5M	96KB
SDitH-gf251-L5-thr	1.40	3.6M	8.75	22.7M	664KB	1.25	3.2M	173KB

6 Security Analysis

6.1 Security definition

The SD-in-the-Head signature scheme is claimed to achieve *unforgeability against chosen message attacks* (EUF-CMA). In this setting, the adversary is given a randomly generated public key pk and they can ask an oracle (called the *signature oracle*) to sign messages $(m_1, \ldots, m_r)$ that they can chose at will. The goal of the adversary is to produce a pair (m, σ) such that m is not one of the requests to the signature oracle and such that σ is a valid signature of m with respect to pk.

6.2 Security assumptions

Our main security assumption is that our proposed SD instances cannot be solved in complexity lower than 2^κ (in gate-count metric) with $\kappa = 143$ for Category-I instances, $\kappa = 207$ for Category-III instances, and $\kappa = 272$ for Category-V instances. This assumption is tailored with respect to state-of-the-art algorithms for solving syndrome decoding (see Section 7).

We further assume that the used XOF is a secure pseudorandom generator with 128-bit security level for Category-I instances, 192-bit security level for Category-III instances, and 256-bit security level for Category-V instances.

Finally, we assume that the used hash function *behaves as a random function.* Namely, our security results hold in the Random Oracle Model (ROM) and the Quantum Random Oracle Model (QROM).

6.3 Security in the ROM

We refer the reader to [FJR22; FR22; AMGH+23] for security proofs of the SD-in-the-Head scheme in the random oracle model. Soundness of the underlying IDS and the HVZK property are separately proven, and then the EUF-CMA security is proven via a series of game hops from a signing oracle to an HVZK simulator which is efficiently simulable by any spectator who knows pk. In [AMHJ+23] a further detail, multi-transcript HVZK is proven to cover the case where transcripts are parallel composed via the Fiat Shamir transform.

6.4 Security in the QROM

In a recent work [AMHJ+23], the authors provide a proof of security against EUF-CMA for the hypercube variant in the QROM. The proof proceeds by presenting an argument that the five round protocol presented in [AMGH+23] can in fact be interpreted as a three round sigma protocol. Mechanically the schemes are near-identical (save for a small optimization that has no bearing on security in the QROM).

Instead of getting challenge points from the verifier, the prover derives the evaluation points via a hash and PRG procedure. Breaking the scheme hence reduces to a search problem over the space of evaluation (and mask) points in order to find points $\{r_i, \varepsilon_i\}$ such that the predicate $S \cdot Q = P \cdot F$ is not satisfied, yet is true when evaluated at those points, i.e. $S(r_i)Q(r_i) = P(r_i)F(r_i)$. The cost of cheating at this stage in the QROM is equivalent to the cost of Grover search.

What results is a three round sigma protocol with a single Fiat Shamir transform, for which the authors can straightforwardly apply the results of [DFM+22] in which the security of

commit-and-open schemes is analyzed, but only for three round schemes. The security parameters for parallel repetition must still be subject to the Kales-Zaverucha (KZ) attack [KZ20a], as despite the three round presentation, the choice of how many parallel repetitions on which to break the polynomial check is still at the discretion of the attacker, with the remaining repetitions broken by correctly guessing challenge parties. The cost of such an attack must be greater than 2^{λ} where λ is the security parameter corresponding to the NIST security level.

The approach of this recent work is directly applicable to the threshold approach too, though it is not explicitly analyzed in this work.

6.5 Security of the d-split syndrome decoding problem

We stress that the d-split SD problem which is used in some instances of our signature scheme (see Section 4) does not rely on a different weaker assumption than the standard SD problem. Any d-split SD instance is at least as secure as a standard SD instance with slightly degraded parameters. This degradation is formally given by the following theorem

Theorem 6.1 ([FJR22])**.** *Let $\mathbb{F}$ be a finite field. Let m, k, w be positive integers such that $m > k$, $m > w$, $d \mid w$ and $d \mid m$. Let $\mathcal{A}_d$ be an algorithm which solves a random $(\mathbb{F}, m, k, w)$-instance of the d-split syndrome decoding problem in time t with success probability ε_d. Then there exists an algorithm $\mathcal{A}_1$ which solves a random $(\mathbb{F}, m, k, w)$-instance of the standard syndrome decoding problem in time t with probability ε_1, where*

$$\varepsilon_1 \geq \frac{\binom{m/d}{w/d}^d}{\binom{m}{w}} \cdot \varepsilon_d \ .$$

Informally, the above result holds because an instance of the standard SD problem is an instance of the d-split syndrome decoding problem with probability $\binom{m/d}{w/d}^d / \binom{m}{w}$. Moreover, a standard syndrome decoding instance can be "randomized" and input to the d-split adversary as much as desired.

