On Threshold Signatures from MPC-in-the-Head

Eliana Carozza[1] and Geoffroy Couteau[2]

[1] IRIF, Université Paris Cité, Paris, France.
carozza@irif.fr
[2] CNRS, IRIF, Université Paris Cité, Paris, France.
couteau@irif.fr

Abstract. We initiate the study of threshold signatures built from MPC-in-the-Head signatures. While recent impossibility results (Doerner, Kondi and Rosenbloom, CRYPTO'24) show that any such construction must have a signature size that grows with the number n of users (or else require a prohibitive non-black-box use of cryptography), we show that this dependency in the number of users can be confined to a term of the form $\lambda^2 n + O(1)$, where λ is a security parameter and $O(1)$ a constant that depends on the signature scheme. We provide a concrete instantiation of our framework by building a threshold signature on top of the scheme of (Carozza, Couteau and Joux, EUROCRYPT'23).

keywords. Signatures, MPC-in-the-Head, threshold signatures

1 Introduction

With the upcoming advent of quantum computers, an important research effort is devoted to the construction of post-quantum primitives such as key encapsulations and signatures. The NIST call for post-quantum KEMs and signatures has accelerated this trend. More recently, a second NIST call for additional post-quantum signature proposals, targeted more specifically at non-lattice-based proposals, has led to a flurry of works. Among the many candidates, the MPC-in-the-Head paradigm [39] has emerged as one of the most promising paradigms to construct quantum-resistant signature schemes, with the most recent schemes in this category achieve signature sizes in the 3 5kB range [8,38,44] even while being (sometimes) based on the hardness of inverting well-studied symmetric primitives such as AES [9]. MPC-in-the-Head (MPCitH) offers an appealing combination of concrete efficiency and no requirement of any algebraic structure of the underlying assumption, which increases our confidence in their security.

Threshold signatures. Yet, signature schemes are far from the end of the story: many cryptographic and real-world applications require signature schemes equipped with more advanced features (this includes ring and group signatures, anonymous credentials, multisignatures, and many more). Among them, *threshold signatures* figure among the most widely studied. A threshold signatures lets a group of participants jointly sign a document such that only a subset of the participants larger than the threshold can produce an accepting signature. Threshold signatures enjoy several attractive features: they are resilient to failures or loss of data and they add a layer of decentralization, among other benefits. In the past decades, a widespread research effort has been devoted to the design of efficient threshold signature schemes, and a call for standardizing threshold schemes has been recently announced by the National Institute of Standards and Technology (NIST) [23].

Unfortunately, the impressive success of the MPC-in-the-Head, that has led to a large number of new efficient schemes in the past few years [1–6, 8–11, 13, 24, 26, 29, 30, 33, 34, 38, 41, 42, 44, 52], does not seem to extend to the desirable setting of threshold signatures. In fact, and while there are several threshold signature schemes in the post-quantum setting (e.g. [45]), there is to our knowledge no known efficient candidate threshold signature based on an MPC-in-the-Head signature (and, more generally, no efficient threshold signature whose underlying signature relies solely on symmetric assumptions).

Barrier towards efficient MPCitH threshold signatures. To anyone familiar with the inner working of MPC-in-the-Head signatures, this should not come as a surprise: efficient MPCitH signatures rely on primitives such as GGM trees which seem particularly complex to securely evaluate in a distributed setting. Plugging an MPCitH signature into a generic MPC protocol results in a prohibitive communication and computation, due in particular to the need to run expensive protocol

making non-black-box use of cryptographic primitives. Unfortunately, this complexity appears to be inherent: in a recent work [32], Doerner, Kondi and Rosenbloom proved that every threshold MPC-in-the-Head signature scheme must necessarily make a non-black-box use of hash functions / PRGs, or else tolerate signature sizes that grow with the number of signers.

1.1 Our Contribution

In this work, we explore the extend to which it is feasible to design efficient threshold signatures from MPCitH signatures in spite of the strong impossibility result of [32]. Our main contribution is a general recipe for converting any MPCitH signature into a "threshold-friendly" signature scheme. In doing so, we do not circumvent the impossibility result of [32]: rather, we bite the bullet and let *the size of the signature scheme grow with the number of users*. Equivalently, one can view the approach taken in our work as follows: given that [32] implies that all threshold-friendly MPCitH signatures must grow with the number of users (or else make expensive non-black-box use of cryptography), we ask how much we can reduce this linear dependency in the number of users.

Our main finding is the following: there is a general template transforming any MPCitH signature into a threshold-friendly signature of size $\lambda^2 n + O(1)$ bits, where the constant depends on the concrete scheme. When $\lambda = 128$, this amounts to a size of 2kB per user. This represent a significant improvement over the naive strategy where all users concatenate independent signatures. Of course, these numbers might feel somewhat underwhelming, and in particular, are not competitive with lattice-based threshold signatures such as [45]. However, we believe that they represent a useful datapoint regarding the best-possible signature size that can be achieved in a threshold-friendly setting. We hope that it will motivate further works in the area.

An interesting consequence of our result is that in the context of threshold signatures, the size difference between existing MPCitH candidates gets pushed to the $O(1)$ term, which becomes largely dominated as n grows. It has the conceptually intriguing effect of inverting, in this context, the notion of the "best" MPCitH signature: we found that older schemes, which are not state-of-the-art anymore and have larger sizes, tend to have a much simpler structure, which in turn results in much more efficient threshold signature schemes (when considering the communication and computation required to distributively generate a signature). We illustrate this behavior by providing a detailed case study of applying our template to the MPCitH scheme from [26], a scheme based on the regular syndrome decoding assumption which has since been superseded by more recent works [24,29,44] (which achieve significantly more compact signatures), but whose simple structure allows to build a very simple and efficient distributed signing procedure on top of its threshold-friendly variant.

To provide concrete numbers: using the parameters $K = 1842$, $k = 1017$, $w = 307$, $\tau = 11$ and $N = 2^{13}$ from [26], our threshold signature scheme has amortized communication 71.2kB per user (when instantiating the underlying "modulus-switching" functionality using the protocol given in Appendix A) and produces signatures of size approximately $2n + 6$ kilobytes.

2 Preliminaries

In this work, we will always call *parties* the virtual participants emulated in its head by the prover of an MPC-in-the-Head protocol, and *users* actual participants that interact to distributively generate a signature.

Given a set S, we write $s \leftarrow\!\!\$\, S$ to indicate that s is uniformly sampled from S. Given a probabilistic Turing machine $\mathcal{A}$ and an input x, we write $y \leftarrow\!\!\$\, \mathcal{A}(x)$ to indicate that y is sampled by running $\mathcal{A}$ on x with a uniform random tape, or $y \leftarrow \mathcal{A}(x; r)$ when we want to make the random coins explicit. Given an integer $n \in \mathbb{N}$, we denote by $[n]$ the set $\{1, \cdots, n\}$. We use λ to denote the computational security parameter. For every n, w such that w divides n, we let $\mathsf{Reg}_w(\mathbb{F}_2^n)$ denote the set of all w-regular vectors over $\mathbb{F}_2^n$ (that is, vectors which are a concatenation of w length-n/w unit vectors).

Given an integer n, we let $\mathsf{Perm}[n]$ denote the set of all permutations of $[n]$. In this work, we typically use permutations over $[n]$ to shuffle the entries of a length-n vector. Given a vector $\mathbf{v} \in \mathbb{F}^n$ and a permutation $\pi : [n] \to [n]$, we write $\pi(\mathbf{v})$ to denote the vector $(v_{\pi(1)}, v_{\pi(2)}, \cdots, v_{\pi(n)})$.

Given a vector x of length $6n$, viewed as a concatenation of n "blocks" of size 6, we let $\mathsf{BHW}_6(x)$ denote the vector over $\mathbb{F}_3$ whose i-th entry is the Hamming weight of the i-th block of x modulo 3.

2.1 Regular Syndrome Decoding Problem

Code parameters. In this work, K always denotes the number of columns in the parity-check matrix H, and k denote the number of its rows. Equivalently, K is the codeword length, and $K - k$ is the dimension of the code. We let w denote the weight of the noise, which will always divide K. We let $\mathsf{bs} \leftarrow K/w$ denote the block size: a w-regular noise vector is sampled as a concatenation of w random unit vectors (the *blocks*) of length bs. We write Reg_w to denote the set of all length-K w-regular vectors.

RSD. Given a weight parameter w, the syndrome decoding problem asks to find a solution of Hamming weight w (under the promise that it exists) to a random system of linear equations $H \cdot \mathbf{x}$ over $\mathbb{F}_2$. There exist several well-established variants of the syndrome decoding problem, with different matrix distributions, underlying fields, or noise distributions. In this work, we focus on a relatively well-studied variant known as the *regular syndrome decoding* (RSD) problem, introduced in 2003 in [7] as the assumption underlying the FSB candidate to the NIST hash function competition. In RSD, the solution $\mathbf{x}$ is sampled randomly from the set Reg_w of w-regular vectors (*i.e.*, $\mathbf{x}$ is a concatenation of w unit vectors of length K/w). This variant has been used (and analyzed) quite often in the literature [7, 12, 17–21, 28, 37, 46, 49, 51].

Definition 1 (Regular Syndrome Decoding Problem). *Let K, k, w be three integers, with $K > k > w$. The syndrome decoding problem with parameters (K, k, w) is defined as follows:*

- *(Problem generation) Sample $H \leftarrow\!\!\$\ \mathbb{F}_2^{k\times K}$ and $\mathbf{x} \leftarrow\!\!\$\ \mathsf{Reg}_w$. Set $\mathbf{y} \leftarrow H \cdot \mathbf{x}$. Output $(H, \mathbf{y})$.*
- *(Goal) Given $(H, \mathbf{y})$, find $\mathbf{x} \in \mathsf{Reg}_w$ such that $H \cdot \mathbf{x} = \mathbf{y}$.*

2.2 The MPC-in-the-Head Paradigm

The MPC-in-the-head paradigm was initiated by the work of Ishai et al [39] and provided a compiler that can build honest-verifier zero-knowledge (HVZK) proofs for arbitrary circuits from secure MPC protocols. Assume we have an MPC protocol with the following properties:

- N parties $(P_1, \cdots, P_N)$ securely and jointly evaluate a function $f : \{0,1\}^* \to \{0,1\}$ on $\mathbf{x}$ while each party possess an additive share $[\![\mathbf{x}]\!]_i$ of input $\mathbf{x}$,
- Secure against passive corruption of $N - 1$ parties i.e any $(N - 1)$ parties can not recover any information about the secret $\mathbf{x}$.

Then the HVZK proof of knowledge of $\mathbf{x}$ such that $f(\mathbf{x}) = 1$ is constructed as:

- Prover generates the additively shares of the witness $\mathbf{x}$ into $([\![\mathbf{x}_1]\!], \cdots, [\![\mathbf{x}_N)]\!])$ among N virtual parties $(P_1, \cdots, P_N)$ and emulate the MPC protocol "in-the-head".
- Prover commits to the view of each party and sends commitments to the verifier.
- Verifier chooses randomly $(N - 1)$ parties and asks the prover to reveal the view of these parties except one. Verifier later accepts if all the views are consistent with an honest execution of MPC protocol with output 1 and agrees with the commitments.

Security of MPC protocol implies that the verifier learns nothing about the input $\mathbf{x}$ from the $N - 1$ shares, and MPC correctness guarantees that the Prover can only cheat with probability $1/N$. Security can then be amplified with parallel repetitions.

3 Technical Overview

Our starting point is the observation that all modern MPC-in-the-Head and VOLE-in-the-Head signatures share a common high level template (up to, sometimes, minor variations). A rough outline of the template is as follows:

- The signer, using a root key and some random salt, derives root keys and salts for each round (where the rounds correspond to the number of repetitions τ of the underlying identification scheme to achieve negligible soundness error).

- For each round $k \leq \tau$, using the round root key and salt, the signer expands the root key into $N = 2^d$ pseudorandom (leaf seed, commitment) pairs, denoted $(\mathsf{sd}_i^k, \mathsf{com}_i^k)$, using a full binary tree *a la* GGM, where each internal nodes has two children, computed by applying a length-doubling pseudorandom generator on the value of the parent node.
- The concatenation of all commitments of the rounds is hashed with a collision-resistant hash function, and all hashes of all rounds are hashed together into a single hash.
- For each round, the leaf seeds are further expanded via a PRG into 2^d pseudorandom additive shares $(s_j^k)_{j \leq N}$ of a target tuple, that usually consists of the witness together with (potentially) some correlated random coins. An auxiliary string, or *shift* is constructed to offset the sum of the shares such that it reconstructs to the target tuple (alternatively, the shifts can be computed after the collapsing step that follows).
- For each round k, the 2^d shares are collapsed into d aggregated values, where the i-th value is computed by aggregating all shares s_j^k such that the i-th bit of j (viewed as a bitstring over $\{0, 1\}^d$) is 0. The MPC-in-the-Head view on this process is that of collapsing one 2^d-party (virtual) MPC instance into d 2-party (virtual) MPC instances by aggregating the shares along the hyperplanes of the d-dimensional boolean hypercube. The VOLE-in-the-Head view differs in semantics (the collapsing process is viewed as a reduction from an $(N-1)$-out-of-N oblivious transfer instance to a (subspace) vector-OLE correlation over the subspace $\mathbb{Z}_d$) but the process is identical.
- (Optional) A *consistency challenge* (for each round) is derived from the shifts and the hash of the commitments. This challenge will be used to check the consistency of the shifted aggregated shares with respect to the correlation they are supposed to satisfy.
- (Optional) A proof of consistency is derived from the challenge and the aggregated shares. Usually, this proof has a very simple structure and involves only challenge-dependent linear combinations of the aggregated shares.
- A *protocol challenge* (for each round) is derived from the shifts and the hash of the commitments (or, if a consistency challenge was used, from the consistency challenge and the consistency proof). This challenge is used for the execution of the main virtual MPC protocol (for MPC-in-the-Head proofs) or VOLE-based ZK proof (for VOLE-in-the-Head proofs) and is typically a "Schwartz-Zippel" challenge (used to collapse the verification of a set of multivariate polynomial equations).
- For each round, the virtual MPC protocol, or the designated-verifier VOLE-based ZK proof, is run.
- An *opening challenge* for each round is derived from the previous challenge. This challenge is used to define the virtual parties to open in the MPC-in-the-Head protocol (or, in VOLE-in-the-Head language, it defines the VOLE MAC key for the designated-verifier VOLE-based ZK proof).
- Eventually, an opening of each round is computed. It contains the opening to all-but-one of the leaf seeds (if the challenge is $\Delta_k \in \{0, 1\}^d$ in round k, the opening contains the values on the nodes of the co-path from the root to the leaf Δ_k in the k-th GGM tree) as well as $\mathsf{com}_{\Delta_k}^k$.

3.1 Challenges in thresholdizing MPCitH signatures

Any signature scheme can be generically converted into a threshold signature scheme via generic maliciously secure MPC. However, the structure of MPC-in-the-Head signature schemes makes them especially ill-suited for such generic approaches, and results in extremely inefficient constructions. Concretely, the main challenge lies in the construction of the GGM tree: to generate a signature, the signer starts by sampling τ root keys (and salts) and uses them to generate τ full binary trees via a length-doubling PRG (instantiated in practice with SHA3 [3], AES in counter mode [9] or fixed-key AES [24]). The number of leaves in the tree is a tunable parameter that typically ranges from $N = 2^8$ ("fast") to $N = 2^{16}$ ("shortest"). We represent such a tree with $N = 2^4$ on Figure 1. Then, each leaf is stretched (via a PRG) into a *commitment* and a *virtual party share*, represented respectively in green and purple on Figure 1. Afterwards, an auxiliary string is constructed from the (aggregated) N virtual party shares (to correct the share of the N-th virtual party so that the shares reconstruct to the right values, typically the witness and some correlated randomness) and the N commitments are concatenated and hashed (to create a succinct commitment of the tuple). Finally, once the challenge leaf i is determined, a succinct opening to all leaves except i is added to the signature by including all seeds on the co-path to the selected leaf node (the co-path is represented in blue on Figure 1, and the selected leaf in red).

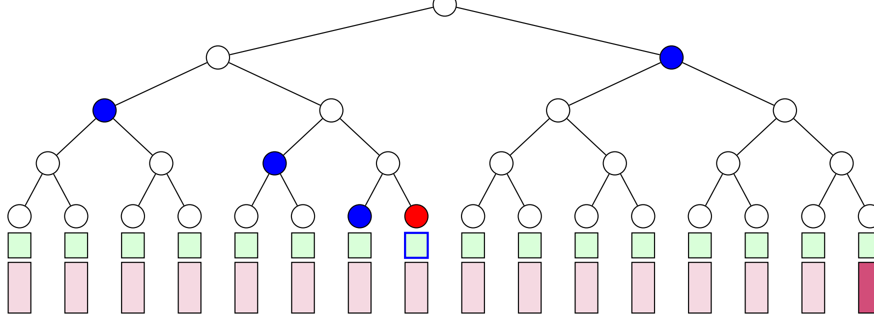

Fig. 1: A full GGM tree with $N = 16$. The seeds on the leaves are stretched into two strings, a commitment string (in green) and a virtual party share (in purple) The red-colored leaf denotes the seed that is not revealed in the signature, and the blue nodes denote the seeds on the co-path to the selected leaf. The signature contains the co-path seeds as well as the commitment string to the red leaf (denoted by a green-filled blue rectangle). The N-th virtual party share, represented in a darker purple, is computed from the aggregated virtual party shares from 1 to $N-1$ instead of being pseudorandomly generated from the N-th leaf seed.

The cost of distributing MPCitH signatures. Now, to distributively generate an MPC-in-the-Head signature, the users would have to

- Run (τ times in parallel) a maliciously secure protocol that, given shares of a root seed, expands it into a *full binary tree* of PRG evaluations, and stretches each leaf seed via a PRG. Crucially, no individual user should see any virtual party share in this process, hence the virtual party hares must remain secretely shared between the parties.
- Reveal the N commitments (they can be made public and locally hashed into a short digest by the users)
- Securely compute the N-th virtual party share (the auxiliary string, or shift) from the aggregated virtual party shares from $i = 1$ to $N - 1$.
- Once the non-opened leaf node is determined, reconstruct the seeds on the co-path to the selected leaf, and include them into the signature.

