FAEST v2: Algorithm Specifications

Version 2.0

Carsten Baum[2], Ward Beullens[8], Lennart Braun[9], Cyprien Delpech de Saint Guilhem[3], Michael Klooß[4], Christian Majenz[2], Shibam Mukherjee[5,10], Emmanuela Orsini[7], Sebastian Ramacher[6], Christian Rechberger[5], Lawrence Roy[1], and Peter Scholl[1]

[1] Aarhus University
[2] Technical University of Denmark
[3] imec-COSIC, KU Leuven
[4] Aalto University
[5] Graz University of Technology
[6] AIT Austrian Institute of Technology
[7] Bocconi University
[8] IBM Research Europe
[9] Université Paris Cité, CNRS, IRIF
[10] Know Center

Table of Contents

1 Introduction

This document describes and specifies version 2 of the FAEST digital signature algorithm. It presents the underlying cryptographic components and specifies the building blocks used to construct the FAEST algorithm.

The design of FAEST is intended to provide security against attacks by quantum computers by relying only on information-theoretic and symmetric-key cryptographic primitives. In particular, in addition to the standard SHA3 hash function, the security of FAEST is tightly linked to the security of AES128, AES192 and AES256, based on which the NIST security categories 1, 3, and 5 are defined.

Overview. A key pair $(\mathsf{pk}, \mathsf{sk})$ for the FAEST signature algorithm is defined as $\mathsf{pk} = (x, y)$ and $\mathsf{sk} = k$ such that $E_k(x) = y$, where E is a block cipher-based encryption function, k is a secret key and x is a plaintext for E. The signature is derived from a non-interactive argument of knowledge of sk, similarly to other post-quantum signature algorithms such as Picnic [CDG+17, ZCD+20] or Banquet [BDK+21]. However, the argument system used in FAEST is not constructed in the MPC-in-the-Head (MPCitH) framework [IKOS07], but using a new tool called VOLE-in-the-Head (VOLEitH)[11] [BBD+23b] which enables the use of efficient VOLE-based proof systems.

There is some similarity between the VOLEitH technique and MPCitH constructions such as Picnic, for instance, both can be seen as initially creating an N-out-of-N secret sharing of the witness for the zero-knowledge proof. However, the zero-knowledge proof phase of a VOLEitH construction like FAEST is very different, and not based on MPC (multi-party computation). Instead, VOLEitH is closer to secure two-party computation, since VOLE is a two-party primitive, and also relies on techniques from other two-party protocols such as oblivious transfer extension [IKNP03, Roy22].

To construct the FAEST signature algorithm, the interactive argument system resulting from combining VOLEitH with a variant of the QuickSilver information-theoretic proof system [YSWW21] is made non-interactive by the Fiat–Shamir transform [FS87]. Two security proofs are presented, one in the random oracle model (ROM) and one in the quantum-accessible ROM (QROM). The ROM proof has a modular structure and models FAEST as closely as possible. The QROM proof is less modular by necessity, and gives up on faithfully modeling one detail of FAEST. The QROM proof yields a bound that gives meaningful concrete security guarantees against quantum adversaries. Up to the expected speed-ups due to quantum search, collision-finding and any additional known algorithm, it is similar to the ROM bound. This is achieved by employing a lossy-key argument and round-by-round soundness, avoiding the security loss associated with most extraction techniques for the Fiat Shamir transformation. The ROM proof, using the coarse-grained attack complexity measure of query complexity, provides a bound on the number of queries necessary to break FAEST that is close to the bit security target. Accepting the premise of the ROM, this implies that FAEST provably reaches its main goal of existential unforgeability against chosen message attacks.

Design choices. In addition to the VOLEitH construction, the main design choices made for the FAEST algorithm were:

- Using QuickSilver as the information-theoretic proof system.
- Instantiating E with the standardized primitive of AES [AES01].
- A specialized set of degree-3 constraints for proving AES inside QuickSilver.
- Customized PRG-based commitment schemes for committing to pseudorandom values, which are cheaply instantiated using AES.

[11] VOLE stands for vector oblivious linear evaluation.

This document also specifies alternative variants for security categories 1, 3 and 5, called FAEST-EM, based on the Even-Mansour construction, which improve performance through a less standard use of either AES or its precursor Rijndael.

Hardness assumptions. Other than the encryption algorithm E, the security of FAEST relies on the security of the VOLEitH construction and of the QuickSilver protocol. Since the first is constructed only from symmetric-key primitives (PRGs and hash functions), and the second is information-theoretically secure, the EUF-CMA security of FAEST does not require any number theoretic or other structured hardness assumptions. As mentioned above, our security proofs also rely on the random oracle model; while this is an idealized model, it is widely accepted in the cryptographic community as a reasonable way to model security when other methods are not available.

In FAEST-EM, one-wayness of the function E is based on the Even-Mansour construction, which requires the assumption that $\mathsf{AES}_x(k) \oplus k$ is hard to invert, where x is a *public* AES key and k a *secret* input to the block cipher. Additionally, to optimize the PRG-based commitments, FAEST-EM relies on an assumption related to the collision-resistance of 2λ-bit AES outputs, where λ is the security parameter.

Outline of this document. Section 2 provides a detailed overview of FAEST, the VOLEitH tool and the QuickSilver proof system. In Section 3, we present the main FAEST and FAEST-EM parameter sets, and some preliminaries for this document: the notation, the data types and their conversions, and the elementary cryptographic primitives. Section 4 describes two additional components: the AES and Rijndael algorithms and universal hashing algorithms.

After these introductory sections, the document specifies the FAEST signature algorithm. Section 5 specifies the VOLEitH construction. Section 6 specifies the computation of the zero-knowledge proof for the AES and Rijndael circuits used in FAEST and FAEST-EM. Section 7 specifies the key generation, signing and verification algorithms.

The final part of this document analyses the FAEST signature algorithm. Section 8 provides the performance results of the implementations. Section 9 provides both a formal proof as well as an analysis of the concrete attacks that are relevant to the building blocks of FAEST. Section 10 discusses the advantages and limitations provided by the algorithm.

1.1 Changelog

FAEST version 2 has undergone several changes to improve performance and security, compared with the first version. The most significant of these are:

- An optimized batch, all-but-one vector commitment construction using GGM trees, based on [BBM+24]. This reduces the size of the vector commitment openings, while incorporating a 'grinding' optimization when computing Fiat-Shamir challenges to reduce signature size.
- When committing to the leaves of the GGM trees, we now use PRG-based commitments instead of hash-based commitments in version 1. This reduces signing and verification time, for which the leaf commitments are a bottleneck.
- A new method of proving AES inside QuickSilver, using degree-3 constraints derived through finite field norms. As well as reducing the proof size compared with the previous, degree-2 constraints, this removes the restriction that every S-box input must be non-zero.

Other, minor tweaks to the algorithms are as follows:

– Each use of AES-CTR within a different part of the signing algorithm now uses a distinct IV, derived from the per-signature IV. This improves concrete security by preventing preprocessing attacks.
– The key generation now uses CTR mode instead of ECB mode, for the 192- and 256-bit variants where two AES blocks are required.
– The key generation now requires the first two bits of the secret key are not both one. This allows for a classical and quantum security proof of FAEST when using the new leaf commitments.
– The per-signature IV is now passed through a hash function (modeled as a random oracle). This is used in the security analysis of the leaf commitments.
– A detailed security argument in the Quantum Random Oracle Model is provided.
– ...

In addition to the above tweaks, the security proofs of use a different strategy, which is needed to obtain a reduction when using the new leaf commitments (which do not allow for extraction, unlike the previous random oracle commitments). Furthermore, to improve readability, various improvements have been made to notation and the write-up of this document.

2 Overview of Algorithms

This section gives a high-level overview of the main algorithms used for the FAEST signature scheme.

2.1 VOLE and VOLE-in-the-Head

Let λ be the security parameter. FAEST uses *VOLE correlations* over the finite field $\mathbb{F}_{2^\lambda}$ as a form of linearly homomorphic commitment scheme. A VOLE (vector oblivious linear evaluation) correlation of length ℓ is defined by a set of random bits $u_i \in \mathbb{F}_2$ and random field elements $v_i \in \mathbb{F}_{2^\lambda}$, together with a random global key $\Delta \in \mathbb{F}_{2^\lambda}$ and local keys $q_i \in \mathbb{F}_{2^\lambda}$, such that:[12]

$$q_i = u_i \cdot \Delta + v_i \qquad \text{for } i = 0, \dots, \ell - 1. \tag{1}$$

The values u_i and v_i should be known only to the prover (in FAEST, the signer), while q_i and Δ should be given to the verifier.

This can be seen as committing the prover to the random bits u_i via a linearly homomorphic commitment scheme. The scheme is hiding, because the random v_i mask u_i in the verifier's values q_i, and the scheme is binding, because opening to a different value u_i' requires the prover to come up with a tag $v_i' = q_i - u_i'\Delta$, but then the prover would have successfully guessed $\Delta = (v_i - v_i')/(u_i' - u_i)$, which can only happen with probability $2^{-\lambda}$. The linearity of Equation (1) implies that the commitments are linearly homomorphic, a particularly useful property for building efficient zero-knowledge proofs [WYKW21, YSWW21, BMRS21].

A VOLE correlation can be created with a secure two-party protocol [BCGI18, BCG+19, WYKW21, Roy22]; in FAEST, however, since we want to obtain zero-knowledge proofs and signatures that are publicly verifiable, we instead use the VOLEitH technique [BBD+23b]. Here, the prover first generates its values u_i, v_i and commits to them using a special type of *VOLE commitment*. The commitment is set up such that the verifier can later send to the prover the random key Δ, after which, the prover can send an opening that allows the verifier to learn its q_i values, satisfying Equation (1), and nothing more.

[12] This is technically a *subfield VOLE*, since the u_i's are restricted to be in $\mathbb{F}_2$, a subfield of $\mathbb{F}_{2^\lambda}$.

Note. It is important that Δ is only given to the signer *after* running the main steps of the zero-knowledge proof, since as soon as Δ is known, the binding property of the homomorphic commitments is trivially broken.

Generating VOLE Commitments. Instead of directly generating VOLE correlations over $\mathbb{F}_{2^\lambda}$, VOLEitH first generates several VOLE instances over two smaller finite fields $\mathbb{F}_{2^{k_0}}$ and $\mathbb{F}_{2^{k_1}}$. If we create τ_0 VOLEs in the k_0-bit field and τ_1 in the k_1-bit field, then we want to set the parameters such that

$$\tau_0 k_0 + \tau_1 k_1 = \lambda$$

This ensures that, if the small VOLEs are set up such that the signer is committed to the *same* $\mathbf{u}$ vector in each instance, then they can be concatenated to build a final VOLE correlation in $\mathbb{F}_{2^\lambda}$.

In the following, we use k to denote the bit-length of the small field (which will be either k_0 or k_1) and let $N = 2^k$.

All-but-One Vector Commitments. The main building block of our VOLEitH approach is the technique of committing to a vector of N pseudo-random seeds by deriving them from a tree of length-doubling PRGs. This is also known as the GGM construction, which builds a puncturable PRF from a PRG [GGM84, KPTZ13, BW13, BGI14]. This can be used to build an *all-but-one vector commitment scheme*, where the signer commits to N seeds by sending a single hash value, and can later open $N-1$ of them with only $O(\log N)$ communication. In what follows, we describe a variant that yields a batch of τ all-but-one vector commitments.

The idea of the construction is to build a complete binary tree with L leaves, where, starting at the root node with a random seed r, the two children of any node are defined by evaluating the parent seed with a length-doubling PRG that outputs two new seeds. After expanding the tree, we obtain L leaf values, k_i, each of which is used as randomness in a LeafCommit algorithm, to derive a new value sd_i and commitment com_i. The seed sd_i is the i-th committed seed, which will later be expanded and used as part of a small-field VOLE, while com_i serves as a commitment to the seed. Finally, to create a succinct commit to all of the seeds, the signer applies a hash function (modelled as a random oracle) to all the com_i's.

We view the above as a commitment to τ vectors of length N, where $L = \tau \cdot N$, by the following ordering:

To open all-but-τ of the seeds, the signer defines the set of tree nodes S such that the leaves that are descendants of any node in S correspond exactly to the $L - \tau$ seeds to be opened. The signer then sends over the nodes in S to the verifier, together with the leaf commitment values com_i for the τ *unopened* nodes. The verifier can then reconstruct exactly the $L - \tau$ seeds. Furthermore, by recomputing the LeafCommit values for the leaves it knows, it can check the original commitment by recomputing the hash of all leaf commitments.

We note that the size of the set S varies depending on the positions of the unopened nodes. If $\tau = 1$, S consists of the siblings of all nodes on the path from the root to the unopened leaf (excluding the root node), so has size $\lceil \log L \rceil$. In general, when $\tau > 1$, the size of the opening depends on how much overlap there is among the paths to the unopened leaves; the more overlap there is, the smaller S gets.

Optimization 1: Rejection Sampling for the Opening. In the FAEST interactive argument, the τ unopened indices are sampled as a random challenge by the verifier. To reduce the size of S, we have the prover reject the challenge whenever the opening size exceeds a

certain threshold, T_{open}, and request a new challenge from the verifier. Note that, after applying Fiat-Shamir, this rejection is done entirely by the prover, by incrementing a counter and computing a new hash value for each attempt at obtaining a challenge.

This modification allows to significantly reduce the opening size, at a small extra cost of the prover computing some additional hash values. Furthermore, despite the apparent reduction in the size of the challenge space, this optimization does not affect the security of FAEST. Each challenge is still sampled from the same domain as before, just rejected with some small probability. Even if the prover can predict ahead of time which challenges will be rejected, it still has to recompute the hash value for each challenge attempt, and each such attempt has exactly the same chance of being a "bad" challenge (that is, one that allows it to cheat) as previously.

Optimization 2: PRG-Based Leaf Commitments. In version 1 of FAEST, the leaf commitments were computed using the SHAKE hash function. Because of the large number of leaves, this turned out to be one of the dominant costs in signing and verification time. To reduce this cost, instead of using SHAKE, we use PRG-based commitments, which can be instantiated efficiently with AES. We propose two instantiations, which (simplifying slightly) are as follows:

$$\mathsf{LeafCommit}_u(r) = (\mathsf{sd}, c) = (\underbrace{\mathsf{PRG}_{4\lambda}(r)[0]}_{=\mathsf{sd}}, \mathsf{sd} \cdot u + \mathsf{PRG}_{4\lambda}(r)[1..3])$$

$$\mathsf{LeafCommit}(r) = (\mathsf{sd}, c) = (r, \mathsf{PRG}_{2\lambda}(r))$$

In the first version, the committed seed is the first λ-bit block of a 4-block output of a PRG applied to the randomness r. The commitment, c, which can be seen as a variant of Naor's commitment [Nao91, MRZ15], uses a public, random 3λ-bit string u, then performs an $\mathbb{F}_{2^{3\lambda}}$ field multiplication of sd with u, before XORing with the remaining randomness output by the PRG. The commitment is easily seen to be computationally hiding, due to the pseudorandomness of the PRG. The construction is also statistically binding, because finding two openings r, r' that are valid for the same c would imply

$$\mathsf{sd} \cdot u + \mathsf{PRG}(r) = \mathsf{sd}' \cdot u + \mathsf{PRG}(r')$$

and hence, $u = (\mathsf{PRG}(r') - \mathsf{PRG}(r))/(\mathsf{sd}' - \mathsf{sd})$. Since r, r' are each λ bits long, and entirely determine $\mathsf{sd}, \mathsf{sd}'$, this leaves only $2^{2\lambda}$ possible values of u for which such a binding break exists. Hence, if u is a uniform 3λ-bit string, the PRG is perfectly binding except with probability $2^{-\lambda}$.

In the second version, the commitment to seed r is simply a 2λ bit PRG output applied to r. This results in a more efficient and smaller commitment, but now relies on the hardness of finding two different λ-bit seeds that map to the same 2λ PRG output. While there are no known attacks better than standard collision attacks when instantiating the PRG with AES, this is a slightly non-standard assumption.[13] We use this version in the FAEST-EM variants.

The formal pseudocode for the batch vector commitment and leaf commitment procedures is given in Section 5.2, and their security is analyzed in Section 9.4.

From Batch Vector Commitments to VOLE. An all-but-one vector commitment of length $N = 2^k$ can be used to obtain a *VOLE commitment* in a finite field $\mathbb{F}_{2^k}$. If the signer holds N committed seeds $\mathsf{sd}_0, \ldots, \mathsf{sd}_{N-1}$, it first expands them using a PRG, to obtain N strings $\mathbf{r}_0, \ldots, \mathbf{r}_{N-1} \in \{0,1\}^\ell$. Then, it computes, in $\mathbb{F}_{2^k}$,

[13] In our security proofs, we actually rely on a stronger assumption which allows the IV defining the AES-based PRG to be switched into a "perfectly binding" mode where no collisions exist. For details, we refer to Section 9.

$$\mathbf{u} = \sum_{i=0}^{N-1} \mathbf{r}_i, \quad \mathbf{v} = \sum_{i=0}^{N-1} i \cdot \mathbf{r}_i$$

where i is encoded as an element of $\mathbb{F}_{2^k}$.

To see how this can be used to get a VOLE correlation, consider a verifier who later learns seeds sd_i for all $i \neq j^*$, for some index $j^* \in \{0, \ldots, N-1\}$ (viewed as an $\mathbb{F}_{2^k}$ element). The verifier can compute

$$\begin{aligned}\mathbf{q} &= \sum_{i=0}^{N-1} (j^* - i) \cdot \mathbf{r}_i \\ &= j^* \cdot \sum_{i=0}^{N-1} \mathbf{r}_i - \sum_{i=0}^{N-1} i \cdot \mathbf{r}_i \\ &= j^* \cdot \mathbf{u} - \mathbf{v}\end{aligned}$$

giving the desired VOLE in $\mathbb{F}_{2^k}$. This step is specified in Section 5.4, which uses an optimized divide-and-conquer algorithm for computing $\mathbf{u}, \mathbf{v}$ and $\mathbf{q}$ with fewer XOR operations.

From τ Small VOLEs to One Big VOLE. After creating the batch vector commitments, the signer has τ random pairs $(\mathbf{u}_i, \mathbf{V}_i)$, making up its side of the small-field VOLE correlations. Next, the signer must send correction values so that the VOLEs can be fixed to use the *same* $\mathbf{u}$ value, say, $\mathbf{u}_0$, by sending $\mathbf{c}_i = \mathbf{u}_i - \mathbf{u}_0$, for $i = 1$ to $\tau - 1$. This allows the verifier to later adjust its output so that all VOLE relations hold with respect to $\mathbf{u}_0$.

Denote the corrected VOLEs as $(\mathbf{u}, \mathbf{V}_i)$ for $i = 0, \ldots, \tau - 1$, where we now view $\mathbf{V}_i$ as a matrix in $\{0,1\}^{\ell \times k}$, instead of a vector in $\mathbb{F}_{2^k}^{\ell}$. When the VOLEs are eventually opened, the verifier will learn $(\Delta_i, \mathbf{Q}_i)$ such that

$$\mathbf{Q}_i = \mathbf{V}_i + \left(\delta_0 \cdot \mathbf{u}_i \cdots \delta_{k-1} \cdot \mathbf{u}_i\right)$$

where $\delta_0, \ldots, \delta_{k-1}$ is the bit decomposition of Δ_i.

Once this has been done, the parties can simply concatenate the τ VOLE instances, by forming the $\ell \times \lambda$ matrices

$$\mathbf{V} = \left(\mathbf{V}_0 \cdots \mathbf{V}_{\tau-1}\right), \quad \mathbf{Q} = \left(\mathbf{Q}_0 \cdots \mathbf{Q}_{\tau-1}\right)$$

Viewing the rows $\mathbf{V}[i], \mathbf{Q}[i]$ of the above as elements of $\mathbb{F}_{2^\lambda}$, these form a VOLE over $\mathbb{F}_{2^\lambda}$, where the new Δ value is formed from the bits of each Δ_i. That is, we have the relation:

$$\mathbf{Q}[i] = \mathbf{V}[i] + \mathbf{u}[i] \cdot \Delta$$

VOLE Consistency Check. Finally, we need a way of ensuring that the signer does not cheat when sending the correction values: if any $\mathbf{c}_i$ is wrong, the VOLE relation will be incorrect and we will not obtain the same guarantees from the ZK proof system. This is done using the consistency check from [Roy22], where the verifier challenges the signer to open a random, linear universal hash function applied to $\mathbf{u}$ and $\mathbf{V}$. The linear hash function is represented by a compressing matrix $\mathbf{H}$, and the prover sends

$$\widetilde{\mathbf{u}} = \mathbf{H}\mathbf{u}, \quad \widetilde{\mathbf{V}} = \mathbf{H}\mathbf{V}.$$

The verifier then computes $\widetilde{\mathbf{Q}} = \mathbf{H}\mathbf{Q}$ and checks that the VOLE relation still holds between $\widetilde{\mathbf{u}}, \widetilde{\mathbf{V}}$ and $\widetilde{\mathbf{Q}}$. The details of the universal hash family we use are given in Section 4.3.

Padding. The above description assumes $\mathbf{u}$ is of length ℓ bits, the same as the extended witness for the OWF relation. In FAEST, we actually run the VOLE commitment on length $\hat{\ell} = \ell + 3\lambda + B$, because the VOLE consistency check reveals a $(\lambda + B)$-bit linear function of $\mathbf{u}$ to the verifier, which needs to hide the underlying witness. After the check, we discard the additional bits and use the length $\ell + 2\lambda$ VOLE for the ZK proof stage. In the ZK proof, the first ℓ bits of $\mathbf{u}$ are used to commit to the extended witness, while the final 2λ bits will be used as a mask for a second consistency check as part of QuickSilver.

2.2 ZK Proof Phase Using QuickSilver

After setting up the VOLE correlation, the next step in FAEST is for the signer to run a variant of the QuickSilver proof system, to prove knowledge of the secret one-way function key used to generate the public key. QuickSilver, by Yang, Sarkar, Weng and Wang [YSWW21], is an interactive zero-knowledge proof system that uses VOLE, building upon the line-point zero-knowledge paradigm of Dittmer et al. [DIO21]. As QuickSilver is interactive, it is described here as a protocol executed between two parties: a prover and a verifier. When integrated into the non-interactive FAEST signature scheme, these roles will be played by the signer and the verifier, respectively.

QuickSilver assumes a VOLE setup, where the prover holds random $u_i \in \mathbb{F}_2, v_i \in \mathbb{F}_{2^\lambda}$ and verifier holds a random key $\Delta \in \mathbb{F}_{2^\lambda}$, and $q_i = u_i \cdot \Delta + v_i$. Given this, the QuickSilver proof that we compute in the FAEST algorithm proceeds in three main stages.

Witness extension. The prover takes the secret key $\mathsf{sk} = k$, which is the witness for the QuickSilver relation, as well as the public input $\mathsf{pk} = (x, y)$ and executes the AES operations required to compute $E_k(x)$. During this execution, certain bits related to the S-box inputs and outputs are recorded into an *extended witness*, called $\boldsymbol{w} \in \mathbb{F}_2^\ell$. (The exact bits that are recorded are specified in Section 6.5.)