All our instances of the d-split SD problem are chosen such that the corresponding standard SD instance achieves the upgraded security level which compensates the degradation. We stress that this might be overly conservative though it does not have a strong impact on performances for the chosen parameters.

7 Analysis of known attacks

7.1 Attacks against the SD problem

To begin, we recall the syndrome decoding problem: given $H \in \mathbb{F}_q^{(m-k)\times m}$ and $y \in \mathbb{F}_q^{m-k}$, the goal is to find $x \in \mathbb{F}_q^m$ such that $\mathrm{wt}(x) = w$ and $Hx = y$.

The most obvious attack is to guess the vector x by brute force. Since there are $\binom{m}{w}q^w$ possible such vectors (and only one solution), this is not feasible for standard parameters such as ours. The next best thing, then, is trying to make "educated guesses", using the information at our disposal. This idea was first fleshed out by Prange [Pra62], which kickstarted a long and fertile line of work yielding what is now known as the *Information-Set Decoding (ISD)* family of algorithms.

In its simplest form, an information-set decoding algorithm is an iterative procedure, where in each step one guesses a set I of k indexes (the information set). Let $J = [1, \ldots, m] \setminus I$, i.e. the set of columns *not* indexed by I. A successful step requires that the submatrix H_J is invertible and the support of x is entirely contained in J. Indeed, if this is the case, then x can be found by computing $x_J = H_J^{-1}y$ and setting the other entries of x to 0. This attack can equivalently be expressed using the dual code, using a generator matrix G instead of H.

Note that the original idea of Prange does not admit errors in the information set. An early improvement due to Lee and Brickell [LB88] changes this by allocating p positions to the coordinates in I, and the remaining $w - p$ to the coordinates to J; in other words, this variant works by searching for a weight-p word in I and checking the weight using the remaining columns. This idea was expanded by Leon [Leo88] and Krouk [Kor89], and yields a noticeable speed-up over the original approach.

A successive work by Stern [Ste89] brings in new ideas that result in further efficiency gain. Originally, Stern's algorithm was intended to solve a different problem, namely, that of finding a low-weight codeword (and particularly, a minimum-weight codeword). However, such a technique can intuitively be used to solve SDP as well. Indeed, if $\mathcal{C}$ is the linear code of length n defined by H, and y is a word at distance w from a codeword $c \in \mathcal{C}$, then x is the minimum-weight codeword in $\mathcal{C} \oplus \langle y \rangle$. Stern builds on the previous ideas of splitting the positions among various coordinates (as in Lee-Brickell, Leon, Krouk) in an even more sophisticated way: the algorithm partitions the information set into subsets X and Y, allocating p positions in each, and the remaining $w - 2p$ in J (with the exception of a set of columns set to 0, of size corresponding to a certain parameter ℓ). The algorithm is able to improve over its predecessors by exploiting *collisions* among words in the respective coordinates, and additionally by choosing each column adaptively as a result of pivots in previous, thus avoiding the cost of a restart and guaranteeing an invertible submatrix.

The literature is rich with several successive works (e.g. [BLP11; MMT11; BJM+12]), all bringing in their own improvement; however, for our purposes, it is sufficient to stop at Stern's algorithm. In fact, the majority of such successive improvements are tailored to the binary case (i.e. $q = 2$): for instance, the representation technique dubbed "$1 + 1 = 0$" in [BJM+12]. As noted by Meurer in [Meu13], sophisticated algorithms become significantly less powerful for large values of q. It is worth noting also that such algorithms also come with an increasingly heavy memory cost, so much that, in practice, it is debatable whether they truly represent the best options, for parameters of cryptographic interest, over the early ISD variants, even in the binary case.