The above procedure is, however, extremely inefficient. To give a back-of-the-enveloppe estimate, even in the two-party setting, using a state-of-the-art maliciously secure constant-round protocol for AES [47] requires on average 6.7ms per AES computation, and 5.2MB of total communication. The τ full binary trees each have $2N$ nodes. Assuming a minimal number of two additional AES evaluations on the leaf nodes (to create the commitment and the virtual party share – note that in most MPCitH schemes, this number is actually much higher), we get a total of $4\tau N$ AES evaluations. Using the parameters of FAEST-128s [9], we have $N = 2^{12}$ and $\tau = 11$. This translates to a total communication of almost a Terabyte (937GB) and more than 20 minutes of computation *only to expand the binary tree*. For a larger number of users, the numbers would get significantly worse.

At a more abstract level, there are two clear downsides to the above strategy: (1) the protocol makes a non-black-box use of the underlying PRG, resulting in high computational costs, and (2) the communication scales with N, when the entire purpose of N in MPCitH protocol is to enable a communication/computation tradeoffs, where larger values of N result in more computation but less communication, since the computation of the full binary tree happens entirely in the signer's head. When distributively generating the signature, this computation must be taken "out of the signer's head", defeating its purpose.

A black-box barrier. The strategy outlined above is, of course, nothing more than the naive direct approach to distributing an MPCitH proof. Even though distributively computing the GGM trees *seems* unavoidable, could there be a more clever strategy that circumvents the need for a non-black-box use of the PRG? Unfortunately, the answer turns out to be no *as soon as the size of the signature does not scale with the number of users*: in a recent work [32], Doerner, Kondi and Rosenbloom proved that every threshold MPC-in-the-Head signature scheme must necessarily make a non-black-box use of hash functions / PRGs, or else tolerate signature sizes that grow with the number of signers. Concretely, this implies that there is no hope to do much better than the naive strategy outlined above without modifying the signature scheme. In fact, [32] explicitely mentions FAEST (and Picnic) as MPCitH signatures that are captured by their impossibility result.

3.2 Biting the bullet: MPCitH signatures that grow with the number of users

The impossibility result of [32] is quite strong, and we do not see any way to circumvent it. Instead, we choose to bite the bullet: we modify the MPC-in-the-Head approach to make it "threshold-friendly". In doing so, we let the size of the signature grow with a bound on the number of signers.

There is a trivial and uninteresting way of making a signature scheme threshold-friendly by increasing its size with the number of users: given n signing keys $(\mathsf{sk}_1, \cdots, \mathsf{sk}_n)$, define $\mathsf{Sign}'(m, (\mathsf{sk}_1, \cdots, \mathsf{sk}_n)) := \mathsf{Sign}(m, \mathsf{sk}_1)||\cdots||\mathsf{Sign}(m, \mathsf{sk}_n)$. Clearly, Sign' can be easily computed among n users $\mathcal{U}_1, \cdots, \mathcal{U}_n$ by letting each user $\mathcal{U}_j$ sample its own signing key sk_j, and concatenating the signatures locally produced by each user. Our main observation is that in the context of MPC-in-the-Head signatures, one can do much better: we only let the number of *GGM trees* scale with the number of users.

Concretely, an MPC-in-the-Head signature always contains τ "co-path", corresponding to seeds on the co-path from the root node of each GGM tree to the selected leaf. The total size of these co-paths is actually independent of N, and identical for essentially all existing MPC-in-the-Head schemes:[3] the size of each co-path is $\lambda \cdot \log N$ (where λ is set to 128) and τ is chosen approximately equal to $\lambda / \log N$ (to achieve soundness $1/2^\lambda$ after τ repetitions, since each round gives soundness $1/N$). This yields a total size of $\tau\lambda \log N \approx \lambda^2$, which amounts to 2kB using $\lambda = 128$. This yields a significant reduction in size for n users compared to naively concatenating n signatures.

For example, FAEST-128s and FAEST-128f signatures have size 5kB and 6.3kB respectively. In the n-user setting, this translates to $2n + 3$ and $2n + 4.3$ kilobytes respectively, compared to $5n$ and $6.3n$. For other MPCitH signatures, the gain is much higher: for example, for the signature scheme of [26] has signature sizes ranging from 12.5kB (fast variant) to 9kB (short variant). Plugged in our compiler, this yields signatures of size $2n + 10.5$ or $2n + 7$ kilobytes, much smaller than the $12.5n$ and $9n$ cost of the naive approach. As discussed in the introduction, a major effect of this transform is that it asymptotically erases the size difference between different signature schemes: for a largeish n, $2n + 10.5$ is not much worse than $2n + 3$. This changes the efficiency metric to compare among MPCitH signatures in the threshold setting: signature size does not matter much anymore, and a much more relevant metric is the (computational, communication) cost of distributively computing the signature. As we will see, some signature schemes such as [26] which are not state-of-the-art anymore (due to their sub-standard signature size) fare especially well in this setting, while other state-of-the-art signature schemes such as FAEST [9] appear much harder to distributively evaluate with reasonable concrete efficiency.

Our approach. In slightly more details, in our approach, we let the signature generate n independent GGM trees from n independent roots (for each of the τ rounds), and we view the pseudorandom strings computed from the seed leaves of each tree as *n-user shares of the commitments and virtual party shares*. Equivalently, we *define* the i-th virtual party commitment and share to be the XOR of the i-th leaf strings of each of the n GGM trees. In a threshold setting, this implies that we can let each user *locally* generate its own GGM tree. This way, the users hold *without any communication* n-wise shares of the virtual party shares. Given these shares, distributively computing a signature boils down to the following four tasks:

1. computing the hash of the concatenation of all commitments,
2. distributively computing the auxiliary string,
3. distributively computing the virtual MPC protocol,
4. appending all the $\tau \cdot n$ co-path to the final signature.

Items 2 and 3 can typically be performed efficiently: the auxiliary string is computed via a distributed protocol from the locally-aggregated strings of each of the n users. Hence, the total communication is independent of the number N of leaves. Furthermore, by design, the virtual MPC protocol operates on $d = \log N$ virtual party shares (after collapsing the N shares into d 2-party shares via the hypercube technique [4]); the threshold version amounts to extending this virtual protocol to an $n \cdot N$-party protocol (where each user controls the shares of N out of $n \cdot N$ virtual parties). For several schemes of interests, the extension to $n \cdot N$ parties is almost for free. Item 4 yields an increase in the signature size (of $2(n-1)$kB when using $\lambda = 128$), but as discussed earlier, a linear-in-n increase is

[3] The recent works of [8] and [38] introduced techniques to —slightly— reduce the size of these co-path, using respectively a "one tree to rule them all" strategy and the half-tree technique from [36], but both techniques can be applied to all existing MPCitH signatures.

unavoidable for schemes whose threshold version makes a black-box use of the underlying primitive.[4] Eventually, item 1 requires some care: the natural approach would be to let all parties reconstruct the N commitments (for each of the τ rounds) and locally hash them. However, this would have a significant impact on communication for large value of N: 5.8MB per user for $\tau = 11, N = 2^{12}$, or even up to 75.5MB per user for $\tau = 9, N = 2^{16}$ (a "short" parameter set used in many recent works, e.g. [4, 24, 38]).

To avoid this large communication overhead, we proceed differently: instead of reconstructing the commitments as the XOR of the commitment shares before hashing, we let each party locally hash the concatenation of its commitment shares, and broadcast the hash. Then, all uses locally hash the concatenation of the n hashes. This dramatically reduces the communication (to 32 bytes per user) at the cost of increasing again slightly the signature size (by $16 \cdot n$ bytes) as all n shares of the selected leaf commitment (the blue rectangle on Figure 1) must now be included into the signature to allow the verification of the hashes.

An MPC-in-the-Head template. We represent on Figure 2 a more precise description of the typical MPC-in-the-Head template, indicating the subroutines involved, their inputs and outputs, and using colors to provide a high level overview of the difficulties that arise in a threshold setting. The term "PPRF" stands for puncturable pseudorandom function [16,22,43] and captures the abstract primitive realized by the GGM PRF [35]: a pseudorandom function with domain size N equipped with an efficient puncturing algorithm that, given a PPRF key K and a point $\Delta \leq N$, generates a succinct key $K\{\Delta\}$ that allows to recompute the PPRF on all points except Δ. "VOLE" stands for vector oblivious linear evaluation, and denotes the (linear) aggregation procedure that converts a 2-party $(N-1)$-out-of-N oblivious transfer into additive shares of $\Delta \cdot u$, where Δ is the punctured point (the selected leaf) and u is a pseudorandom string known to the signer. This step is syntactically identical to the hypercube technique [4] that collapses an N-party virtual MPC protocol into $d = \log N$ instances of a 2-party protocol (but the alternative VOLE-style view has proven conceptually useful and is at the heart of the line of work on VOLE-in-the-Head signatures).

We outline below the color code of Figure 2:

▭ : the red color denotes the subroutines that are challenging to generate in a distributed setting, and their naive distributed evaluation is highly non black-box. We handle it by duplicating the PPRF instances, letting each user locally run its own PPRF instances and compute the VOLEs / hypercube aggregated shares. The VOLEs obtained by each users are viewed as additive shares of the global VOLE of the signature.

▭ : the blue color denotes calls to a hash function on the private outputs of the PPRF functionality. We let instead each party locally hash its shares of the commitments, and all parties hash the concatenation of the local hashes.

▭ : the green color denotes functionalities which are typically much simpler to distributively evaluate (often involving mostly linear operations). When adapting our high-level template to a concrete choice of MPCitH signature, the bulk of the work is the design of efficient MPC protocols for computing the green subroutines.

⬚ : the gray color denotes the optional Quicksilver proof challenge [50]. This check is typically present in VOLE-in-the-Head protocols [8–10,29,44], but absent from MPC-in-the-Head protocols [1–6, 11, 13, 24, 26, 30, 33, 34, 38, 41, 52] that directly rely on a virtual MPC protocol to prove the target relation (though in principle, such a protocol can also receive a challenge, typically to introduce batching and make verification more efficient).

→ : the red arrow denotes outputs that will be directly appended to the signature, or that can be computed from values included in the signature.

While the above template is described at an abstract level, we provide later on a full-fledged case analysis of the application of this template to a concrete MPC-in-the-Head signature scheme. Before we move on to this concrete case analysis, however, we shall discuss important technical subtleties that arise when attempting to prove security of the resulting threshold signature scheme.

3.3 Threshold signatures from threshold-friendly MPCitH signatures

Given a threshold-friendly MPCitH signature scheme (KeyGen, Sign, Verify) that follows the template outlined in the previous section, we sketch a distributed signing procedure. Let Commit denote an

[4] It is an interesting open question whether the cost could be lowered below 2kB/user. We leave it to future work, but note that it seems particularly challenging.

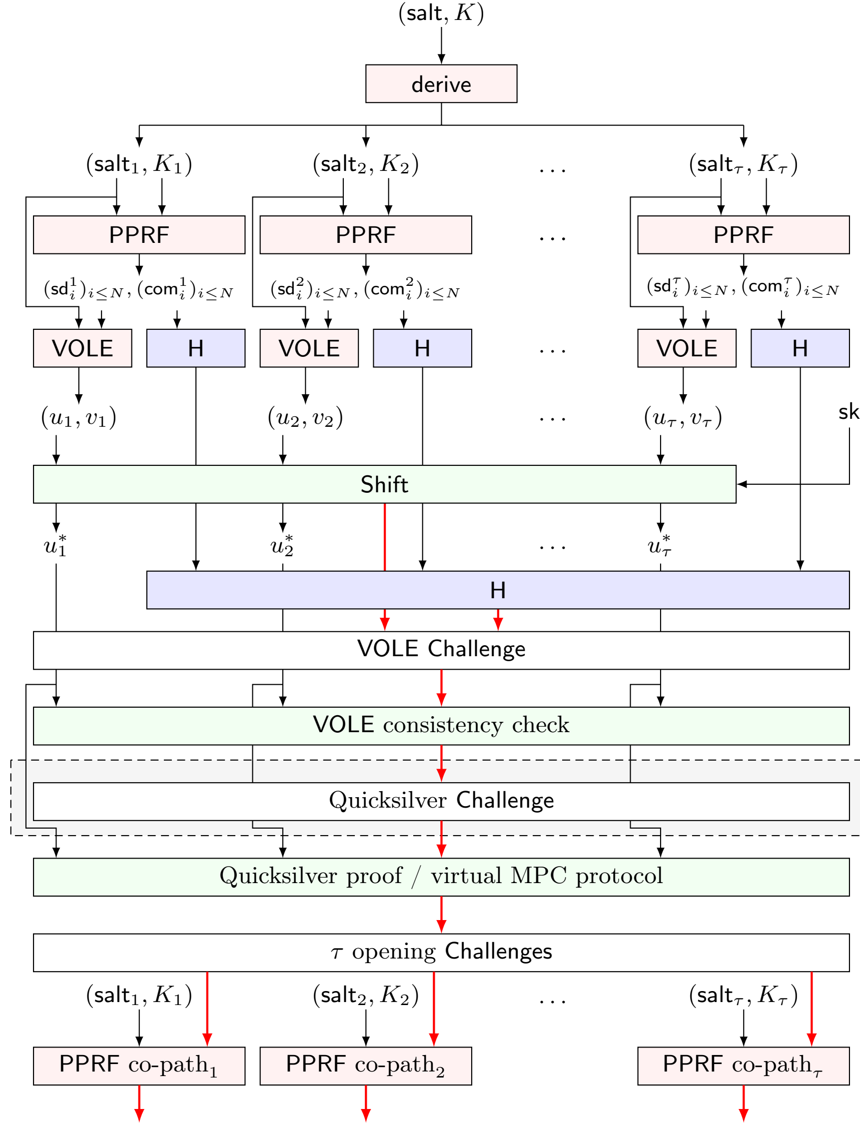

Fig. 2: High level representation of the common structure of existing MPC-in-the-Head and VOLE-in-the-Head signature schemes. sk denotes the secret key, or *witness* (the preimage of some one-way function). We refer the reader to Section 3.2 for a breakdown of the components and an outline of the color code of the template.

extractable and equivocable commitment scheme. We use the notations from the template of Figure 2. The protocol involves n users $\mathcal{U}_1, \cdots, \mathcal{U}_n$. We assume that the setup is executed by a trusted dealer.[5]

Trusted setup. Sample $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$. Sample n shares $(\mathsf{sk}_1, \cdots, \mathsf{sk}_n)$ of sk. Output pk as a public output, and send sk_j to each user $\mathcal{U}_j$. // Additionally, the trusted setup might append other correlated material to facilitate the executions of distributed protocols during the signing phases, such as e.g. PCG seeds.

Commit-and-open phase. Each user $\mathcal{U}_j$ samples τ root keys $(K_{1,j}, \cdots, K_{\tau,j})$. A master salt salt is sampled via a secure coin-flipping protocol, and τ salts $(\mathsf{salt}_1, \cdots, \mathsf{salt}_\tau)$ are derived from salt using a PRG modelled as a random oracle. Each user $\mathcal{U}_j$ locally computes $(\mathsf{sd}_i^{e,j}, \mathsf{com}_i^{e,j})_{i \leq N} \leftarrow \mathsf{PPRF}(\mathsf{salt}_e, K_{e,j})$ for $e = 1$ to τ and sets $h^j \leftarrow \mathsf{H}(\mathsf{com}_1^{1,j}, \cdots, \mathsf{com}_N^{1,j}, \cdots, \mathsf{com}_1^{\tau,j}, \cdots, \mathsf{com}_N^{\tau,j})$.

[5] This is a reasonable assumption in the common scenario where a signer wants to delegate its signing capability to n untrusted servers. In other contexts, no signer is assumed to know the full secret key and the trusted setup must be replaced by a distributed protocol. For simplicity, we focus on the former scenario in this work.

Commit: each user $\mathcal{U}_j$ sends $c_j \leftarrow\$ \ \mathsf{Commit}(h^j)$.
Open: each user $\mathcal{U}_j$ opens c_j to h^j.
Hash: all users compute $h_1 \leftarrow \mathsf{H}(h^1, \cdots, h^n)$.

Shift. Each user $\mathcal{U}_j$ generates τ VOLE pairs $(u_{e,j}, v_{e,j}) \leftarrow \mathsf{VOLE}(\mathsf{sd}_1^{e,j}, \cdots, \mathsf{sd}_N^{e,j})$ (equivalently, τ pairs of hypercube-aggregated shares). For every e, the pairs $u_{e,j}, v_{e,j})_{j \leq n}$ are viewed as additive shares of a single VOLE pair (u_e, v_e). All parties engage in a maliciously secure MPC protocol to securely instantiate the n-party functionality $\mathcal{F}_{\mathsf{shift}}$ that takes as input shares of $(u_e, v_e)_{e \leq \tau}$ and publicly outputs the shift shift (in VOLE-in-the-Head terminology) or auxiliary string aux (in MPC-in-the-Head terminology).

VOLE consistency check. All parties derive from (h_1, shift) a first challenge chall_1. In the VOLE-in-the-Head setting user $\mathcal{U}_j$ locally computes shares of the VOLE consistency check from $(\mathsf{chall}_1, \mathsf{shift}, (u_{e,j}, v_{e,j})_{e \leq \tau})$. The users broadcast their shares and reconstruct the VOLE consistency check (this is possible because the check is performed via a linear universal hash function). In the MPC-in-the-Head setting, the auxiliary string typically has a more complex structure (e.g. Beaver triples in [3, 11], shares of the same pseudorandom bit over $\mathbb{F}_2$ and $\mathbb{F}_3$ in [26], or shares of a pseudorandom integer and its one-hot-vector representation in [24]). Accordingly, the check becomes more involved. Nevertheless, in existing MPC-in-the-Head protocol, this consistency check remains fully linear (it involves sacrificing in [3, 11] and randomly shuffling the correlated randomness in [24, 26] using a public random permutation derived from the challenge).

(Optional) Quicksilver challenge. All parties publicly derive from (h_1, shift) and from the VOLE consistency check a challenge chall_2 for the Quicksilver proof.