The prover then commits to the extended witness by sending $\boldsymbol{d} = \boldsymbol{u} + \boldsymbol{w}$ to the verifier, where $\boldsymbol{u} \in \mathbb{F}_2^\ell$ is the vector of the prover's u_i values. This allows the verifier to update its VOLE outputs and learn $q_i' := q_i + d_i \cdot \Delta = v_i + u_i \cdot \Delta$.

Evaluating Constraints. Next, the parties want to evaluate various, low-degree constraints on the committed witness. Let the constraints be determined by functions $f_1, \ldots, f_C$, where each function can be represented as a degree-d circuit, and the goal of the proof is to show that for all i,

$$f_i(w_0, \ldots, w_{\ell-1}) = 0$$

To evaluate a constraint function, the parties will evaluate its circuit description in a gate-by-gate manner. Recall that linear operations can be applied to VOLE commitments without any interaction, thanks to linearity of the VOLE relation; therefore, addition gates are straightforward. To handle multiplication gates, the prover views its VOLE outputs (u_i, v_i) as the coefficients of a degree-1 polynomial, $\rho_i(X) = u_i X + v_i$. Then, multiplying two VOLE commitments is done by just multiplying the polynomials, obtain a *degree-2* output $\rho_i(X)\rho_j(X)$. Meanwhile, the verifier's q_i value equals $\rho_i(\Delta)$, so the verifier can multiply two VOLE values q_i and q_j by simply computing $q_i \cdot q_j = \rho_i(\Delta) \cdot \rho_j(\Delta)$.

This process can continue up to higher degrees, by allowing the degree of the prover's polynomials to grow. One thing we have to take of is that when adding two polynomials of different degrees, they must first be aligned to have the same degree, such that the "message" is always stored in the highest coefficient.

In FAEST, we use a maximum degree of $d = 3$. We also use a slight extension of the above, where we don't just do addition and multiplication of witness values over $\mathbb{F}_2$,

but will at times lift the committed witness and obtain commitments to $\mathbb{F}_{2^8}$ elements, embedded in $\mathbb{F}_{2^\lambda}$.

Proving Constraints (Challenge/Response). After evaluating the C constraint functions, the prover wants to show that each constraint evaluates to zero. The verifier first sends a random challenge, which can be thought of as a random vector $\boldsymbol{r} \in \mathbb{F}_{2^\lambda}^C$. This is used to define a single, combined constraint:

$$\hat{f} = \sum_i \boldsymbol{r}_i \cdot p_i(X) = 0$$

Both parties can apply this linear combination to their respective values (polynomials for the prover, or evaluations for the verifier) to obtain a single constraint polynomial to be checked. Note that since we're working over a field, it holds that if any of the original constraints f_i were incorrect, then $\hat{f}$ is also incorrect, except with probability at most $2^{-\lambda}$.

Now, to prove correctness of the degree-d constraint $\hat{f}$, the prover could just send over the coefficients $(a_0, \ldots, a_d)$ of its polynomial, at which point the verifier can check that $a_d = 0$ and $\sum a_i \Delta^i$ matches with the evaluation point it computed. If the constraint was incorrect, then a cheating prover can only pass this check with probability $d/2^\lambda$, since there are at most d possible values of Δ for which the check can pass, each corresponding to a root of the polynomial with coefficients $(a_0 + a'_0, \ldots, a_d + a'_d)$, where the a'_is are the coefficients sent by a cheating prover.

Combining the soundness of this check with the random linear combination, we obtain an overall soundness error of $2^{-\lambda} + d \cdot 2^{-\lambda}$.

Using Universal Hashes as a Random Linear Combination. Similarly to the VOLE check, instead of having the verifier choose C random coefficients for its challenge, we use a universal hash function. This hash must be linear over $\mathbb{F}_{2^\lambda}$, and allows for a smaller challenge size while preserving the $\approx 2^{-\lambda}$ failure probability of the naive method.

Adding Zero-Knowledge. QuickSilver protocol adds one more component to the constraint check in order to guarantee the zero-knowledge property of the protocol. Indeed, revealing $(a_0, \ldots, a_d)$ to the verifier leaks information about the circuit values used to compute them. To prevent this, the prover and the verifier jointly request $(d-1)\lambda$ further random VOLE elements, which are used to mask the check above. Each set of λ VOLEs can be combined into a single VOLE (u_i, v_i, q_i) where u_i is *uniform* over the field. This can then be used to mask one of the a_i coefficients. Because there are effectively only $d-1$ unknown degrees of freedom for the verifier (who knows its evaluation point as well as $a_d = 0$), it suffices to have $d-1$ random mask commitments to ensure zero-knowledge.

2.3 Putting Things Together

In FAEST, VOLE commitments and QuickSilver are combined to form a 3-challenge (or 7-message) interactive argument, for proving knowledge of the one-way function preimage. This flow is illustrated in Figure 2.1

Transforming the Argument to a Non-Interactive Signature Scheme. To obtain the final signature scheme, we apply the Fiat-Shamir transformation to make the argument non-interactive, using a hash function (modelled as a random oracle). This begins by computing an initial hash μ of the message being signed and other public information. Then, the prover runs the protocol, but instead of receiving challenges from the verifier, computes:

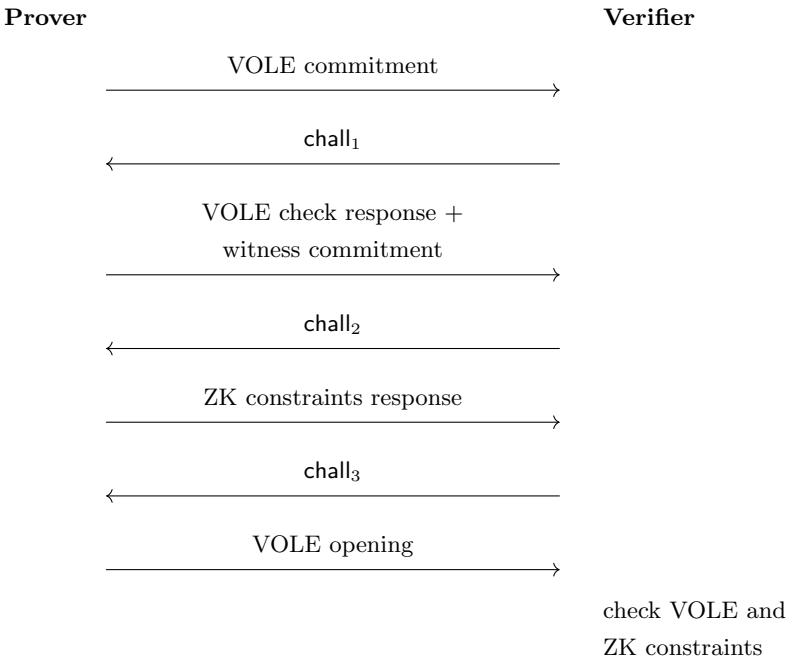

Fig. 2.1: Interactive argument used in FAEST

$$\mathsf{chall}_i = \mathsf{H}(\mathsf{chall}_{i-1} \| \mathsf{msg}_i)$$

where msg_i is the previous message sent by the prover, and we define $\mathsf{chall}_0 := \mu$.

Finally, the prover could define the signature on m to be the tuple $(\mathsf{msg}_1, \mathsf{msg}_2, \mathsf{msg}_3, \mathsf{msg}_4)$, allowing the verifier to reconstruct all challenges and check the transcript. Instead, we actually define the signature in a slightly more compact way, as $(\mathsf{msg}'_1, \mathsf{msg}'_2, \mathsf{msg}_3, \mathsf{msg}_4, \mathsf{chall}_3)$, where msg'_1 and msg'_2 are certain substrings of msg_1 and msg_2. This still works, because the missing information in msg_1 and msg_2 can be reconstructed given chall_3 and the VOLE opening, msg_4. Thus, the verifier can still reconstruct a complete transcript, and check that chall_3 matches with the challenge from that transcript to ensure soundness.

3 Main Parameters and Building Blocks

3.1 FAEST and FAEST-EM Parameter Sets

We let λ denote the computational security parameter. In this section, we describe parameter sets for FAEST-λ and FAEST-EM-λ for $\lambda \in \{128, 192, 256\}$ (categories $\{1, 3, 5\}$).

To reduce the number of parameters explicitly passed to the algorithms that require them, we define the following main parameter structure given to all algorithms:

$$\texttt{param} := (\lambda, \tau, w_{\mathsf{grind}}, T_{\mathsf{open}}, n_{\mathsf{leafcom}}, B, \ell)$$

These parameters, described in Table 3.1, are sufficient to determine the signature size and all other parameters in any given FAEST or FAEST-EM instance.

Parameter	Description
λ	Security parameter in $\{128, 192, 256\}$
τ	Number of small-VOLE instances
w_{grind}	Grinding parameter: w upper bits of Δ must be zero
T_{open}	Threshold for the maximum opening size of the GGM tree
n_{leafcom}	Number of λ-bit blocks in each leaf commitment
B	Extra length padding for the VOLE check
$E_k(x)$	One-way function used to generate the public key
ℓ	Witness length for ZK proof (in bits)

Table 3.1: Main parameters for FAEST and FAEST-EM

Scheme	OWF $E_k(x)$	ℓ	τ	w_{grind}	T_{open}	B	n_{leafcom}	sizes (bytes)	
								pk	sig.
FAEST-128s	$\mathsf{AES128}_k(x)$	1280	11	7	102	16	3	32	4 506
FAEST-128f		1280	16	8	110	16	3	32	5 924
FAEST-192s	$\mathsf{AES192}_k(x) \Vert \mathsf{AES192}_k(x \oplus 1)$	2496	16	12	162	16	3	48	11 260
FAEST-192f		2496	24	8	163	16	3	48	14 948
FAEST-256s	$\mathsf{AES256}_k(x) \Vert \mathsf{AES256}_k(x \oplus 1)$	3104	22	6	245	16	3	48	20 696
FAEST-256f		3104	32	8	246	16	3	48	26 548
FAEST-EM-128s	$\mathsf{AES128}_x(k) \oplus k$	960	11	7	103	16	2	32	3 906
FAEST-EM-128f		960	16	8	112	16	2	32	5 060
FAEST-EM-192s	$\mathsf{Rijndael192}_x(k) \oplus k$	1728	16	8	162	16	2	48	9 340
FAEST-EM-192f		1728	24	8	176	16	2	48	12 380
FAEST-EM-256s	$\mathsf{Rijndael256}_x(k) \oplus k$	2688	22	6	218	16	2	64	17 984
FAEST-EM-256f		2688	32	8	234	16	2	64	23 476

Table 3.2: One-way functions and parameters for the FAEST-λ and FAEST-EM-λ variants. ℓ is the number of VOLE correlations required for the ZK proof; τ is the number of repetitions; k (or $k-1$) is the bit length of the small VOLEs; B is a padding parameter affecting security of the VOLE check; n_{leafcom} is the number of λ-bit blocks in each leaf commitment

Signature size. Given the main parameters, the signature size (in bits) can be calculated as follows.

$$\text{size} = \underbrace{\tau \cdot (\ell + 3\lambda + B)}_{\substack{\text{witness/mask commitment+} \\ \text{responses to VOLE/ZK checks}}} + \underbrace{T_{\mathsf{open}} \cdot \lambda + n_{\mathsf{leafcom}} \lambda \tau}_{\text{GGM tree opening}} + \underbrace{\lambda}_{\Delta \text{ challenge}} + \underbrace{128}_{\text{salt}} + \underbrace{32}_{\text{counter}}$$

3.1.1 Additional Parameters

VOLE and VOLE-in-the-Head. Table 5.1 details some additional parameters used in the vector commitment and VOLE algorithms in Section 5. We define:

$$\mathtt{param}_{\mathsf{VOLE}} := (n_{\mathsf{leafcom}}, B, \tau_1, \tau_0, k, L, \{N_i\}_{i=0}^{\tau-1})$$

One-Way Functions and their Parameters. FAEST uses one-way functions based on the AES standard and its predecessor, Rijndael. For the FAEST parameter sets with security levels $\lambda \in \{128, 192, 256\}$, these are defined using the AES block cipher as follows:

$$\begin{aligned} E_{128} : \{0,1\}^{128} \times \{0,1\}^{128} &\to \{0,1\}^{128} \\ (k, x) &\mapsto \mathsf{AES128}_k(x) \\ E_{192} : \{0,1\}^{192} \times \{0,1\}^{128} &\to \{0,1\}^{256} \\ (k, x) &\mapsto \mathsf{AES192}_k(x) \| \mathsf{AES192}_k(x \oplus 1) \\ E_{256} : \{0,1\}^{256} \times \{0,1\}^{128} &\to \{0,1\}^{256} \\ (k, x) &\mapsto \mathsf{AES256}_k(x) \| \mathsf{AES256}_k(x \oplus 1) \end{aligned}$$

For $\lambda = 192, 256$, the $x \oplus 1$ operation XORs with the 128-bit, little-endian string consisting of a single 1 followed by 127 zeroes. Note that for these settings, one 128-bit AES block does not suffice for one-wayness, since there are likely to be many valid preimages and an adversary can expect to find one with only 2^{128} block cipher operations.

For FAEST-EM, the one-way function is instead based on the Rijndael block cipher with a λ-bit key and λ-bit block size (note that for $\lambda = 128$, Rijndael is the same as AES128). This is defined as:

$$\begin{aligned} E_\lambda^{\mathsf{EM}} : \{0,1\}^\lambda \times \{0,1\}^\lambda &\to \{0,1\}^\lambda \\ (k, x) &\mapsto \mathsf{Rijndael}\text{-}\lambda_x(k) \oplus k \end{aligned}$$

Table 3.3 describes some parameters of these one-way functions that are used in the zero-knowledge proof phase. It includes, for example, the number of rounds R in the block cipher, the block size $N_{\mathsf{st\text{-}bits}}$, the number of S-boxes in a single encryption block, S_{enc}, or in the key schedule, S_{ke}, as well as an integer $\beta \in \{1, 2\}$ used to indicate the number of AES blocks according to the security level.

We define the following structure, which is used in the algorithms specific to the one-way function:

$$\mathtt{param}_{\mathsf{OWF}} := (N_{\mathsf{k}}, N_{\mathsf{st}}, N_{\mathsf{st\text{-}bytes}}, N_{\mathsf{st\text{-}bits}}, \beta, R, S_{\mathsf{ke}}, S_{\mathsf{enc}}, \ell_{\mathsf{ke}}, \ell_{\mathsf{enc}}, \ell, C).$$

3.2 Data Types and Conversions

Finite field Arithmetic. FAEST uses finite field arithmetic over $\mathbb{F}_{2^8}$ (as part of AES), $\mathbb{F}_{2^{64}}$, as well as $\mathbb{F}_{2^\lambda}$ and $\mathbb{F}_{2^{3\lambda}}$, for each λ-bit security variant. These fields are defined as polynomials over $\mathbb{F}_2$, taken modulo an irreducible polynomial P. The irreducible polynomials are

Param. λ	Formula	Description	FAEST 128	FAEST 192	FAEST 256	FAEST-EM 128	FAEST-EM 192	FAEST-EM 256
N_{k}	$\frac{\lambda}{32}$	no. 32-bit words in key	4	6	8	4	6	8
N_{st}	4 or $\frac{\lambda}{32}$	block size in 32-bit words	4	4	4	4	6	8
$N_{\mathsf{st\text{-}bytes}}$	$4N_{\mathsf{st}}$	block size in bytes	16	16	16	16	24	32
$N_{\mathsf{st\text{-}bits}}$	$32N_{\mathsf{st}}$	block size in bits	128	128	128	128	192	256
β	$\lceil \frac{\lambda}{N_{\mathsf{st\text{-}bits}}} \rceil$	no. message blocks	1	2	2	1	1	1
R	$\max(N_{\mathsf{k}}, N_{\mathsf{st}}) + 6$	no. encryption rounds	10	12	14	10	12	14
S_{ke}	$-\frac{\lambda}{8} + 56 + 28 \lfloor \frac{\lambda}{256} \rfloor$ or 0	no. S-Boxes in key sch.	40	32	52	0	0	0
S_{enc}	$4 \cdot N_{\mathsf{st}} \cdot R$	no. S-Boxes in enc.	160	192	224	160	288	448
ℓ_{ke}	$\lambda + 8S_{\mathsf{ke}}$	no. witness bits for key sch.	448	448	672	128	192	256
ℓ_{enc}	$6S_{\mathsf{enc}} - N_{\mathsf{st\text{-}bits}}$	no. witness bits for enc.	832	1 024	1 216	832	1 536	2 432
ℓ	$\ell_{\mathsf{ke}} + \beta\ell_{\mathsf{enc}}$	witness length in bits	1 280	2 496	3 104	960	1 728	2 688
C	$2S_{\mathsf{ke}} + \beta \cdot \frac{3}{2}S_{\mathsf{enc}} + 1$	no. constraints	321	641	777	241	433	673

Table 3.3: Unified OWF parameters for all instances

taken from a table of low Hamming weight irreducible polynomials [Ser98], which agrees with the AES specification [AES01] for P_8.

$$
\begin{aligned}
P_8(\alpha) &= \alpha^8 + \alpha^4 + \alpha^3 + \alpha^1 + 1 \\
P_{64}(\alpha) &= \alpha^{64} + \alpha^4 + \alpha^3 + \alpha^1 + 1 \\
P_{128}(\alpha) &= \alpha^{128} + \alpha^7 + \alpha^2 + \alpha^1 + 1 \\
P_{192}(\alpha) &= \alpha^{192} + \alpha^7 + \alpha^2 + \alpha^1 + 1 \\
P_{256}(\alpha) &= \alpha^{256} + \alpha^{10} + \alpha^5 + \alpha^2 + 1 \\
P_{384}(\alpha) &= \alpha^{384} + \alpha^{12} + \alpha^3 + \alpha^2 + 1 \\
P_{576}(\alpha) &= \alpha^{576} + \alpha^{13} + \alpha^4 + \alpha^3 + 1 \\
P_{768}(\alpha) &= \alpha^{768} + \alpha^{19} + \alpha^{17} + \alpha^4 + 1
\end{aligned}
$$

An implementation does not necessarily need to support general arithmetic in all of these fields, however. In particular, in the fields $\mathbb{F}_{2^{3\lambda}}$, we only ever do multiplication between a λ-bit element (where only the lowest λ coefficients are non-zero) and a 3λ-bit element, which can be performed significantly faster than regular multiplication.

Conversions To/From Bits, Field Elements and Integers. The following algorithms are used to convert and manipulate finite field elements.

- ToField$(\mathbf{x}; k)$: maps $\mathbf{x} \in \{0,1\}^{nk}$, for $n \geq 1$ into a (vector of) field element(s) $\boldsymbol{x} \in \mathbb{F}_{2^k}^n$ using little-endian ordering.
- ToBits$(\boldsymbol{x}; \boldsymbol{k}, \boldsymbol{n})$: maps $\boldsymbol{x} \in \mathbb{F}_{2^k}^n$, for $n \geq 1$, into a bit string $\mathbf{x} \in \{0,1\}^{nk}$.
- ByteCombine$(\mathbf{x}; \lambda)$: takes a vector of exactly 8 bits, $\mathbf{x} \in \{0,1\}^8$, and combines them into a single element in $\mathbb{F}_{2^\lambda}$ using powers of an $\mathbb{F}_{2^8}$ generator within $\mathbb{F}_{2^\lambda}$ and little-endian ordering. (The precise generators we use are specified in Appendix A.1.)
  This ordering, where the most significant bit of the byte indicates the highest power of the α_8 generator element matches the interpretation of bytes as polynomials in the AES standard [AES01].

In an implementation, depending on the chosen representation of finite field elements, ToField and ToBits may not require any operations (e.g. if field elements are stored as

ToField($\mathbf{x}, k$)

1 : **let** $\alpha_k \in \mathbb{F}_{2^k}$ // The α element of $\mathbb{F}_{2^k}$
2 : **if** $\mathbf{x} \in \{0,1\}^k$
3 : **return** $x := \sum_{i=0}^{k-1} \mathbf{x}[i] \cdot \alpha_k^i$
4 : **else if** $\mathbf{x} \in \{0,1\}^{nk}$
5 : **new** $\boldsymbol{x} \in \mathbb{F}_{2^k}^n$
6 : **for** $i \in [0..n)$ **do**
7 : $\boldsymbol{x}[i] := \sum_{j=0}^{k-1} \mathbf{x}[ni+j] \cdot \alpha_k^j$
8 : **return** $\boldsymbol{x}$
9 : **else**
10 : **return** $\perp$

ToBits($\boldsymbol{x}; k, n$)

1 : **new** $\mathbf{x} := \{\} \in \{0,1\}^{nk}$
2 : **for** $i \in [0..n)$ **do**
3 : Parse $x_0 + x_1\alpha_k + \cdots + x_{k-1}\alpha_k^{k-1} = \boldsymbol{x}[i]$ for $x_j \in \mathbb{F}_2$
4 : $\mathbf{x}_i := x_0 \| \cdots \| x_{k-1} \in \{0,1\}^k$
5 : $\mathbf{x} := \mathbf{x} \| \mathbf{x}_i$
6 : **return** $\mathbf{x} \in \{0,1\}^{nk}$

ByteCombine($\mathbf{x}; k$)

1 : **let** $\alpha_8 \in \mathbb{F}_{2^k}$ // Generator of $\mathbb{F}_{2^8}$ within $\mathbb{F}_{2^k}$ (Appendix A.1)
2 : // Viewing $\mathbf{x}[i] \in \mathbb{F}_2 \subset \mathbb{F}_{2^\lambda}$
3 : **return** $x = \sum_{i=0}^{7} \mathbf{x}[i] \cdot \alpha_8^i$

Fig. 3.4: Data conversion functions

BitDec(i, d)

1 : **for** $j \in [0 \ldots d-1]$
2 : $b_j := i \bmod 2$
3 : $i \leftarrow (i - b_j)/2$
4 : **return** $(b_0, \ldots, b_{d-1})$

NumRec($d, (b_0, \ldots, b_{d-1})$)

1 : **return** $\sum_{j=0}^{d-1} b_j \cdot 2^j$

Fig. 3.5: Bit decomposition and reconstruction algorithms, little endian representation

packed byte arrays of polynomial coefficients). We use these functions in this specification document to make explicit when we refer to field elements or to bit strings, and to emphasize either to which finite field the elements belong, or the length of the bit strings.

In Figure 3.5, we describe the bit decomposition algorithm BitDec which decomposes and integer i into d bits. The output is in little endian notation, i.e. the bit b_0 is the parity bit of i. Additionally, Figure 3.5 contains the integer reconstruction algorithm, which maps a bit-string of length d uniquely into an integer in the interval $[0..2^d)$. Clearly, $\mathsf{NumRec}(d, \mathsf{BitDec}(i, d)) = i$ for all $i \in [0, 2^d)$.