An adaptation of Lee-Brickell to the generic q-ary case is straightforward, easy to understand, and useful as a loose estimation of the complexity, for given parameters. On the other hand, the

work of Peters [Pet10] provides an efficient adaptation of Stern's algorithm to $\mathbb{F}_q$, incorporating all relevant improvements, and providing a clear security assessment.
The cost at each iteration is given by

$$C_{iter} = \frac{1}{2}(n-k)^2(n+k) + \left(\left(\frac{k}{2}-p+1\right) + \left(\binom{\lfloor k/2 \rfloor}{p} + \binom{k-\lfloor k/2 \rfloor}{p}\right)(q-1)^p\right)\ell + \frac{q}{q-1}(w-2p+1)2p\left(1+\frac{q-2}{q-1}\right)\frac{\binom{\lfloor k/2 \rfloor}{p}\binom{k-\lfloor k/2 \rfloor}{p}(q-1)^{2p}}{q^\ell},$$

whereas the probability of success at each step is given by

$$p_{succ} = \binom{\lfloor k/2 \rfloor}{p}\binom{k-\lfloor k/2 \rfloor}{p}\binom{n-k-\ell}{w-2p}\Big/\binom{n}{w}.$$

The total cost of ISD can then be obtained as

$$\frac{C_{iter}\cdot \log_2 q}{p_{succ}}.$$

As shown in [NPC+17], it is possible to employ some dedicated techniques to provide a small gain. The authors fully exploit the field structure by computing multiples of the target syndrome, of the form αy for $\alpha \in \mathbb{F}_q^*$, and then looking for solutions x'; clearly, if $Hx' = \alpha y$, then $x' = \alpha x$. Their technique allows to gain a factor of $\sqrt{q-1}$ [NPC+17].

7.2 Signature forgery attacks

When we apply the Fiat-Shamir transformation to a zero-knowledge proof of knowledge, there is a security drop. The forgery cost of the obtained signature scheme can be lower than $\frac{1}{\varepsilon}$, where ε is the soundness error of the original proof of knowledge. The best forgery attack against FS-based schemes with several parallel repetition ($\tau > 1$) is currently the attack of [KZ20b].

Hypercube approach Given a parameter $\tau^* \in \{0, \ldots, \tau\}$, the [KZ20b]'s forgery attack aims to build commitments for the τ parallel executions such that the corresponding first challenges (the MPC challenges) lead to a valid verification for at least τ^* executions. More precisely, the adversary tries to guess the τ challenges $(\mathsf{chal}_1'[e])_{e\in[1:\tau]}$, builds the corresponding commitments, computes h_1 and expands the real challenge $(\mathsf{chal}_1[e])_{e\in[1:\tau]}$. She repeats this process until $\mathsf{chal}_1'[e] = \mathsf{chal}_1[e]$ for τ^* executions. It will be repeated in average $\mathsf{PMF}(\tau,\tau^*,p) := \sum_{k=\tau^*}^{\tau}\binom{\tau}{k}p^k(1-p)^{\tau-k}$ times before being successful, where p is the false positive rate of the MPC protocol (see Theorem 2.1). Then, the adversary simply needs to guess the view challenges for the $\tau - \tau^*$ remaining executions and build the corresponding responses. Since the size of the second challenge set is N per execution, she will try in average $N^{\tau-\tau^*}$ times before guessing correctly all the view challenges. At the end, the forgery cost of the attack is the cost for the optimal τ^*:

$$\mathsf{cost}_{forgery} = \max_{\tau^*}\left\{\frac{1}{\mathsf{PMF}(\tau,\tau^*,p)} + N^{\tau-\tau^*}\right\}.$$

The scheme parameters proposed in Table 5 have been chosen such that the associated forgery cost is at least of 128-bits for Category I, of 192-bits for Category III and of 256-bits for Category V.