Quicksilver proof / virtual MPC protocol. In this step, the parties distributively run the virtual MPC protocol (alternatively, distributively generate the quicksilver proof). We note that the cost of this step varies widely from one signature scheme to the other: in some signature schemes such as [24, 26] the virtual MPC protocol is reduced to a bare minimum, involving only local linear operations and broadcasting shares. As a consequence, distributively emulating this virtual MPC protocol among n users is straightforward. In contrast, in schemes such as FAEST (and all VOLE-in-the-Head schemes), this step requires executing a maliciously secure protocol for distributively generating a Quicksilver proof, which appears considerably more challenging. In particular, in FAEST, it requires running a maliciously secure n-user evaluation of the AES circuit over an extension field.[6]

Opening. Eventually, all parties publicly derive from (h_1, shift) and the transcript π of the virtual MPC protocol / the Quicksilver proof π a challenge chall_3 that encodes the τ positions $(i_1, \cdots, i_\tau) \in [N]^\tau$ that should not be opened. Each user $\mathcal{U}_j$ computes $(\mathsf{CoPath}^{e,j}, \mathsf{com}^e_{i_e})_{e \leq \tau}$, where $\mathsf{CoPath}^{e,j}$ denotes the $\log N$-sized tuple of seeds on the co-path from the root $K_{e,j}$ to the i_e-th leaf. They output the final signature

$$\sigma = (\mathsf{salt}, h_1, (\mathsf{chall}_2), \mathsf{chall}_3, (\mathsf{CoPath}^{e,j}, \mathsf{com}^e_{i_e})_{e \leq \tau, j \leq n}, \pi, \mathsf{shift}).$$

Obstacles towards proving security. The above template threshold signature seems intuitively secure. However, we cannot reduce security to the unforgeability of the underlying (threshold-friendly) MPC-in-the-Head scheme. To understand the issue, let us run through the first steps of the security analysis. Assume that all users except $\mathcal{U}_j$ are corrupted, and let Sim denote a simulator that emulates $\mathcal{U}_j$. In the setup phase, Sim receives pk from the signature functionality, and samples $n-1$ random shares $(\mathsf{sk}_\ell)_{\ell \neq j}$. Then, during the commit-and-open phase, Sim commits to a dummy string (it can adapt its opening later on as the commitment scheme is equivocable) and extracts $(h^\ell)_{\ell \neq j}$ from the commitments $(c_\ell)_{\ell \neq j}$.

However, this is where the proof gets essentially stuck. Recall that the goal of Sim is to play the role of an interface between the corrupted parties and the signing oracle. To this end, Sim is not given access to the secret key, and must be able to simulate given only access to the signing oracle. By the unforgeability of the signature scheme, this implies that the only way for Sim to properly simulate a threshold signing session on a message m is to somehow force the signing session to output the exact same signature as it received from the signing oracle.[7]

[6] The work of [48] reports more than 20s of computation for 14 parties, and about 3 minutes for 128 parties, when evaluating the plain AES circuit; the version we need here would be a large constant factor larger, since the AES circuit must be evaluated over an extension field.

[7] Technically, Sim could possibly force outputting a different signature if the signature scheme is unforgeable but not strongly unforgeable. However, to our knowledge all known MPCitH schemes are plausibly strongly unforgeable so it seems unlikely that one could implement such a strategy.

Now, given access to a signing oracle, Sim receives a signature $\sigma = (\mathsf{salt}, h_1, (\mathsf{chall}_2), \mathsf{chall}_3, (\mathsf{CoPath}^{e,j}, \mathsf{com}^e_{i_e})_{e\leq\tau,j\leq n}, \pi$ on a message m. From the signature, it obtains $h_1 = \mathsf{H}(\tilde{h}^1, \cdots, \tilde{h}^n)$. To simulate, Sim should at this stage find h^j such that $\mathsf{H}(h^1, \cdots, h^n) = \mathsf{H}(\tilde{h}^1, \cdots, \tilde{h}^n)$, which would break the collision-resistance of H (hence also the security of the signature scheme). Worse, the co-paths and commitments $(\mathsf{CoPath}^{e,j}, \mathsf{com}^e_{i_e})$ included in the signature allow recomputing the $\tilde{h}^\ell$. Because Sim must force the signing session to output the same co-paths as contained in σ, the adversary will necessarily notice from the final signature of the signing session that the $\tilde{h}^\ell$ recomputed from the co-paths are not equal to the h^ℓ. In other words, Sim cannot possibly simulate the interaction at this stage.

Candidate workarounds. First, we note that this issue is a direct consequence of our handling of item 1 in Section 3.2: if the parties were instead computing h_1 as a hash of the XOR of their individual commitment shares, then Sim could reconstruct from σ the commitment tuple $(\mathsf{com}^e_i)_{i\leq N,e\leq\tau}$, extract the commitment shares of the corrupted users from the $(c_\ell)_{\ell\neq j}$, and adapt the opening of its commitment c_j to the XOR of the commitment tuple and the corrupted parties' commitment shares. This is in fact similar to the strategy employed in standard threshold signature schemes such as threshold variants of Schnorr. But as we saw, in the MPCitH context this would incur a prohibitive communication.

To preserve the low communication of our solution, the most natural candidate workaround would be to modify the threshold signing protocol to give the ability to Sim to somehow force the corrupted users to output the same hashes $\tilde{h}^\ell$ as contained in σ. However, the only way to achieve this is to compute the hash of each user via a distributed protocol instead of letting the user compute it locally. But computing the hash values h^ℓ via a distributed protocol would again incur a prohibitive communication overhead, because the size of the input to each hash is very large, completely negating the communication advantage of our method! Furthermore, this distributed computation would have to make a non-black-box use of H, which would again require heavy computations.

3.4 Our solution: corruptible existential unforgeability

To resolve this conundrum, we adopt a different strategy: we do not change the threshold signing protocol, and instead modify the *unforgeability game* of the signature. Our main observation, which is very natural in retrospect, is the following: in the course of modifying the signature scheme to make it threshold-friendly, we also made it *more secure* in a specific sense that we will clarify shortly. The security proof of existential unforgeability of an MPCitH signature relies at some point on a sequence of game hops to replace the co-path and the selected leaf seed with uniformly random strings, by invoking $\log n$ times the security of the PRG. In turns, this allows to replace the share generated from the selected leaf with a random string. As a portion of this string is used to mask the witness, this guarantees that the witness remains hidden (statistically, at this point).

Now, in our threshold-friendly variant, there are n GGM trees and n co-paths, and the $n \cdot N$ strings stretched from the leaf seeds of all trees form additive shares of the N virtual parties' shares. The same analysis as with a single co-path carries over to this setting, but crucially, it suffices to replace the selected leaf share of a *single GGM tree* with a random string (this is easy to see: the "shares of shares" form a $n \cdot N$-party additive sharing of the witness, and replacing a single share with a random string suffices to statistically mask the witness). This implies that the scheme satisfies the following stronger security property: it remains secure even if *all but one* of the root keys are controlled by the adversary (instead of being randomly sampled by the signing oracle), as long as one root key is guaranteed to remain honestly and secretly sampled by the signing oracle. In other words, our threshold-friendly variant achieves a form of resistance against partial corruption of the random tape of the signature scheme.

We formalize this observation by introducing the notion of *corruptible existential unforgeability*. Informally, a signature scheme $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ satisfies (n-users) corruptible existential unforgeability against chosen-message attacks (CEUF-CMA) if the randomness of Sign is an n-tuple $(r_1, \cdots, r_n)$, and no adversary can forge a signature after making an arbitrary polynomial number of queries to the following *corruptible* signing oracle: the oracle receives queries $(m, i, (r_j)_{j\neq i})$, samples r_i, and returns $\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk}; (r_1, \cdots, r_n))$.

Going back to the security analysis of the threshold signature scheme, Sim can now emulate the commit-and-open phase of the protocol as follows:

- It extracts the hashes $(h^\ell)_{\ell\neq j}$ from the corrupted users' commitments

– From each h^ℓ, Sim extracts the root keys $(K_{1,\ell}, \cdots, K_{\tau,\ell})$. We will come back to this extraction step later (but in short, it relies on modeling H as a random oracle, and the AES cipher used to instantiate the GGM PPRF as an ideal cipher, and letting Sim observe the queries).
– Sim sends $(m, j, (K_{1,\ell}, \cdots, K_{\tau,\ell})_{\ell \neq j})$ to the corruptible signing oracle, and receives a signature σ. Note that by design, the hash h_1 contained in σ is of the form $\mathsf{H}(h^1, \cdots, h^j, \cdots, h^n)$, where the $(h^\ell)_{\ell \neq j}$ are the same as extracted from the commitments c_ℓ (as they have been computed from the same root keys).
– Sim recomputes h^j from σ using $(\mathsf{CoPath}^{e,j}, \mathsf{com}^e_{i_e})_{e \leq \tau}$ and adapts the opening of c_j to h^j using equivocability.

3.5 A dummy attack

Alas, the strategy given above does not yet quite work, due to an annoying "dummy attack" that a corrupted party could run. In the analysis above, we assumed that given the hashes h^ℓ extracted from $\mathcal{U}_\ell$'s commitments (where $\mathcal{U}_\ell$ denotes the ℓ-th user), Sim could recover the root keys $(K_{1,\ell}, \cdots, K_{\tau,\ell})$ of $\mathcal{U}_\ell$ by observing its queries to the random oracle and to the ideal cipher. But what happens if Sim does *not* find a preimages $(K_{1,\ell}, \cdots, K_{\tau,\ell})$ among the queries which are consistent with h^ℓ? There are a few reasons why this could happen. Perhaps $\mathcal{U}_\ell$ created a query collision, or did simply not query anything and sampled h^ℓ at random. These are easy to rule out: with overwhelming probability, the collision-resistance and the one-wayness of the random oracle guarantees that the user will fail to produce an accepting signature, and Sim can proceed through the simulation with dummy values.

However, there is one specific way that $\mathcal{U}_\ell$ could never query one of the root keys $K_{e,\ell}$, yet still manage to produce an accepting signature with non-negligible probability: $\mathcal{U}^\ell$ can *guess a selected leaf* i^*, sample uniformly random seeds on the co-path to i^* (the blue nodes on Figure 1) and sample a uniformly random seed on i^* (the red node on Figure 1). With these "fake" seeds, $\mathcal{U}_\ell$ can recompute all seed leaves of the GGM tree, and run the entire threshold signing protocol. At the end of the protocol, during the opening phase, a challenge i^e will be generated. With probability $1/N$, it might happen that $i^e = i^*$, in which case by adding its fake co-path to the signature, $\mathcal{U}_\ell$ will create a valid signature! (Meaning, not a honestly distributed signature, but a signature that passes the verification nonetheless). This is of course a dummy attack that achieves nothing —$\mathcal{U}_\ell$ must still know a share sk_ℓ of sk to compute the rest of the signature unless it manages to produce fake co-paths for *all* $e \in [\tau]$, which happens only with negligible probability— but it still implies that our simulation breaks down, as Sim cannot simply assume that if it fails to extract a root key, then the signature will necessarily not verify.

We handle this last challenge by returning to the (corruptible) threshold-friendly signature scheme construction. We modify the syntax of the scheme to let it take as random input either root seeds, or pairs (co-path, selected leaf seed). Then, the signature algorithm proceeds as before, reconstructing the leaf seeds either from the root or from the co-path. If some root $K_{e,\ell}$ was replaced with a co-path to a leaf i^* but $i^* \neq i^e$, the signing algorithm raises a flag $\mathsf{flag} = \perp$ (indicating that the signature generation failed to produce a valid signature), but outputs the invalid signature anyway. Fortunately, this syntactic change has almost no impact on the $\mathsf{CEUFCMA}$ security analysis of the scheme (as the analysis relies essentially on replacing the *uncorrupted* root seed by a random co-path to the right selected leaf), though it makes the description of the scheme more tedious.

With this modification, we can circumvent this dummy attack as follows: during the commit-and-open phase, Sim will attempt to extract *either* root keys $K_{e,\ell}$ or pairs (co-path, selected leaf seed) from the preimages. If neither extraction attempt succeeds, Sim proceeds through the rest of the simulation with dummy values, and the signature is guaranteed to be invalid (with overwhelming probability over the choice of the random oracle and ideal cipher). Else, if extraction succeeds, Sim feeds the extracted values to the corruptible signing functionality (which accepts either root keys or (co-path, leaf) pairs as randomness input) and can proceed with the rest of the simulation using the signature returned by the functionality (which might or might not be a valid signature, depending on whether a corrupted party attempted a dummy guess attack and guessed wrongly).

3.6 Case analysis: RSD-based threshold signatures

Our template is not an automated compiler: we cannot, unfortunately, produce a full-fledged abstract template from which one could automatically derive a threshold signature scheme given any MPC-in-the-Head scheme as input, with an accompanying formal proof of security. This stems from the fact

that existing MPC-in-the-Head schemes often differ on subtle low-level details: in how exactly they instantiate the GGM PPRF, in how they inject salt into the PRG evaluations to avoid the collision attack of [31], in how they hash the $\tau \cdot N$ commitments (either hashing the full concatenation, or first hashing individually the τ blocks of N commitments before hashing the τ hashes), in whether they compute the auxiliary string directly from the virtual party share or from the VOLE, etc, etc.

Each MPCitH scheme also comes with its own security analysis, and though many broadly follow the same template, some again differ in their low-level details. Furthermore, there are two broad template security analysis which all schemes more or less choose between: most MPC-in-the-Head protocols follow the original Picnic analysis [52], sometimes updated to either model directly the GGM PPRF in the random oracle model to facilitate extraction [3] or modelling instead the GGM PPRF as a multi-instance PPRF to achieve tight security while relying only on AES instead of a hash function for better efficiency [24, 38]. In parallel, VOLE-in-the-Head protocols typically follow the proof methodology introduced by FAEST [9], which differs significantly from the analysis of previous schemes. At a high level, while other MPC-in-the-Head scheme prove standard soundness and suffer from the Kales-Zaverucha attack [40] when the underlying identification scheme has more than three rounds (and must therefore adjust their parameters accordingly), the VOLE consistency check used in FAEST and follow-up works allows them to achieve the stronger notion of *round-by-round soundness* [25, 27] that, in particular, does not result in any security loss when compiling multi-round identification schemes via Fiat-Shamir.

As a result of these discrepancies, we view our abstract template as more of a recipe: a methodology that signature designers can adopt to convert their favorite signature scheme into a threshold-friendly signature scheme, prove its CEUF-CMA security as a relatively direct adaptation of the original EUF-CMA proof of the base scheme, and construct a threshold signing scheme from that. We believe that it would be possible in theory to have a "grand unification" of all MPC-in-the-Head and VOLE-in-the-Head security analyses in a general formal framework, and from there to provide a formal generic way to derive a CEUF-CMA threshold-friendly scheme and an accompanying threshold signing protocol. However, coming up with such a grand unification framework is out of the scope of this work (and we expect it to be challenging, to say the least).

Instead, we provide in the body of the paper a detailed case analysis of applying our "recipe" to a concrete MPCitH signature scheme. We expect (but do not formally claim) our concrete analysis to carry over in a very similar form to most other MPC-in-the-Head [1–6, 11, 13, 24, 26, 30, 33, 34, 38, 41, 52] and VOLE-in-the-Head [8–10, 29, 44] protocols. We select the scheme from [26], which is based on the regular syndrome decoding assumption, because of the very simple structure of its virtual MPC protocol: concretely, securely computing the auxiliary string and distributively running the virtual MPC protocol boils down to securely instantiating a functionality that converts "modulo 2" shares into "modulo 3" shares of the same value. As we will see, recent results from the MPC literature provide efficient instantiations of this functionality. This simple structure makes it an especially favorable candidate for our transform, even though the signature scheme itself is not state-of-the-art (we are aware of at least three MPCitH signature schemes based on regular syndrome decoding that achieve shorter signature sizes [24, 29, 44], but we expect all three of them to yield less efficient threshold signatures).

4 Threshold-Friendly MPC-in-the-Head Signatures

4.1 Corruptible existential unforgeability

We introduce the *Corrupted Existential Unforgeability under Chosen Message Attack* (**CEUF-CMA**) security model built upon the classical EUF-CMA notion by introducing adversarial control over specific seed inputs. In this model the adversary interacts with the challenger by querying signatures on tuples $(m, i, (r_j)_{j\neq i})$, where the chosen seeds will be used in the signature by the challenger for all-but-one of the trees in the MPC. The security of the scheme hinges on ensuring that even with such control, the adversary cannot produce a valid forgery on a fresh message-seed pair. This model captures a more refined adversarial advantage, strengthening the analysis of existential unforgeability in scenarios with partial input corruption.

Notation 2 *Let $n = n(\lambda)$ be a polynomial, and let $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme. We say that Σ is an $(n+1)$-source signature scheme if there exists $n+1$ sets $(\mathcal{R}, \mathcal{R}_1, \cdots, \mathcal{R}_n)$ such that the random tape of* Sign *is sampled from the randomness domain $\mathcal{R} \times \mathcal{R}_1 \times \cdots \times \mathcal{R}_n$.*

Definition 3. *Let $n = n(\lambda)$ be a polynomial, and let $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be an $(n+1)$-source signature scheme with randomness domain $\mathcal{R} \times \mathcal{R}_1 \times \cdots \times \mathcal{R}_n$. Consider the following experiment $\mathsf{Exp}_{\Sigma}^{\mathsf{CEUFCMA}}(\mathcal{A})$ played between a challenger and an adversary $\mathcal{A}$:*

1. *The challenger generates a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$, and gives the public key pk to $\mathcal{A}$.*
2. *The adversary $\mathcal{A}$ may adaptively request signatures on tuples $(m, i, (r_j)_{j \neq i})$, where $i \in [n]$ is chosen by the adversary himself. For each query $(m, i, (r_j)_{j \neq i})$, the challenger randomly chooses $r \leftarrow\$ \mathcal{R}$ (the "uncorruptible" part of the randomness) and the remaining uncorrupted coin $r_i \leftarrow\$ \mathcal{R}_i$, computes a signature $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m, (r, r_1, \ldots, r_m))$, and returns σ to $\mathcal{A}$.*
3. *Finally, the adversary outputs a forgery (m^*, σ^*). The experiment outputs 1 if:*
   - $\mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*) = 1$*, and*
   - *no tuple $(m^*, i, (r_j)^*_{j \neq i})$ was previously queried by the adversary.*

   Otherwise, the experiment outputs 0.