3.3 Cryptographic Primitives

In addition to the one-way functions defined in Section 3.1.1, FAEST uses the following symmetric primitives:

– $\mathsf{PRG} : \{0,1\}^\lambda \times \{0,1\}^{128} \times \{0,1\}^{32} \to \{0,1\}^*$, a pseudo-random generator taking as input a λ-bit seed, 128-bit initialization vector and 32-bit tweak
– $\mathsf{H}_0 : \{0,1\}^{128} \to \{0,1\}^{3\lambda}$: hash function for commitment keys
– $\mathsf{H}_1 : \{0,1\}^* \to \{0,1\}^{2\lambda}$, collision-resistant hash for vector commitments
– $\mathsf{H}_2^j : \{0,1\}^* \to \{0,1\}^*$, for $j \in \{1,2,3\}$, hash function for the j-th Fiat-Shamir challenge; modeled as a random oracle
– $\mathsf{H}_3 : \{0,1\}^* \to \{0,1\}^{\lambda+128}$, hash function for secret randomness derivation
– $\mathsf{H}_4 : \{0,1\}^* \to \{0,1\}^{128}$, hash function for IV derivation

PRG. The PRG can incorporate both a 128-bit initialization vector iv, and a 32-bit tweak. These separate different uses of the PRG, increasing concrete security by preventing multi-target attacks (see Section 9.1.2). The iv is freshly sampled for each signature, while the tweak is modified for each distinct use within the signing and verification procedures.

We implement PRG using AES in CTR mode, as shown in Figure 3.6. The algorithm takes the output bit-length m as an additional input, and starts by using the secret seed sd to create an expanded AES key schedule. The tweak is then incorporated by adding it to the upper 32 bits (in little-endian ordering) of the IV, using AddToUpperWord. Next, the algorithm runs AES-CTR to obtain m bits of output, by iteratively encrypting and incrementing the lower 32 bits of the IV modulo 2^{32}. Using the lower bits for the counter and the upper bits for the tweak ensures that there is no conflict between these separation mechanisms, and PRG is suitable for up to 2^{32} blocks of output and 2^{32} distinct tweaks with the same iv input; both of these limits are far above what's needed in a single run of the signing algorithm.

$\mathsf{AddToUpperWord}(\mathsf{iv} = (\mathsf{iv}_0, \ldots, \mathsf{iv}_3), \alpha)$

INPUT: $\mathsf{iv}_i \in \{0,1\}^{32}$, $\alpha \in [0..2^{32})$
OUTPUT: $\mathsf{iv}' \in \{0,1\}^{128}$
1 : // iv_3 is "upper" 32 bits in little-endian order
2 : $\mathsf{iv}'_3 \leftarrow \mathsf{NumRec}(32, \mathsf{iv}_3) + \alpha \bmod 2^{32}$
3 : **return** $(\mathsf{iv}_0, \mathsf{iv}_1, \mathsf{iv}_2, \mathsf{BitDec}(\mathsf{iv}'_3))$

$\mathsf{AddToLowerWord}(\mathsf{iv}, \alpha)$

INPUT: $\mathsf{iv}_i \in \{0,1\}^{32}$, $\alpha \in [0..2^{32})$
OUTPUT: $\mathsf{iv}' \in \{0,1\}^{128}$
1 : // iv_0 is "lower" 32 bits in little-endian order
2 : $\mathsf{iv}'_0 \leftarrow \mathsf{NumRec}(32, \mathsf{iv}_0) + \alpha \bmod 2^{32}$
3 : **return** $(\mathsf{BitDec}(\mathsf{iv}'_0), \mathsf{iv}_1, \mathsf{iv}_2, \mathsf{iv}_3)$

$\mathsf{PRG}(\mathsf{sd}, \mathsf{iv}, \mathsf{twk}; m)$

INPUT: $\mathsf{sd} \in \{0,1\}^\lambda, \mathsf{iv} \in \{0,1\}^{128}, \mathsf{twk} \in \{0,1\}^{32}, m \in \mathbb{N}$
OUTPUT: $s \in \{0,1\}^m$
1 : $h := \lfloor m/128 \rfloor$
2 : $rem := m - 128 \cdot h$
3 : $\overline{\mathbf{k}} \leftarrow \mathsf{AES}\text{-}\lambda.\mathsf{KeyExpansion}(\mathsf{sd})$
4 : $\mathsf{iv}' \leftarrow \mathsf{AddToUpperWord}(\mathsf{iv}, \mathsf{twk})$
5 : **for** $i \in [0..h)$ **do**
6 : $\quad \mathbf{s}_i \leftarrow \mathsf{AES}\text{-}\lambda.\mathsf{Encrypt}(\overline{\mathbf{k}}, \mathsf{AddToLowerWord}(\mathsf{iv}', i))$
7 : **return** $\mathbf{s}_0 \parallel \cdots \parallel \mathbf{s}_{h-2} \parallel \mathbf{s}_{h-1}[0..rem-1]$

Fig. 3.6: Tweakable PRG based on AES-CTR

Hash functions. The hash functions are instantiated using SHAKE128, if $\lambda = 128$, and SHAKE256 otherwise. As with PRG, for H_2^j we write $\mathsf{H}_2^j(x; \ell)$ to specify the output length ℓ in bits. To ensure domain separation, we append a single byte i to the message, as follows:

– $\mathsf{H}_i(m) := \text{SHAKE}(m \| i, \ell)$, if $i \in \{0, 1, 3, 4\}$
– $\mathsf{H}_2^j(m) := \text{SHAKE}(m \| 8 + j, \ell)$

Here, SHAKE is either SHAKE128 or SHAKE256, depending on λ.

In our security proofs, we model $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^j$ as random oracles, while H_3 is modeled as a pseudo-random function (where the last λ input bits are the key).

4 Additional Building Blocks

In this section we present two important components of our scheme, namely AES and the universal hash functions used in the consistency checks described in Section 2.

4.1 AES and Rijndael

The Advanced Encryption Standard (AES) algorithm is a symmetric-key cipher with a 128-bit block size and key length of 128, 192 or 256 bits, running in $R = 10$, 12 or 14 rounds (depending on the key length) [AES01]. These three versions of the AES algorithm will be denoted AES128, AES192 and AES256 respectively. The AES algorithm is a standardised variant of the Rijndael algorithm [DR02] which also accepts plaintext blocks of length 192 and 256 bits.

Each execution of the AES algorithm uses three routines: key expansion (which generates round keys), encryption (called "cipher" in the standard) and decryption (or inverse cipher). The Rijndael algorithm uses the same routines, with different size parameters. In this document, we will ignore the decryption routine as we will use encryption (including key expansion) as a one-way function (OWF). To describe these routines we will use the following terms.

(Cipher) Key. Secret, cryptographic key that is used by the key expansion routine to generate the round keys (also called expanded key); it can be pictured as a rectangular array of bytes, having four rows and N_k columns. Both AES and Rijndael accept $N_\mathsf{k} \in \{4, 6, 8\}$.

Round key. Round keys are values derived from the key using the key expansion routine; they are applied to the state in the encryption routine.

State. Intermediate Cipher result that can be pictured as a rectangular array of bytes, having four rows and N_st columns. In the case of AES $N_\mathsf{st} = 4$ is fixed by the standard. The Rijndael algorithm also accepts states with $N_\mathsf{st} \in \{6, 8\}$.

S-box. Non-linear substitution table used to perform one-to-one byte substitutions.

The Rijndael algorithm sometimes runs more rounds in the encryption routine than AES does for the same N_k, depending on the block size N_st. The following table gives the different values of R; it can be summarised as $R := \max(N_\mathsf{k}, N_\mathsf{st}) + 6$.

R	$N_\mathsf{st} = 4$	$N_\mathsf{st} = 6$	$N_\mathsf{st} = 8$
$N_\mathsf{k} = 4$	10	12	14
$N_\mathsf{k} = 6$	12	12	14
$N_\mathsf{k} = 8$	14	14	14

The State. As mentioned above, the intermediary result of the AES encryption, on which the next round of operations is about to be performed, can be arranged in a $4 \times N_{\mathsf{st}}$ rectangular array of bytes called the state. In this section, we denote this state array by s, and refer to each byte of s as $s_{r,c}$, where $0 \leq r < 4$ and $0 \leq c < N_{\mathsf{st}}$ denote respectively the row and column indices of the byte.

We note that we follow the approach of the standard and view the state as an array of columns when appropriate. That is, to interpret the $4 \times N_{\mathsf{st}}$ array as a $4 \cdot N_{\mathsf{st}}$-byte string, we read the rectangular array by first going down the first column, and then moving on to the second (i.e. in column-major order):

$$s \to s_{0,0}s_{1,0}s_{2,0}s_{3,0}s_{0,1} \ldots s_{1,N_{\mathsf{st}}-1}s_{2,N_{\mathsf{st}}-1}s_{3,N_{\mathsf{st}}-1}.$$

4.1.1 The AES S-box. The only non-linear component of the AES and Rijndael algorithms is the S-box byte-for-byte substitution; the SubBytes transformation is then obtained by applying the S-box substitution independently to each byte of the state. The S-box itself is composed of two transformations:

1. Taking the multiplicative inverse in the finite field $GF(2^8)$, and mapping 0 to itself.
2. Applying the following $GF(2)$-affine transformation:

$$b_i \mapsto b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i,$$

for $0 \leq i < 8$, where b_i is the i-th bit of the byte, and c_i is the i-th bit of a byte c with value `0110 0011`. We denote this transformation on the byte b as $b' = L(b)$ but stress that it is only affine over $GF(2)$.

4.1.2 The AES and Rijndael Algorithms. In addition to SubBytes, there are three other operations defined by the AES standard: ShiftRows, MixColumns and AddRoundKey. These same operations are similarly defined for the Rijndael algorithm [DR02]. These are used as part of two routines, KeyExpansion and Encrypt, which respectively expand the λ-bit key $\mathbf{k}$ into $R+1$ round keys and transform the input array (plaintext block) into the output array (ciphertext block).

ShiftRows$_{N_{\mathsf{st}}}$. In this first transformation, the bytes in the rows of the State are cyclically shifted left by different increments. For the AES algorithm with $N_{\mathsf{st}} = 4$, the first row is not changed, the second row shifts by one, the third row shifts by two, and the last row shifts by three. This effects the following permutation on the state:

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

$\longrightarrow$

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,1}$	$s_{1,2}$	$s_{1,3}$	$s_{1,0}$
$s_{2,2}$	$s_{2,3}$	$s_{2,0}$	$s_{2,1}$
$s_{3,3}$	$s_{3,0}$	$s_{3,1}$	$s_{3,2}$

For the Rijndael algorithm, these shifts are the same when $N_{\mathsf{st}} = 6$, but differ for $N_{\mathsf{st}} = 8$. In this second case, the third row is shifted by three (instead of two) and the fourth row is shifted by four (instead of three).

MixColumns. This second transformation applies a $\mathbb{F}_{2^8}$-linear transformation to each of the columns of the state, identically and independently. Since each column of the state containes exactly four bytes in both AES and Rijndael, this transformation is identical for both algorithms.

While the standard presents this transformation first using polynomial multiplication in $GF(2^8)[x]$ followed by reduction modulo x^4+1, we present it here directly as a matrix multiplication:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} \{02\} & \{03\} & \{01\} & \{01\} \\ \{01\} & \{02\} & \{03\} & \{01\} \\ \{01\} & \{01\} & \{02\} & \{03\} \\ \{03\} & \{01\} & \{01\} & \{02\} \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \qquad \text{for } 0 \le c < 4.$$

Here, the bytes of the state are viewed as $\mathbb{F}_{2^8}$ elements according to the ByteCombine routine and the matrix coefficients are taken from the following values:

Byte	Field element
{01}	1
{02}	α
{03}	$\alpha+1$

where α generates $\mathbb{F}_{2^8}$.

AddRoundKey. This final transformation adds one of the round keys to the state with a simple bit-wise XOR operation. Each round key is made up of N_{st} words of 4 bytes each, derived from the key expansion routine, which are each added onto the state such that

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \oplus \begin{bmatrix} \bar{\mathbf{k}}[\mathtt{round}*4+c]_0 \\ \bar{\mathbf{k}}[\mathtt{round}*4+c]_1 \\ \bar{\mathbf{k}}[\mathtt{round}*4+c]_2 \\ \bar{\mathbf{k}}[\mathtt{round}*4+c]_3 \end{bmatrix} \qquad \text{for } 0 \le c < 4,$$

where the $\bar{\mathbf{k}}[i]$ are the expanded key words, and $0 \le \mathtt{round} \le R$ indicates the current round of the algorithm.

Key Expansion routine. Before starting the R rounds of the encryption routine, the AES and Rijndael algorithms perform the key expansion routine on the cipher key $\mathbf{k}$ to obtain a total of $N_{\mathsf{st}} \cdot (R+1)$ expanded key words. To do so, the key expansion routine makes use of two transformations:

SubWord Takes a 4-byte input word and returns an output word by applying the AES S-box to each of the four bytes.

RotWord Takes a 4-byte input word $[a_0, a_1, a_2, a_3]$ and performs a cyclic permutation to return the 4-byte output word $[a_1, a_2, a_3, a_0]$.

In addition, a round constant with value $[\mathtt{Rcon[i]}\|\{00\}\|\{00\}\|\{00\}]$ is added at certain intervals during the key expansion (after every multiple of N_{k} words); the value $\mathtt{Rcon[i]}$ is computed as $\{02\}^i = \alpha_8^i \in \mathbb{F}_{2^8}$. The different values of $\mathtt{Rcon[i]}$ are for the AES algorithm with $N_{\mathsf{st}} = 4$ are listed in Table 4.1; the values of $\mathtt{Rcon[i]}$ for greater values of i for the Rijndael algorithm can be derived by continuing the sequence.

The KeyExpansion routine (Fig. 4.2) first places the N_{k} words of the key $\mathbf{k}$ into the first N_{k} words of the expanded key $\bar{\mathbf{k}}$, and then computes each next word $\bar{\mathbf{k}}[i]$ as the XOR of the one before, $\bar{\mathbf{k}}[i-1]$, with the one N_{k} words back, $\bar{\mathbf{k}}[i-N_{\mathsf{k}}]$. Every N_{k} words, a transformation to $\bar{\mathbf{k}}[i-1]$ is applied before the XOR: first with RotWord, then with SubWord, and finally with an XOR with the round constant word $[\mathtt{Rcon[i/N_k - 1]}\|\{00\}\|\{00\}\|\{00\}]$. When $\lambda = 256$, for which $N_{\mathsf{k}} = 8$, there is an additional SubWord operation applied to $\bar{\mathbf{k}}[i-1]$ when the word index $i \bmod 8 = 4$, but without the RotWord transformation or the XOR with the round constant word.

Round index i	0	1	2	3	4	5	6	7	8	9
Rcon[i]	{01}	{02}	{04}	{08}	{10}	{20}	{40}	{80}	{1b}	{36}

Table 4.1: Round constant values for the AES algorithm, in bytes.

KeyExpansion($\mathbf{k}; \mathtt{param}_{\mathsf{OWF}}$)

1 : **new** $\bar{\mathbf{k}} \in [\{0,1\}^{32}; N_{\mathsf{st}}(R+1)]$ // Empty expanded key, as an array of words

2 : **for** $i \in [0..N_{\mathsf{k}})$ **do**

3 : $\quad \bar{\mathbf{k}}[i] := \mathbf{k}[i]$ // Here $\mathbf{k} \in [\{0,1\}^{32}; N_{\mathsf{k}}]$ is viewed as an array of words

4 : **for** $i \in [N_{\mathsf{k}}..N_{\mathsf{st}}(R+1))$ **do**

5 : $\quad$ **tmp** $:= \bar{\mathbf{k}}[i-1]$

6 : $\quad$ **if** $i \bmod N_{\mathsf{k}} = 0$ **then**

7 : $\quad\quad$ **tmp** $:=$ SubWord(RotWord(**tmp**)) $+ [\mathtt{Rcon}[i/N_{\mathsf{k}} - 1] \| \{00\} \| \{00\} \| \{00\}]$

8 : $\quad$ **if** $N_{\mathsf{k}} > 6$ **and** $i \bmod N_{\mathsf{k}} = 4$ **then**

9 : $\quad\quad$ **tmp** $:=$ SubWord(**tmp**)

10 : $\quad \bar{\mathbf{k}}[i] := \bar{\mathbf{k}}[i - N_{\mathsf{k}}] +$ **tmp**

11 : **return** $\bar{\mathbf{k}}$

Fig. 4.2: The AES and Rijndael key expansion routines

Encryption routine. The Encrypt routine (Fig. 4.3) matches that given by the AES standard [AES01, Figure 5] and the original Rijndael specification. First, the input plaintext $\mathbf{in} \in \{0,1\}^{32 \cdot N_{\mathsf{st}}}$ is bit-wise XOR-ed with the first N_{st} words of the expanded key at line 3. Second, for all but the last round, the SubBytes, ShiftRows and MixColumns transformations are applied in sequence to the state, before a new XOR with the next N_{st} words of the expanded key (lines 5–9). Finally, the last round applies the same transformations with the exception of MixColumns. All of this leaves the state containing the ciphertext array $\mathbf{out} \in \{0,1\}^{32 \cdot N_{\mathsf{st}}}$.

4.2 Galois Conjugates and Field Norm

Definition 4.1 (Galois Conjugates). *Let $K = \mathbb{F}_{p^m}$ and $L = \mathbb{F}_{p^n}$ be finite fields of characteristic p where $m|n$. Then the map $\phi : L \to L; x \mapsto x^p$ is called the* Frobenius endomorphism. *For $Gal(L/K) = \{\phi^{i \cdot m}\}_{i \in \{0,\ldots,\frac{n}{m}-1\}}$ and for every $a \in L$, the values $\sigma_1(a), \sigma_2(a), \ldots$ with $\sigma_i \in Gal(L/K)$ are the* Galois conjugates *of a for $Gal(L/K)$.*

Definition 4.2 (Norm and Trace). *Let $K = \mathbb{F}_q$ and $L = \mathbb{F}_{q^n}$ be finite fields and $a \in L$, then*

$$N_{L/K}(a) := \prod_{\sigma \in Gal(L/K)} \sigma(a)$$

is called the norm *of a and*

$$Tr_{L/K}(a) := \sum_{\sigma \in Gal(L/K)} \sigma(a)$$

is called the trace *of a.*

```
Encrypt(in, k̄; param_OWF)
 1 : new state ∈ {0,1}^{32·N_st}
 2 : state := in
 3 : AddRoundKey(state, k̄[0..N_st − 1])

 4 : for r ∈ [1..R) do
 5 :   SubBytes(state)
 6 :   ShiftRows_{N_st}(state)
 7 :   MixColumns(state)
 8 :   AddRoundKey(state, k̄[N_st r..N_st(r + 1) − 1])

 9 : SubBytes(state)
10 : ShiftRows_{N_st}(state)
11 : AddRoundKey(state, k̄[N_st R..N_st(R + 1) − 1])

12 : return out := state
```

Fig. 4.3: The encryption routine

It is well-known that $N_{L/K}(a), Tr_{L/K}(a) \in K$. We will make use of the following fact.

Remark 4.3. Let L, K be defined as above. Then L is a K-vector space of dimension n and the functions $F_\beta : L \to K; x \mapsto Tr_{L/K}(\beta \cdot x)$ are exactly all the K-linear functions from vector space L to K when considering all $\beta \in L$.

4.3 Universal Hashing

Two of the consistency checks used in FAEST require a family of linear, universal hash functions. In order to get tight bounds and fast algorithms, we use a combination of small matrix hashes and polynomial hashes, which are designed to take advantage of CPUs with 64-bit binary polynomial multipliers, whilst supporting security parameters $\lambda \in \{128, 192, 256\}$.

The hash functions are specified in Figure 4.4, and analyzed in the remainder of this section. We need the hashes to be ε-almost universal, as defined below. For some intermediate building blocks, we will also use the ε-almost uniform property.

Definition 4.4. *A family of linear hash functions is a family of matrices $\mathcal{H} \subseteq \mathbb{F}_q^{r \times n}$. The family is* ε-almost universal *if for any non-zero $\mathbf{x} \in \mathbb{F}_q^n$,*

$$\Pr_{\mathbf{H} \leftarrow \mathcal{H}}[\mathbf{Hx} = 0] \leq \varepsilon.$$

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

$$\Pr_{\mathbf{H} \leftarrow \mathcal{H}}[\mathbf{Hx} = \mathbf{v}] \leq \varepsilon.$$

Note that the algorithms in Figure 4.4 specify how a random bit string sd of the appropriate length is used to sample a function from the family and evaluate it on a given input $\mathbf{x}$.

Our hashes must also satisfy the following hiding property.

Definition 4.5. *A matrix $\mathbf{H} \in \mathbb{F}_q^{r \times (n+h)}$ is $\mathbb{F}_q^n$-hiding if the distribution of $\mathbf{Hv}$ is independent from $\mathbf{v}_{[0..n)}$ when $\mathbf{v}_{[n..n+h)} \leftarrow \mathbb{F}_q^h$. A hash family $\mathcal{H} \subseteq \mathbb{F}_q^{r \times (n+h)}$ is $\mathbb{F}_q^n$-hiding if every $\mathbf{H} \in \mathcal{H}$ is $\mathbb{F}_q^n$-hiding.*

VOLEHash(sd, $(\mathbf{x}_0, \mathbf{x}_1) \in \{0,1\}^{\ell+2\lambda} \times \{0,1\}^{\lambda+B}$)

// λ-bit $\mathbf{r}_i, \mathbf{s}$; 64-bit $\mathbf{t}$
Parse $\mathsf{sd} = (\mathbf{r}_0 \| \mathbf{r}_1 \| \mathbf{r}_2 \| \mathbf{r}_3 \| \mathbf{s} \| \mathbf{t}) \in \{0,1\}^{5\lambda+64}$
$r_i := \mathsf{ToField}(\mathbf{r}_i, \lambda)$, **for** $i \in [0..3]$
$s := \mathsf{ToField}(\mathbf{s}, \lambda)$
$t := \mathsf{ToField}(\mathbf{t}, 64)$
$\ell' := \lambda \cdot \lceil (\ell + \lambda)/\lambda \rceil$

$\mathbf{x}_0 := \mathbf{x}_0 \| 0^{\ell' - (\ell+\lambda)}$ // pad to multiple of λ
$\hat{\boldsymbol{y}} := \mathsf{ToField}(\mathbf{x}_0, \lambda)$
$\overline{\boldsymbol{y}} := \mathsf{ToField}(\mathbf{x}_0, 64)$
$h_0 := \sum_{i=0}^{\ell'/\lambda - 1} s^{\ell'/\lambda - 1 - i} \cdot \hat{\boldsymbol{y}}[i]$ (in $\mathbb{F}_{2^\lambda}$)
$h_1 := \sum_{i=0}^{\ell'/64 - 1} t^{\ell'/64 - 1 - i} \cdot \overline{\boldsymbol{y}}[i]$ (in $\mathbb{F}_{2^{64}}$)
$h_1' := \mathsf{ToField}(\mathsf{ToBits}(h_1) \| 0^{\lambda-64}, \lambda)$
$(h_2, h_3) := (r_0 h_0 + r_1 h_1', r_2 h_0 + r_3 h_1')$
$\mathbf{h} := (\mathsf{ToBits}(h_2) \| \mathsf{ToBits}(h_3)[0..B)) \oplus \mathbf{x}_1$
return h

ZKHash(sd, $(\boldsymbol{x}_0, x_1) \in \mathbb{F}_{2^\lambda}^{\ell} \times \mathbb{F}_{2^\lambda}$)

// λ-bit $\mathbf{r}_i, \mathbf{s}$; 64-bit $\mathbf{t}$
Parse $\mathsf{sd} = (\mathbf{r}_0 \| \mathbf{r}_1 \| \mathbf{s} \| \mathbf{t}) \in \{0,1\}^{3\lambda+64}$
$r_i := \mathsf{ToField}(\mathbf{r}_i, \lambda)$, **for** $i \in \{0,1\}$
$s := \mathsf{ToField}(\mathbf{s}, \lambda)$
$t := \mathsf{ToField}(\mathbf{t} \| 0^{\lambda-64}, \lambda)$
$h_0 := \sum_{i=0}^{\ell-1} s^{\ell-1-i} \cdot \boldsymbol{x}_0[i]$
$h_1 := \sum_{i=0}^{\ell-1} t^{\ell-1-i} \cdot \boldsymbol{x}_0[i]$
$\mathbf{h} := \mathsf{ToBits}(r_0 h_0 + r_1 \cdot h_1 + x_1)$
return h

Fig. 4.4: Universal hashing algorithms

We will use the following, straightforward method of transforming a uniform hash family into a universal family that is hiding.