Threshold approach We can apply the [KZ20b]'s attack to the threshold approach. However, there is two variations:

- the MPC challenge (*i.e.* the first challenge) is the same for all the τ parallel executions, thus instead of having $\tau + 1$ strategies (for τ^*), we only have two:
  1. either the adversary tries to guess this unique MPC challenge,
  2. or she only focus on the view challenges (*i.e.* the second challenges) of the τ executions.
- As explained in [FR22], the adversary is not forced to commit a valid sharing for each execution. With the strategy 2 where she tries to guess the view challenges, committing an invalid sharing will not help to decrease the attack cost. The strategy 1 aims to produce some false positive in the MPC protocol. We can lower bound the cost of this strategy by the cost to obtain a false positive for a single witness encoded by a subset of $\ell + 1$ shares among N (in at least one of the τ parallel executions). The latter cost is given by

$$\frac{1}{1-(1-p)^{\tau\cdot\binom{N}{\ell+1}}} \approx \frac{1}{\tau\cdot\binom{N}{\ell+1}\cdot p},$$

  where p is the false positive rate of the MPC protocol (see Theorem 2.1)

Thus, when we adapt [KZ20b]'s attack to our scheme variant relying on the threshold approach, we get a forgery cost which is lower bounded by

$$\max\left\{\frac{1}{1-(1-p)^{\tau\cdot\binom{N}{\ell+1}}}, \binom{N}{\ell}^{\tau}\right\}.$$

The scheme parameters proposed in Table 5 have been chosen such that this lower bound is at least of 128-bits for Category I, of 192-bits for Category III and of 256-bits for Category V.

8 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. There are more specific advantages, and also limitations, in which we discuss hereafter.

8.1 Advantages of SD-in-the-Head

Conservative hardness assumption. Our signature scheme is based on the presumably hardest problem in code-based cryptography: the 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. We also utilize two conservative base finite fields to define our instances, namely $\mathbb{F}_{251}$ and $\mathbb{F}_{2^8}$, which give better performances (compared to $\mathbb{F}_2$) and are still expected to provide high security.

Adaptive and tunable parameters. Instead of using permutations like most of the previous zero-knowledge protocols for syndrome decoding, we rely on the MPC-in-the-head (MPCitH) paradigm in which the task of proving the low Hamming weight of the SD solution is reduced to proving some relations between specific polynomials. Using MPCitH enables us to tailor parameters, in particular the number of parties, meaning (like SPHINCS$^+$) that we can provide a variety of parameter sets tailored to different use cases. Although this specification targets small signature sizes, it is possible to increase the number of MPC parties (giving smaller signatures at the cost of slower timings) or decreasing the number of MPC parties (giving faster performance at the cost of larger signatures). We can also support different base fields (such as, e.g., $\mathbb{F}_2$).

We propose two variants of the SD-in-the-Head signature scheme which provide different trade-offs between efficiency and signature size. Our *hypercube variant* allows us to increase the number of parties while mitigating the computational overhead and thus obtaining a smaller signature size. Our *threshold variant* allows us to decrease the MPC computation to a small number of parties and get an efficient signature verification at the cost of a slightly increased signature size. For both variants, the main part of the computation can be pre-computed in a message-independent "offline" phase to leave a very small (< 1 ms) online cost which can become important for constrained embedded devices.

Small code-based signatures. The SD-in-the-Head signature scheme achieves among the smaller signature sizes for code-based signatures to-date, and is particularly performant in terms of the common "signature size + public-key size" metric. When comparing to other PQC signature schemes, our scheme competes most closely to SPHINCS$^+$; with similar signature and key sizes, better performances, and maintaining a similar standard of conservative security, albeit from an alternative (yet well-established) hardness assumption.

Compared to other code-signature signature schemes (see for example [FJR22, Table 6]), only Durandal [ABG+19], Wave [DST19], and LESS-FM II [BBP+21] have smaller signature sizes. However, Wave has a very large public-key, and Durandal and LESS-FM II are based on a much less conservative hardness assumption.

We can also compare SD-in-the-Head to other signature schemes based on the MPCitH paradigm, such as Picnic [ZCD+20] and Banquet [BdK+21]. The SD-in-the-Head signature

+ key sizes are much smaller than Picnic3 and Banquet (fast), we are actually much closer in size and performance to Picnic4 and Banquet (short) while still outperforming these schemes.

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 SPHINCS$^+$. 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.