We define the advantage of an adversary $\mathcal{A}$ in breaking the CEUFCMA *security of the scheme Σ as:*

$$\mathsf{Adv}_{\Sigma}^{\mathsf{CEUFCMA}}(\mathcal{A}) = \Pr\left[\mathsf{Exp}_{\Sigma}^{\mathsf{CEUFCMA}}(\mathcal{A}) = 1\right]$$

where $\mathsf{Exp}_{\Sigma}^{\mathsf{CEUFCMA}}(\mathcal{A})$ is defined as above. We say that a signature scheme Σ is n-user CEUFCMA-*secure if the advantage $\mathsf{Adv}_{\Sigma}^{\mathsf{CEUFCMA}}(\mathcal{A})$ of any polynomial-time adversary $\mathcal{A}$ is negligible.*

We note that every signature scheme can trivially be converted into a corruptible $(n+1)$-source signature scheme as defined above, simply by defining $\mathsf{Sign}^*(m, \mathsf{sk}; (r, r_1, \cdots, r_n)) := \mathsf{Sign}(m, \mathsf{sk}; r)$ (that is, ignoring the corruptible part of the randomness and using only the uncorruptible part). However, not every $(n+1)$-source signature scheme is a $(n+1)$-source corruptible signature scheme. Our results build upon the observation that specific constructions of $(n+1)$-source MPC-in-the-Head schemes are CEUFCMA secure. Looking ahead, r in our construction will correspond to the salt (which must be sampled randomly for each signature scheme, but is publicly revealed by the signature), and each r_i will be a τ-tuple of seed roots for computing GGM trees.

4.2 A Threshold-Friendly variant of the signature scheme

We outline below a threshold-friendly variant of the signature scheme from [26], which is based on the regular syndrome decoding assumption. We refer the reader to [26] for detailed discussions on the RSD assumption, its cryptanalysis, and the parameter selection process. We note however that in light of existing cryptanalysis, RSD appears to be a conservative assumption and has attracted some attention in the context of MPC-in-the-Head signatures [24, 26, 29, 44].

Beyond modifying the construction of [26] using the recipe introduced in the technical overview to make it "threshold-friendly", we also update it along the way to include some optimizations that appeared more recently, such as the hypercube technique (and a few more minor optimizations). As such, the reader might find that the description differs significantly from that of [26] (but we stress that it is the same signature scheme with a few modern generic optimizations). In the following, we let $D = \log N$ denote the logarithm of the number N of virtual parties.

The PPRF. Our construction relies on a multi-instance puncturable pseudorandom function, as defined in [24]. In short, a multi-instance puncturable PRF remains secure if the adversary is allowed to query punctured keys for a large number of different keys, and must distinguish whether the challenges have all been computed by evaluating the PPRF at the punctured points, or have all been sampled at random. For completeness, we recall the formal definition of multi-instance PPRFs in Appendix B.

We introduce some notations. We let F denote the GGM PPRF, instantiated with a length-doubling PRG. When the length doubling PRG is instantiated as $\mathsf{PRG}_{\mathsf{salt}}(x) = (\mathsf{AES}_{\mathsf{salt}_0}(x) \oplus x, \mathsf{AES}_{\mathsf{salt}_1}(x) \oplus x)$, where $\mathsf{salt}_0, \mathsf{salt}_1$ are two λ-bit random strings (parsed from the 2λ-bit random salt salt given as input to the multi-instance PPRF), and when AES is modelled as an ideal cipher, [24] proved that the resulting PPRF is indeed (tightly) multi-instance secure. Given a salt salt, a root seed sd, and an index i, we write $\mathsf{sd}_i \leftarrow \mathsf{F}_{\mathsf{salt}}(\mathsf{sd}, i)$ to denote the i-th leaf of the GGM tree computed using $\mathsf{PRG}_{\mathsf{salt}}$.

To simplify notation, given a tuple $\mathsf{cp} = (i^*, v^1, \cdots, v^D, \mathsf{sd}^*)$, we denote $\mathsf{F}_{\mathsf{salt}}(\mathsf{cp}, i)$ the following procedure that computes the PPRF outputs from a co-path instead of a root seed:

– view $v^1, \cdots, v^D$ as the λ-bit seeds on the nodes of the co-path from the root to the i^*-th leaf (i.e., the blue nodes on Figure 1);
– if $i \neq i^*$, compute sd_i using $\mathsf{PRG}_{\mathsf{salt}}$ from its closer ancestor on the co-path (for example, if $i = 1$, that would be the seed v^2 on the leftmost blue node on Figure 1);
– else, if $i = i^*$, output $\mathsf{sd}_{i^*} = \mathsf{sd}^*$.

Eventually, given a salt salt, a root seed sd, and a leaf index i, we let $\mathsf{CoPath}_{\mathsf{salt}}(\mathsf{sd}, i)$ denote the procedure which recomputes the entire GGM tree and outputs the D-tuple of seeds on the nodes on the co-path to the leaf i.

Key generation. In our signature scheme, the key generation algorithm randomly samples a syndrome decoding instance (H, y) with solution x. We describe it on Figure 1.

Algorithm $\mathsf{KeyGen}(1^\lambda)$

Inputs: A security parameter λ.
1. Sample $\mathsf{sd} \leftarrow \{0,1\}^\lambda$ and set $H \leftarrow \mathsf{PRG}(\mathsf{sd})$ where $H = (H'|I) \in \mathbb{F}_2^{k \times K}$ is a parity-check matrix in systematic form.
2. Sample $(x|x_2) \leftarrow\$ \mathsf{Reg}_w(\mathbb{F}_2^K)$ with $x \in \mathbb{F}_2^{K-k}$ and set $y \leftarrow H' \cdot x \oplus x_2$.
3. Divide x into n additive shares $\tilde{x}_i$ for $i \in [n]$;
4. Set $\mathsf{pk} = (\mathsf{sd}, y)$ and $\mathsf{sk} = (H, (\tilde{x}_1, \cdots, \tilde{x}_n), y)$.

Algorithm 1: Key generation algorithm of the signature scheme

Signing. We represent the signing algorithm below.

Algorithm $\mathsf{Sign}(m, \mathsf{sk}; (r, r_1, \cdots, r_n))$

Inputs: A message $m \in \{0,1\}^{2\lambda}$ and secret key $\mathsf{sk} = (H, (\tilde{x}_1, \cdots, \tilde{x}_n), y)$.
Randomness:
– Parse r as $\mathsf{salt} \in \{0,1\}^{2\lambda}$ and derive $(\mathsf{salt}_1, \cdots, \mathsf{salt}_\tau) \leftarrow \mathsf{PRG}(\mathsf{salt})$.
– Parse each r_j as $(K^{e,j})_{e \leq \tau}$ and further parse each $K^{e,j}$ as either $\mathsf{sd}^{e,j} \in \{0,1\}^\lambda$ or as $\mathsf{cp}^{e,j} = (i^*_{e,j}, v^1_{e,j}, \cdots, v^D_{e,j}, \mathsf{sd}^*_{e,j}) \in [N] \times (\{0,1\}^\lambda)^{D+1}$. // $\mathsf{cp}^{e,j}$ corresponds to using as randomness input a co-path to a leaf $i^*_{e,j}$ and a $i^*_{e,j}$-th leaf seed $\mathsf{sd}^*_{e,j}$ instead of a root seed $\mathsf{sd}^{e,j}$. Note that forcing a choice of co-path to a leaf $i^*_{e,j}$ yields a correct signature only in the event that $i^e = i^*_{e,j}$ in Phase 4. This randomness input is never used in an honest use of the signing algorithm, but is allowed in order to specify the behavior of Sign given corrupted inputs.

Phase 1. For each iteration $e \in [\tau]$ and each $j \in [n]$:
– For $d = 1$ to D, set $(X^{e,j}_{d,0}, R^{e,j}_{d,0}, U^{e,j}_{d,0}) \leftarrow (0,0,0) \in \mathbb{F}_2^{K-k} \times \mathbb{F}_2^{K-k} \times \mathbb{F}_3^{K-k}$;
– Set $x^{e,j}_N \leftarrow \tilde{x}_j$, $u^e_N \leftarrow 0$, and $r^{e,j} \leftarrow 0$;
– **For** $i = 1$ **to** $N-1$:
  1. Compute $\mathsf{sd}^{e,j}_i \leftarrow \mathsf{F}_{\mathsf{salt}}(\mathsf{sd}^{e,j}, i)$; // $\mathsf{F}_{\mathsf{salt}}(\mathsf{cp}^{e,j}, i)$ if $K^{e,j} = \mathsf{cp}^{e,j}$
  2. $(x^{e,j}_i, r^{e,j}_i, u^{e,j}_i, \mathsf{com}^{e,j}_i) \leftarrow \mathsf{PRG}(\mathsf{sd}^{e,j}_i)$;
     // $(x^{e,j}_i, r^{e,j}_i, u^{e,j}_i, \mathsf{com}^{e,j}_i) \in \mathbb{F}_2^{K-k} \times \mathbb{F}_2^{K-k} \times \mathbb{F}_3^{K-k} \times \{0,1\}^\lambda$.
  3. $x^{e,j}_N \leftarrow x^{e,j}_N \oplus x^{e,j}_i$, $r^{e,j} \leftarrow r^{e,j} \oplus r^{e,j}_i$ and $u^{e,j}_N \leftarrow u^{e,j}_N + u^{e,j}_i \bmod 3$;
  4. Compute the virtual parties' views

$$(x^e_i, r^e_i, u^e_i) = \left(\bigoplus_j x^{e,j}_i, \bigoplus_j r^{e,j}_i, \sum_j u^{e,j}_i \bmod 3 \right).$$

– **On node** N: // computing the auxiliary string
  1. Compute $\mathsf{sd}^{e,j}_N \leftarrow \mathsf{F}_{\mathsf{salt}}(\mathsf{sd}^{e,j}, N)$ // $\mathsf{F}_{\mathsf{salt}}(\mathsf{cp}^{e,j}, N)$ if $K^{e,j} = \mathsf{cp}^{e,j}$
  2. Set $(r^{e,j}_N, \mathsf{com}^{e,j}_N) \leftarrow \mathsf{PRG}(\mathsf{sd}^{e,j}_N)$;
  3. Set $r^{e,j} \leftarrow r^{e,j} \oplus r^{e,j}_N$, $r^e \leftarrow \bigoplus_j r^{e,j}$, $u^e \leftarrow \sum_{i=1}^{N-1} u^e_i \bmod 3$, and $u^e_N \leftarrow r^e - u^e \bmod 3$.

4. Set the N-th virtual party's view

$$(x_N^e, r_N^e, u_N^e) = \left(\bigoplus_j x_N^{e,j}, \bigoplus_j r_N^{e,j}, u_N^e \bmod 3 \right)$$

5. Define $\mathsf{aux}_N^e \leftarrow (x_N^e, u_N^e)$;

– **For** $i = 1$ **to** N: // hypercube aggregation
  - For all $d \le D$ such that $i[d] = 0$, set: // $i[d]$ is the d-th bit of i.
    - $X_{d,0}^{e,j} \leftarrow X_{d,0}^{e,j} \oplus x_i^{e,j}$;
    - $R_{d,0}^{e,j} \leftarrow R_{d,0}^{e,j} \oplus r_i^{e,j}$;
    - $U_{d,0}^{e,j} \leftarrow U_{d,0}^{e,j} + u_i^{e,j} \bmod 3$;
  - Set the hypercube-aggregated views to

$$(X_{d,0}^e, R_{d,0}^e, U_{d,0}^e) = \left(\bigoplus_j X_{d,0}^{e,j}, \bigoplus_j R_{d,0}^{e,j}, \sum_j U_{d,0}^{e,j} \bmod 3 \right).$$

– Get $h_1^j \leftarrow \mathsf{H}_1(\mathsf{com}_1^{1,j}, \cdots, \mathsf{com}_N^{1,j}, \cdots, \mathsf{com}_1^{\tau,j}, \cdots, \mathsf{com}_N^{\tau,j})$; // Accumulate the commitments inside the hash rather than storing and hashing all at once.

– Set $h_1 \leftarrow \mathsf{H}_1(m, \mathsf{salt}, h_1^1, \cdots, h_1^n)$;

Phase 2.

1. $(\pi^e)_{e \le \tau} \leftarrow \mathsf{PRG}_1(h_1)$. // $\pi^e \in \mathsf{Perm}([K-k])$.

Phase 3. For each iteration $e \in [\tau]$:

1. $z_1^e \leftarrow x \oplus \pi^e(r^e)$, $z_2^e \leftarrow H' \cdot z_1^e \oplus y$,
   and $z^e \leftarrow (z_1^e || z_2^e)$;
2. For $d = 1$ to D, set:
   – $z_{d,0}^e[1] \leftarrow X_{d,0}^e \oplus \pi^e(R_{d,0}^e)$, $z_{d,0}^e[2] \leftarrow H' \cdot z_{d,0}^e[1] \oplus y$, and $z_{d,0}^e \leftarrow (z_{d,0}^e[1] || z_{d,0}^e[2])$;
   – $z_{d,1}^e[1] \leftarrow z_1^e \oplus z_{d,0}^e$, $z_{d,1}^e[2] \leftarrow z_2^e \oplus z_{d,0}^e[2]$, and $z_{d,1}^e \leftarrow (z_{d,1}^e[1] || z_{d,1}^e[2])$;
   – $\bar{x}_{d,0}^e \leftarrow z^e + (1 - z^e) \cdot \pi^e(U_{d,0}^e)$;
   – $\bar{x}_{d,1}^e \leftarrow x - \bar{x}_{d,0}^e \bmod 3$.
   – For $b = 0, 1$, set $\mathsf{msg}_{d,b}^e \leftarrow (z_{d,b}^e, \mathsf{BHW}_3(\bar{x}_{d,b}^e))$.
3. Get $h_2 \leftarrow \mathsf{H}_2(m, \mathsf{salt}, h_1, (\mathsf{msg}_{d,b}^e)_{d \le D, b \in \{0,1\}, e \le \tau})$;

Phase 4.

– Set $(i^e)_{e \le \tau} \leftarrow \mathsf{PRG}_2(h_2)$. // $i^e \in [N]$.

– For $e = 1$ to τ, if there exists $j \in [n]$ such that $K^{e,j} = \mathsf{cp}^{e,j}$, denoting $i_{e,j}^*$ the first component of $\mathsf{cp}^{e,j}$, if $i_{e,j}^* \neq i^e$, raise a flag $\mathsf{flag} = \perp$. Else, raise a flag $\mathsf{flag} = \top$.

– For all e, j such that $K^{e,j} = \mathsf{cp}^{e,j} = (i_{e,j}^*, v_{e,j}^1, \cdots, v_{e,j}^D, \mathsf{sd}_{e,j}^*)$, define $\mathsf{copath}^{e,j} = (v_{e,j}^1, \cdots, v_{e,j}^D)$.

– For all e, j such that $K^{e,j} = \mathsf{sd}^{e,j}$, define $\mathsf{copath}^{e,j} = \mathsf{CoPath}_{\mathsf{salt}}(\mathsf{sd}^{e,j}, i^e)$.

Phase 5. Output

$$\sigma = \left(\mathsf{salt}, h_1, h_2, \left(\mathsf{copath}^{e,j}, \mathsf{com}_{i_e}^{e,j} \right)_{e \le \tau,\, j \le n}, (z^e, \mathsf{aux}_N^e)_{e \le \tau} \right), \mathsf{flag}.$$

Algorithm 2: Signing algorithm of the signature scheme

Verification. We represent the verification algorithm below.

Algorithm $\mathsf{Verify}(m, \mathsf{pk}, \sigma)$

Inputs. A public key $\mathsf{pk} = (H, y)$, a message $m \in \{0,1\}^*$, a signature σ.

1. Parse the signature as follows:

$$\sigma = \left(\mathsf{salt}, h_1, h_2, \left(\mathsf{copath}^{e,j}, \mathsf{com}_{i_e}^{e,j} \right)_{e \le \tau,\, j \le n}, (z^e, \mathsf{aux}_N^e)_{e \le \tau} \right)$$

2. Recompute $(\pi^e)_{e\leq\tau} \leftarrow \mathsf{PRG}_1(h_1)$, where $\pi^e \in \mathsf{Perm}([K-k])$;
3. Recompute $(i^e)_{e\leq\tau} \leftarrow \mathsf{PRG}_2(h_2)$ and parse each i^e as a D-bit string $(b^e_d)_{d\leq D}$.
4. For each iteration $e \in [\tau]$,
   - For $d = 1$ to D:
     - Denote $b = 1 - b^e_d$;
     - Set $(X^e_{d,b}, R^e_{d,b}, U^e_{d,b}) \leftarrow (0,0,0) \in \mathbb{F}_2^{K-k} \times \mathbb{F}_2^{K-k} \times \mathbb{F}_3^{K-k}$;
     - For each $i \neq i^e$:
       * Recompute $\mathsf{sd}_i^{e,j}$ from $\mathsf{copath}^{e,j}$ for each $j \in [n]$;
       * If $i \neq N$, recompute $(x_i^{e,j}, r_i^{e,j}, u_i^{e,j}, \mathsf{com}_i^e) \leftarrow \mathsf{PRG}(\mathsf{sd}_i^{e,j})$; else, parse aux_N^e as (x_N^e, u_N^e), and compute $r_N^e = \bigoplus_j r_N^{e,j}$ where $r_N^{e,j} \leftarrow \mathsf{PRG}(\mathsf{sd}_N^{e,j})$;
       * If $i[d] = b$, update:
         - $X^e_{d,b} \leftarrow X^e_{d,b} \oplus \bigoplus_j x_i^e$;
         - $R^e_{d,b} \leftarrow R^e_{d,b} \oplus \bigoplus_j r_i^e$;
         - $U^e_{d,b} \leftarrow U^e_{d,b} + \sum_j u_i^e \bmod 3$;
     - Recompute $(\mathsf{msg}^e_{d,b})_{d\leq D, b\in\{0,1\}, e\leq\tau}$ by simulating the Phase 3 of the signing algorithm as below:
       * $z^e_{d,0} \leftarrow \left(X^e_d \oplus \pi^e(R^e_{d,0}) || H' \cdot z^e_{d,0}[1] \oplus y\right)$;
       * $z^e_{d,1} \leftarrow \left(z^e_1 \oplus z^e_{d,0} || z^e_2 \oplus z^e_{d,0}[2]\right)$;
       * $\bar{x}^e_{d,b} \leftarrow z^e + (1 - z^e) \cdot \pi(U^e_{d,b})$;
       * $\mathsf{msg}^e_{d,b} \leftarrow (z^e_{d,b}, \mathsf{BHW}_6(\bar{x}^e_{d,b}))$;
       * $\mathsf{msg}^e_{d,1-b} \leftarrow (z^e_{d,1-b}, 1 - \mathsf{BHW}_6(\bar{x}^e_{d,b}) \bmod 3)$
5. Check if $h_1 = \mathsf{H}_1(m, \mathsf{salt}, h_1^1, \cdots, h_1^n)$ where

$$h_1^j \leftarrow \mathsf{H}_1(\mathsf{com}_1^{1,j}, \cdots, \mathsf{com}_N^{1,j}, \cdots, \mathsf{com}_1^{\tau,j}, \cdots, \mathsf{com}_N^{\tau,j});$$

6. Check if $h_2 = \mathsf{H}_2(m, \mathsf{salt}, h_1, (\mathsf{msg}^e_{d,b})_{d\leq D, b\in\{0,1\}, e\leq\tau})$;
7. Output 1 iff both conditions are satisfied.