Proposition 4.6. *Let $\mathcal{H} \subseteq \mathbb{F}_q^{r \times n}$ be an ε-almost uniform hash family. Let $\mathcal{H}' \subseteq \mathbb{F}_q^{r \times (n+r)}$ be the family $\{[\mathbf{H}\ I_r] : \mathbf{H} \in \mathcal{H}\}$, where I_r is the $r \times r$ identity matrix. Then, it holds that (1) $\mathcal{H}'$ is ε-almost universal, and (2) $\mathcal{H}'$ is $\mathbb{F}_q^n$-hiding.*

Proof. Let $\mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \end{bmatrix}$ be non-zero, for $\mathbf{x}_0 \in \mathbb{F}_q^n$ and $\mathbf{x}_1 \in \mathbb{F}_q^r$. If $\mathbf{H}' \leftarrow \mathcal{H}'$ then, $\mathbf{H}\mathbf{x} = 0$ implies $\mathbf{H}\mathbf{x}_0 = \mathbf{x}_1$, and since at most one of $\mathbf{x}_0, \mathbf{x}_1$ are zero, we cannot have $\mathbf{x}_0 = 0$, so this holds with probability at most ε. For the second part, the hiding property holds because I_r ensures that if the last r elements of the input to the hash are uniform then they perfectly mask the rest. □

4.3.1 Standard Constructions. As building blocks, we use two well-known constructions of linear universal hash families [CW79, BJKS94]. The first is a simple matrix hash family, where $\mathcal{H} = \mathbb{F}_q^{r \times n}$, which is q^{-r}-uniform. The second is a polynomial-based hash, where the input $\mathbf{v} \in \mathbb{F}_q^n$ is parsed as the coefficients of a polynomial of degree up to $n-1$, and sampling a hash function involves evaluating the polynomial at a randomly chosen point in $\mathbb{F}_q$. Since the polynomial has at most $n-1$ roots over $\mathbb{F}_q$, this hash family is $(n-1)/q$-almost universal. We also use a variant of this where the random point is restricted to a subset $S \subset \mathbb{F}_q$, which is $(n-1)/|S|$-universal.

4.3.2 Composition and Truncation of Hashes. We rely on the following composition results. Similar properties have been shown in e.g. [Sti92, Roy22].

Proposition 4.7. *Let $\mathcal{H}, \mathcal{H}'$ be ε and ε'-almost universal families. Then, the concatenation $\left\{ \begin{bmatrix} \mathbf{H} \\ \mathbf{H}' \end{bmatrix} : \mathbf{H} \in \mathcal{H}, \mathbf{H}' \in \mathcal{H}' \right\}$ is $\varepsilon\varepsilon'$-almost universal.*

Proof. Follows from independence of $\mathbf{H}$ and $\mathbf{H}'$. □

Proposition 4.8. *Let $\mathcal{H} \subseteq \mathbb{F}_q^{r' \times n}$ be ε-almost universal and $\mathcal{H}' \subseteq \mathbb{F}_q^{r \times r'}$ be ε'-almost uniform. Then, the product $\{\mathbf{H}'\mathbf{H} : \mathbf{H} \in \mathcal{H}, \mathbf{H}' \in \mathcal{H}'\}$ is $(\varepsilon + \varepsilon')$-almost uniform.*

Proof. Let $\mathbf{x} \in \mathbb{F}_q^n$ be non-zero. Then, $\mathbf{x}' = \mathbf{H}\mathbf{x}$ is non-zero with probability at least $1 - \varepsilon$. If $\mathbf{x}'$ is non-zero, then for any $\mathbf{v} \in \mathbb{F}_q^r$, $\mathbf{H}'\mathbf{x}' \neq \mathbf{v}$ with probability at least $1 - \varepsilon'$, by the uniformity of $\mathcal{H}'$. It follows that $\Pr[\mathbf{H}'\mathbf{H}\mathbf{x} \neq \mathbf{v}] \geq (1 - \varepsilon)(1 - \varepsilon') \geq 1 - \varepsilon - \varepsilon'$, and so the product is an $(\varepsilon + \varepsilon')$-almost uniform family. □

Proposition 4.9. *Let $\delta \in \mathbb{N}$ and $\mathcal{H} \subseteq \mathbb{F}_q^{r \times n}$ be an ε-almost uniform family. Then, the truncated family $\{\mathbf{H}_{[0..r-\delta)} : \mathbf{H} \in \mathcal{H}\}$ is εq^δ-uniform.*

Proof. For each $\mathbf{H} \in \mathcal{H}$, write $\mathbf{H} = \begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \end{bmatrix}$, where $\mathbf{H}_0 \in \mathbb{F}_q^{(r-\delta) \times n}$ and $\mathbf{H}_1 \in \mathbb{F}_q^{\delta \times n}$. Let $\mathbf{y} = \begin{bmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \end{bmatrix} \in \mathbb{F}_q^r$ and $\mathbf{x} \in \mathbb{F}_q^n \setminus \{0\}$. If $\mathbf{H} \leftarrow \mathcal{H}$ we have $\Pr[\mathbf{H}\mathbf{x} = \mathbf{y}] \leq \varepsilon$. Applying conditional probability, we get

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

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

4.3.3 VOLE Universal Hash. The first check, to verify consistency of the VOLE correlations, requires a family that is linear over $\mathbb{F}_2$. It's evaluated on inputs in $\mathbb{F}_2^{\hat{\ell}}$, where $\hat{\ell} = \ell + 3\lambda + B$, ℓ is the witness length and $B = 16$ is a parameter chosen for security.[14]

To build the hash, we map the seed sd into $(r_0, r_1, r_2, r_3, s, t) \in \mathbb{F}_{2^\lambda}^5 \times \mathbb{F}_{2^{64}}$. The input $\mathbf{x} \in \mathbb{F}_2^{\hat{\ell}}$ is first split into $(\mathbf{x}_0, \mathbf{x}_1)$, where $\mathbf{x}_0 \in \{0,1\}^{\ell+\lambda}$, and then $\mathbf{x}_0$ is parsed twice, first as a vector $\hat{\boldsymbol{y}}$ of $\mathbb{F}_{2^\lambda}$ elements, and then as a vector $\overline{\boldsymbol{y}}$ of $\mathbb{F}_{2^{64}}$ elements. Then, compute

$$\begin{aligned} h_0 &= \hat{y}_0 s^{\hat{\ell}/\lambda - 1} + \hat{y}_1 s^{\hat{\ell}/\lambda - 2} + \cdots + \hat{y}_{\hat{\ell}/\lambda - 2} s + \hat{y}_{\hat{\ell}/\lambda - 1} \quad \text{in } \mathbb{F}_{2^\lambda}, \\ h_1 &= \overline{y}_0 t^{\hat{\ell}/64 - 1} + \overline{y}_1 t^{\hat{\ell}/64 - 2} + \cdots + \overline{y}_{\hat{\ell}/64 - 1} t + \overline{y}_{\hat{\ell}/64 - 1} \quad \text{in } \mathbb{F}_{2^{64}} \end{aligned}$$

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

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

Finally, take the first $\lambda + B$ bits of the concatenation of the field elements h_2 and h_3, and XOR this with $\mathbf{x}_1$ to obtain the output.

We argue security of the construction below. Note that instead of aiming for $\varepsilon = 2^\lambda$, we aim for $2^{-\lambda - B}$, where $B = 16$ (Table 3.2). The extra few bits of security compensate for the $\binom{\tau}{2}$ security loss in the proof of the SoftSpokenVOLE protocol from [BBD+23b].

[14] Our signature scheme actually calls VOLEHash on input an $\hat{\ell} \times \lambda$ matrix, which is translated into computing the hash on each column separately, with the same seed.

Lemma 4.10. VOLEHash *is an* ε_{v}*-almost universal hash family in* $\mathbb{F}_2^{(\lambda+B)\times\hat{\ell}}$*, for* $\varepsilon_{\mathsf{v}} = 2^{-\lambda-B}(1+2^{B-50})$*, if* $\hat{\ell} \le 2^{13}$*. Furthermore,* VOLEHash *is* $\mathbb{F}_2^{\ell+\lambda}$*-hiding.*

Proof. We show ε_{v}-uniformity of the hash that outputs the first $\lambda+B$ bits of (h_2, h_3), i.e. without adding $\mathbf{x}_1$. By Proposition 4.6, this implies ε_{v}-universality of the final hash, as well as the hiding property.

The first part of the hash — computing h_0, h_1 — is a concatenation of two polynomial hashes, over either $\mathbb{F}_{2^\lambda}$ or $\mathbb{F}_{2^{64}}$. These are ε-universal with $\varepsilon = d/|\mathbb{F}|$, where d is the polynomial degree and $\mathbb{F}$ is the field, and we have $d \le \hat{\ell}/64$. Since binary field multiplication is bilinear over $\mathbb{F}_2$, both of these hashes are also $\mathbb{F}_2$-linear. Applying Proposition 4.7, their concatenation is then ε_0-universal with $\varepsilon_0 \le \hat{\ell}^2/2^{\lambda+76}$. Note that for $\hat{\ell} \le 2^{13}$, we have $\varepsilon_0 \le 2^{-\lambda-50}$.

The second part of the hash starts with a 2×2 matrix hash, which is $2^{-2\lambda}$-uniform. After truncation, the resulting hash is ε_1-uniform for $\varepsilon_1 = 2^{-\lambda-B}$, by Proposition 4.9. The final combined hash is the product of these two parts, so applying Proposition 4.8 and summing the probabilities, we get that for all $\hat{\ell} \le 2^{13}$, the hash is ε_{v}-uniform for $\varepsilon_{\mathsf{v}} = \varepsilon_0 + \varepsilon_1 \le 2^{-\lambda-B}(1+2^{B-50})$. □

4.3.4 ZK Universal Hash.

Our second hash, used for verifying AES constraints in ZK, must be linear over $\mathbb{F}_{2^\lambda}$. It works on inputs of $\ell' = \ell + 1$ field elements. We map the seed sd into $r_0, r_1, s, t \in \mathbb{F}_{2^\lambda}$, where r_0, r_1 and s are uniform, while t (viewed as an $\mathbb{F}_2$ polynomial) is zero in all its degree ≥ 64 coefficients (and uniform otherwise). Let

$$\mathbf{r}^\top := \begin{bmatrix} r_0 & r_1 \end{bmatrix} \begin{bmatrix} s^{\ell-1} & s^{\ell-2} & \cdots & s & 1 \\ t^{\ell-1} & t^{\ell-2} & \cdots & t & 1 \end{bmatrix}$$

The hash of an input $\boldsymbol{x} = (\boldsymbol{x}_0, x_1) \in \mathbb{F}_{2^\lambda}^\ell \times \mathbb{F}_{2^\lambda}$ is simply $h = \mathbf{r}^\top \boldsymbol{x}_0 + x_1$.

Lemma 4.11. ZKHash *is an* $\varepsilon_{\mathsf{zk}}$*-almost universal hash family in* $\mathbb{F}_{2^\lambda}^{1\times\ell'}$*, for* $\varepsilon_{\mathsf{zk}} = 2^{-\lambda}(1+2^{-38})$*, if* $\ell' \le 2^{13}$*. Furthermore,* ZKHash *is* $\mathbb{F}_{2^\lambda}^{\ell}$*-hiding.*

Proof. As with VOLEHash, by Proposition 4.6 it suffices to show $\varepsilon_{\mathsf{zk}}$-uniformity of the hash defined by $\mathbf{r}^\top \mathbf{x}_0$.

This hash is a product of two separate hashes. The outer hash — multiplication by $(r_0\ r_1)$ — is a standard matrix hash, which is $2^{-\lambda}$-almost uniform. The inner hash is a concatenation of two polynomial evaluations, the first of which defines an almost-universal hash for $\varepsilon_0 = (\ell'-1)/2^\lambda$, and the second for $\varepsilon_1 = (\ell'-1)/2^{64}$; together, this gives an $\varepsilon_0\varepsilon_1$-almost universal hash (via Proposition 4.7), where $\varepsilon_0\varepsilon_1 \le (\ell')^2/2^{\lambda+64} \le 2^{-\lambda-38}$, for all $\ell' \le 2^{13}$. Combining the inner and outer parts, from Proposition 4.8 we get that ZKHash is $\varepsilon_{\mathsf{zk}}$-almost uniform for $\varepsilon_{\mathsf{zk}} = 2^{-\lambda}(1+2^{-38})$. □

5 VOLE-in-the-Head Functions

In this section we describe the algorithms required to build the VOLEitH protocol. This is obtained from a batch all-but-one vector commitment scheme (BAVC), presented in Section 5.2, whose outputs are transformed in VOLE correlations using the ConvertToVOLE algorithm described in Section 5.4. Finally, both of these are used to build the VOLEitH protocol, described in Section 5.5.

The main parameters used by the VOLEitH protocol are listed in Table 5.1. As described informally in Section 2.1, our BAVC scheme implements a single large GGM tree of size $L = \sum_i N_i$ that collectively commits to τ vectors $\{(m_{i,0}, \ldots, m_{i,N_i-1})\}_{i=0}^{\tau-1}$, of which τ_1 of bit-length k and τ_0 of bit-length $k-1$. N_i represents the length of vector i, $i = [0..\tau)$.

Parameter	Formula	Description
n_{leafcom}	3 for FAEST, 2 for FAEST-EM	Number of λ-bit blocks in leaf commitments
τ_1	$(\lambda - w_{\mathsf{grind}}) \bmod \tau$	Number of larger small-VOLE instances
τ_0	$\tau - \tau_1$	Number of smaller small-VOLE instances
k	$\lfloor (\lambda - w_{\mathsf{grind}})/\tau \rfloor + 1$	Bit-length of the larger small-VOLE instances
$k-1$	$k-1$	Bit-length of the smaller small-VOLE instances
N_i	2^k if $i < \tau_1$, 2^{k-1} otherwise	Field size in the i-th small-VOLE instance
L	$\sum_{i=0}^{\tau-1} N_i = \tau_1 2^k + \tau_0 2^{k-1}$	Number of leaves in the GGM tree
B	16	Extra length padding for the VOLE check
$\hat{\ell}$	$\ell + 3\lambda + B$	Witness length plus extra randomness for VOLE + ZK checks

Table 5.1: Extra parameters used in the VOLE-in-the-head components of FAEST

5.1 Building blocks for BAVC and FAEST commitments

Before describing the main BAVC scheme, we give some auxiliary algorithms.

– BAVC.PosInTree. It uses the fact that each leaf node in the GGM tree corresponds to exactly one position (i, j) in the τ vectors. More specifically, the function maps a pair (i, j), corresponding to the position j in the vector i out of τ total vectors, to a unique leaf index $\alpha \in [L-1..2L-1)$.

BAVC.PosInTree(i, j)

INPUT: $i \in [0..\tau)$, $j \in [0..N_i)$
OUTPUT: $\alpha \in [L-1..2L-1)$
1 : **if** $j < 2^{k-1}$ **then**
2 : **return** $L - 1 + \tau j + i$
3 : **else**
4 : **return** $L - 1 + \tau 2^{k-1} + \tau_1 (j \bmod 2^{k-1}) + i$

Fig. 5.2: Leaf position utility function in BAVC

FAEST.LeafHash(uhash, $\mathbf{x}$)

INPUT: uhash $\in \{0,1\}^{3\lambda}$, $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1) \in \{0,1\}^{\lambda} \times \{0,1\}^{3\lambda}$
OUTPUT: $\mathbf{h} \in \{0,1\}^{3\lambda}$
1 : $u \leftarrow$ ToField(uhash, 3λ)
2 : $x_0 \leftarrow$ ToField($\mathbf{x}_0 \| 0^{2\lambda}, 3\lambda$)
3 : $x_1 \leftarrow$ ToField($\mathbf{x}_1, 3\lambda$)
4 : $h \leftarrow x_0 \cdot u + x_1$
5 : $\mathbf{h} \leftarrow$ ToBits(h)
6 : **return** $\mathbf{h}$

FAEST.LeafCommit(r, iv, twk, uhash)

INPUT: $r \in \{0,1\}^{\lambda}$, iv $\in \{0,1\}^{128}$
twk $\in \{0,1\}^{32}$, uhash $\in \{0,1\}^{3\lambda}$
OUTPUT: sd $\in \{0,1\}^{\lambda}$, com $\in \{0,1\}^{3\lambda}$
1 : (sd, s_0, s_1, s_2) $\leftarrow$ PRG(r, iv, twk; 4λ)
2 : (sd, $\mathbf{x}_1$) $\leftarrow$ PRG(r, iv, twk; 4λ)
3 : com $\leftarrow$ LeafHash(uhash, (sd$\|s_0\|s_1\|s_2$))
4 : com $\leftarrow$ LeafHash(uhash, (sd$\|\mathbf{x}_1$))
5 : **return** (sd, com)

FAEST-EM.LeafCommit(r, iv, twk)

INPUT: $r \in \{0,1\}^{\lambda}$, iv $\in \{0,1\}^{128}$, twk $\in \{0,1\}^{32}$
OUTPUT: sd $\in \{0,1\}^{\lambda}$, com $\in \{0,1\}^{2\lambda}$
1 : com $\leftarrow$ PRG(r, iv, twk; 2λ)
2 : sd $\leftarrow r$
3 : **return** (sd, com)

Fig. 5.3: New AES-based leaf commitment functions for BAVC

– BAVC.LeafCommit. It represents a direct random oracle-based leaf commitment function for BAVC, that could be used if we don't want to rely on an AES-based approach. This algorithm combines a short seed s, a 128-bit vector iv and a 32-bit vector twk into a single input for a random oracle that outputs 3λ bits. The first λ bits become the seed sd, while the remaining 2λ bits form the commitment.
– FAEST.LeafHash. It describes the hash used in FAEST.LeafCommit. To build the hash, it maps the seed uhash $\in \{0,1\}^{3\lambda}$ and the input vector $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1) \in \{0,1\}^{\lambda} \times \{0,1\}^{3\lambda}$ into field elements $u, x_0, x_1 \in \mathbb{F}_{2^{3\lambda}}$. Then it computes the hash $h = x_0 \cdot u + x_1 \in \mathbb{F}_{2^{3\lambda}}$, and finally converts back to bits.
– FAEST.LeafCommit. This functions expands a λ-bit seed r via PRG into 4λ bits. The first block of λ bits becomes leaf sd; the other 3λ bits $\mathbf{x}_1$, combined with an additional uhash input, are processed by FAEST.LeafHash to produce a 3λ-bit commitment.
– FAEST-EM.LeafCommit. This simpler variant of LeafCommit calls PRG on $(r, \mathsf{iv}, \mathsf{twk})$, but only requests 2λ bits of output. It returns the original seed r as leaf seed and uses the 2λ-bit PRG output directly as the commitment.

5.2 Batch All-but-One Vector Commitments

In this section we describe our BAVC algorithms given in Figure 5.4 and Figure 5.5.

BAVC.Commit. It generates a batch vector commitment as follows. It starts by generating seeds for each internal node of a GGM tree using PRG (Figure 3.6). The resulting tree has L leaves, enough to cover all positions across the τ vectors. For each leaf, a leaf commitment function LeafCommit is called, producing both a leaf seed $\mathsf{sd}_{i,j}$ and a commitment $\mathsf{com}_{i,j}$. The algorithm then groups these commitments by vector, i.e., for each $i \in [0..\tau - 1]$, it collects $\{\mathsf{com}_{i,j}\}_{j=0}^{N_i-1}$ and hashes them together obtaining

h_i. The final commitment com is the hash of all τ commitments $h_0, \ldots, h_{\tau-1}$. The algorithm returns (1) the commitment com, (2) the full collection of GGM seeds and leaf commitments in decom for later decommitment, and (3) the leaf seeds $\mathsf{sd}_{i,j}$, $i \in 0, \tau - 1$.

BAVC.Open. Given opening values $\Delta_i, i \in [0..\tau - 1]$, to keep hidden in each of the τ committed vectors, this algorithm produces a "partial opening" decom_I. It does so by marking the unopened leaves in the GGM tree and revealing the seeds on those internal nodes that suffice to reconstruct all other leaves. If the chosen pattern of hidden leaves is too large to open within a threshold T_{open}, the algorithm aborts. Otherwise, the revealed seeds and unopened leaf commitments are collected as decom_I.

BAVC.Reconstruct. This algorithm takes the partial opening decom_I, along with the index vector $I = (\Delta_0, \ldots, \Delta_{\tau-1})$, and reconstructs every leaf except those at the unopened positions. Specifically, it parses the revealed seeds from decom_I, walks up the GGM tree to recover missing seeds, and re-computes the per-leaf commitments $\mathsf{com}_{i,j}$. The final output is (1) the recomputed commitment com and (2) the set of seeds for all opened leaves.

5.3 BAVC Correctness

We show that the BAVC scheme presented in the previous section satisfies *correctness with aborts*. Informally, by "correctness," we mean that if Open returns a non-$\perp$ partial opening for an index vector $I = (\Delta_0, \ldots, \Delta_{\tau-1})$, then Reconstruct indeed recovers the original commitment com and the correct hidden seeds for all opened positions.

Proposition 5.1 (BAVC Correctness with Aborts). *Let* BAVC *be the batch all-but-one vector commitment scheme described in Figure 5.4 and Figure 5.5. Then*

$$\forall(\mathsf{com}, \mathsf{decom}, \{\mathsf{sd}_{i,j}\}) \leftarrow \mathsf{BAVC.Commit}(r, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}}),$$

where $r \leftarrow \{0,1\}^\lambda$ *is the root seed,* $\mathsf{iv} \leftarrow \{0,1\}^{128}$ *is an AES-CTR IV, and* $\mathtt{param}, \mathtt{param}_{\mathsf{VOLE}}$ *are public parameters, and* $\forall\, \mathsf{decom}_I \leftarrow \mathsf{BAVC.Open}(\mathsf{decom}, I = (\Delta_0, \ldots, \Delta_{\tau-1}))$,

if Open *does not abort (so that* $\mathsf{decom}_I \neq \perp$*), then*

$$(\hat{\mathsf{com}}, \{\hat{\mathsf{sd}}_{i,j}\}_{j \neq \Delta_i}) \leftarrow \mathsf{BAVC.Reconstruct}(\mathsf{decom}_I, I, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$$

is such that $\hat{\mathsf{com}} = \mathsf{com}$ *and* $\hat{\mathsf{sd}}_{i,j} = \mathsf{sd}_{i,j}$ *for all* $j \neq \Delta_i$.