8.2 Limitations of SD-in-the-Head

Quadratic growth w.r.t. the security level. As other MPCitH-based signature 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 the signature size. In practice, the size of SD-in-the-Head signatures increase of $\sim$140% while going from Category I to Category III and of $\sim$75% while going from Category III to Category V.

Randomness required for the hypercube approach. The novel trade-off that the hypercube approach brings, compared to the original SD-in-the-Head scheme, trades expensive MPC computations against less expensive randomness generation [AMGH$^+$23, Table 6]. However, for systems without good hardware support for symmetric primitives, the performances of the hypercube variant may be degraded.

Signature size for the threshold approach. The threshold approach has a bigger signature size in comparison to the hypercube approach. The sizes in comparison are roughly an extra 25% bigger. However, verification for the threshold approach is in-turn much faster.

References

[ABG+19] N. Aragon, O. Blazy, P. Gaborit, A. Hauteville, and G. Zémor. Durandal: A rank metric based signature scheme. In Y. Ishai and V. Rijmen, editors, *EUROCRYPT 2019, Part III*, pages 728–758. Springer, Heidelberg, 2019 (cited on page 53).

[AMGH+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. https://eprint.iacr.org/2022/1645 (cited on pages 2, 10, 12, 55).

[AMGH+23] C. Aguilar-Melchor, N. Gama, J. Howe, A. Hülsing, D. Joseph, and D. Yue. The return of the SDitH. In *EUROCRYPT*, 2023. For full version see [AMGH+22] (cited on pages 3, 13, 48, 54).

[AMHJ+23] C. Aguilar-Melchor, A. Hülsing, D. Joseph, C. Majenz, E. Ronen, and D. Yue. Sdith in the qrom. Cryptology ePrint Archive, Paper 2023/756, 2023. https://eprint.iacr.org/2023/756 (cited on page 48).

[BBC+19] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini. A finite regime analysis of information set decoding algorithms. *Algorithms*, (10):209, 2019 (cited on page 42).

[BBP+21] A. Barenghi, J.-F. Biasse, E. Persichetti, and P. Santini. LESS-FM: fine-tuning signatures from the code equivalence problem. In J. H. Cheon and J.-P. Tillich, editors, *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021*, pages 23–43. Springer, Heidelberg, 2021 (cited on page 53).

[BdK+21] C. Baum, C. de Saint Guilhem, D. Kales, E. Orsini, P. Scholl, and G. Zaverucha. Banquet: short and fast signatures from AES. In J. Garay, editor, *PKC 2021, Part I*, pages 266–297. Springer, Heidelberg, 2021 (cited on page 53).

[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, Heidelberg, 2012 (cited on page 50).

[BLP11] D. J. Bernstein, T. Lange, and C. Peters. Smaller decoding exponents: ball-collision decoding. In P. Rogaway, editor, *CRYPTO 2011*, pages 743–760. Springer, Heidelberg, 2011 (cited on page 50).

[BMVT78] E. Berlekamp, R. McEliece, and H. Van Tilborg. On the inherent intractability of certain coding problems (corresp.) *IEEE Transactions on Information Theory*, (3):384–386, 1978 (cited on page 3).

[BN20] C. Baum and A. Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In A. Kiayias, M. Kohlweiss, P. Wallden, and V. Zikas, editors, *PKC 2020, Part I*, pages 495–526. Springer, Heidelberg, 2020 (cited on page 6).

[CT23] K. Carrier and J.-P. Tillich. Round 1 (Additional Signatures) OFFICIAL COMMENT: SDitH. NIST PQC Forum, 2023-08-03. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/d_BcUfFGl5o/m/IyHkazJeAQAJ (cited on page 1).

[DFM+22] J. Don, S. Fehr, C. Majenz, and C. Schaffner. Efficient NIZKs and signatures from commit-and-open protocols in the QROM. In Y. Dodis and T. Shrimpton, editors, *CRYPTO 2022, Part II*, pages 729–757. Springer, Heidelberg, 2022 (cited on page 48).

[dOT21] C. de Saint Guilhem, E. Orsini, and T. Tanguy. Limbo: efficient zero-knowledge MPCitH-based arguments. In G. Vigna and E. Shi, editors, *ACM CCS 2021*, pages 3022–3036. ACM Press, 2021 (cited on page 15).