Algorithm 3: Verification algorithm of the signature scheme

4.3 Security Analysis of the Threshold-Friendly Signature Scheme

In this section, we prove that the signature scheme described on Algorithm 1, Algorithm 2 and Algorithm 3 satisfies n-user corruptible existential unforgeability against chosen-message attacks. We recall in Appendix B the notion of existential unforgeability against key-only attacks. Note that since the adversary is not allowed any signature query in the EUFKO game, we do not need to consider a corruptible variant of the notion.

Corruptible existential unforgeability

Theorem 4. *Assume that* F *is a* (q_s, τ)*-instance* $(t, \epsilon_{\mathsf{F}})$*-secure PPRF, that* PRG *is a* (q_s, τ)*-instance* $(t, \epsilon_{\mathsf{PRG}})$*-secure PRG, and that any adversary running in time t has at advantage at most* ϵ_{SD} *against the regular syndrome decoding problem. Model the hash functions* $\mathsf{H}_1, \mathsf{H}_2$ *as random oracles with output of length* 2λ*-bit and the pseudorandom generator* PRG_2 *as a random oracle. Then corrupted chosen-message adversary against the signature scheme described in Figure 2, running in time t, making q_s signing queries, and making q_1, q_2, q_3 queries, respectively, to the random oracles* $\mathsf{H}_1, \mathsf{H}_2$ *and* PRG_2*, succeeds in outputting a valid forgery with probability*

$$\mathsf{Adv}_{\Sigma}^{\mathsf{CEUFCMA}}(\mathcal{A}) \leq \frac{q_s\left(q_s + q_1 + q_2 + q_3\right)}{2^{2\lambda}} + \epsilon_{\mathsf{F}} + \varepsilon_{\mathsf{PRG}} + \epsilon_{\mathsf{SD}} + \Pr[X + Y = \tau] + \frac{1}{2^\lambda},$$

where $\varepsilon_{\mathsf{SD}}$ *bounds the advantage of* $\mathcal{A}$ *against the regular syndrome decoding problem,* $X = \max_{\alpha\in Q_1}\{X_\alpha\}$ *and* $Y = \max_{\beta\in Q_2}\{Y_\beta\}$ *with* $X_\alpha \sim \mathsf{Binomial}(\tau, \mathsf{p})$ *and* $Y_\beta \sim \mathsf{Binomial}\left(\tau - X, \frac{1}{N}\right)$ *where* Q_1 *and* Q_2 *are sets of all queries to oracles* H_1 *and* H_2 *and* p *is a statistical failure event bounded in Lemma 7 of [26], and set to* 2^{-132} *in their concrete parameter choices.*

Proof. The security analysis proceeds in two parts: first bounding $\mathsf{Adv}^{\mathsf{EUFKO}}(\mathcal{A})$, then bounding $\mathsf{Adv}^{\mathsf{CEUFCMA}}(\mathcal{A})$ using $\mathsf{Adv}^{\mathsf{EUFKO}}(\mathcal{A})$. The first half of the analysis, bounding $\mathsf{Adv}^{\mathsf{EUFKO}}(\mathcal{A})$, is identical

to the analysis in [26] (up to replacing the single GGM tree by n GGM trees whose leaves are summed everywhere in the analysis) and yields

$$\mathsf{Adv}_{\Sigma}^{\mathsf{EUFKO}}(\mathcal{A}) \leq \epsilon_{\mathsf{SD}} + \Pr[X + Y = \tau] + \frac{1}{2^{\lambda}},$$

where $\varepsilon_{\mathsf{SD}}$, X, Y are as as defined in the statement of Theorem 4. The crux of the analysis in our context lies in reducing corruptible existential unforgeability against chosen message attacks to EUFKO security. This is captured by the following lemma:

Lemma 5 ($\mathsf{EUFKO} \implies \mathsf{CEUFCMA}$).

$$\mathsf{Adv}^{\mathsf{CEUFCMA}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{EUFKO}}(\mathcal{A}) + \frac{q_s\,(q_s + q_1 + q_2 + q_3)}{2^{2\lambda}} + \epsilon_{\mathsf{F}} + \epsilon_{\mathsf{PRG}}$$

We now prove Lemma 5. Let us consider an adversary $\mathcal{A}$ against the $\mathsf{CEUFCMA}$ property of the signature scheme. To prove security we will define a sequence of experiments involving $\mathcal{A}$, where the first corresponds to the experiment in which $\mathcal{A}$ interacts with the real signature scheme, and the last one is an experiment in which $\mathcal{A}$ is using only random element independent from the witness.
Game 1 (Gm^1). This corresponds to the actual interaction of $\mathcal{A}$ with the real signature scheme. Upon receiving a query $(m, j, (r_j)_{i \neq j})$ from $\mathcal{A}$, the signing oracle samples $r_j := (\mathsf{sd}^{e,j})_{e \leq \tau} \leftarrow\$ (\{0,1\}^{\lambda})^{\tau}$, and return $\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk}; (r_1, \cdots, r_n))$. We denote $\mathsf{Gm}^{\ell}(\mathsf{Forge})$, the event that after interacting with the corruptible signing oracle in Game ℓ, $\mathcal{A}$ generates a valid signature σ^* for a message m^* that were not previously queried to the signing oracle.
Game 2 (Gm^2). In this game, we abort if the sampled salt salt collides with the value sampled in any of the previous queries to the hash functions H_1 or H_2, or if the input to PRG_2 collides with the value obtained in any of the previous queries. Therefore we can bound this probability by

$$|\Pr[\mathsf{Gm}^1(\mathsf{Forge})] - \Pr[\mathsf{Gm}^2(\mathsf{Forge})]| \leq \frac{q_s \cdot (q_s + q_1 + q_2 + q_3)}{2^{2\lambda}}$$

Game 3 (Gm^3). The difference with the previous game is that now before signing a message we choose uniformly random values h_1, h_2 and $(i^e)_{e \leq \tau}$. Since Phase 1, Phase 3 and Phase 5 are computed as before and the only change compared to the previous game is that we randomly set the output of H_1 as h_1, the output of H_2 as h_2 and the output of $\mathsf{PRG}_2(h_2)$ as $(i^e)_{e \leq \tau}$. A difference in the forgery probability can only happen in the event that a query to H_1, H_2 or PRG_2 was made before; however, in this scenario, Game 2 aborts. Therefore,

$$\Pr[\mathsf{Gm}^2(\mathsf{Forge})] = \Pr[\mathsf{Gm}^3(\mathsf{Forge})].$$

Game 4 (Gm^4) in this game, upon receiving a query $(m, j, (r_i)_{i \neq j})$ from $\mathcal{A}$, after sampling $r_j := (\mathsf{sd}^{e,j})_{e \leq \tau} \leftarrow\$ (\{0,1\}^{\lambda})^{\tau}$, compute the leaf seed $\mathsf{sd}^{e,j}_{i_e} = \mathsf{F}_{\mathsf{salt}}(\mathsf{sd}^{e,j}, i^e)$ and the corresponding co-path $\mathsf{copath}^{e,j} = \mathsf{CoPath}_{\mathsf{salt}}(\mathsf{sd}^{e,j}, i^e)$. Set $\mathsf{cp}^{e,j} = (i^e, \mathsf{copath}^{e,j}, \mathsf{sd}^{e,j}_{i^e})$ for $e = 1$ to τ and $r_j = (\mathsf{cp}^{e,j})_{u \leq \tau}$. Run $\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk}; (r_1, \cdots, r_n))$.
This game is a purely syntactic change: by design of the signing algorithm, it will output exactly the same signature as in Game 3 (observe that since PRG_2 is programmed to output $(i^e)_e$, no aborts of the signing algorithm will be triggered). Therefore, we have

$$\Pr[\mathsf{Gm}^3(\mathsf{Forge})] = \Pr[\mathsf{Gm}^4(\mathsf{Forge})].$$

Game 5 (Gm^5) in this game, upon receiving a query $(m, j, (r_i)_{i \neq j})$ from $\mathcal{A}$, instead of sampling $r_j := (\mathsf{sd}^{e,j})_{e \leq \tau} \leftarrow\$ (\{0,1\}^{\lambda})^{\tau}$, sample for $e = 1$ to τ a uniformly random leaf seed $\mathsf{sd}^{e,j}_{i_e}$ and a uniformly random corresponding co-path $\mathsf{copath}^{e,j}$ (note that from Game 3, the indices i^e are sampled ahead of time). Set $\mathsf{cp}^{e,j} = (i^e, \mathsf{copath}^{e,j}, \mathsf{sd}^{e,j}_{i^e})$ for $e = 1$ to τ and $r_j = (\mathsf{cp}^{e,j})_{u \leq \tau}$. Run $\sigma \leftarrow \mathsf{Sign}(m, \mathsf{sk}; (r_1, \cdots, r_n))$.
The only difference between Game 3 and Game 4 is that the leaf seed $\mathsf{sd}^{e,j}_{i^e}$ and its corresponding co-path $\mathsf{copath}^{e,j}$ are sampled uniformly at random. Distinguishing between the two games reduces therefore immediately to breaking the (Q, τ)-instance strong security of the PPRF, where Q is a bound on the number of signing queries from $\mathcal{A}$ (cf 11). Therefore, we have

$$|\Pr[\mathsf{Gm}^5(\mathsf{Forge})] - \Pr[\mathsf{Gm}^4(\mathsf{Forge})]| \leq \mathsf{Adv}^{\mathsf{F}}(\mathcal{A}) = \varepsilon_{\mathsf{F}}.$$

Game 6 (Gm^6). In this game, upon receiving a query $(m, j, (r_i)_{i \neq j})$ from $\mathcal{A}$ and for $e = 1$ to τ, we sample $(x^{e,j}_{i^e}, r^{e,j}_{i^e}, u^{e,j}_{i^e}, \mathsf{com}^{e,j}_{i^e}) \leftarrow\$ \mathbb{F}_2^{K-k} \times \mathbb{F}_2^{K-k} \times \mathbb{F}_3^{K-k} \times \{0,1\}^\lambda$ at random. As the seed $\mathsf{sd}^{e,j}_{i^e}$ is uniformly random (independent of $\mathsf{copath}^{e,j}$ and never revealed by the signature (it is only used in Game 5 to construct $(x^{e,j}_{i^e}, r^{e,j}_{i^e}, u^{e,j}_{i^e}, \mathsf{com}^{e,j}_{i^e}) \leftarrow \mathsf{PRG}(\mathsf{sd}^{e,j}_{i^e})$), this game is indistinguishable from the previous one by a direct application of the multi-instance security of PRG (Definition 9), and we have

$$|\Pr[\mathsf{Gm}^5(\mathsf{Forge})] - \Pr[\mathsf{Gm}^4(\mathsf{Forge})]| \leq \mathsf{Adv}^{\mathsf{PRG}}(\mathcal{A}) = \epsilon_{\mathsf{PRG}}.$$

Game 7 (Gm^7). In this game, we modify the behavior of the emulation in Phase 3. Namely, we construct the messages $(\mathsf{msg}^e_{d,b})_{d \leq D, b \in \{0,1\}, e \leq \tau}$ using instead the same procedure as the verifier in step 4 of the Verify algorithm (Algorithm 3). Note that this process yields an identical construction of the $\mathsf{msg}^e_{d,b}$ by correctness of the verification algorithm, and can be used in the emulation because we sample the challenges i^e ahead of time. Note also that the witness x is at this stage only used in two places: in the computation of x^e_N in aux^e_N, and in the computation of $z^e_1 = x \oplus \pi^e(r^e)$. We have

$$\Pr[\mathsf{Gm}^7(\mathsf{Forge})] = \Pr[\mathsf{Gm}^6(\mathsf{Forge})].$$

Game 8 (Gm^8). In this game, we instead sample $\mathsf{aux}^e_N \leftarrow\$ \mathbb{F}_2^{K-k} \times \mathbb{F}_3^{K-k}$ and $z^e_1 \leftarrow\$ \mathbb{F}_3^{K-k}$ for $e = 1$ to τ. Note that $\mathsf{aux}^e_N = (x^e_N, u^e_N)$ is constructed as

$$x^e_N = x \oplus \bigoplus_{i=1}^{N} \bigoplus_{j=1}^{n-1} x^{e,j}_i, \qquad u^e_N = r^e - \sum_{i=1}^{N-1} \sum_{j=1}^{n} u^{e,j}_i \bmod 3.$$

Because each of these terms is masked by a uniformly random value (respectively $x^{e,j}_{i^e}$ and $u^{e,j}_{i^e}$), and because $x \oplus \pi^e(r^e)$ is masked by $\pi^e(r^{e,j}_{i^e})$, all are uniformly random over $\mathbb{F}_2^{K-k}$, $\mathbb{F}_3^{K-k}$, and $\mathbb{F}_2^{K-k}$ respectively, and we have

$$\Pr[\mathsf{Gm}^8(\mathsf{Forge})] = \Pr[\mathsf{Gm}^7(\mathsf{Forge})].$$

Observe that in Game 8, the emulation does not use the witness x anymore, hence it does not need the secret key sk. Therefore, an adversary outputting a forgery in Game 8 immediately implies an adversary with the same success probability against the EUFKO security of the signature scheme:

$$\Pr[\mathsf{Gm}^8(\mathsf{Forge})] = \mathsf{Adv}^{\mathsf{EUFKO}}(\mathcal{A}).$$

This concludes the proof of Lemma 5 and Theorem 4.

5 Threshold signatures from threshold-friendly signatures

In this section, we introduce a threshold signature scheme constructed from the threshold-friendly signature scheme introduced in section 4. Our construction assumes the following building blocks and functionalities:

- We let Commit denote an extractable and equivocable commitment scheme. Concretely, a possible instantiation of Commit is as $\mathsf{Commit}(m; r) = \mathsf{H}'(m; r)$, where H' is a random oracle to which the simulator will be given programmable access.
- $\mathcal{F}_{\mathsf{salt}}$ is a random coin-sampling functionality: upon receiving the signing session id from all users, it samples $\mathsf{salt} \leftarrow\$ \{0,1\}^{2\lambda}$ and outputs it to all users. This functionality can be instantiated via a simple commit-and-open protocol.
- $\mathcal{F}_{2,3}$ denotes the "mod2-to-mod3" functionality, that converts additive shares of a vector modulo 2 into additive shares of the same vector modulo 3. We represent the ideal functionality on Functionality 1. In Appendix A, we introduce efficient protocols for securely instantiating the functionality using pseudorandom correlation generators.

5.1 The functionality $\mathcal{F}_{2,3}$

We represent on Functionality 1 the mod2-to-mod3 functionality. We describe a "corruptible" variant of the functionality, where we let the corrupted parties define their output shares, and sample the honest parties' output share consistently with the corrupted parties' shares.

Functionality $\mathcal{F}_{2,3}$

Parameters. The functionality interacts with n users $\mathcal{U}_1, \cdots, \mathcal{U}_n$. We let $\mathcal{C}$ denote the (possibly empty) subset of corrupted users.

Input. Each user $\mathcal{U}_j$ sends a vector $u_j \in \mathbb{F}_2^t$. Additionally, each corrupted user $\mathcal{U}_j$ sends a vector $u'_j \in \mathbb{F}_3^t$. The functionality aborts if it receives incorrectly formatted inputs, or if they do not all have the same length.

Functionality.
- Compute $u = \bigoplus_{j \le n} u_j$.
- Compute $u' = \sum_{j \in \mathcal{C}} u'_j$.
- Sample $n - |\mathcal{C}|$ uniformly random shares $(u'_j)_{j \in [n] \setminus \mathcal{C}}$ of $u - u'$ mod 3 over $\mathbb{F}_3$.
- Output u'_j to each honest user $\mathcal{U}_j$.

Functionality 1: ideal functionality for converting sums from mod2 to mod3

5.2 The threshold signing protocol

Protocol Threshold Signing

Key-generation (trusted setup).

Inputs: A security parameter λ.
1. Run $(\mathsf{pk}, \mathsf{sk}) \leftarrow\$ \mathsf{KeyGen}(1^\lambda)$.
2. Parse $\mathsf{sk} = (H, (\tilde{x}_1, \cdots, \tilde{x}_n)$. Output $\mathsf{sk}_j = (H, \tilde{x}_j, y)$ to each $\mathcal{U}_j$.

Sampling the salt. All parties invoke $\mathcal{F}_{\mathsf{salt}}$ and receive a global salt $\mathsf{salt} \in \{0,1\}^{2\lambda}$. All parties derive $(\mathsf{salt}_1, \cdots, \mathsf{salt}_\tau) \leftarrow \mathsf{PRG}(\mathsf{salt})$.

Building the trees. For each iteration $e \in [\tau]$, each user $\mathcal{U}_j$ proceeds as follows:
- Sample $\mathsf{sd}^{e,j} \leftarrow\$ \{0,1\}^\lambda$;
- For $d = 1$ to D, set $(X_{d,0}^{e,j}, R_{d,0}^{e,j}, U_{d,0}^{e,j}) \leftarrow (0,0,0)$;
- Set $x_N^{e,j} \leftarrow \tilde{x}_j$, $u_N^e \leftarrow 0$, and $r^{e,j} \leftarrow 0$;
- **For** $i = 1$ **to** $N-1$:
  1. Compute $\mathsf{sd}_i^{e,j} \leftarrow \mathsf{F}_{\mathsf{salt}}(\mathsf{sd}^{e,j}, i)$;
  2. $(x_i^{e,j}, r_i^{e,j}, u_i^{e,j}, \mathsf{com}_i^{e,j}) \leftarrow \mathsf{PRG}(\mathsf{sd}_i^{e,j})$;
  3. $x_N^{e,j} \leftarrow x_N^{e,j} \oplus x_i^{e,j}$, $r^{e,j} \leftarrow r^{e,j} \oplus r_i^{e,j}$ and $u_N^{e,j} \leftarrow u_N^{e,j} + u_i^{e,j}$ mod 3;
- **On node** N**:**
  1. Compute $\mathsf{sd}_N^{e,j} \leftarrow \mathsf{F}_{\mathsf{salt}}(\mathsf{sd}^{e,j}, N)$
  2. Set $(r_N^{e,j}, \mathsf{com}_N^{e,j}) \leftarrow \mathsf{PRG}(\mathsf{sd}_N^{e,j})$
  3. Set $r^{e,j} \leftarrow r^{e,j} \oplus r_N^{e,j}$
  4. Send $r^{e,j}$ to the $\mathcal{F}_{2,3}$ functionality in order to obtain $u^{e,j}$. // This is the **Shift** phase of the template protocol in Section 3.3.
  5. Set $u_N^{e,j} \leftarrow u^{e,j} - u_N^{e,j}$ mod 3
  6. Broadcast $(x_N^{e,j}, u_N^{e,j})$. Upon receiving all shares, all user reconstruct $\mathsf{aux}_N^e = (x_N^e, u_N^e) = \left(\bigoplus_j x_N^{e,j}, \sum_j u_N^{e,j} \text{ mod } 3\right)$;
- For $i = 1$ to N, for all $d \le D$ such that $i[d] = 0$, set: // $i[d]$ is the d-th bit of the integer i.
  - $X_{d,0}^{e,j} \leftarrow X_{d,0}^{e,j} \oplus x_i^{e,j}$;
  - $R_{d,0}^{e,j} \leftarrow R_{d,0}^{e,j} \oplus r_i^{e,j}$;
  - $U_{d,0}^{e,j} \leftarrow U_{d,0}^{e,j} + u_i^{e,j}$ mod 3.