Proof. (Sketch.) We prove that any $\mathsf{decom}_I \neq \perp$ output by BAVC.Open contains exactly the information needed for BAVC.Reconstruct to re-compute and recover every unhidden leaf's seed ($\mathsf{sd}_{i,j}, j \neq \Delta_i$), as well as re-compute the final hash com.

First, we recall from BAVC.Commit (Figure 5.4) that each internal node seed k_α of the GGM tree is expanded via a deterministic pseudorandom generator:

$$(k_{2\alpha+1}, k_{2\alpha+2}) \leftarrow \mathsf{PRG}(k_\alpha, \mathsf{iv}, \alpha;\, 2\lambda).$$

Hence, if the same node α and parent seed k_α appear in two different executions, they will produce the same child seeds. This implies that all node seeds in the unhidden part of the tree are fixed as soon as we fix $k_0 = r$ and the expansions up to those unhidden nodes.

When $\mathsf{BAVC.Open}(\mathsf{decom}, I)$ runs, it marks the leaf corresponding to $\mathsf{BAVC.PosInTree}(i, \Delta_i)$ for each vector i as "hidden" and includes exactly the subset of child seeds needed to re-derive all other leaves. This selective opening does not give out the seeds on a path if that path leads to a fully hidden leaf. The algorithm also checks that the total size of revealed seeds stays below some threshold T_{open}; if not, it aborts with $\perp$. Concretely, in the final partial decommitment decom_I, generated in BAVC.Open, we see:

BAVC.Commit$(r, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$

INPUT: $r \in \{0,1\}^{\lambda}, \mathsf{iv} \in \{0,1\}^{128}$
OUTPUT: $\mathsf{com} \in \{0,1\}^{2\lambda}, \mathsf{decom} \in \{0,1\}^{(2L-1)\lambda + n_{\mathsf{leafcom}}\lambda\tau}, \mathsf{seeds} \in \{0,1\}^{k\lambda}$

1: **if** variant is FAEST-λ **do**
2: $(\mathsf{uhash}_0, \ldots, \mathsf{uhash}_{\tau-1}) \leftarrow \mathsf{H}_0(\mathsf{iv}; 3\lambda\tau)$
3: $\mathsf{uhash} \leftarrow \mathsf{H}_0(\mathsf{iv}; 3\lambda)$
4: // For FAEST-EM, ignore uhash
5: $k_0 \leftarrow r$
6: **for** $\alpha \in [0..L-2]$ **do**
7: $(k_{2\alpha+1}, k_{2\alpha+2}) \leftarrow \mathsf{PRG}(k_\alpha, \mathsf{iv}, \alpha; 2\lambda)$
8: **for** $i \in [0..\tau)$ **do**
9: **for** $j \in [0..N_i)$ **do**
10: $\alpha \leftarrow \mathsf{BAVC.PosInTree}(i, j)$
11: $(\mathsf{sd}_{i,j}, \mathsf{com}_{i,j}) \leftarrow \mathsf{LeafCommit}(k_\alpha, \mathsf{iv}, i + L - 1, \mathsf{uhash}_i)$
12: $h_i \leftarrow \mathsf{H}_1(\mathsf{com}_{i,0} \| \ldots \| \mathsf{com}_{i,N_i-1})$
13: $\mathsf{com} \leftarrow \mathsf{H}_1(h_0 \| \ldots \| h_{\tau-1})$
14: $\mathsf{decom} \leftarrow \left((k_\alpha)_{\alpha \in [0..2L-1)}, (\mathsf{com}_{i,j})_{i \in [0..\tau), j \in [0..N_i)}\right)$
15: **return** $(\mathsf{com}, \mathsf{decom}, (\mathsf{sd}_{0,i}, \ldots, \mathsf{sd}_{N_i-1,i})_{i \in [0..\tau)})$

BAVC.Open$(\mathsf{decom}, I = (\Delta_0, \ldots, \Delta_{\tau-1}); \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$

INPUT: $\mathsf{decom} \in \{0,1\}^{(2L-1)\lambda + n_{\mathsf{leafcom}}\lambda\tau}, I \in [0..N_0) \times \cdots \times [0..N_{\tau-1})$
OUTPUT: $\mathsf{decom}_I \in \{0,1\}^{n_{\mathsf{leafcom}}\lambda\tau + T_{\mathsf{open}}\lambda} \cup \{\bot\}$

1: Parse $\mathsf{decom} := \left((k_\alpha)_{\alpha \in [0..2L-1)}, (\mathsf{com}_{i,j})_{i \in [0..\tau), j \in [0..N_i)}\right)$
2: // Initialize opening with the τ leaf hashes
3: $\mathsf{decom}_I \leftarrow (\mathsf{com}_{0,\Delta_0}, \ldots, \mathsf{com}_{\tau-1,\Delta_{\tau-1}})$
4: // Mark tree nodes that will not be revealed
5: $S \leftarrow 0^{2L-1} \in \{0,1\}^{2L-1}$
6: $n_h \leftarrow 0$ // number of marked hidden nodes
7: **for** $i \in [0..\tau)$ **do**
8: // Mark each unopened leaf and its descendants
9: $\alpha \leftarrow \mathsf{BAVC.PosInTree}(i, \Delta_i)$
10: $S[\alpha] \leftarrow 1$
11: $n_h \leftarrow n_h + 1$
12: **while** $\alpha > 0$ and $S[\lfloor (\alpha - 1)/2 \rfloor] = 0$ **do**
13: $\alpha \leftarrow \lfloor (\alpha - 1)/2 \rfloor$
14: $S[\alpha] \leftarrow 1$
15: $n_h \leftarrow n_h + 1$
16: **if** $n_h - 2\tau + 1 > T_{\mathsf{open}}$ **then**
17: **return** $\bot$
18: // Add tree nodes to opening information
19: **for** $i = L - 2$ to 0 **do**
20: // If exactly one child needs to be opened, open it
21: **if** $S[2i+1] \oplus S[2i+2] = 1$ **then**
22: $\alpha \leftarrow 2i + 1 + S[2i+1]$
23: Append k_α to decom_I
24: Zero-append decom_I to length $n_{\mathsf{leafcom}}\lambda\tau + T_{\mathsf{open}}\lambda$ bits
25: **return** decom_I

Fig. 5.4: Commit and open algorithms for batch all-but-one vector commitment

BAVC.Reconstruct($\mathsf{decom}_I, I = (\Delta_0, \ldots, \Delta_{\tau-1}), \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}}$)

INPUT: $\mathsf{decom}_I \in \{0,1\}^{n_{\mathsf{leafcom}}\lambda\tau + T_{\mathsf{open}}\lambda}, I \in [0..N_0) \times \cdots \times [0..N_{\tau-1})$
OUTPUT: $\mathsf{com} \in \{0,1\}^{2\lambda}$ and $L - \tau$ seeds in $\{0,1\}^\lambda$

Parse $\mathsf{decom}_I = (\mathsf{com}_{0,\Delta_0}, \ldots, \mathsf{com}_{\tau-1,\Delta_{\tau-1}}, \mathsf{nodes})$
if variant is FAEST-λ **do**
  $(\mathsf{uhash}_0, \ldots, \mathsf{uhash}_{\tau-1}) \leftarrow \mathsf{H}_0(\mathsf{iv}; 3\lambda\tau)$
  $\mathsf{uhash} \leftarrow \mathsf{H}_0(\mathsf{iv}; 3\lambda)$
// For FAEST-EM, ignore uhash
// Mark tree nodes that are not revealed
$S \leftarrow 0^{2L-1} \in \{0,1\}^{2L-1}$
for $i \in [0..\tau)$ **do**
  // Mark each unopened leaf node
  $\alpha \leftarrow$ BAVC.PosInTree(i, Δ_i)
  $S[\alpha] \leftarrow 1$
// Walk up the tree, copying nodes from decom_I
for $i = L - 2$ to 0 **do**
  $S[i] \leftarrow S[2i+1] \vee S[2i+2]$
  // If exactly one child has been opened, take seed from the opening
  if $S[2i+1] \oplus S[2i+2] = 1$ **then**
    if nodes is empty **then**
      return $\perp$
    $\alpha \leftarrow 2i + 1 + S[2i+1]$
    Parse $\mathsf{nodes} = (k_\alpha, \mathsf{nodes}')$ // Extract node $k_\alpha \in \{0,1\}^\lambda$
    $\mathsf{nodes} \leftarrow \mathsf{nodes}'$
if nodes is not empty, or not all zeroes **then**
  return $\perp$
// Expand the tree
for $i \in [0..L-1)$ **do**
  if $S[i] = 0$ **do**
    $(k_{2i+1}, k_{2i+2}) \leftarrow \mathsf{PRG}(k_i, \mathsf{iv}, i; 2\lambda)$
for $i \in [0..\tau)$ **do**
  for $j \in [0..N_i)$ **do**
  $\alpha \leftarrow$ BAVC.PosInTree(i, j)
  if $S[\alpha] = 1$ **then**
    $h_{i,j} \leftarrow \mathsf{com}_{i,j}$
  else
    $(\mathsf{sd}_{i,j}, h_{i,j}) \leftarrow \mathsf{LeafCommit}(k_\alpha, \mathsf{iv}, i + L - 1, \mathsf{uhash}_i)$

  $h_i \leftarrow \mathsf{H}_1(h_{i,0} \| \ldots \| h_{i,N_i})$
$\mathsf{com} \leftarrow \mathsf{H}_1(h_0 \| \ldots \| h_{\tau-1})$
return $(\mathsf{com}, (\mathsf{sd}_{i,j})_{j \in [0..N_i) \setminus \{\Delta_i\}})_{i \in [0..\tau)}$

Fig. 5.5: Reconstruction algorithm for batch all-but-one vector commitment

1. The τ hidden leaf commitments $\{\mathsf{com}_{i,\Delta_i}\}_{i=0}^{\tau-1}$.
2. The seeds of children at internal nodes having "exactly one hidden child." These seeds allow re-expansion of the unmarked subtrees, but do not reveal the seeds on the fully hidden paths.

Hence $\mathsf{decom}_I \neq \perp$ implies BAVC.Open has given the verifier *sufficient* information to re-derive all unhidden leaves.

When Reconstruct is called on input decom_I, it does as follows:

(a) It parses the hidden leaf commitments $\{\mathsf{com}_{i,\Delta_i}\}$;
(b) It re-expands the unhidden portion of the GGM tree from the revealed node seeds;
(c) It runs the same leaf-commit function to get $(\mathsf{sd}_{i,j}, \mathsf{com}_{i,j})$ for $j \neq \Delta_i$. All expansions are deterministic, so by comparing with BAVC.Commit's expansions, one sees that $\mathsf{sd}_{i,j}$ and $\mathsf{com}_{i,j}$ must coincide for the unhidden leaves.

Finally, BAVC.Reconstruct re-computes the vector-level partial commitments

$$h_i \;=\; \mathsf{H}_1\big(\mathsf{com}_{i,0} \| \cdots \| \mathsf{com}_{i,N_i-1}\big)$$

and concatenates them, hashing again:

$$\mathsf{com}' \;=\; \mathsf{H}_1\big(h_0 \| \ldots \| h_{\tau-1}\big).$$

For unhidden leaves, $\mathsf{com}_{i,j}$ are precisely the ones re-derived before. For the hidden leaves, the partial opening decom_I has already included the correct commitments $\mathsf{com}_{i,\Delta_i}$. Therefore the entire set $\{\mathsf{com}_{i,j}\}_{i,j\in[N_i],\, j\neq\Delta_i} \cup \{\mathsf{com}_{i,\Delta_i}\}$ matches the set from BAVC.Commit. Hence $\mathsf{com}' = \mathsf{com}$, except with negligible probability in the event of a hash collision.

Other than the commitment com, the BAVC.Reconstruct algorithm also returns $\{\mathsf{sd}_{i,j}\}_{j\neq\Delta_i}$ as re-derived from the GGM expansions and leaf commits. By the same argument as before, each such $\mathsf{sd}_{i,j}$ is the same as in BAVC.Commit. Meanwhile, the seeds for the hidden indices (i, Δ_i) never appear in decom_I and remain unknown to the verifier.

Since every unhidden seed and leaf commit re-expands to the same value, we have that BAVC.Reconstruct(decom_I, I) re-creates exactly the same com from BAVC.Commit. No new information about the hidden leaves $\{\mathsf{sd}_{i,\Delta_i}\}$ is revealed, beyond the $\mathsf{com}_{i,\Delta_i}$. Hence, provided Open does not abort, the batch all-but-one vector commitment satisfies correctness. □

5.4 Seed Expansion and Conversion to VOLE

After committing to N random seeds using the all-but-one vector commitment, each seed is expanded to a longer vector of $\mathbb{F}_q$ elements using PRG. The set of N vectors is then converted into a VOLE correlation over $\mathbb{F}_N$, for $N = 2^d$, which we represent as d VOLE correlations over $\mathbb{F}_2$. Recall from Section 2.1 that this conversion requires the signer to compute, in $\mathbb{F}_N$

$$\mathbf{u} = \sum_{i=0}^{N-1} \mathsf{PRG}(\mathsf{sd}_i), \quad \mathbf{v} = \sum_{i=0}^{N-1} i \cdot \mathsf{PRG}(\mathsf{sd}_i),$$

where PRG expands each seed to a vector in $\{0,1\}^{\hat{\ell}}$ and i is viewed as an element of $\mathbb{F}_N$. The verifier, when given some index Δ and sd_i for all $i \neq \Delta$, will compute

$$\mathbf{q} = \sum_{i=0}^{N-1} (\Delta - i) \cdot \mathbf{r}_i$$

Instead of computing the above directly, the signer and verifier use a divide-and-conquer method [Roy22] to iteratively compute the vectors $\mathbf{v}_j$ and $\mathbf{q}_j$, which represent the j-th bits extracted from $\mathbf{v}, \mathbf{q}$. This algorithm is shown in Figure 5.6.

This algorithm is run by the prover using all N seeds sd_i, while the verifier will run the same algorithm where one of the seeds is unknown. It therefore sets one of the seeds to $\perp$ and ignores the $\mathbf{u}$ part of the output. For correctness, the verifier must additionally permute its seeds according to the permutation $i \mapsto i \oplus \Delta$.

$\mathsf{ConvertToVOLE}(\mathsf{sd}_0, \ldots, \mathsf{sd}_{N-1}, \mathsf{iv}, \mathsf{twk}; \hat{\ell})$

INPUT: $\mathsf{sd}_i \in \{0,1\}^\lambda$, $\mathsf{iv} \in \{0,1\}^{128}$, $\mathsf{twk} \in \{0,1\}^{32}$, $\hat{\ell} \in \mathbb{N}$
OUTPUT: $\mathbf{u}, \mathbf{v}_i \in \{0,1\}^{\hat{\ell}}$

1 : $d := \log N$
2 : $\mathbf{r}_{0,0} := 0^{\hat{\ell}}$ **if** $\mathsf{sd}_0 = \perp$ **else** $\mathsf{PRG}(\mathsf{sd}_0, \mathsf{iv}, \mathsf{twk}; \hat{\ell})$
3 : **for** $i \in [1..N)$ **do**
4 : $\quad \mathbf{r}_{0,i} := \mathsf{PRG}(\mathsf{sd}_i, \mathsf{iv}, \mathsf{twk}; \hat{\ell})$
5 : $\mathbf{v}_0 := \cdots := \mathbf{v}_{d-1} := 0^{\hat{\ell}}$
6 : **for** $j \in [0..d)$ **do**
7 : $\quad$ **for** $i \in [0..N/2^{j+1})$ **do**
8 : $\quad\quad \mathbf{v}_j := \mathbf{v}_j \oplus \mathbf{r}_{j,2i+1}$
9 : $\quad\quad \mathbf{r}_{j+1,i} := \mathbf{r}_{j,2i} \oplus \mathbf{r}_{j,2i+1}$
10 : $\mathbf{u} := \mathbf{r}_{d,0}$
11 : **return** $(\mathbf{u}, \mathbf{v}_0, \ldots, \mathbf{v}_{d-1}) \in (\mathbb{F}_2^{\hat{\ell}})^{d+1}$

Fig. 5.6: Seed expansion and conversion to VOLE

When run by prover and verifier on consistent inputs, the ConvertToVOLE algorithm gives the following correlation guarantee.

Proposition 5.2. *Let* $(\mathsf{sd}_0, \ldots, \mathsf{sd}_{N-1})$ *and* $(\mathsf{sd}'_0, \ldots, \mathsf{sd}'_{N-1})$ *be seeds such that* $\mathsf{sd}'_i = \mathsf{sd}_{i \oplus \Delta}$ *for all* $i > 0$, *for some* $\Delta \in [0..N)$, *and* $\mathsf{sd}'_0 = \perp$. *Then, if*

$$(\delta_0, \ldots, \delta_{d-1}) := \mathsf{BitDec}(\Delta, d), (\mathbf{u}, \mathbf{v}_0, \ldots, \mathbf{v}_{d-1}) := \mathsf{ConvertToVOLE}(\mathsf{sd}_0, \ldots, \mathsf{sd}_{N-1}; \hat{\ell}),$$

and

$$(\mathbf{u}', \mathbf{q}_0, \ldots, \mathbf{q}_{d-1}) := \mathsf{ConvertToVOLE}(\mathsf{sd}'_0, \ldots, \mathsf{sd}'_{N-1}; \hat{\ell}),$$

it holds that

$$\mathbf{q}_j = \mathbf{v}_j \oplus \delta_j \cdot \mathbf{u}, \quad \text{for } j \in [0..d).$$

Proof. For convenience, we will consider the inputs of the algorithm with the PRG already applied, i.e. view the strings $\mathbf{r}_{0,0}, \ldots, \mathbf{r}_{0,N-1}$ as the prover's input. Denoting the verifier's input by $\mathbf{r}'_{0,1}, \ldots, \mathbf{r}'_{0,N-1}$, so it holds that $\mathbf{r}'_{0,i} = \mathbf{r}_{0,i \oplus \Delta}$ for all $i > 0$.

First, we consider the $\mathbf{r}_{j,i}$ values computed by the prover. For the value $\mathbf{r}_{1,i}$ we see that $\mathbf{r}_{1,0} = \mathbf{r}_{0,0} \oplus \mathbf{r}_{0,1}$, $\mathbf{r}_{1,1} = \mathbf{r}_{0,2} \oplus \mathbf{r}_{0,3}$ etc. Hence for $\mathbf{r}_{2,0}$ we obtain $\mathbf{r}_{2,0} = \mathbf{r}_{1,0} \oplus \mathbf{r}_{1,1} = \mathbf{r}_{0,0} \oplus \cdots \oplus \mathbf{r}_{0,3}$. We can therefore write $\mathbf{r}_{j,i} = \sum_{k=i \cdot 2^j}^{(i+1) \cdot 2^j - 1} \mathbf{r}_{0,k}$ and $\mathbf{u} = \mathbf{r}_{d,0} = \mathbf{r}_{0,0} \oplus \cdots \oplus \mathbf{r}_{N-1,0}$.

We now argue correctness by induction on d. With $d = 1$, the prover, on input $(\mathbf{r}_{0,0}, \mathbf{r}_{0,1})$, obtains output $\mathbf{v}_0 = \mathbf{r}_{0,1}$ and $\mathbf{u} = \mathbf{r}_{0,0} \oplus \mathbf{r}_{0,1}$. The verifier, on input $(0^{\hat{\ell}}, \mathbf{r}'_{0,1} = \mathbf{r}_{0,1 \oplus \Delta})$, outputs $\mathbf{q}_0 = \mathbf{r}_{0,1 \oplus \Delta}$, which equals $\mathbf{v}_0 \oplus \Delta \cdot \mathbf{u}$ as required. Suppose the algorithm is correct for inputs of length $N/2 = 2^{d-1}$. We first show that the $j = 0$ output is correct for inputs of length N. The prover's relevant output is $\mathbf{v}_0 = \bigoplus_{i=0}^{N/2-1} \mathbf{r}_{0,2i+1}$, the sum of

all the odd-indexed $\mathbf{r}_0$ values. On the verifier's side, if $\delta_0 = 0$ then $\mathbf{q}_0 = \mathbf{v}_0$. Otherwise, if $\delta_0 = 1$ then $\mathbf{q}_0$ is the sum of all the even-indexed $\mathbf{r}_0$ values, because for all $i \in [0..N/2)$ we have $\mathbf{r}'_{2i+1} = \mathbf{r}_{(2i+1)\oplus\Delta}$, and $(2i+1)\oplus\Delta$ is even. It follows that $\mathbf{q}_0 \oplus \mathbf{v}_0 = \bigoplus_{i=0}^{N-1} \mathbf{r}_i = \delta_j \cdot \mathbf{u}$.

When $j = 1$, we can see the remaining iterations as recursively running the same algorithm again, but on the set of $N/2$ inputs $\mathbf{r}_{1,0}, \ldots, \mathbf{r}_{1,N/2-1}$ by the prover and $\mathbf{r}'_{1,0}, \ldots, \mathbf{r}'_{1,N/2-1}$ by the verifier. Let $\Delta' = \sum_{j=1}^{d-1} 2^{j-1}\delta_j$, so $\Delta = 2\Delta' + \Delta_0$. For $i > 0$, it holds that

$$\begin{aligned}\mathbf{r}'_{1,i} = \mathbf{r}_{0,2i\oplus\Delta} \oplus \mathbf{r}_{0,(2i+1)\oplus\Delta} &= \mathbf{r}_{0,2i\oplus\Delta} \oplus \mathbf{r}_{0,2i\oplus\Delta\oplus 1} \\ &= \mathbf{r}_{0,2(i\oplus\Delta')} \oplus \mathbf{r}_{0,2(i\oplus\Delta')\oplus 1} = \mathbf{r}_{1,i\oplus\Delta'}\end{aligned}$$

Therefore, in the recursive step, the verifier's inputs for $i > 0$ are a permutation of the prover's, according to $i \mapsto i \oplus \Delta'$. By the induction hypothesis, it follows that the final outputs $\mathbf{v}_j, \mathbf{q}_j$, for $j = 1, \ldots, d-1$, satisfy $\mathbf{q}_j = \mathbf{v}_j \oplus \delta_j \cdot \mathbf{u}$. □

5.5 VOLEitH: Commitment and Reconstruction

The main FAEST algorithms use the vector commitments and ConvertToVOLE procedure to commit the signer to a batch of τ VOLE instances of length $\hat{\ell}$. These are later verified, when the prover is challenged to open all-but-one of each set of vector commitment messages, allowing the verifier to reconstruct its VOLE output.

We describe this with the following three algorithms.

5.5.1 Challenge Decomposition. Let chall be a challenge bit string of length λ and $0 \le i < \tau_0 + \tau_1$ be an integer, the algorithms $\mathsf{DecodeChall}_3$ and $\mathsf{DecodeAllChall}_3$(Figure 5.7) work together to generate the challenges for the τ VOLE instance from chall. The parameters τ, τ_1, τ_0, k and w_{grind} define precisely how chall is split into different challenges. More precisely, for every $i \in [0..\tau-1]$, $\mathsf{DecodeAllChall}_3$ compiles the full list of sub-challenges $\Delta_0, \ldots, \Delta_{\tau-1}$ and returns them. Each Δ_i is thus derived by a distinct chunk of chall, mapped to either k-bit or $(k-1)$-bit blocks according to whether $i < \tau_1$ or $i \ge \tau_1$.