[DST19] T. Debris-Alazard, N. Sendrier, and J.-P. Tillich. Wave: A new family of trapdoor one-way preimage sampleable functions based on codes. In S. D. Galbraith and S. Moriai, editors, *ASIACRYPT 2019, Part I*, pages 21–51. Springer, Heidelberg, 2019 (cited on page 53).

[Dwo15] M. Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf (cited on page 43).

[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, Heidelberg, 2022 (cited on pages 2, 3, 6, 7, 9, 10, 44, 48, 49, 53).

[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. https://eprint.iacr.org/2022/1407 (cited on pages 2, 3, 10, 13, 15, 17, 48, 52).

[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, Heidelberg, 1987 (cited on page 3).

[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).

[KKW18] J. Katz, V. Kolesnikov, and X. Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, 2018 (cited on pages 10, 14).

[Kor89] E. A. Koruk. Decoding complexity bound for linear block codes. *Problemy Peredachi Informatsii*, (3):103–107, 1989 (cited on page 50).

[KZ20a] D. Kales and G. Zaverucha. An attack on some signature schemes constructed from five-pass identification schemes. In *Cryptology and Network Security: 19th International Conference, CANS 2020, Vienna, Austria, December 14–16, 2020, Proceedings*, pages 3–22. Springer, 2020 (cited on page 49).

[KZ20b] D. Kales and G. Zaverucha. An attack on some signature schemes constructed from five-pass identification schemes. In S. Krenn, H. Shulman, and S. Vaudenay, editors, *CANS 20*, pages 3–22. Springer, Heidelberg, 2020 (cited on pages 43, 51, 52).

[LB88] P. J. Lee and E. F. Brickell. An observation on the security of mceliece's public-key cryptosystem. In D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, and C. G. Günther, editors, *Advances in Cryptology — EUROCRYPT '88*, pages 275–280, Berlin, Heidelberg. Springer Berlin Heidelberg, 1988. ISBN: 978-3-540-45961-3 (cited on page 50).

[Leo88] J. S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Transactions on Information Theory*, (5):1354–1359, 1988 (cited on page 50).

[Meu13] A. Meurer. *A coding-theoretic approach to cryptanalysis*. PhD thesis, Verlag nicht ermittelbar, 2013 (cited on page 50).

[MMT11] A. May, A. Meurer, and E. Thomae. Decoding random linear codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In D. H. Lee and X. Wang, editors, *ASIACRYPT 2011*, pages 107–124. Springer, Heidelberg, 2011 (cited on page 50).

[MS10] F. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes. 9th Edition*. Elsevier Science, 1978, 2010. ISBN: 9780444851932 (cited on page 17).

[NIS22] NIST. Call for additional digital signature schemes for the post-quantum cryptography standardization process, 2022. https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf (cited on page 42).

[NPC+17] R. Niebuhr, E. Persichetti, P.-L. Cayrel, S. Bulygin, and J. Buchmann. On lower bounds for information set decoding over 𝕗q and on the effect of partial knowledge. *International Journal of Information and Coding Theory*, (1):47–78, 2017 (cited on page 51).

[Pet10] C. Peters. Information-set decoding for linear codes over F_q. In N. Sendrier, editor, *The Third International Workshop on Post-Quantum Cryptography, PQCRYPTO 2010*, pages 81–94. Springer, Heidelberg, 2010 (cited on page 51).

[Pra62] E. Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, (5):5–9, 1962 (cited on page 50).

[Sha79] A. Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, (11):612–613, 1979 (cited on page 13).

[Ste89] J. Stern. A method for finding codewords of small weight. In G. Cohen and J. Wolfmann, editors, *Coding Theory and Applications*, pages 106–113, Berlin, Heidelberg. Springer Berlin Heidelberg, 1989. ISBN: 978-3-540-46726-7 (cited on page 50).

[TS16] R. C. Torres and N. Sendrier. Analysis of information set decoding for a sublinear error weight. In T. Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, pages 144–161. Springer, Heidelberg, 2016 (cited on page 42).

[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 pages 2, 53).