Commit-and-open phase. For each iteration $e \in [\tau]$, each user $\mathcal{U}_j$ proceeds as follows:
- Set $h_1^j \leftarrow \mathsf{H}_1(\mathsf{com}_1^{1,j}, \cdots, \mathsf{com}_N^{1,j}, \cdots, \mathsf{com}_1^{\tau,j}, \cdots, \mathsf{com}_N^{\tau,j})$.
- Broadcast $c^j \leftarrow\$ \mathsf{Commit}(h^j)$.
- Upon receiving all $(c^\ell)_{\ell \in [n]}$, broadcast h^j and the opening of c^j.
- If an opening does not verify, abort the protocol. Else, all users compute $h_1 \leftarrow \mathsf{H}_1(m, \mathsf{salt}, h_1^1, \cdots, h_1^n)$.
- Compute $(\pi^e)_{e \le \tau} \leftarrow \mathsf{PRG}_1(h_1)$. // The use of a permutation to shuffle the correlated randomness replaces the VOLE consistency check in [26].

Distributed virtual protocol. Let $(1^p_j)_{j\in[n]}$ denote arbitrary shares of 1 modulo $p = 2$ or 3.

- Each user $\mathcal{U}_j$ sets $z_1^{e,j} \leftarrow \tilde{x} \oplus \pi^e(r^{e,j})$, $z_2^{e,j} \leftarrow H' \cdot z_1^{e,j} \oplus y \cdot 1^2_j$, and $z^{e,j} \leftarrow (z_1^{e,j}||z_2^e)$.
- All users broadcast $z^{e,j}$ and reconstruct $z^e = \bigoplus_j z^{e,j}$.
- For $d = 1$ to D, set:
  - $z^{e,j}_{d,0}[1] \leftarrow X^{e,j}_{d,0} \oplus \pi^e(R^{e,j}_{d,0})$, $z^{e,j}_{d,0}[2] \leftarrow H' \cdot z^{e,j}_{d,0}[1] \oplus y$, and $z^{e,j}_{d,0} \leftarrow (z^{e,j}_{d,0}[1]||z^{e,j}_{d,0}[2])$;
  - $z^{e,j}_{d,1}[1] \leftarrow z_1^{e,j} \oplus z^{e,j}_{d,0}$, $z^{e,j}_{d,1}[2] \leftarrow z_2^{e,j} \oplus z^{e,j}_{d,0}[2]$, and $z^{e,j}_{d,1} \leftarrow (z^{e,j}_{d,1}[1]||z^{e,j}_{d,1}[2])$;
  - $\bar{x}^{e,j}_{d,0} \leftarrow z^e \cdot 1^3_j + (1 - z^e) \cdot \pi^e(U^{e,j}_{d,0}) \bmod 3$
  - $\bar{x}^e_{d,1} \leftarrow \tilde{x}_j - \bar{x}^{e,j}_{d,0} \bmod 3$.
  - For $b = 0, 1$, set $\mathsf{msg}^{e,j}_{d,b} \leftarrow (z^{e,j}_{d,b}, \mathsf{BHW}_6(\bar{x}^{e,j}_{d,b}))$.
  - For $b = 0, 1$, all users broadcast $\mathsf{msg}^{e,j}_{d,b}$ and reconstruct $\mathsf{msg}^e_{d,b} = \sum_j \mathsf{msg}^{e,j}_{d,b} \bmod 3$.
- Set $h_2 \leftarrow \mathsf{H}_2(m, \mathsf{salt}, h_1, (\mathsf{msg}^e_{d,b})_{d\leq D, b\in\{0,1\}, e\leq\tau})$.
- Set $(i^e)_{e\leq\tau} \leftarrow \mathsf{PRG}_2(h_2)$.
- Each user $\mathcal{U}_j$ broadcasts $\mathsf{copath}^{e,j} = \mathsf{CoPath}_{\mathsf{salt}}(\mathsf{sd}^{e,j}, i^e)$ and $\mathsf{com}^{e,j}_{i_e}$ for $e = 1$ to τ. // This is the **Opening** phase in the template protocol of Section 3.3.

Output. The parties abort if the signature σ below does not verify.

$$\sigma = \left(\mathsf{salt}, h_1, h_2, \left(\mathsf{copath}^{e,j}, \mathsf{com}^{e,j}_{i_e}\right)_{e\leq\tau,\, j\leq n}, (z^e, \mathsf{aux}^e_N)_{e\leq\tau}\right).$$

5.3 Security analysis

Theorem 6. *Let $\mathcal{A}$ denote an adversary corrupting at most $n-1$ users, engaging in any polynomial number* Ns *of threshold signing sessions, making at most* Q *queries to the random oracle* H_1 *and to the ideal cipher, and outputting a forgery (m^*, σ^*) after all sessions, where m^* is a message that was never signed during a signing session. Then in the $(\mathcal{F}_{\mathsf{salt}}, \mathcal{F}_{2,3})$-hyrid model, it holds that*

$$\Pr[\mathsf{Verify}(m, \mathsf{pk}, \sigma^*) = 1] \leq \frac{\mathsf{Q}^2 + \mathsf{Ns}n\mathsf{Q}}{2^{2\lambda}} + \mathsf{Adv}^{\mathsf{Commit}}(\mathcal{A}) + \mathsf{Adv}^{\mathsf{CEUFCMA}}(\mathcal{A}).$$

Proof. Let $\mathcal{A}$ denote an adversary corrupting all users except $\mathcal{U}_\ell$ (the case of a smaller number of corrupted parties proceeds similarly to the all-but-one corruption case). We describe below a simulator Sim that interacts with the corruptible signing functionality and is given (non-programming) access to the random oracle H_1 and to the ideal cipher underlying the multi-instance PPRF.

KeyGen. Sim invokes the key generation of the corruptible signing oracle and receives $\mathsf{pk} = (\mathsf{sd}, y)$. It recomputes $H \leftarrow \mathsf{PRG}(\mathsf{sd})$ and picks $(\tilde{x}_j)_{j\neq\ell} \leftarrow\$ (\mathbb{F}_2^{K-k})^{n-1}$ uniformly at random. It outputs $\mathsf{sk}_j = (H, \tilde{x}_j, y)$ to each corrupted user $\mathcal{U}_j$.

Sampling the salt. Sim honestly emulate $\mathcal{F}_{\mathsf{salt}}$.

Building the trees. Sim stores the inputs $\tilde{r}^{e,j}$ of all corrupted users to $\mathcal{F}_{2,3}$ and randomly samples $(u^{e,j})_{j\neq\ell} \leftarrow\$ (\mathbb{F}_3^{K-k})^{n-1}$. It returns $u^{e,j}$ to each corrupted user $\mathcal{U}_j$.

Commit-and-open phase. Sim broadcasts a dummy commitment c^ℓ. Upon receiving all commitments $(c^j)_{j\neq\ell}$, for each $j \neq \ell$,

- Sim extracts the value $\tilde{h}^j$ contained in c^j
- Sim searches among all queries to H_1 for a preimage of $\tilde{h}^j$ of the form $\mathbf{com}^j = (\mathsf{com}_1^{1,j}, \cdots, \mathsf{com}_N^{1,j}, \cdots, \mathsf{com}_1^{\tau,j}, \cdots, \mathsf{com}_N^{\tau,j})$. If there is no such preimages, or if there are multiple preimages, or if the preimage is not correctly formatted, Sim raises a flag fail.
- For all tuples $\mathbf{com}^j$ for which Sim did not raise a flag fail, for $e = 1$ to τ, Sim searches among all queries to the ideal cipher for preimages on the nodes of a GGM tree with salt salt_e and commitments $(\mathsf{com}_1^{e,j}, \cdots, \mathsf{com}_N^{e,j})$ at the leaves. If it does not find either a root seed $\mathsf{sd}^{j,e}$ or a set of seeds $(v^1_{e,j}, \cdots, v^D_{e,j})$ on the co-path to a leaf $i^*_{e,j}$ together with the $i^*_{e,j}$-th leaf seed $\mathsf{sd}^*_{e,j}$, it raises a flag fail. Else, it sets $K^{e,j}$ to be either $\mathsf{sd}^{e,j}$ or $\mathsf{cp}^{e,j} = (i^*_{e,j}, v^1_{e,j}, \cdots, v^D_{e,j}, \mathsf{sd}^*_{e,j})$ and $r^j \leftarrow (K^{e,j})_{e\leq\tau}$.
- If Sim did not raise a flag fail, it sends $(m, \ell, (r^j)_{j\neq\ell})$ to the corruptible signing functionality. Upon receiving a signature σ, it recomputes h_1^ℓ from the signature via the verification algorithm.

- Else, Sim picks random root seeds $\mathsf{sd}^{e,\ell}$ and constructs h_1^ℓ as a honest user, using $u^{e,\ell} \leftarrow\!\!\$ \, \mathbb{F}_3^{K-k}$ as simulated output of $\mathcal{F}_{2,3}$.
- Sim adapts the opening of c^ℓ to h_1^ℓ.

Distributed virtual protocol ($\mathsf{flag} \neq \mathsf{fail}$)**.** Provided that it did not raise a flag fail, Sim can recompute at this stage from the r^j all values $z^{e,j}, \mathsf{msg}_{d,b}^{e,j}$ for $j \neq \ell$, $e = 1$ to τ, $d = 1$ to D, and $b \in \{0,1\}$. From σ, by running the verification algorithm, Sim obtains z^e and $\mathsf{msg}_{d,b}^e$ for $e = 1$ to τ, $d = 1$ to D, and $b \in \{0,1\}$.
- For $e = 1$ to τ, Sim defines $z^{e,\ell} \leftarrow z^e \oplus \bigoplus_{j \neq \ell} z^{e,j}$ and broadcasts $z^{e,\ell}$.
- For $e = 1$ to τ, $d = 1$ to D, and $b \in \{0,1\}$, Sim broadcasts $\mathsf{msg}_{d,b}^{e,\ell} \leftarrow \mathsf{msg}_{d,b}^e - \sum_{j \neq \ell} \mathsf{msg}_{d,b}^{e,j} \bmod 3$.
- Eventually, Sim obtains from σ and broadcasts $(\mathsf{copath}^{e,\ell}, \mathsf{com}_{i^e}^{e,\ell})$ for $e = 1$ to τ.

Distributed virtual protocol ($\mathsf{flag} = \mathsf{fail}$)**.** If $\mathsf{flag} = \mathsf{fail}$, Sim broadcasts uniformly random $z^{e,\ell}, \mathsf{msg}_{d,b}^{e,\ell}$, as well as the correct co-path and commitment $(\mathsf{copath}^{e,\ell}, \mathsf{com}_{i^e}^{e,\ell})$ computed from its random root seeds.

As Sim is only given access to the corruptible signing functionality, it is clear that the advantage of $\mathcal{A}$ in outputting a valid forgery (m^*, σ^*) after interacting polynomially many times with Sim is at most $\mathsf{Adv}^{\mathsf{CEUFCMA}}(\mathcal{A})$. We now show that Sim's emulation is indistinguishable from a real interaction with $\mathcal{U}_\ell$. We proceed via a sequence of game hops.

Game 1 (Gm^1). This is the real game. The game honestly runs the key generation and distributes sk_j to each corrupted user $\mathcal{U}_j$. In each signing phase, the game honestly emulates $\mathcal{U}_\ell$.

Game 2 (Gm^2). In this game, we raise a flag fail if any collision occurs in H_1 or in the ideal cipher. As the total number of queries to either H_1 or the ideal cipher is at most Q, we have $\Pr[\mathsf{fail}(\mathsf{Gm}^2)] \leq \frac{\mathsf{Q}^2}{2^{2\lambda}}$.

Game 3 (Gm^3). In this game, we raise a flag fail if a signature contains $h_1^j = \mathsf{H}_1(\mathbf{com}^j)$ such that $\mathbf{com}^j$ was not queries to H_1 prior to sending c^j, and a flag ver if any of the two checks of this same signature verifies. By the one-wayness of H_1, we have $\Pr[\mathsf{ver} \mid \mathsf{fail}(\mathsf{Gm}^3)] \leq \frac{\mathsf{Ns}n\mathsf{Q}}{2^{2\lambda}}$, where the bound is reached only if all signing sessions are performed in parallel and the adversary attempts to find a preimage to any c^j of any signing session after sending c^j (and before running the distributed virtual protocol).

Game 4 (Gm^4). In this game, when $\mathsf{flag} = \mathsf{fail}$, we emulate $\mathcal{U}_\ell$ as the simulator Sim, computing a honest GGM tree but sampling dummy $u^{e,\ell}$, $z^{e,\ell}$, $\mathsf{msg}_{d,b}^{e,\ell}$. Conditioned on $\mathsf{flag} = \mathsf{fail}$, with probability at least $1 - \frac{\mathsf{Ns}n\mathsf{Q}}{2^{2\lambda}}$, both checks (for h_1 and h_2) fail due to an incorrect h_1^j. In this case, all honestly computed $z^{e,\ell}$, $\mathsf{msg}_{d,b}^{e,\ell}$ are distributed as random elements over $\mathbb{F}_2^K$ and $\mathbb{F}_3^K \times \mathbb{F}_3^w$ respectively (they are random share of values for which one share is missing).

Game 5 (Gm^5). This game is the simulation, where Sim does not use the secret key sk. Instead, Sim interacts with the corruptible signing oracle. Observe that whenever $\mathsf{flag} \neq \mathsf{fail}$, the only difference between the transcript of the simulated interaction and the transcript of Game 4 is the commitment c^ℓ, which Sim computes as an equivocable commitment to a dummy string. Therefore, the advantage of any adversary $\mathcal{A}$ in distinguishing Game 4 from Game 5 is at most $\mathsf{Adv}^{\mathsf{Commit}}(\mathcal{A})$. This concludes the proof of Theorem 6.

References

1. Aaraj, N., Bettaieb, S., Bidoux, L., Budroni, A., Dyseryn, V., Esser, A., Gaborit, P., Kulkarni, M., Mateu, V., Palumbi, M., Perin, L., Tillich, J.: PERK. Tech. rep., National Institute of Standards and Technology (2023), available at `https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures`
2. Adj, G., Rivera-Zamarripa, L., Verbel, J., Bellini, E., Barbero, S., Esser, A., Sanna, C., Zweydinger, F.: MiRitH — MinRank in the Head. Tech. rep., National Institute of Standards and Technology (2023), available at `https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures`
3. Aguilar-Melchor, C., Feneuil, T., Gama, N., Gueron, S., Howe, J., Joseph, D., Joux, A., Persichetti, E., Randrianarisoa, T.H., Rivain, M., Yue, D.: SDitH — Syndrome Decoding in the Head. Tech. rep., National Institute of Standards and Technology (2023), available at `https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures`
4. Aguilar-Melchor, C., Gama, N., Howe, J., Hülsing, A., Joseph, D., Yue, D.: The return of the SDitH. pp. 564–596. LNCS (2023)
5. Aragon, N., Bardet, M., Bidoux, L., Chi-Domínguez, J.J., Dyseryn, V., Feneuil, T., Gaborit, P., Joux, A., Rivain, M., Tillich, J., Vinçotte, A.: RYDE. Tech. rep., National Institute of Standards and Technology (2023), available at `https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures`