When $\mathsf{DecodeAllChall}_3$ is called, it begins by iterating over all $i \in [0..\tau-1]$. For each i, it calls $\mathsf{DecodeChall}_3(\mathsf{chall}, i; \mathtt{param})$. That function first checks whether i is within the valid range. If this is not the case, the function aborts; otherwise, $\mathsf{DecodeChall}_3$ treats chall as a bitstring of length $\lambda - w_{\mathsf{grind}}$. Then the function branches on whether $i < \tau_1$. In this way, each challenge sub-block can have either length k or $k-1$, depending on which partition of chall it falls into.

Once the appropriate substring indices are chosen, $\mathsf{DecodeChall}_3$ extracts the corresponding slice of chall and returns it. The $\mathsf{DecodeAllChall}_3$ procedure then takes that returned bitstring, interprets it as an integer via NumRec, and stores it as Δ_i.

5.5.2 VOLE Commitment. The algorithms VOLECommit and VOLEReconstruct use the sub-procedures from our batch all-but-one vector commitment to turn committed values into the sender values of VOLE correlations as well as to create the receiver parts of VOLE correlations from commitment openings, respectively.

The algorithm VOLECommit() takes as input a λ-bit seed r, a 128-bit initialization vector iv, and a parameter $\hat{\ell}$. It outputs a commitment com, a decommitment decom, a set of "correction" vectors $\{\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}\}$, the secret vector $\boldsymbol{u}$ for the VOLE correlation, and a matrix $\mathbf{V}$ whose columns are the vectors $\boldsymbol{v}$ of the τ VOLEs.

First, the algorithm invokes $\mathsf{BAVC.Commit}(r, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$ to perform a single batch all-but-one vector commitment over all τ VOLE vectors. This call creates one large GGM tree and returns a global commitment com, a decommitment structure decom with all internal and leaf values, and the per-leaf seeds $\mathsf{sd}_{i,j}$ for each vector $i \in [0..\tau-1]$).

$\mathsf{DecodeChall}_3(\mathsf{chall}, i; \mathtt{param})$

1 : If $i \notin [0 \dots \tau_0 + \tau_1)$ **then abort**
2 : Parse $\mathsf{chall} \in \{0,1\}^{\lambda - w_{\mathsf{grind}}}$
3 : **if** $i < \tau_1$ **then**
4 : $\quad \mathtt{lo} := i \cdot k$
5 : $\quad \mathtt{hi} := (i+1) \cdot k - 1$
6 : **else**
7 : $\quad t := i - \tau_1$
8 : $\quad \mathtt{lo} := \tau_1 k + t(k-1)$
9 : $\quad \mathtt{hi} := \tau_1 k + (t+1)(k-1) - 1$
10 : **return** $\mathsf{chall}[\mathtt{lo} \dots \mathtt{hi}]$

$\mathsf{DecodeAllChall}_3(\mathsf{chall}; \mathtt{param})$

1 : **for** $i \in [0..\tau)$ **do**
2 : $\quad \Delta_i \leftarrow \mathsf{NumRec}(\mathsf{DecodeChall}_3(\mathsf{chall}, i; \mathtt{param}))$
3 : **return** $(\Delta_0, \dots, \Delta_{\tau-1})$

Fig. 5.7: Challenge decoding algorithm

Next, the function loops over each vector i. For each such vector, it calls the function $\mathsf{ConvertToVOLE}(\mathsf{sd}_{i,0}, \dots, \mathsf{sd}_{i,N_i-1}, \mathsf{iv}, i + 2^{31}; \hat{\ell})$. This function merges the seeds $\mathsf{sd}_{i,j}$ into one $\hat{\ell}$-bit secret $\boldsymbol{u}_i$ and a set of column vectors $\mathbf{v}_{i,0}, \dots, \mathbf{v}_{i,k_i-1}$. The column vectors are placed in a matrix $\mathbf{V}_i \in \mathbb{F}_2^{\hat{\ell} \times k_i}$, where each column is $\hat{\ell}$ bits. After doing this for all τ vectors, the procedure concatenates these sub-matrices (plus any zero-padding) into a global matrix $\mathbf{V} \in \mathbb{F}_2^{\hat{\ell} \times \lambda}$. This generates $\tau \cdot \hat{\ell}$ VOLE correlations with different secrets $\boldsymbol{u}_i$. To correct the sender secret to the same secret vector, the algorithm designates $\boldsymbol{u}_0$ from the first vector as the secret $\boldsymbol{u}$, and, for each $i \geq 1$, it computes $\boldsymbol{c}_i = \boldsymbol{u}_0 \oplus \boldsymbol{u}_i$. These "correction" vectors allow the other VOLE instances to share the same underlying secret $\boldsymbol{u}$. Finally, the function returns com, the decommitment decom, all correction vectors $\{\boldsymbol{c}_i\}$, the secret $\boldsymbol{u}$, and the large matrix $\mathbf{V}$.

$\mathsf{FAEST.VOLECommit}(r, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$

Input: $r \in \{0,1\}^\lambda, \mathsf{iv} \in \{0,1\}^{128}$
Output: $\mathsf{com} \in \{0,1\}^{2\lambda}, \mathsf{decom} \in \{0,1\}^{n_{\mathsf{leafcom}}\lambda\tau + T_{\mathsf{open}}\lambda}, \mathbf{c}_i, \mathbf{u} \in \{0,1\}^{\hat{\ell}}, \mathbf{V} \in \{0,1\}^{\hat{\ell} \times \lambda}$

1 : $(\mathsf{com}, \mathsf{decom}, (\mathsf{sd}_{i,0}, \dots, \mathsf{sd}_{i,N_i-1})_{i \in [0,\tau)}) \leftarrow \mathsf{BAVC.Commit}(r, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$
2 : $\tau_0 := \lambda \bmod \tau$
3 : **for** $i \in [0..\tau)$ **do**
4 : $\quad (\boldsymbol{u}_i, \mathbf{v}_{i,0}, \dots, \mathbf{v}_{i,k_b-1}) := \mathsf{ConvertToVOLE}(\mathsf{sd}_{i,0}, \dots, \mathsf{sd}_{i,N_i-1}, \mathsf{iv}, i + 2^{31}; \hat{\ell})$
5 : $\quad \mathbf{V}_i := [\mathbf{v}_{i,0} \cdots \mathbf{v}_{i,k_i-1}] \in \mathbb{F}_2^{\hat{\ell} \times k_i}$ // stored in column major representation
6 : $\mathbf{V} := [\mathbf{V}_0 \cdots \mathbf{V}_{\tau-1}\ \mathbf{0}_{\hat{\ell},w}] \in \mathbb{F}_2^{\hat{\ell} \times \lambda}$ // $\mathbf{0}$ is $(\hat{\ell} \times w)$ bits of zero-padding
7 : $\boldsymbol{u} := \boldsymbol{u}_0$
8 : **for** $i \in [1 \dots \tau)$ **do**
9 : $\quad \boldsymbol{c}_i := \boldsymbol{u} \oplus \boldsymbol{u}_i$
10 : **return** $(\mathsf{com}, \mathsf{decom}, \mathbf{c}_1, \dots, \mathbf{c}_{\tau-1}, \boldsymbol{u}, \mathbf{V})$

Fig. 5.8: FAEST VOLE commitments

5.5.3 VOLE Reconstruction Once the signer has committed to its execution of the VOLEitH protocol with VOLECommit, the verifier can use the commitments, together with

the challenge that generates the Δ_i keys, to reconstruct the matching VOLE keys using VOLEReconstruct.

FAEST.VOLEReconstruct($\mathsf{chall}_3, \mathsf{decom}_I, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}}$)

INPUT: $\mathsf{chall}_3 \in \{0,1\}^\lambda, \mathsf{decom}_I \in \{0,1\}^{n_{\mathsf{leafcom}}\lambda\tau + T_{\mathsf{open}}\lambda}, \mathbf{c}_i \in \{0,1\}^\lambda, \mathsf{iv} \in \{0,1\}^{128}$

OUTPUT: $\mathsf{com} \in \{0,1\}^\lambda, \mathbf{Q} \in \{0,1\}^{\hat{\ell}\times\lambda}$

$I = (\Delta_0, \ldots, \Delta_{\tau-1}) \leftarrow \mathsf{DecodeAllChall}_3(\mathsf{chall}_3)$

if $I = \bot$ **then return** $\bot$

else do

  $\mathsf{rec} := (\mathsf{com}, (\mathsf{sd}_{i,0}, \ldots, \mathsf{sd}_{i,N_i-1})_{i\in[0..\tau)}) \leftarrow \mathsf{BAVC.Reconstruct}(\mathsf{decom}_I, I, \mathsf{iv}; \mathtt{param})$

  if $\mathsf{rec} = \bot$ **then return** $\bot$

  else

    for $i \in [0 \ldots \tau)$ **do**

      for $j \in [1 \ldots N_i)$ **do**

        $\mathsf{sd}'_{i,j} \leftarrow \mathsf{sd}_{i,j\oplus\Delta_i}$ // XOR j and Δ_i as bit-strings

      $(\boldsymbol{u}'_i, \mathbf{q}_{i,0}, \ldots, \mathbf{q}_{i,k_i-1}) := \mathsf{ConvertToVOLE}(\bot, \mathsf{sd}'_{i,1}, \ldots, \mathsf{sd}'_{i,N_i-1}, \mathsf{iv}, i + 2^{31}; \hat{\ell})$

      $(\delta_{i,0}, \ldots, \delta_{i,k_i}) \leftarrow \mathsf{BitDec}(\Delta_i, k_i)$

      if $i = 0$ **then**

        $\mathbf{Q}_i := [\mathbf{q}_{i,0} \cdots \mathbf{q}_{i,k_i-1}] \in \mathbb{F}_2^{\hat{\ell}\times k_i}$ // stored in column major representation

      else

        $\mathbf{Q}_i \leftarrow [\mathbf{q}_{i,0} \cdots \mathbf{q}_{i,k_i-1}] + [\delta_{i,0} \cdot \mathbf{c}_i \cdots \delta_{i,k_i-1} \cdot \mathbf{c}_i] \in \mathbb{F}_2^{\hat{\ell}\times k_i}$

    $\mathbf{Q} := [\mathbf{Q}_0 \cdots \mathbf{Q}_{\tau-1}\ \mathbf{0}_{\hat{\ell},w}] \in \mathbb{F}_2^{\hat{\ell}\times\lambda}$ // $\mathbf{0}$ is $(\hat{\ell} \times w)$ bits of zero-padding

    return $(\mathsf{com}, \mathbf{Q})$

Fig. 5.9: FAEST VOLE reconstruction

More concretely, the algorithm VOLEReconstruct takes, as input, a λ-bit challenge chall_3, a partial decommitment decom_I from the all-but-one vector commitment, a set of correction vectors $\{\mathbf{c}_i\}$, a 128-bit iv, and a VOLE length $\hat{\ell}$. It returns both the commitment com (verifying consistency with the committed values) and a matrix $\mathbf{Q}$ that encodes the receiver's VOLE messages.

First, the algorithm decodes chall_3 into $\Delta_0, \ldots, \Delta_{\tau-1}$ by calling

$$I \;=\; (\Delta_0, \ldots, \Delta_{\tau-1}) \;\leftarrow\; \mathsf{DecodeAllChall}_3(\mathsf{chall}_3).$$

Each Δ_i indicates which leaf in the i-th vector remains hidden. Next, it partially reconstructs the batch vector commitment by invoking

$$\big(\mathsf{com},\ (\mathsf{sd}_{i,0}, \ldots, \mathsf{sd}_{i,N_i-1})_{i\in[0..\tau)}\big) \;\leftarrow\; \mathsf{BAVC.Reconstruct}(\mathsf{decom}_I,\ I,\ \mathsf{iv};\ \mathtt{param}).$$

This call returns commitment com (matching the one from VOLECommit) and, for every i, all leaf seeds except for the one at index Δ_i. The algorithm then iterates over each vector i. Within each vector, it applies ConvertToVOLE on input seeds $\mathsf{sd}'_{i,j} = \mathsf{sd}_{i,j\oplus\Delta_i}$, returning $\big(\boldsymbol{u}'_i,\ \mathbf{q}_{i,0}, \ldots, \mathbf{q}_{i,k_i-1}\big)$ (Proposition 5.2). Additionally, it interprets Δ_i as bits $(\delta_{i,0}, \ldots, \delta_{i,k_i-1})$. If $i = 0$, the resulting columns $\mathbf{q}_{i,j}$ are stored directly in $\mathbf{Q}_0$. If $i \geq 1$, each column $\mathbf{q}_{i,j}$ is "shifted" by $\delta_{i,j} \cdot \mathbf{c}_i$ to make it consistent with the main VOLE secret $\boldsymbol{u}_0$ from VOLECommit. Finally, it concatenates $\mathbf{Q}_i$ for all i (possibly with a zero-padding block $\mathbf{0}_{\hat{\ell},w}$) into a single matrix $\mathbf{Q} \in \mathbb{F}_2^{\hat{\ell}\times\lambda}$. At the end, the function outputs the recovered commitment com and the matrix $\mathbf{Q}$, which together allow the verifier to validate that

the partial opening and VOLE messages correspond to the original batch commitment generated by FAEST.VOLECommit.

6 Zero-Knowledge Constraints for AES and Rijndael

This section specifies how to compute the QuickSilver proof of knowledge of the OWF for the FAEST. Before jumping into the details, we first introduce the VOLE commitment notation and its operations and then give a high level overview of how an AES-128 encryption under a secret key $k \in \{0,1\}^{128}$ can be performed efficiently using the VOLE commitment scheme.

6.1 VOLE-ZK Operations

We use the notation $\langle x \rangle_e$ to denote that a value $x \in \mathbb{F}_{2^e}$, where e is 2 or λ, has been committed through VOLE. This means the signer knows x and a value $x_0 \in \mathbb{F}_{2^\lambda}$, and the verifier knows $\gamma_x := x_0 + x \cdot \Delta$, where $\Delta \in \mathbb{F}_{2^\lambda}$ is the VOLE-ZK challenge.

We generalize this to degree-d commitments, as follows.

Definition 6.1 (VOLE Commitment). *Let $x \in \mathbb{F}_{2^e}$ be known to $\mathcal{P}$. We write $\langle x \rangle_e^d$ to represent a* degree-d, VOLE commitment *to x, meaning that $\mathcal{P}$ knows the coefficients of a commitment polynomial $\rho_x(X) = \rho_0 + \rho_1 \cdot X + \cdots + \rho_d \cdot X^d$, where $\rho_d = x$ and $\rho_0, \ldots, \rho_{d-1} \in \mathbb{F}_{2^\lambda}$, and $\mathcal{V}$ holds the evaluation $\gamma_x = \rho_x(\Delta)$, for a key $\Delta \in \mathbb{F}_{2^\lambda}$.*

We need to perform various operations on VOLE-committed values, detailed in Figure 6.1.

Remark 6.2.

- When computing constraints, we usually end up with some commitment $\langle z \rangle^d$ such that z is supposed to be zero. In these cases, the signer does not need to store the full polynomial, since the highest coefficient (z) is known to be zero. This can also save computation, since when computing $\langle z \rangle^d$, the signer can skip computation of the degree-d coefficient.
- By convention, if $[d]$ is omitted, we assume $d = 1$. Similarly, if $[e]$ is omitted, we assume $e = 2$.

Signer: VOLE-ZK operations	**Verifier:** VOLE-ZK operations
// Represent $\langle x \rangle_e^d$ as $(x_0, \ldots, x_{d-1}, x) \in \mathbb{F}_{2^\lambda}^d \times \mathbb{F}_{2^e}$,	// Represent $\langle x \rangle_e^d$ as $\gamma_x \in \mathbb{F}_{2^\lambda}$
// coefficients of $\rho_x(X) = x_0 + x_1 X + \cdots + xX^d$	// Verifier also holds global key $\Delta \in \mathbb{F}_{2^\lambda}$
ADD: $\langle x \rangle_{e_1}^{d_1} + \langle y \rangle_{e_2}^{d_2} \to \langle z \rangle_e^d$	ADD: $\langle x \rangle^{d_1} + \langle y \rangle^{d_2} \to \langle z \rangle^d$
(where $d = \max(d_1, d_2), e = \max(e_1, e_2)$)	
1 : $\rho_z(X) \leftarrow X^{d-d_1} \cdot \rho_x(X) + X^{d-d_2} \cdot \rho_y(X)$	
2 : **return** $(z_0, \ldots, z_d)$ (coeffs of ρ_z)	**return** $\Delta^{d-d_1} \cdot \gamma_x + \Delta^{d-d_2} \cdot \gamma_y$
ADD CONSTANT: $\langle x \rangle_e^d + c \to \langle z \rangle_e^d$	ADD CONSTANT: $\langle x \rangle_e^d + c \to \langle z \rangle_e^d$
3 : **return** $\rho_z(X) \leftarrow \rho_x(X) + X^d \cdot c$	**return** $\gamma_z = \gamma_x + \Delta^d \cdot c$
MULTIPLY: $\langle x \rangle_{e_1}^{d_1} \cdot \langle y \rangle_{e_2}^{d_2} \to \langle z \rangle_e^d$	MULTIPLY: $\langle x \rangle^{d_1} \cdot \langle y \rangle^{d_2} \to \langle z \rangle^d$
(where $d = d_1 + d_2, e = \max(e_1, e_2)$)	
4 : $\rho_z(X) \leftarrow \rho_x(X)\rho_y(X)$	
5 : **return** $(xy, z_1, \ldots, z_{d_1+d_2})$ (coeffs of ρ_z)	**return** $\gamma_x \cdot \gamma_y$

Fig. 6.1: VOLE-ZK operations. The extension degrees e, e_1, e_2 may be any combination in $\{2, \lambda\}$. The output degree d is always 1, 2, or 3.

Lemma 6.3. *Let $\langle x \rangle_e^d$ be a VOLE commitment. Assume that $\mathcal{P}$ sends $\rho' \in \mathbb{F}_{2^\lambda}$ to $\mathcal{V}$ such that $\deg(\rho') \leq d$ but $\rho_d \neq x$. Let $\mathcal{V}$ accept if $\rho'(\Delta) = \gamma_x$, and otherwise reject. Then $\mathcal{V}$ accepts with probability at most $d/2^\lambda$.*

Proof. By definition, we have that $\gamma_x = \rho(\Delta)$. Since $\rho' \neq \rho$ as $x' \neq x$. As $\rho'(X) = \rho(X)$ for at most d choices of $X \in \mathbb{F}_{2^\lambda}$ (as otherwise $\rho' = \rho$ by the fundamental theorem of algebra) and since Δ is chosen uniformly at random and independent of ρ', the claim follows. □

6.2 An overview of the OWF evaluation using VOLE commitments

An overview of the OWF evaluation is also given in Figure 6.2. At the outset, we assume that the prover has committed to the individual bits of k as $\langle k_1 \rangle_1^1, \ldots, \langle k_{128} \rangle_1^1$.

Computing the key expansion. Towards computing KeyExpansion on the aforementioned commitments to obtain commitments $\langle \bar{k}_1 \rangle_1^1, \ldots, \langle \bar{k}_{\lambda(R+1)} \rangle_1^1$ to $\bar{\mathbf{k}}$, all operations besides SubWord can be performed directly on either $\langle k_1 \rangle_1^1, \ldots, \langle k_{128} \rangle_1^1$ or on commitments to outputs of SubWord without increasing the degree. Therefore, for every input a to SubWord in KeyExpansion, the prover will commit to $b = \mathsf{SubWord}(a)$ as the bits $\langle b_1 \rangle_1^1, \ldots, \langle b_{32} \rangle_1^1$. This, as mentioned above, then allows to obtain degree-1 commitments of all bits of $\bar{\mathbf{k}}$ such that the commitments each are of degree $d = 1$. To check that the commitments $\langle b_1 \rangle_1^1, \ldots, \langle b_{32} \rangle_1^1$ to the outputs are actually consistent with the inputs a for SubWord (or rather, with the input commitments $\langle a_1 \rangle_1^1, \ldots, \langle a_{32} \rangle_1^1$), we perform the following operations:

1. Apply the inverse of step 2 of SubBytes on each byte of b, leading to commitments $\langle \bar{b}_1 \rangle_1^1, \ldots, \langle \bar{b}_{32} \rangle_1^1$. Note that the computation is only linear.
2. Let $\alpha \in \mathbb{F}_{2^8}$ be the element generating $\mathbb{F}_{2^8}$ from $\mathbb{F}_2$. For each byte, do the following as exemplified for the first byte of a, b: set $\langle \hat{a} \rangle_8^1 = \sum_{i=1}^8 \langle a_i \rangle_1^1 \cdot \alpha^{i-1}$ and $\langle \hat{b} \rangle_8^1 = \sum_{i=1}^8 \langle \bar{b}_i \rangle_8^1$. Then check that the first step of the AES S-box holds.

To verify the first step, we verify that $\langle \hat{b}^2 \rangle_8^1 \cdot \langle \hat{a} \rangle_8^1 - \langle \hat{b} \rangle_8^1$ as well as $\langle \hat{a}^2 \rangle_8^1 \cdot \langle \hat{b} \rangle_8^1 - \langle \hat{a} \rangle_8^1$ are commitments to 0. This is sound, due to the following:

Proposition 6.4. *Let $\mathbb{F}$ be a field and $x, y \in \mathbb{F}$. If $x^2 \cdot y = x$ and $x \cdot y^2 = y$, then*

- *either $x \neq 0$ and $y = x^{-1}$; or*
- *$x = y = 0$.*

Proof. Assume that $x \neq 0$. Dividing both sides of $x^2 \cdot y = x$ by x shows that $y = x^{-1}$. Assume that $x = 0$. By the second equation, we then have that $x \cdot y^2 = 0 = y$. □

Note that, in the process, this only generates commitments of degree 2: computing $\hat{a}^2$ from $\hat{a}$ is a $\mathbb{F}_2$-linear operation - by computing the linear operation on the $\langle a_i \rangle_1^1$-commitments before lifting to $\mathbb{F}_{2^8}$ we can compute the square without a multiplication. This observation also holds for $\hat{b}^2$.

Evaluating the round functions of AES. Both the input block $x \in \{0,1\}^{128}$ and the output block $y \in \{0,1\}^{128}$ of the AES-128 instance that we compute on the commitments are defined by the verification key. Moreover, consistent commitments (of degree 1) to the bits of the round key for each round are available as described above. We therefore evaluate Encrypt on input $\mathbf{in} = x$ with commitments $\langle \bar{k}_1 \rangle_1^1, \ldots, \langle \bar{k}_{\lambda(R+1)} \rangle_1^1$ to the round keys, using the homomorphism of the commitment scheme, and revealing that the commitment to **out** as being identical to y.