6. Aragon, N., Bardet, M., Bidoux, L., Chi-Domínguez, J., Dyseryn, V., Feneuil, T., Gaborit, P., Neveu, R., Rivain, M., Tillich, J.: MIRA. Tech. rep., National Institute of Standards and Technology (2023), available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures
7. Augot, D., Finiasz, M., Sendrier, N.: A fast provably secure cryptographic hash function. Cryptology ePrint Archive, Report 2003/230 (2003), https://eprint.iacr.org/2003/230
8. Baum, C., Beullens, W., Mukherjee, S., Orsini, E., Ramacher, S., Rechberger, C., Roy, L., Scholl, P.: One tree to rule them all: Optimizing GGM trees and OWFs for post-quantum signatures. In: ASIACRYPT 2024 (to appear) (2024), https://eprint.iacr.org/2024/490
9. Baum, C., Braun, L., de Saint Guilhem, C.D., Klooß, M., Majenz, C., Mukherjee, S., Orsini, E., Ramacher, S., Rechberger, C., Roy, L., Scholl, P.: FAEST. Tech. rep., National Institute of Standards and Technology (2023), available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures
10. Baum, C., Braun, L., de Saint Guilhem, C.D., Klooß, M., Orsini, E., Roy, L., Scholl, P.: Publicly verifiable zero-knowledge and post-quantum signatures from vole-in-the-head. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V. Lecture Notes in Computer Science, vol. 14085, pp. 581–615. Springer (2023), https://doi.org/10.1007/978-3-031-38554-4_19
11. Baum, C., Delpech de Saint Guilhem, C., Kales, D., Orsini, E., Scholl, P., Zaverucha, G.: Banquet: Short and fast signatures from AES. pp. 266–297. LNCS (2021)
12. Bernstein, D.J., Lange, T., Peters, C., Schwabe, P.: Really fast syndrome-based hashing. In: Nitaj, A., Pointcheval, D. (eds.) AFRICACRYPT 11. LNCS, vol. 6737, pp. 134–152 (Jul 2011)
13. Bettale, L., Kahrobaei, D., Perret, L., Verbel, J.: Biscuit. Tech. rep., National Institute of Standards and Technology (2023), available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures
14. Bombar, M., Bui, D., Couteau, G., Couvreur, A., Ducros, C., Servan-Schreiber, S.: FOLEAGE: $\mathbb{F}_4$OLE-based multi-party computation for boolean circuits. In: ASIACRYPT 2024 (to appear) (2024), https://eprint.iacr.org/2024/429
15. Bombar, M., Couteau, G., Couvreur, A., Ducros, C.: Correlated pseudorandomness from the hardness of quasi-abelian decoding. pp. 567–601. LNCS (2023)
16. Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part II. LNCS, vol. 8270, pp. 280–300 (Dec 2013)
17. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 896–912. ACM Press (Oct 2018)
18. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Resch, N., Scholl, P.: Correlated pseudorandomness from expand-accumulate codes. pp. 603–633. LNCS (2022)
19. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Rindal, P., Scholl, P.: Efficient two-round OT extension and silent non-interactive secure computation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 291–308. ACM Press (Nov 2019)
20. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 489–518 (Aug 2019)
21. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators from ring-LPN. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part II. LNCS, vol. 12171, pp. 387–416 (Aug 2020)
22. Boyle, E., Goldwasser, S., Ivan, I.: Functional signatures and pseudorandom functions. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 501–519 (Mar 2014)
23. Brandão, L.T.A.N., Peralta, R.: NIST First Call for Multi-Party Threshold Schemes. Tech. Rep. NIST IR 8214C (Initial Public Draft), National Institute of Standards and Technology (January 2023), https://doi.org/10.6028/NIST.IR.8214C.ipd, public comment period closed on April 10, 2023.
24. Bui, D., Carozza, E., Couteau, G., Goudarzi, D., Joux, A.: Short signatures from regular syndrome decoding, revisited. In: ASIACRYPT 2024 (to appear) (2024), https://eprint.iacr.org/2024/252
25. Canetti, R., Chen, Y., Holmgren, J., Lombardi, A., Rothblum, G.N., Rothblum, R.D., Wichs, D.: Fiat-Shamir: from practice to theory. In: Charikar, M., Cohen, E. (eds.) 51st ACM STOC. pp. 1082–1090. ACM Press (Jun 2019)
26. Carozza, E., Couteau, G., Joux, A.: Short signatures from regular syndrome decoding in the head. pp. 532–563. LNCS (2023)
27. Chiesa, A., Manohar, P., Spooner, N.: Succinct arguments in the quantum random oracle model. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019, Part II. LNCS, vol. 11892, pp. 1–29 (Dec 2019)
28. Couteau, G., Rindal, P., Raghuraman, S.: Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. pp. 502–534. LNCS (2021)
29. Cui, H., Liu, H., Yan, D., Yang, K., Yu, Y., Zhang, K.: ReSolveD: Shorter signatures from regular syndrome decoding and VOLE-in-the-head. pp. 229–258. LNCS (2024)

30. Delpech de Saint Guilhem, C., De Meyer, L., Orsini, E., Smart, N.P.: BBQ: Using AES in picnic signatures. In: Paterson, K.G., Stebila, D. (eds.) SAC 2019. LNCS, vol. 11959, pp. 669–692 (Aug 2019)
31. Dinur, I., Nadler, N.: Multi-target attacks on the Picnic signature scheme and related protocols. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part III. LNCS, vol. 11478, pp. 699–727 (May 2019)
32. Doerner, J., Kondi, Y., Rosenbloom, L.N.: Sometimes you can't distribute random-oracle-based proofs. pp. 323–358. LNCS (2024)
33. Feneuil, T., Joux, A., Rivain, M.: Syndrome decoding in the head: Shorter signatures from zero-knowledge proofs. pp. 541–572. LNCS (2022)
34. Feneuil, T., Rivain, M.: MQOM — MQ on my Mind. Tech. rep., National Institute of Standards and Technology (2023), available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures
35. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. Journal of the ACM 33(4), 792–807 (Oct 1986)
36. Guo, X., Yang, K., Wang, X., Zhang, W., Xie, X., Zhang, J., Liu, Z.: Half-tree: Halving the cost of tree expansion in COT and DPF. pp. 330–362. LNCS (2023)
37. Hazay, C., Orsini, E., Scholl, P., Soria-Vazquez, E.: TinyKeys: A new approach to efficient multi-party computation. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part III. LNCS, vol. 10993, pp. 3–33 (Aug 2018)
38. Huth, J., Joux, A.: MPC in the head using the subfield bilinear collision problem. pp. 39–70. LNCS (2024)
39. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: Johnson, D.S., Feige, U. (eds.) 39th ACM STOC. pp. 21–30. ACM Press (Jun 2007)
40. Kales, D., Zaverucha, G.: An attack on some signature schemes constructed from five-pass identification schemes. In: Krenn, S., Shulman, H., Vaudenay, S. (eds.) CANS 20. LNCS, vol. 12579, pp. 3–22 (Dec 2020)
41. Kales, D., Zaverucha, G.: Improving the performance of the Picnic signature scheme. IACR TCHES 2020(4), 154–188 (2020), https://tches.iacr.org/index.php/TCHES/article/view/8680
42. Katz, J., Kolesnikov, V., Wang, X.: Improved non-interactive zero knowledge with applications to post-quantum signatures. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 525–537. ACM Press (Oct 2018)
43. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 669–684. ACM Press (Nov 2013)
44. Ouyang, Y., Tang, D., Xu, Y.: Code-based zero-knowledge from VOLE-in-the-head and their applications: Simpler, faster, and smaller. In: ASIACRYPT 2024 (to appear) (2024), https://eprint.iacr.org/2024/1414
45. Pino, R.D., Katsumata, S., Maller, M., Mouhartem, F., Prest, T., Saarinen, M.J.O.: Threshold raccoon: Practical threshold signatures from standard lattice assumptions. pp. 219–248. LNCS (2024)
46. Rindal, P., Schoppmann, P.: VOLE-PSI: Fast OPRF and circuit-PSI from vector-OLE. pp. 901–930. LNCS (2021)
47. Wang, X., Ranellucci, S., Katz, J.: Authenticated garbling and efficient maliciously secure two-party computation. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 21–37. ACM Press (Oct / Nov 2017)
48. Wang, X., Ranellucci, S., Katz, J.: Global-scale secure multiparty computation. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 39–56. ACM Press (Oct / Nov 2017)
49. Weng, C., Yang, K., Katz, J., Wang, X.: Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. pp. 1074–1091. IEEE Computer Society Press (2021)
50. Yang, K., Sarkar, P., Weng, C., Wang, X.: QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. pp. 2986–3001. ACM Press (2021)
51. Yang, K., Weng, C., Lan, X., Zhang, J., Wang, X.: Ferret: Fast extension for correlated OT with small communication. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 20. pp. 1607–1626. ACM Press (Nov 2020)
52. Zaverucha, G., Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., Katz, J., Wang, X., Kolesnikov, V., Kales, D.: Picnic. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions

A Instantiating the functionality

In this section, we provide simple protocols to securely instantiate $\mathcal{F}_{2,3}$ in the malicious setting using recent results on pseudorandom correlation generators [17, 19, 20]. Before we proceed with their description, we make two observations:

- The simulation of our threshold signing protocol is oblivious to whether the output of $\mathcal{F}_{2,3}$ is correct or not. This is because Sim emulates the protocol without using the share output by $\mathcal{F}_{2,3}$ (or using a random share instead when $\mathsf{flag} = \mathsf{fail}$), and because the protocol does not verify that the corrupted parties are actually using the correct output obtained via the functionality. This is a relatively standard behavior in threshold signatures: the correct behavior of the participants is guaranteed by the validity of the signature produced by the protocol (in fact, many existing threshold signatures leverage a similar observation to obtain more efficient constructions). In our context, this means that we only need to instantiate the functionality which guarantees correctness when the users are honest, and *privacy of the honest parties' inputs* when some participants are malicious. This dispenses us from the need to use any expensive zero-knowledge proofs or similar mechanisms.
- In our model, where the key generation is executed by a trusted entity (typically the owner of sk), we can assume that the trusted dealer additionally generates and include in the key shares sk_j *helper strings* (e.g. correlated randomness) that the parties can use to facilitate the secure instantiation of $\mathcal{F}_{2,3}$.

A construction for $n = 2$. Equipped with these observations, we now proceed with our constructions, starting with the two-party setting ($n = 2$). We first recall some background on PCGs. A PCG for a target correlation C (a 2-party correlation is a distribution over pairs of values) is a pair $(\mathsf{Setup}, \mathsf{Expand})$ such that

- $\mathsf{Setup}(1^\lambda)$ produces short keys c_0, c_1, and
- $\mathsf{Expand}(\sigma, c_\sigma)$ outputs a long string y_σ,

such that (y_0, y_1) are indistinguishable from a random sample from C. We refer the reader to Section B.4 of the Appendix B for a more formal definition (taken almost verbatim from [15]). The (length-t) OLE correlation over a field $\mathbb{F}$ refers to the following correlation: the two users $\mathcal{U}_1,\mathcal{U}_2$ receive $(\mathbf{z}_1, \mathbf{x}) \in (\mathbb{F}^t)^2$ and $(\mathbf{z}_2, \mathbf{y}) \in (\mathbb{F}^t)^2$ respectively, such that

$$\mathbf{z}_0 + \mathbf{z}_1 = \mathbf{x} \odot \mathbf{y},$$

where $\odot$ denotes the Schur (i.e., component-wise) product. We say that a PCG for the OLE correlation is *programmable* if, informally, the randomness used to generate $\mathbf{x}$ or $\mathbf{y}$ can be fixed across different instances (such that $\mathcal{U}_1$ can obtain an OLE $(\mathbf{z}_1, \mathbf{x})$ with a user $\mathcal{U}_2$, and a second tuple $(\mathbf{z}'_1, \mathbf{x})$ with the same $\mathbf{x}$ with another user $\mathcal{U}_3$. We again refer the reader to Appendix B.4 for a formal definition. We use the following result from [14, 15]:

Lemma 7. *Assuming the quasi-abelian syndrome decoding assumption, there exists a programmable PCG for the length-t OLE correlation over $\mathbb{F}_q$ for any $q > 2$. Furthermore, the key size is bounded by $O(\lambda^3 \log t)$ and the runtime of* Expand *is $\tilde{O}(t)$.*

We note that the work of [14] introduces a number of algorithmic and low-level optimizations that demonstrate that this PCG achieves a very good concrete performance over small fields: [14] reports generating 12 millions OLEs over $\mathbb{F}_4$. While our setting is slightly different as we work over $\mathbb{F}_3$, most of their optimizations carry over directly to our setting and we expect a similar, if not better, efficiency.

Protocol $\Pi^2_{2,3}$

Parameters. A PCG $(\mathsf{Setup}, \mathsf{Expand})$ for length-t OLEs over $\mathbb{F}_3$
Trusted setup. The dealer generates $(c_1, c_2) \leftarrow\$ \mathsf{Setup}(1^\lambda)$ and adds them to the secret keys of $\mathcal{U}_1,\mathcal{U}_2$ respectively.
Protocol.

1. For $i = 1, 2$, given inputs $\mathbf{r}_i \in \mathbb{F}_2^t$, user $\mathcal{U}_i$ generates $(\mathbf{z}_i, \mathbf{x}_i) \leftarrow \mathsf{Expand}(i, c_i)$ and broadasts $\mathbf{v}_i = \mathbf{x}_i + \mathbf{r}_i \bmod 3$.
2. User $\mathcal{U}_1$ outputs $\mathbf{z}_1 - \mathbf{v}_2 \odot \mathbf{x}_1 + \mathbf{r}_1 \bmod 3$.
3. User $\mathcal{U}_2$ outputs $\mathbf{z}_2 + \mathbf{v}_1 \odot \mathbf{r}_2 + \mathbf{r}_2 \bmod 3$.

Protocol 1: A 2-party protocol for securely instantiating the functionality $\mathcal{F}_{2,3}$

We first show that Protocol 1 is correct. Over $\mathbb{F}_3$, we have

$$\begin{aligned}&(\mathbf{z}_1 - \mathbf{v}_2 \odot \mathbf{x}_1 + \mathbf{r}_1) + (\mathbf{z}_2 + \mathbf{v}_1 \odot \mathbf{r}_2 + \mathbf{r}_2)\\ =&(\mathbf{z}_1 + \mathbf{z}_2) + (\mathbf{r}_1 + \mathbf{x}_1) \odot \mathbf{r}_2 - (\mathbf{r}_2 + \mathbf{x}_2) \odot \mathbf{x}_1 + \mathbf{r}_1 + \mathbf{r}_2\\ &= \mathbf{x}_1 \odot \mathbf{x}_2 + \mathbf{r}_1 \odot \mathbf{r}_2 + \mathbf{x}_1 \odot \mathbf{r}_2 - \mathbf{r}_2 \odot \mathbf{x}_1 - \mathbf{x}_1 \odot \mathbf{x}_2 + \mathbf{r}_1 + \mathbf{r}_2\\ &= \mathbf{r}_1 \odot \mathbf{r}_2 + \mathbf{r}_1 + \mathbf{r}_2\\ &= \mathbf{r}_1 \oplus \mathbf{r}_2.\end{aligned}$$

Then, security follows immediately from the fact that by the PCG security, $\mathbf{x}_i$ is computationally indistinguishable from a random vector over $\mathbb{F}_3^t$ given c_{3-i}, hence $\mathbf{v}_i$ computationally masks $\mathbf{r}_i$ over $\mathbb{F}_3$.

General case. For an arbitrary number of users, we rely on a generalization of protocol $\Pi_{2,3}$ in a tree-based fashion. First, we start by describing a protocol $\Pi_{\mathsf{xor}}^{2^i}$ where 2^i parties, given as input additive shares over $\mathbb{F}_3$ of two bits a, b, generate additive shares over $\mathbb{F}_3$ of the xor $a \oplus b$.

Protocol $\Pi_{\mathsf{xor}}^{2^i}$

Trusted setup. The dealer samples $(u, v) \leftarrow\$ \mathbb{F}_3^2$ and set $(u_j, v_j, w_j)_{j \leq 2^i}$ to be random shares of $(u, v, u \cdot v)$ over $\mathbb{F}_3$. Each user $\mathcal{U}_j$ receives (u_j, v_j, w_j).

Protocol. The users have $\mathbb{F}_3$-shares $(a_j)_{j \leq 2^i}$ and $(b_j)_{j \leq 2^i}$ of bits a, b.

1. Each user $\mathcal{U}_j$ broadcasts $u_j + a_j \bmod 3$ and $v_j + b_j \bmod 3$. All users reconstruct $\sum_j (u_j + a_j) = u + a$ and $\sum_j (v_j + b_j) = v + b$.
2. Each user $\mathcal{U}_j$ outputs $z_j = (a + u)b_j - (v + b)u_j + w_j + a_j + b_j \bmod 3$.

Protocol 2: A 2^i-party protocol for computing shares of the XOR of two bits shared over $\mathbb{F}_3$

Correctness follows again easily by inspection, as $\sum_j z_j = (a+u)b-(v+b)u+uv+a+b = ab+a+b = a \oplus b$ over $\mathbb{F}_3$, and security follows from the fact that each a_j (resp. b_j) is perfectly masked by u_j (resp. v_j) over $\mathbb{F}_3$. Equipped with protocol $\Pi_{\mathsf{xor}}^{2^i}$, we describe a protocol (in the $\mathcal{F}_{\mathsf{xor}}^{2^i}$-hybrid model) that securely instantiates $\mathcal{F}_{2,3}$ with n users.

Protocol $\Pi_{2,3}^n$

Input. The parties $\mathcal{U}_j$ each have an input $r_j \in \mathbb{F}_2$.

Protocol. Assume that $n = 2^d$ is a power of 2. Place the n on the leaves of a full binary tree of depth d. The protocol proceeds in d rounds. In round i, all users who share a common ancestor on the $i+1$-th layer interact. Set $i = 1$ to be the leaf layer.

1. Every 2^i-tuples S of users who have an ancestor on the layer $i + 1$ (there are 2^{d-i-1} disjoint tuples) partition S into two equal-sized sets S_0, S_1 of users that share a common ancestor on the i-th layer. The users $(\mathcal{U}_j)_{j \in S_b}$ have additive shares of a bit r_{S_b} over $\mathbb{F}_3$. By adding dummy 0 shares, we equivalently assume that all users $(\mathcal{U}_j)_{j \in S}$ have 2^j-user shares of (r_{S_0}, r_{S_1}).
2. The users $(\mathcal{U}_j)_{j \in S}$ invoke $\mathcal{F}_{\mathsf{xor}}^{2^i}$ to obtain shares of $r_S := r_{S_0} \oplus r_{S_1}$ and set $i = i + 1$
3. When $i = d$, all users output their $\mathbb{F}_3$-shares of $r_{[2^d]} = \bigoplus r_j$.

Protocol 3: An n-party protocol for securely instantiating the functionality $\mathcal{F}_{2,3}^n$ in the $\mathcal{F}_{\mathsf{xor}}^{2^i}$-hybrid model

The proof of correctness and privacy against malicious users of Protocol 3, in the $\mathcal{F}_{\mathsf{xor}}^{2^i}$-hybrid, is straightforward. The protocol $\Pi_{2,3}^n$ can be generalized to producing $\mathbb{F}_3$-shares of the XOR of $\mathbb{F}_2^t$-vectors via a direct parallel repetion.

Eventually, instantiating $\mathcal{F}_{\mathsf{xor}}^{2^i}$ via $\Pi_{\mathsf{xor}}^{2^i}$ requires access to a trusted source of random 2^i-user Beaver triples over $\mathbb{F}_3$. This can be instantiated efficiently using again the PCG for OLEs over $\mathbb{F}_3$ from [15]: as their PCG is programmable, and as explained in their work, if each pair of users is given a pair of (programmed) PCG seeds, they can locally recombine the pseudorandom outputs into a 2^i-user $\mathbb{F}_3$-Beaver triples. This yields the following efficient instantiation of $\mathcal{F}_{2,3}$: in the trusted setup phase, the trusted dealer generates as many pairwise PCG seeds as required to instantiate $\Pi_{\mathsf{xor}}^{2^i}$ at every level of the tree-based construction from Protocol 3. Then, the parties locally expand their seeds into arbitrary length Beaver triples over $\mathbb{F}_3$, and run the efficient protocol from Protocol 3 to convert $\mathbb{F}_2$-shares into $\mathbb{F}_3$-shares.

B Additional Preliminaries

B.1 Existential Unforgeability against Key-Only Attacks

Definition 8 (EUF-KO security). *Given a signature scheme* $\mathsf{Sig} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *and security parameter* λ*, we say that* Sig *is* EUFKO*-secure if any PPT algorithm* $\mathcal{A}$ *has negligible advantage in the EUF-KO game, defined as*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{EUFKO}} = \Pr\left[\mathsf{Verify}(\mathsf{pk}, \mu^*, \sigma^*) = 1 \,\middle|\, \begin{array}{c} (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Setup}(\{0,1\}^\lambda) \\ (\mu^*, \sigma^*) \leftarrow \mathcal{A}(\mathsf{pk}) \end{array}\right].$$

B.2 Multi-instance PRGs

We recall the notion of multi-instance PRG from [24]. We use $(\mathsf{F}_0, \mathsf{F}_1)$ to denote functions that compute the left half and right half of the length-doubling PRG output. The definitions of this section are taken essentially verbatim from [24].

Definition 9 ((Q,τ)-instance (t,ϵ)-secure PRG). *A PRG* $\mathsf{PRG} = (\mathsf{F}_0, \mathsf{F}_1)$ *with* $\mathsf{F}_b : \{0,1\}^{2\lambda} \to \{0,1\}^\lambda$ *is an* (Q,τ)*-instance* (t,ϵ)*-secure length-doubling* PRG *if for every non-uniform PPT distinguisher* $\mathcal{D}$ *running in time at most* t*, it holds that for all sufficiently large* λ*,*

$$\mathsf{Adv}^{\mathsf{PRG}}(\mathcal{D}) = |\Pr[\mathsf{Exp}_{\mathcal{D}}^{\mathsf{rw\text{-}prg}}(\lambda) = 1] - \Pr[\mathsf{Exp}_{\mathcal{D}}^{\mathsf{iw\text{-}prg}}(\lambda) = 1]| \le \epsilon(\lambda),$$

where $\mathsf{Exp}_{\mathcal{D}}^{\mathsf{rw\text{-}prg}}(\lambda)$ *and* $\mathsf{Exp}_{\mathcal{D}}^{\mathsf{iw\text{-}prg}}(\lambda)$ *are defined below.*

$\mathsf{Exp}_{\mathcal{D}}^{\mathsf{rw\text{-}prg}}(\lambda)$:	$\mathsf{Exp}_{\mathcal{D}}^{\mathsf{iw\text{-}prg}}(\lambda)$:
– $(\mathsf{salt}_1, \mathsf{salt}_2, \ldots, \mathsf{salt}_{2Q}) \leftarrow_r \{0,1\}^\lambda$ – $(\mathsf{sd}_{i,e})_{i \le Q, e \le \tau} \leftarrow_r (\{0,1\}^\lambda)^{Q \cdot \tau}$ – $\forall i \le Q, e \le \tau$: • $y_{2i-1,e} \leftarrow \mathsf{F}_0(\mathsf{sd}_{i,e}, \mathsf{salt}_{2i-1})$ • $y_{2i,e} \leftarrow \mathsf{F}_1(\mathsf{sd}_{i,e}, \mathsf{salt}_{2i})$	– $(\mathsf{salt}_1, \mathsf{salt}_2, \ldots, \mathsf{salt}_{2Q}) \leftarrow_r \{0,1\}^\lambda$ – $(y_{i,e})_{i \le 2Q, e \le \tau} \leftarrow_r (\{0,1\}^\lambda)^{2Q \cdot \tau}$
Output $b \leftarrow \mathcal{D}\big((\mathsf{salt}_i, (y_{i,e})_{e \le \tau})_{i \le 2Q}\big)$	**Output** $b \leftarrow \mathcal{D}\big((\mathsf{salt}_i, (y_{i,e})_{e \le \tau})_{i \le 2Q}\big)$

The definition extends immediately to PRGs that stretch their seeds by a larger factor. The definition assumes each of F_0 and F_1 to take a distinct λ-bit salt, in line with how the salt is actually used in the concrete AES-based construction, where $\mathsf{F}_b(\mathsf{sd}, \mathsf{salt}_b)$ is defined to be $\mathsf{AES}_{\mathsf{salt}_b}(\mathsf{sd})$.

B.3 Multi-instance puncturable pseudorandom functions

Pseudorandom functions [35], are families of keyed functions F_k such that no adversary can distinguish between a black-box access to F_k for a random key k and access to a truly random function. A puncturable pseudorandom function (PPRF) [16,22,43] is a PRF F such that given an input x, and a PRF key k, one can generate a *punctured* key, denoted $k\{x\} = F.\mathsf{Punc}(K, x)$, which allows evaluating F at every point except for x (*i.e.*, there is an algorithm $F.\mathsf{Eval}$ such that $F.\mathsf{Eval}(k\{x\}, x') = F_K(x')$ for all $x' \ne x$), and such that $F_k(x)$ is indistinguishable from random given $k\{x\}$. The definition of multi-instance PPRF below is taken essentially verbatim from [24].

Definition 10 ((Q, τ)-instance (t, ϵ)-secure PPRF). *A function family $F = \{F_K\}$ with input domain $[2^D]$, salt domain $\{0,1\}^s$, and output domain $\{0,1\}^\lambda$, is an (Q, τ)-instance (t, ϵ)-secure PPRF if it is a PPRF which additionally takes as input a salt* salt*, and for every non-uniform PPT distinguisher $\mathcal{D}$ running in time at most t, it holds that for all sufficiently large λ,*

$$\mathsf{Adv}^{\mathsf{PPRF}}(\mathcal{D}) = |\Pr[\mathsf{Exp}^{\mathsf{rw\text{-}pprf}}_{\mathcal{D}}(\lambda) = 1] - \Pr[\mathsf{Exp}^{\mathsf{iw\text{-}pprf}}_{\mathcal{D}}(\lambda) = 1]| \le \epsilon(\lambda)$$

where the experiments $\mathsf{Exp}^{\mathsf{rw\text{-}pprf}}_{\mathcal{D}}(\lambda)$ and $\mathsf{Exp}^{\mathsf{iw\text{-}pprf}}_{\mathcal{D}}(\lambda)$ are defined below.

$\mathsf{Exp}^{\mathsf{rw\text{-}pprf}}_{\mathcal{D}}(\lambda)$:	$\mathsf{Exp}^{\mathsf{iw\text{-}pprf}}_{\mathcal{D}}(\lambda)$:
– $((K_{j,e})_{j\le Q,e\le\tau} \leftarrow\$ (\{0,1\}^\lambda)^{Q\cdot\tau}$	– $((K_{j,e})_{j\le Q,e\le\tau} \leftarrow\$ (\{0,1\}^\lambda)^{Q\cdot\tau}$
– $\mathsf{salt} := (\mathsf{salt}_1, \ldots, \mathsf{salt}_Q) \leftarrow\$ \{0,1\}^s$	– $\mathsf{salt} := (\mathsf{salt}_1, \ldots, \mathsf{salt}_Q) \leftarrow\$ \{0,1\}^s$
– $\mathbf{i} := ((i_{1,e})_{e\le\tau}, \ldots, (i_{Q,e})_{e\le\tau}) \leftarrow\$ [2^D]^{Q\cdot\tau}$	– $\mathbf{i} := ((i_{1,e})_{e\le\tau}, \ldots, (i_{1,e})_{e\le\tau}) \leftarrow\$ [2^D]^{Q\cdot\tau}$
– $\forall j \le Q, e \le \tau : K^{i_{j,e}}_{j,e} \leftarrow F.\mathsf{Punc}(K_{j,e}, i_{j,e})$	– $\forall j \le Q, e \le \tau : K^{i_{j,e}}_{j,e} \leftarrow F.\mathsf{Punc}(K_{j,e}, i_{j,e})$
– $(y_{j,e})_{j\le Q,e\le\tau} \leftarrow (F_{K_{j,e}}(i_{j,e}, \mathsf{salt}_j))_{j\le Q,e\le\tau}$	– $(y_{j,e})_{j\le Q,e\le\tau} \leftarrow\$ (\{0,1\}^\lambda)^{Q\cdot\tau}$
Output $b \leftarrow \mathcal{D}(\mathsf{salt}, \mathbf{i}, (K^{i_{j,e}}_{j,e}, y_{j,e})_{j\le Q,e\le\tau})$	**Output** $b \leftarrow \mathcal{D}(\mathsf{salt}, \mathbf{i}, (K^{i_{j,e}}_{j,e}, y_{j,e})_{j\le Q,e\le\tau})$

We also recall a stronger property, satisfied by the AES-based construction of [24], in which indistinguishability is preserved even the ideal world experiment does not only sample $(y_1, \cdots, y_Q)$ uniformly at random, but also samples "fake" punctured keys $K^{x_k}_j$ uniformly at random over an appropriate domain:

Definition 11 ((Q, τ)-instance strongly (t, ϵ)-secure PPRF). *A function family $F = \{F_K\}$ with input domain $[2^D]$, salt domain $\{0,1\}^s$, output domain $\{0,1\}^\lambda$, and punctured key domain $(\{0,1\}^\lambda)^D$ is an (Q, τ)-instance (t, ϵ)-secure PPRF if it is a PPRF which additionally takes as input a salt* salt*, and for every non-uniform PPT distinguisher $\mathcal{D}$ running in time at most t, it holds that for all sufficiently large λ,*

$$\mathsf{Adv}^{\mathsf{PPRF}}(\mathcal{D}) = |\Pr[\mathsf{Exp}^{\mathsf{rw\text{-}pprf}}_{\mathcal{D}}(\lambda) = 1] - \Pr[\mathsf{Exp}^{\mathsf{iw\text{-}spprf}}_{\mathcal{D}}(\lambda) = 1]| \le \epsilon(\lambda),$$

where the experiment $\mathsf{Exp}^{\mathsf{iw\text{-}spprf}}_{\mathcal{D}}(\lambda)$ is defined as $\mathsf{Exp}^{\mathsf{iw\text{-}pprf}}_{\mathcal{D}}(\lambda)$, except that the line $\forall j \le Q, e \le \tau : K^{i_{j,e}}_{j,e} \leftarrow F.\mathsf{Punc}(K_{j,e}, i_{j,e})$ is replaced by $\forall j \le Q, e \le \tau : K^{i_{j,e}}_{j,e} \leftarrow\$ (\{0,1\}^\lambda)^D$.

B.4 Pseudorandom correlation generators

We recall the notion of pseudorandom correlation generator (PCG) from [20]. At a high level, a PCG for some target ideal correlation takes as input a pair of short, correlated seeds and outputs long correlated pseudorandom strings, where the expansion procedure is deterministic and can be applied locally. The definitions below are taken almost verbatim from [21].

Definition 12 (Correlation generator). *A PPT algorithm* C *is called a* correlation generator*, if* C *on input 1^λ outputs a pair of elements in $\{0,1\}^n \times \{0,1\}^n$ for $n \in \mathsf{poly}(\lambda)$.*

The security definition of PCGs requires the target correlation to satisfy a technical requirement, which roughly says that it is possible to efficiently sample from the conditional distribution of $\mathcal{R}_0$ given $\mathcal{R}_1 = r_1$ and vice versa. It is easy to see that this is true for the correlations considered in this paper.

Definition 13 (Reverse-sampleable correlation generator). *Let* C *be a correlation generator. We say* C *is* reverse sampleable *if there exists a PPT algorithm* RSample *such that for $\sigma \in \{0,1\}$ the correlation obtained via:*

$$\{(\mathcal{R}'_0, \mathcal{R}'_1)\ |(\mathcal{R}_0, \mathcal{R}_1) \leftarrow\$ \mathsf{C}(1^\lambda), \mathcal{R}'_\sigma := \mathcal{R}_\sigma, \mathcal{R}'_{1-\sigma} \leftarrow\$ \mathsf{RSample}(\sigma, \mathcal{R}_\sigma)\}$$

is computationally indistinguishable from $\mathsf{C}(1^\lambda)$.

Definition 14 (Pseudorandom Correlation Generator (PCG)). *Let* C *be a reverse-sampleable correlation generator. A* pseudorandom correlation generator (PCG) for C *is a pair of algorithms* (PCG.Setup, PCG.Expand) *with the following syntax:*

– $\mathsf{PCG.Setup}(1^\lambda)$ *is a PPT algorithm that given a security parameter* λ*, outputs a pair of seeds* (c_0, c_1)*;*
– $\mathsf{PCG.Expand}(\sigma, c_\sigma)$ *is a polynomial-time algorithm that given a party index* $\sigma \in \{0,1\}$ *and a seed* c_σ*, outputs a bit string* $\mathcal{R}_\sigma \in \{0,1\}^n$*.*

The algorithms $(\mathsf{PCG.Setup}, \mathsf{PCG.Expand})$ *should satisfy the following:*

– **Correctness.** *The correlation obtained via:*

$$\{(\mathcal{R}_0, \mathcal{R}_1) \mid (c_0, c_1) \leftarrow\!\!\$\; \mathsf{PCG.Setup}(1^\lambda), R_\sigma \leftarrow \mathsf{PCG.Expand}(\sigma, c_\sigma) \text{ for } \sigma \in \{0,1\}\}$$

is computationally indistinguishable from $\mathsf{C}(1^\lambda)$*.*
– **Security.** *For any* $\sigma \in \{0,1\}$*, the following two distributions are computationally indistinguishable:*

$$\begin{aligned} &\{(c_{1-\sigma}, \mathcal{R}_\sigma) \mid (c_0, c_1) \leftarrow\!\!\$\; \mathsf{PCG.Setup}(1^\lambda), \mathcal{R}_\sigma \leftarrow \mathsf{PCG.Expand}(\sigma, c_\sigma)\} \text{ and} \\ &\{(c_{1-\sigma}, \mathcal{R}_\sigma) \mid (c_0, c_1) \leftarrow\!\!\$\; \mathsf{PCG.Setup}(1^\lambda), \mathcal{R}_{1-\sigma} \leftarrow \mathsf{PCG.Expand}(\sigma, c_{1-\sigma}), \\ &\qquad\qquad\qquad\qquad\qquad\qquad R_\sigma \leftarrow\!\!\$\; \mathsf{RSample}(\sigma, \mathcal{R}_{1-\sigma})\} \end{aligned}$$

where $\mathsf{RSample}$ *is the reverse sampling algorithm for correlation* C*.*

Note that $\mathsf{PCG.Setup}$ could simply output a sample from C. To avoid this trivial construction, we also require that the seed size is significantly shorter than the output size.

Programmable PCGs. At a high level, a programmable PCG allows generating multiple PCG keys such that part of the correlation generated remains the same across different instances. Programmable PCGs are necessary to construct n-party correlated randomness from the 2-party correlated randomness generated via the PCG. Informally, this is because when expanding n-party shares (e.g. of Beaver triples) into a sum of 2-party shares, the sum will involve many "cross terms"; using programmable PCGs allows maintaining consistent pseudorandom values across these cross terms. We recall the formal definition below.

Definition 15 (Programmable PCG). *A tuple of algorithms* $\mathsf{PCG} = (\mathsf{PCG.Setup}, \mathsf{PCG.Expand})$ *following the syntax of a standard* PCG*, but where* $\mathsf{PCG.Setup}(1^\lambda)$ *takes additional random inputs* $\rho_0, \rho_1 \in \{0,1\}^*$*, is a* programmable PCG *for a simple bilinear 2-party correlation* C_e^n *(specified by* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$*) if the following holds:*

- ***Correctness.*** *The correlation obtained via:*

$$\left\{((R_0, S_0), (R_1, S_1)) \middle| \begin{array}{l} \rho_0, \rho_1 \xleftarrow{\$} \$, (k_0, k_1) \leftarrow\!\!\$\; \mathsf{PCG.Setup}(1^\lambda, \rho_0, \rho_1), \\ (R_\sigma, S_\sigma) \leftarrow \mathsf{PCG.Expand}(\sigma, k_\sigma) \text{ for } \sigma \in \{0,1\} \end{array}\right\}$$

is computationally indistinguishable from $C_e^n(1^\lambda)$*.*
- ***Programmability*** *There exist public efficiently computable functions* $\phi_0 : \{0,1\}^* \to \mathbb{G}_1^n$*,* $\phi_1 : \{0,1\}^* \to \mathbb{G}_2^n$ *such that*

$$\Pr\left[\begin{array}{c} \rho_0, \rho_1 \leftarrow \$, (k_0, k_1) \leftarrow\!\!\$\; \mathsf{PCG.Setup}(1^\lambda, \rho_0, \rho_1) \\ (R_0, S_0) \leftarrow \mathsf{PCG.Expand}(0, k_0), \\ (R_1, S_1) \leftarrow \mathsf{PCG.Expand}(1, k_1) \end{array} : \begin{array}{c} R_0 = \phi_0(\rho_0) \\ R_1 = \phi_1(\rho_1) \end{array}\right] \geq 1 - \mathsf{negl}(\lambda),$$

where $e : \mathbb{G}_1^n \times \mathbb{G}_2^n \to \mathbb{G}_T^n$ *is the bilinear map obtained by applying* e *componentwise.*
- ***Programmable security*** *The distributions*

$$\left\{(\mathrm{k}_1, (\rho_0, \rho_1)) \mid \rho_0, \rho_1 \leftarrow \$, (\mathrm{k}_0, \mathrm{k}_1) \leftarrow\!\!\$\; \mathsf{PCG.Setup}(1^\lambda, \rho_0, \rho_1)\right\} \text{ and}$$

$$\left\{(\mathrm{k}_1, (\rho_0, \rho_1)) \mid \rho_0, \rho_1, \tilde{\rho_0} \leftarrow \$, (\mathrm{k}_0, \mathrm{k}_1) \leftarrow\!\!\$\; \mathsf{PCG.Setup}(1^\lambda, \tilde{\rho_0}, \rho_1)\right\}$$

as well as

$$\left\{(\mathrm{k}_0, (\rho_0, \rho_1)) \mid \rho_0, \rho_1 \leftarrow \$, (\mathrm{k}_0, \mathrm{k}_1) \leftarrow\!\!\$\; \mathsf{PCG.Setup}(1^\lambda, \rho_0, \rho_1)\right\} \text{ and}$$

$$\left\{(\mathrm{k}_0, (\rho_0, \rho_1)) \mid \rho_0, \rho_1, \tilde{\rho_0} \leftarrow \$, (k_0, k_1) \leftarrow\!\!\$\; \mathsf{PCG.Setup}(1^\lambda, \tilde{\rho_0}, \rho_1)\right\}$$

are computationally indistinguishable.