Just like KeyExpansion, all the operations in the AES round function are $\mathbb{F}_2$-linear, except for the inversion in the S-box. Therefore, one can use the same approach as for evaluating KeyExpansion, i.e., by committing to the output bits of SubBytes and checking

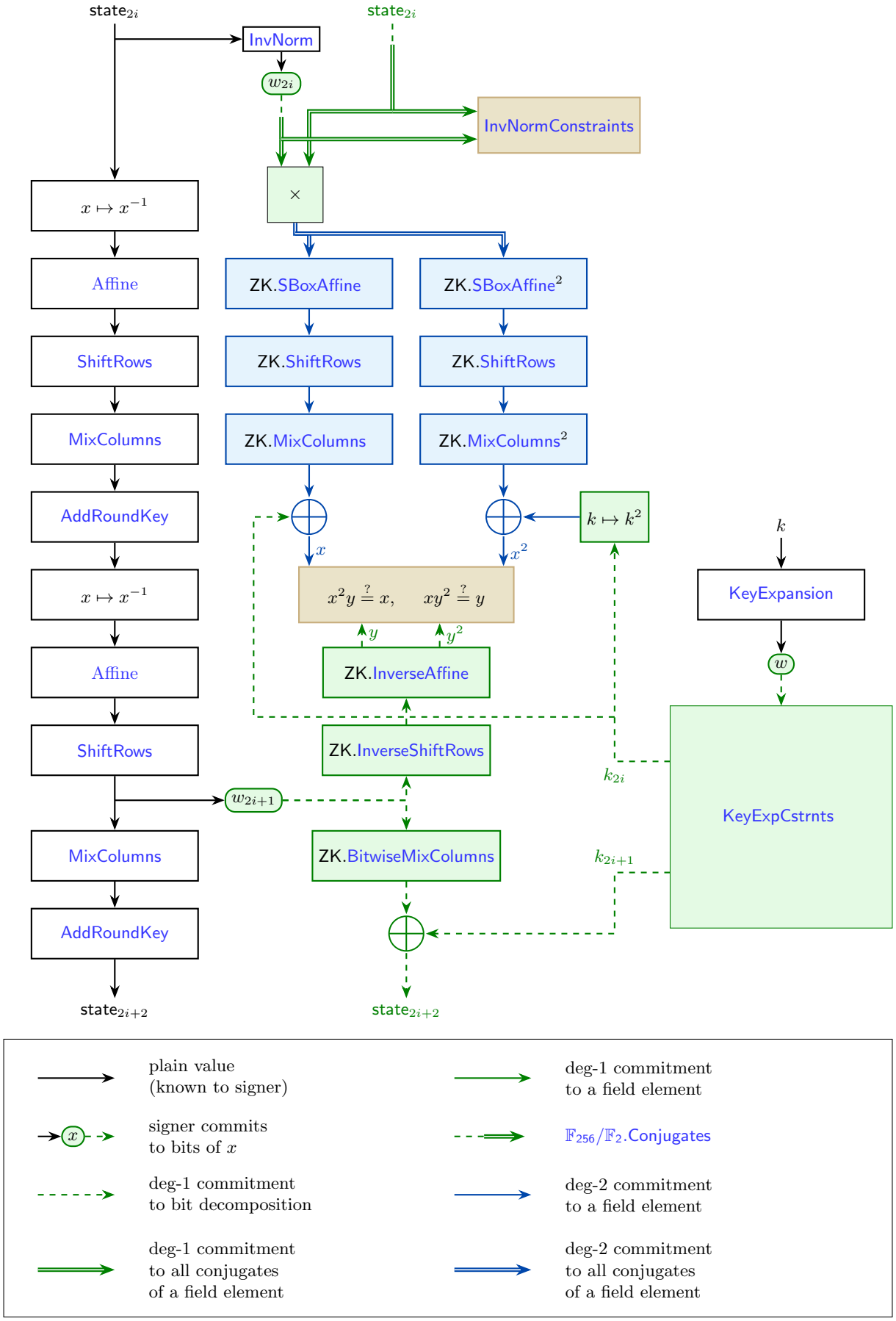

Fig. 6.2: A diagrammatic overview of the QuickSilver proof of knowledge of two consecutive AES rounds in FAEST. The FAEST-EM version is similar, except that the AES key is public, so the key schedule is evaluated in the clear, rather than on degree-1 VOLE commitments.

consistency using relations of the form $x^2 \cdot y = x$ and $x \cdot y^2 = y$. This approach was used in the Round-1 version of FAEST and requires to generate $(R-1) \cdot 128$ bit-commitments to commit to the outputs of each SubBytes-instance except the last SubBytes instances[15].

Using field norm to optimize inversions. In the round-2 version, we use the norm $N_{\mathbb{F}_{2^8}/\mathbb{F}_{2^4}}$ to evaluate each S-Box more efficiently. We treat odd and even rounds differently, where we will commit to the full state in even rounds, and commit only to $\lambda/2$ bits in odd rounds, which results in only having to commit to approximately $(R-1) \cdot 96$ bits. In a nutshell the idea is that in even rounds, we do not commit to the full output of the inversion a^{-1} in the S-box, but we only commit to its norm $c = N_{\mathbb{F}_{2^8}/\mathbb{F}_{2^4}}(a^{-1}) = a^{-17}$, which is an element of $\mathbb{F}_{16}$ and therefore only needs half as many bits to represent. Because $a^{-1} = a^{16} \cdot a^{-17}$ we can obtain a commitment to the inverse a^{-1} by first computing a^{16} (whose bits are $\mathbb{F}_2$-linear combinations of the bits of a), and then multiplying it by the inverse norm $N_{\mathbb{F}_{2^8}/\mathbb{F}_{2^4}}(a^{-1})$.

Computing on Galois Conjugates. A slight complication is that this trick only lets us compute a commitment to the field element a^{-1}, not a commitment to the 8 bits of a^{-1}. This only allows us to perform $\mathbb{F}_{256}$-linear operations on the VOLE commitments, which is a problem, because the affine part of the S-box is $\mathbb{F}_2$-linear but not $\mathbb{F}_{256}$-linear. To work around this, we compute commitments to all the Galois conjugates of a^{-1}, i.e., commitments to $a^{-1}, a^{-2^1}, \ldots, a^{-2^7}$. This is done by doing bit operations to compute commitments to all the conjugates of a and $c = N_{\mathbb{F}_{2^8}/\mathbb{F}_{2^4}}(a^{-1})$, and then doing 8 multiplications to get commitments to all the conjugates of a^{-1}, i.e., using $a^{-1} = a^{16} \cdot c$, $a^{-2} = a^{32} \cdot c^2$, etc. Since any $\mathbb{F}_2$-linear function can be expressed as a $\mathbb{F}_{256}$-linear combination of Galois conjugates (Remark 4.3), this allows us to compute the affine part of the S-box.

Checks. Finally, we need to verify the consistency of the committed bits. For even rounds, we need to check for every inversion that the provided c is indeed equal to $N_{\mathbb{F}_{2^8}/\mathbb{F}_{2^4}}(a^{-1})$, where a is the input to the S-box, which is computed from the commitment to the state in the previous round. This is done by checking that $c \cdot a^2 \cdot a^{16} = a$, which is a degree-3 constraint. For odd rounds, we need to check that the inputs and outputs of the inversions in the S-box are consistent. For an inversion with input x, and output y this is done by checking that $x^2 \cdot y = x$ and $x \cdot y^2 = y$. Note that, since we have degree-1 commitments to the bits of y, we can derive a degree-1 commitment for y^2 by doing bit operations on the commitments. However, since we only have degree-2 commitments to x as field elements, the naive approach to compute x^2 as $x \cdot x$ would result in a degree-4 commitment to x^2, which makes $x^2 y = x$ a degree-5 constraint. To avoid this, we compute x^2 directly, by applying the appropriate operations to the conjugates of the output of the previous layer of inversions. This essentially means we compute SBoxAffine, ShiftRows, MixColumns, and AddRoundKey twice, once normally to compute x, and once starting from the squared state and doing modified versions of the AES operations where all the constants (including the round's subkey) are replaced by their squares, to compute the degree-2 commitment to x^2 (see Figure 6.2). With this approach we can compute degree-3 commitments to $x^2 y - x$ and $xy^2 - y$, which are later checked to be commitments to zero.

Now we proceed with a more formal specification of the AES proof. The principal FAEST algorithms use the following building blocks to perform AES and Rijndael related operations to compute the QuickSilver proof of knowledge of the OWF for the FAEST-λ parameter sets.

– Section 6.5: FAEST.ExtendWitness in Figure 6.15

[15] Their output bits can be derived from the bits of y after reversing the linear operations and by subtracting the last round key.

ZK.ByteCombine($\langle x_0\rangle_2, \ldots, \langle x_7\rangle_2; \lambda$)

INPUT: Committed bits $x_i \in \mathbb{F}_2$

OUTPUT: Committed $x \in \mathbb{F}_{2^\lambda}$, containing embedded $\mathbb{F}_{2^8}$ element

1 : **let** $\alpha_8 \in \mathbb{F}_{2^\lambda}$ // Basis element of $\mathbb{F}_{2^8}$ within $\mathbb{F}_{2^\lambda}$

2 : **return** $\langle x\rangle_\lambda = \sum_{i=0}^{7} \langle x_i\rangle_2 \cdot \alpha_8^i$

Fig. 6.3: Lifting 8 committed bits to a single commitment

StateToBytes($\langle$state$\rangle$)

INPUT: $\langle$state$\rangle$ is an array of $N_{\text{st-bits}}$ commitments to bits

OUTPUT: $N_{\text{st-bytes}}$-byte array $\langle$**out**$\rangle$

1 :

2 : **new** $\langle$**out**$\rangle_\lambda$:= new array of $N_{\text{st-bytes}}$ committed bytes

3 : **for** $i \in [0..N_{\text{st-bytes}} - 1)$ **do**

4 : // extract a chunk of 8 bits

5 : $\langle bt\rangle_\lambda$:= ZK.ByteCombine($\langle$state$[8 \cdot i..8 \cdot i + 7]\rangle$)

6 : $\langle$**out**$\rangle_\lambda := \langle$**out**$\|bt\rangle_\lambda$

7 : **return** $\langle$**out**$\rangle_\lambda$

Fig. 6.4: Converting the AES state matrix to an $N_{\text{st-bytes}}$-byte array in column-major order.

InvNorm(**x**)

INPUT: $\mathbf{x} \in \{0,1\}^8$

OUTPUT: $\mathbf{y} \in \{0,1\}^4$

1 : $x \leftarrow$ ToField($\mathbf{x}, 8$)

2 : **if** $x = 0$ **then**

3 : **return** $(0,0,0,0)$

4 : $\mathbf{y}' \leftarrow$ ToBits($1/x^{17}$)

5 : $\mathbf{y} \leftarrow (\mathbf{y}'[0], \mathbf{y}'[6], \mathbf{y}'[7], \mathbf{y}'[2])$

6 : **return** $\mathbf{y}$

Fig. 6.5: Inverse $\mathbb{F}_{2^4}$ norm function

– Section 6.6: FAEST.KeyExpFwd in Figure 6.16.
– Section 6.6: FAEST.KeyExpBkwd in Figure 6.17.
– Section 6.6: FAEST.KeyExpCstrnts in Figure 6.18.
– Section 6.7: FAEST.EncCstrnts in Figure 6.20.

Before introducing these procedures, we describe some more elementary building blocks.

6.3 Building Blocks for AES in VOLE-ZK

This section introduces some basic functions used to implement the VOLE-based zero-knowledge proof for AES operations.

– ZK.ByteCombine (Figure 6.3): It combines vectors of committed 8 bits into one committed field element of $\mathbb{F}_{2^\lambda}$, which embeds an $\mathbb{F}_{2^8}$ element consistent with the AES polynomial representation.
– StateToBytes (Figure 6.4): The function converts a committed state array given as an array of $N_{\text{st-bits}}$ to a committed byte array **out**.

InvNormToConjugates($\langle \mathbf{x} \rangle = (\langle x_0 \rangle, \ldots, \langle x_3 \rangle)$)

INPUT: $\langle \mathbf{x} \rangle$, where $x_i \in \mathbb{F}_2$

OUTPUT: $\langle \boldsymbol{y} \rangle_\lambda$, where $\boldsymbol{y} \in \mathbb{F}_{2^\lambda}^4$ and $y_j = y_0^{2^j}$ for $j > 0$

1 : **let** $\alpha_8 \in \mathbb{F}_{2^\lambda}$ // Defines basis for $\mathbb{F}_{2^8}$ within $\mathbb{F}_{2^\lambda}$

2 : // $\mathbb{F}_{2^4}$ basis defined by $\beta_4 = \alpha_8^6 + \alpha_8^4$

3 : **for** $j \in [0..4)$ **do**

4 : $\quad \langle y_j \rangle_\lambda \leftarrow (1, \beta_4^{2^j}, \beta_4^{2^{j+1}}, \beta_4^{3 \cdot 2^j}) \cdot \langle \mathbf{x} \rangle_2$ // inner product

5 : **return** $(\langle y_0 \rangle_\lambda, \ldots, \langle y_3 \rangle_\lambda)$

Fig. 6.6: Lift the inverse norm to $\mathbb{F}_{2^8}$ (embedded in $\mathbb{F}_{2^\lambda}$) and compute all its $\mathbb{F}_2$ conjugates

- InvNorm (Figure 6.5): For each S-box with input $x \in \mathbb{F}_{2^8}$, viewed as an extension of $\mathbb{F}_{2^4}$, we compute the field norm $N_{\mathbb{F}_{2^8}/\mathbb{F}_{2^4}}(1/x) = 1/x^{17}$. The result is an element of $\mathbb{F}_{2^4}$, embedded into $\mathbb{F}_{2^8}$. We reverse the embedding by extracting 4 relevant bits of the result, and include this in the extended witness. The algorithm InvNorm defines these bits using the basis $\mathbf{b} = (1, \beta, \beta^2, \beta^3)$ for $\mathbb{F}_{2^4}$ over $\mathbb{F}_{2^8}$, where $\beta = \alpha^6 + \alpha^4$ and α is the basis element of $\mathbb{F}_{2^8}$. That is, given an output $\mathbf{y} \leftarrow \mathsf{InvNorm}(\mathbf{x}) \in \{0,1\}^4$, the norm of $1/x$ equals the inner product $\mathbf{b} \cdot \mathbf{y} \in \mathbb{F}_{2^8}$.
- InvNormToConjugates (Figure 6.6): This procedure lifts a committed 4-bit representation (output of InvNorm) of an $\mathbb{F}_{2^4}$ element to a commitment in $\mathbb{F}_{2^\lambda}$ and computes all its Frobenius conjugates (i.e., y, y^2, y^4, y^8).
- SquareBits, SquareAndCombine (Figure 6.7): It computes the square of VOLE commitments bit-by-bit within $\mathbb{F}_{2^8}$. It crucially uses the linearity of the (Frobenius) map in characteristic 2. SquareBits takes bits as input and returns bits, while SquareAndCombine lifts the bits to $\mathbb{F}_{2^8}$ before outputting the committed field element.
- $\mathbb{F}_{256}/\mathbb{F}_2$.Conjugates (Figure 6.7): the algorithm computes the conjugates of $\mathbb{F}_{2^8}$ field elements through repeated squaring operations. The algorithm takes as input commitments to state bits, and outputs a vector of commitments corresponding to the conjugates of each bytes in the state treated as $\mathbb{F}_{2^8}$ field elements.

6.4 Building Blocks for AES and Rijndael in VOLE-ZK

In this section, we describe the core AES/Rijndael operations in the context of a VOLE-based ZK proof. We focus on how each operation can be implemented on committed data with specific degrees, i.e., linear or quadratic.

6.4.1 S-Box Affine Component. The function ZK.SBoxAffine in Figure 6.8 applies the S-box's affine map to a committed byte (or set of bytes) in $\mathbb{F}_{2^8}$. More specifically, it takes as input:

- A set of degree-2 commitments $\langle \boldsymbol{x}_j \rangle$, each corresponding to $(\boldsymbol{x}_0)^{2^j}$ for $j = 0, \ldots, 7$. Here, $\boldsymbol{x}_0[i]$ is the i-th byte of the input, and $\boldsymbol{x}_j[i]$ is its 2^j-power (a conjugate in characteristic 2).
- A flag sq that indicates whether we apply the map in its usual form (sq = `false`) or we apply its *square* version (sq = `true`).

Internally, ZK.SBoxAffine linearly combines the conjugates $\boldsymbol{x}_0^{2^j}$ using constants $\zeta_0, \ldots, \zeta_7, \zeta_8$. If sq is set, each constant ζ_k is squared, and an index shift $t = 1$ is used. The result is an $\mathbb{F}_{2^8}$ value for each byte position i, returned as $\langle y \rangle$.

SquareBits$(\langle x_0\rangle^d, \ldots, \langle x_7\rangle^d)$

$\langle y_0\rangle^d \leftarrow \langle x_0\rangle^d + \langle x_4\rangle^d + \langle x_6\rangle^d$
$\langle y_1\rangle^d \leftarrow \langle x_4\rangle^d + \langle x_6\rangle^d + \langle x_7\rangle^d$
$\langle y_2\rangle^d \leftarrow \langle x_1\rangle^d + \langle x_5\rangle^d$
$\langle y_3\rangle^d \leftarrow \langle x_4\rangle^d + \langle x_5\rangle^d + \langle x_6\rangle^d + \langle x_7\rangle^d$
$\langle y_4\rangle^d \leftarrow \langle x_2\rangle^d + \langle x_4\rangle^d + \langle x_7\rangle^d$
$\langle y_5\rangle^d \leftarrow \langle x_5\rangle^d + \langle x_6\rangle^d$
$\langle y_6\rangle^d \leftarrow \langle x_3\rangle^d + \langle x_5\rangle^d$
$\langle y_7\rangle^d \leftarrow \langle x_6\rangle^d + \langle x_7\rangle^d$
return $(\langle y_0\rangle^d, \ldots, \langle y_7\rangle^d)$

SquareAndCombine$(\langle x_0\rangle, \ldots, \langle x_7\rangle)$

$\langle y_0\rangle, \ldots, \langle y_7\rangle \leftarrow$ SquareBits$(\langle x_0\rangle, \ldots, \langle x_7\rangle)$
return ZK.ByteCombine$(\langle y_0\rangle, \ldots, \langle y_7\rangle)$

$\mathbb{F}_{256}/\mathbb{F}_2$.Conjugates$(\langle s_0\rangle, \ldots, \langle s_{N_{\text{st-bits}}-1}\rangle)$

Input: $\langle s_i\rangle$: VOLE commitments to state bits
Output: $\langle \boldsymbol{y}_{i,j}\rangle_8$, for $i \in [0..N_{\text{st-bytes}}), j \in [0..8)$: commitments to the conjugates of each byte in the state

for $i \in [0..N_{\text{st-bytes}})$ **do**
  $\langle \mathbf{x}_0\rangle := (\langle s_{8i}\rangle, \ldots, \langle s_{8i+7}\rangle)$
  for $j \in [0..6]$ **do**
    // Compute conjugates x^{2^j} by repeated squaring in $\mathbb{F}_{2^8}$
    $\langle \boldsymbol{y}_{i,j}\rangle_8 \leftarrow$ ZK.ByteCombine$(\langle \mathbf{x}_j\rangle, \lambda)$
    $\langle \mathbf{x}_{j+1}\rangle \leftarrow$ SquareBits$(\langle \mathbf{x}_j\rangle)$
  $\langle \boldsymbol{y}_{i,7}\rangle_8 \leftarrow$ ZK.ByteCombine$(\langle \mathbf{x}_7\rangle, \lambda)$

return $(\langle \boldsymbol{y}_{0,0}\rangle_8, \ldots, \langle \boldsymbol{y}_{N_{\text{st-bytes}}-1,7}\rangle_8)$

Fig. 6.7: Bitwise squaring and conjugates of VOLE commitments in $\mathbb{F}_{2^8}$

Inverse Affine Map. The inverse affine map, Figure 6.9, processes the entire state by operating on each byte, where each byte is represented by eight degree-1 commitments to bits, $\langle x_{i,0}\rangle, \ldots, \langle x_{i,7}\rangle$. For each byte i and bit position $j \in \{0, \ldots, 7\}$, the inverse affine transformation is computed as

$$y_{i,j} = x_{i,(j-1) \bmod 8} \oplus x_{i,(j-3) \bmod 8} \oplus x_{i,(j-6) \bmod 8} \oplus c_j,$$

where c_j equals 1 if $j \in \{0, 2\}$ and 0 otherwise. In the implementation, the constant bit c_j is converted into a VOLE commitment using the function ConstantToVOLE(c_j), which maps the Boolean constant c_j to its corresponding commitment.

6.4.2 MixColumns, ShiftRows and AddRoundKey We describe here the remaining AES/Rijndael operations.

- ZK.MixColumns (Figure 6.10): This procedure implements the MixColumns transformation on committed state bytes. It takes as input a sequence of degree-2 commitments to bytes (elements of $\mathbb{F}_{2^8}$) and, depending on the flag sq, optionally squares the mixing constants. The output is a new sequence of degree-2 commitments obtained by mixing each column according to the MixColumns linear transformation.

- ZK.BitwiseMixColumns (Figure 6.11): This procedure performs the MixColumns operation at the bit level on the committed state bits. It processes the state byte-by-byte

ZK.SBoxAffine($\langle x_{0,0}\rangle_\lambda^2, \ldots, \langle x_{0,7}\rangle_\lambda^2, \ldots, \langle x_{N_{\text{st-bytes}}-1,0}\rangle_\lambda^2, \ldots, \langle x_{N_{\text{st-bytes}}-1,7}\rangle_\lambda^2$; sq = false)

INPUT: $\langle x_{i,j}\rangle_\lambda^2$: degree-2 commitment to $x_{i,j} \in \mathbb{F}_{2^8} \subset \mathbb{F}_{2^\lambda}$, where $x_{i,j} = x_{0,i}^{2^j}$ for $j > 0$; flag sq

OUTPUT: $\langle \boldsymbol{y}\rangle_\lambda^2$, commitments to the affine component of the S-box, applied to each $x_{i,0}$

if sq **then**
  $s \leftarrow 2, t \leftarrow 1$
else
  $s \leftarrow 1, t \leftarrow 0$
// Define the 9 coefficients $\zeta_0, \ldots, \zeta_8 \in \mathbb{F}_{2^\lambda}$, where $\zeta_i = \text{ByteCombine}(\chi_i; \lambda)$
// We can take $(\chi_0, \ldots, \chi_8) = (0x05, 0x09, 0xf9, 0x25, 0xf4, 0x01, 0xb5, 0x8f, 0x63)$ for the affine map,
// and for the square version $(0x11, 0x41, 0x07, 0x7d, 0x56, 0x01, 0xfc, 0xcf, 0xc2)$.
for $i \in [0, N_{\text{st-bytes}})$ **do**
  $\langle y_i\rangle_\lambda^2 \leftarrow \zeta_8^s + \zeta_7^s \langle x_{i,7+t \bmod 8}\rangle^2 + \zeta_6^s \langle x_{i,6+t}\rangle^2 + \cdots + \zeta_0^s \langle x_{i,0+t}\rangle^2$
return $\langle \boldsymbol{y}\rangle_\lambda^2 = (\langle y_0\rangle_\lambda^2, \ldots, \langle y_{N_{\text{st-bytes}}-1}\rangle_\lambda^2)$

Fig. 6.8: Apply the affine map (or its square) of the S-box to every component of the state, given as input all the conjugates of each byte of the state.

ZK.InverseAffine(⟨state-bits⟩)

INPUT: For each byte $i \in [0, N_{\text{st-bytes}})$, commitments $\langle x_{i,j}\rangle$ to $x_{i,j} \in \mathbb{F}_2$ for $j \in \{0, \ldots, 7\}$.
OUTPUT: For each byte i, commitments $\langle y_{i,j}\rangle$ to $y_{i,j} \in \mathbb{F}_2$ for $j \in \{0, \ldots, 7\}$
for $i \in [0, N_{\text{st-bytes}})$ **do**
  for $j \in [0, 8)$ **do**
    $c_j \leftarrow 1$ **if** $j \in \{0, 2\}$ **else** 0
    $\langle y_{i,j}\rangle \leftarrow \langle x_{i,(j-1) \bmod 8}\rangle + \langle x_{i,(j-3) \bmod 8}\rangle + \langle x_{i,(j-6) \bmod 8}\rangle + \text{ConstantToVOLE}(c_j)$
return $\{(\langle y_{i,0}\rangle, \ldots, \langle y_{i,7}\rangle)\}_{i=0}^{N_{\text{st-bytes}}-1}$

Fig. 6.9: Inverse affine layer on the entire state. Each byte is processed as an 8-bit Boolean circuit in $\mathbb{F}_2$.

by first extracting bit-level representations of each byte, then applying a linear combination (multiplication by constants in $\mathbb{F}_{2^8}$) and recombining the results, all while preserving the VOLE commitment structure.

- ZK.ShiftRows (Figure 6.12): This procedure applies the ShiftRows transformation to an AES/Rijndael state where the state is stored as an ordered list of bytes. Each byte is represented as a VOLE commitment in $\mathbb{F}_{2^8}$. Internally, the state is viewed as a $4 \times N_{\text{st}}$ matrix (with N_{st} columns), and the procedure rotates each row r to the left by r positions (with a slight modification when $N_{\text{st}} = 8$ such that the last two rows are rotated by $r + 1$ positions). The output is a commitment to resulting the shifted state.

- ZK.InverseShiftRows (Figure 6.13): This procedure reverses the ShiftRows operation on a bit-level representation of the state. It takes as input a commitment to an $N_{\text{st-bits}}$-bit array, and reorders the bits to undo the row rotations done by the ShiftRows step. The output is a $N_{\text{st-bits}}$-bit array of VOLE commitments.

- ZK.AddRoundKey (Figure 6.14): This procedure implements the AddRoundKey step at the bit level. It takes as input two arrays of degree-1 VOLE commitments for the state bits and one for the round key bits, and outputs an array of VOLE commitments representing the new state after the key addition. For each index i in the range $[0, N_{\text{st-bits}})$,

ZK.MixColumns($\langle \boldsymbol{x} \rangle^2_\lambda$, sq $\in \{0,1\}$)

INPUT: $\langle \boldsymbol{x} \rangle^2_\lambda = (\langle x_0 \rangle^2_\lambda, \ldots, \langle x_{(N_{\text{st-bytes}}-1)} \rangle^2_\lambda)$, each $\langle x_i \rangle^2_\lambda$ is degree-2 commitment to $x_i \in \mathbb{F}_{2^8} \subset \mathbb{F}_{2^\lambda}$

OUTPUT: $\langle \boldsymbol{y} \rangle^2_\lambda = (\langle y_0 \rangle^2_\lambda, \ldots, \langle y_{(N_{\text{st-bytes}}-1)} \rangle^2_\lambda)$, each $\langle y_i \rangle^2_\lambda$ is degree-2 commitment to $y_i \in \mathbb{F}_{2^8} \subset \mathbb{F}_{2^\lambda}$

// Flag sq determines whether to square the matrix constants.
// Let $\nu_1 \leftarrow$ ByteCombine$(\{01\}; \lambda)$, $\nu_2 \leftarrow$ ByteCombine$(\{02\}; \lambda)$, $\nu_3 \leftarrow$ ByteCombine$(\{03\}; \lambda)$.
// If sq $= 1$, then replace each ν_i by its square in $\mathbb{F}_{2^8}$: $\nu_1 \leftarrow \nu_1^2$, $\nu_2 \leftarrow \nu_2^2$, $\nu_3 \leftarrow \nu_3^2$
for $c \in [0..N_{\text{st}})$ **do**
  $i_0 \leftarrow 4c$, $i_1 \leftarrow 4c+1$, $i_2 \leftarrow 4c+2$, $i_3 \leftarrow 4c+3$
  // Compute $\langle y[i_0..i_3] \rangle$ via $(2,3,1,1)$ or $(4,5,1,1)$ in $\mathbb{F}_{2^8}$.
  $\langle y_{i_0} \rangle^2_\lambda \leftarrow \langle x_{i_0} \rangle^2_\lambda \cdot \nu_2 + \langle x_{i_1} \rangle^2_\lambda \cdot \nu_3 + \langle x_{i_2} \rangle^2_\lambda \cdot \nu_1 + \langle x_{i_3} \rangle^2_\lambda \cdot \nu_1$
  $\langle y_{i_1} \rangle^2_\lambda \leftarrow \langle x_{i_0} \rangle^2_\lambda \cdot \nu_1 + \langle x_{i_1} \rangle^2_\lambda \cdot \nu_2 + \langle x_{i_2} \rangle^2_\lambda \cdot \nu_3 + \langle x_{i_3} \rangle^2_\lambda \cdot \nu_1$
  $\langle y_{i_2} \rangle^2_\lambda \leftarrow \langle x_{i_0} \rangle^2_\lambda \cdot \nu_1 + \langle x_{i_1} \rangle^2_\lambda \cdot \nu_1 + \langle x_{i_2} \rangle^2_\lambda \cdot \nu_2 + \langle x_{i_3} \rangle^2_\lambda \cdot \nu_3$
  $\langle y_{i_3} \rangle^2_\lambda \leftarrow \langle x_{i_0} \rangle^2_\lambda \cdot \nu_3 + \langle x_{i_1} \rangle^2_\lambda \cdot \nu_1 + \langle x_{i_2} \rangle^2_\lambda \cdot \nu_1 + \langle x_{i_3} \rangle^2_\lambda \cdot \nu_2$
return $\langle \boldsymbol{y} \rangle^2_\lambda$

Fig. 6.10: MixColumns with optional squared matrix depending on the flag sq on committed byte values

ZK.BitwiseMixColumns$(\langle s_0 \rangle, \ldots, \langle s_{N_{\text{st-bits}}-1} \rangle)$

INPUT: $\langle s_i \rangle$: VOLE commitments to state bits

OUTPUT: $\langle o_i \rangle$: VOLE commitments to state bits after MixColumns Operation

for $c \in [0..N_{\text{st}})$ **do**
  for $r \in [0..4)$ **do**
    // Read byte from input, and multiply by $\alpha \in \mathbb{F}_{256} = \mathbb{F}_2[\alpha]/(\alpha^8 + \alpha^4 + \alpha^3 + \alpha + 1)$
    $\langle \mathbf{a}^{(r)} \rangle := (\langle s_{32 \cdot c + 8 \cdot r} \rangle_1, \ldots, \langle s_{32 \cdot c + 8 \cdot r + 7} \rangle_1)$
    $\langle \mathbf{b}^{(r)} \rangle := \langle (a_7^{(r)}, a_0^{(r)} + a_7^{(r)}, a_1^{(r)}, a_2^{(r)} + a_7^{(r)}, a_3^{(r)} + a_7^{(r)}, a_4^{(r)}, a_5^{(r)}, a_6^{(r)}) \rangle$
  $\langle \mathbf{o}_{4c+0} \rangle \leftarrow \langle \mathbf{b}^{(0)} + \mathbf{a}^{(3)} + \mathbf{a}^{(2)} + \mathbf{b}^{(1)} + \mathbf{a}^{(1)} \rangle$
  $\langle \mathbf{o}_{4c+1} \rangle \leftarrow \langle \mathbf{b}^{(1)} + \mathbf{a}^{(0)} + \mathbf{a}^{(3)} + \mathbf{b}^{(2)} + \mathbf{a}^{(2)} \rangle$
  $\langle \mathbf{o}_{4c+2} \rangle \leftarrow \langle \mathbf{b}^{(2)} + \mathbf{a}^{(1)} + \mathbf{a}^{(0)} + \mathbf{b}^{(3)} + \mathbf{a}^{(3)} \rangle$
  $\langle \mathbf{o}_{4c+3} \rangle \leftarrow \langle \mathbf{b}^{(3)} + \mathbf{a}^{(2)} + \mathbf{a}^{(1)} + \mathbf{b}^{(0)} + \mathbf{a}^{(0)} \rangle$
return $\langle \mathbf{o}_0 \parallel \mathbf{o}_1 \parallel \cdots \parallel \mathbf{o}_{(N_{\text{st-bytes}}-1)} \rangle$

Fig. 6.11: MixColumns operation performed on commitments to bits.

the output is computed as:

$$\langle o_i \rangle \leftarrow \langle s_i \rangle + \langle k_i \rangle,$$

where the addition is performed over $\mathbb{F}_2$ (i.e., a bitwise XOR).

- ZK.AddRoundKeyBytes (Figure 6.14): This procedure performs the AddRoundKey operation at the byte level. It takes as input an array of degree-2 VOLE commitments representing the state bytes and an array of VOLE commitments representing the corresponding round key bytes, where the key commitments have degree $d \in \{1, 2\}$. For each byte index i in the range $[0, N_{\text{st-bytes}})$, it computes:

$$\langle o_i \rangle^2_\lambda \leftarrow \langle s_i \rangle^2_\lambda + \langle k_i \rangle^d_\lambda,$$

yielding new degree-d commitments for the state after the key addition.

ZK.ShiftRows($\langle$state$\rangle$)

INPUT: state $= (\langle s_0\rangle_\lambda, \ldots, \langle s_{(N_{\text{st-bytes}}-1)}\rangle_\lambda)$ in byte order (each $\langle s_i\rangle_\lambda$ is a commitment to s_i in $\mathbb{F}_{2^8}$).
OUTPUT: state$' = (\langle s'_0\rangle_\lambda, \ldots, \langle s'_{(N_{\text{st-bytes}}-1)}\rangle_\lambda)$, with each row r shifted left by r positions.

// Internally, AES stores the $N_{\text{st-bytes}}$ bytes in a $4 \times N_{\text{st}}$ matrix by column.
// Let $\langle s_{4c+r}\rangle_\lambda$ be the commitment corresponding to the (r, c) entry.
// We rotate row r left by r positions, unless $N_{\text{st}} = 8$, when the last two rows are rotated by $r + 1$.
for $r \in [0..4)$ **do**
  for $c \in [0..N_{\text{st}})$ **do**
    if $N_{\text{st}} \neq 8$ or $r \in \{0, 1\}$ **do**
      $\langle s'_{4c+r}\rangle_\lambda \leftarrow \langle s_{4\cdot((c+r) \bmod N_{\text{st}})+r}\rangle_\lambda$
    else
      $\langle s'_{4c+r}\rangle_\lambda \leftarrow \langle s_{4\cdot((c+r+1) \bmod N_{\text{st}})+r}\rangle_\lambda$
return $(\langle s'_0\rangle_\lambda, \ldots, \langle s'_{(N_{\text{st-bytes}}-1)}\rangle_\lambda)$

Fig. 6.12: ShiftRows on a committed state .

ZK.InverseShiftRows($\langle$state-bits$\rangle$)

INPUT: state-bits $= (\langle s_0\rangle, \ldots, \langle s_{N_{\text{st-bits}}-1}\rangle) \in \{0,1\}^{N_{\text{st-bits}}}$, where each $\langle s_i\rangle$ is a commitment to a bit.
OUTPUT: state-bits$' = (\langle s'_0\rangle, \ldots, \langle s'_{N_{\text{st-bits}}-1}\rangle)$

for $r \in [0..4)$ **do**
  for $c \in [0..N_{\text{st}})$ **do**
    if $N_{\text{st}} \neq 8$ or $r \in \{0, 1\}$ **do** // Shift left by r columns
      $i \leftarrow 4\,((c - r) \bmod N_{\text{st}}) + r$
    else // Shift left by $r + 1$ columns
      $i \leftarrow 4\,((c - r - 1) \bmod N_{\text{st}}) + r$
    $(\langle s'_{8(4c+r)}\rangle, \ldots, \langle s'_{8(4c+r)+7}\rangle) \leftarrow (\langle s_{8i}\rangle, \ldots, \langle s_{8i+7}\rangle)$
return $(\langle s'_0\rangle, \ldots, \langle s'_{N_{\text{st-bits}}-1}\rangle)$

Fig. 6.13: Inverse of the ShiftRows operation on a bit-level $N_{\text{st-bits}}$-bit array. The procedure reverses the row rotations applied by ShiftRows.

6.5 Witness Extension

The AES.ExtendWitness algorithm (Figure 6.15) takes the AES key $\mathbf{k}$, the FAEST public key $\mathsf{pk} = (\mathbf{in}, \mathbf{out})$, and the instance parameters $\mathtt{param}$ and $\mathtt{param}_{\mathsf{OWF}}$, returning an extended witness $\mathbf{w} \in \{0,1\}^\ell$ for later use. It can be divided in two main steps as follows.

Key Expansion. It first runs the AES KeyExpansion on $\mathbf{k}$ to get the expanded key $\bar{\mathbf{k}}$, hence it appends the first N_{k} words to $\mathbf{w}$ as $\mathtt{key\_bits}$. For each SubWord operation (i.e. after every 4 or 6 words, depending on λ), the algorithm appends the 32-bit subword, denoted as $\mathtt{non\text{-}lin\_word\_bits}$, Therefore, each subword uses 32 bits in the extended witness.

Encryption Routine. For each plaintext block $b \in \{0, 1\}$ (depending on λ), the algorithm initializes the state as $\mathbf{in}_b$ and perform $R - 1$ rounds. It only stores the state bits after each ShiftRows *on odd rounds* (i.e. on rounds j where $1 \leq j < R$ and $j \bmod 2 = 1$). More specifically, for each of these odd rounds, it appends $N_{\mathsf{st\text{-}bits}}$ bits of *column-major* state into $\mathbf{w}$, denoted as $\mathtt{shift\_row\_bits}_i$. For the other rounds, if j is even, it breaks the 16 bytes of the state into $\{b_0, \ldots, b_{15}\}$ and appends InvNorm(byte) $\in \{0,1\}^4$ for each byte. Therefore, the extended witness $\mathbf{w} \in \{0,1\}^\ell$ which can be represented as:

ZK.AddRoundKey($\langle s_0\rangle, \ldots, \langle s_{N_{\text{st-bits}}-1}\rangle, \langle k_0\rangle, \ldots, \langle k_{N_{\text{st-bits}}-1}\rangle$)

INPUT: $\langle s_i\rangle, \langle k_i\rangle$: commitments to state bits and round key bits

OUTPUT: $\langle o_i\rangle$: commitments to state bits after AddRoundKey

1 : **for** $\in [0..N_{\text{st-bits}})$ **do**

2 : $\quad \langle o_i\rangle \leftarrow \langle s_i\rangle + \langle k_i\rangle$

3 : **return** $(\langle o_0\rangle, \ldots, \langle o_{N_{\text{st-bits}}-1}\rangle)$

ZK.AddRoundKeyBytes($\langle s_0\rangle^2_\lambda, \ldots, \langle s_{N_{\text{st-bytes}}-1}\rangle^2_\lambda, \langle k_0\rangle^d_\lambda, \ldots, \langle k_{N_{\text{st-bytes}}-1}\rangle^d_\lambda$) (where $d \in \{1, 2\}$)

INPUT: $\langle s_i\rangle^2_\lambda, \langle k_i\rangle^d_\lambda$: degree-2 commitments to state bytes, degree-d commitments to round key bytes

OUTPUT: $\langle o_i\rangle^d_\lambda$: degree-$d$ commitments to state bytes after AddRoundKey

1 : **for** $\in [0..N_{\text{st-bytes}})$ **do**

2 : $\quad \langle o_i\rangle^2_\lambda \leftarrow \langle s_i\rangle^2_\lambda + \langle k_i\rangle^d_\lambda$

3 : **return** $(\langle o_0\rangle^2_\lambda, \ldots, \langle o_{N_{\text{st-bytes}}-1}\rangle^2_\lambda)$

Fig. 6.14: AddRoundKey procedure. It XORs consecutive bits or bytes of the expanded key with the current AES/Rijndael state.

$$\mathbf{w} := \underbrace{\texttt{key\_bits}}_{N_\mathsf{k}\text{ words}} \| \underbrace{(\texttt{non-lin\_word\_bits})}_{S_{\mathsf{ke}}/4\text{ non-lin words}} \| \left\{ \underbrace{(\texttt{inv\_norm\_bits}_{2i})}_{16N_{\text{st}}} \| \underbrace{(\texttt{odd round shift\_row\_bits}_{2i+1})}_{N_{\text{st-bits}}} \right\}_{i\in[0..R/2)}.$$

If $\beta = 2$, the algorithm similarly stores an interleaving of `inv_norm_bits` and `shift_row_bits`, but repeats it twice for the two encryption blocks.

6.6 Deriving Constraints for the Key Expansion Routine

The KeyExpCstrnts algorithm, described in Figure 6.18, derives the constraint values required for the QuickSilver proof for the AES circuit corresponding to the key expansion. All computations are performed directly on VOLE commitment. In particular, the algorithm combines the outputs of two forward/backward key schedule routines, namely KeyExpFwd (see Figure 6.16) and KeyExpBkwd (see Figure 6.17), to generate commitments to both the round keys and the corresponding inverse outputs of the $\mathbb{F}_{2^8}$ S-boxes.

KeyExpFwd. Given a vector of VOLE-committed witness values $\langle \boldsymbol{w}\rangle$, this algorithm computes commitments to the $R+1$ round keys for the encryption routine. It initializes the first λ committed bits (or N_k words) of the expanded key directly from the witness values, and then, for each subsequent word, it either directly reads the corresponding values from $\langle \boldsymbol{w}\rangle$ (when ($j \bmod N_\mathsf{k} = 0$) or ($N_\mathsf{k} > 6$ and $j \bmod N_\mathsf{k} = 4$), i.e, when the word j corresponds to a word derived from SubWord); otherwise it computes the new word as the sum of two previous words. The resulting commitments to the round key bits are the output of the function.

KeyExpBkwd. To compute the outputs of the $\mathbb{F}_{2^8}$ inversions (i.e., to recover the S-box inputs or outputs) during key expansion, KeyExpBkwd works entirely on VOLE commitments. It selects the appropriate portions of the expanded key $\langle \boldsymbol{x}\rangle$ and removes the byte that was XOR-ed into the key word, using the corresponding committed key values in $\langle \mathbf{x}_{\mathsf{key}}\rangle$, as well as subtracting the round constant when necessary. Then, it applies the inverse of the S-box's $\mathbb{F}_2$-affine transformation to revert the final linear transformation, so the result matches the $\mathbb{F}_{2^8}$ input to the S-box. To simplify writing into the output array $\langle \boldsymbol{y}\rangle$, KeyExpBkwd iterates over the S_{ke} S-boxes contained within the key expansion. However, since the relevant expanded key words (after the first N_k) are situated at intervals of

FAEST.ExtendWitness($\mathbf{k}$, pk; $\mathtt{param}$, $\mathtt{param}_{\mathsf{OWF}}$)

INPUT: $\mathbf{k} \in \{0,1\}^{\lambda}$, $\mathsf{pk} = (\mathbf{in}, \mathbf{out}) \in \{0,1\}^{N_{\text{st-bits}} \cdot \beta} \times \{0,1\}^{N_{\text{st-bits}} \cdot \beta}$
OUTPUT: $\mathbf{w} \in \{0,1\}^{\ell}$: extended witness

1 : // We set EM = `false` if we are considering AES, otherwise `false`
2 : // Initialize empty extended witness array, eventually of length ℓ
3 : **new** $\mathbf{w} \in \{0,1\}^{\leq \ell}$
4 : // Store round keys as array of 32-bit words
5 : **new** $\bar{\mathbf{k}} \in [\{0,1\}^{32}; N_{\mathsf{st}}(R+1)]$
6 : $\bar{\mathbf{k}} \leftarrow \mathsf{KeyExpansion}(\mathbf{k}, \mathtt{param}_{\mathsf{OWF}})$
7 : $\mathbf{w} \leftarrow \bar{\mathbf{k}}[0..N_{\mathsf{k}} - 1]$ // Saving the first N_{k} words, which contain $\mathbf{k}$
8 : **if** EM = `false` **then** // extract key schedule witness components
9 : $i_{\mathsf{k}} \leftarrow N_{\mathsf{k}}$
10 : **for** $j \in [0..S_{\mathsf{ke}}/4)$ **do**
11 : $\mathbf{w} \leftarrow \mathbf{w} \| \bar{\mathbf{k}}[i_{\mathsf{k}}]$ // Saving the word that depends on SubWord
12 : **if** $\lambda = 192$ **then** $i_{\mathsf{k}} := i_{\mathsf{k}} + 6$
13 : **else** $i_{\mathsf{k}} \leftarrow i_{\mathsf{k}} + 4$
// Encryption routine and saving witness bits
14 : **for** $b \in [0..\beta)$ **do**
15 : **new** `state` $\in \{0,1\}^{N_{\text{st-bits}}}$
16 : `state` $\leftarrow \mathbf{in}_b$
17 : AddRoundKey(`state`, $\bar{\mathbf{k}}[0..N_{\mathsf{st}} - 1]$)
18 : **for** $j \in [0..R-1)$ **do** ;
19 : **if** $j \bmod 2 = 0$ **then**
20 : // Save 4-bit inverse norm of S-box inputs, packed into bytes
21 : **foreach** byte b_i of state **do**
22 : $\mathbf{w} \leftarrow \mathbf{w} \| \mathsf{InvNorm}(b_i)$
23 : SubBytes(`state`)
24 : ShiftRows(`state`)
25 : **if** $j \bmod 2 = 1$ **then**
26 : $\mathbf{w} := \mathbf{w} \|$ `state` // Save S-box outputs, in column-major order
27 : MixColumns(`state`)
28 : AddRoundKey(`state`, $\bar{\mathbf{k}}[N_{\mathsf{st}} \cdot (j+1)..N_{\mathsf{st}}(j+2) - 1]$)
// Last round is not committed to, so not computed
29 : **return** $\mathbf{w} \in \{0,1\}^{\ell}$

Fig. 6.15: Extending the witness from the AES/Rijndael key and the input plaintext block(s).

either 4 (for AES128 and AES256) or 6 (for AES192), the algorithm maintains an alternative index i_{wd} which is increased after every 4 S-box by the appropriate amount (in bits). In addition, since once every four S-Box outputs (one out of every eight for AES256) have the round constant added into them, we use `rconEvry` to track how often round constants must be removed.

KeyExpCstrnts. The procedure derives the S_{ke} commitments corresponding to the key expansion routine's constraints and the round keys. It first invokes KeyExpFwd($\langle \mathbf{w} \rangle$) to produce commitments $\langle \mathbf{k} \rangle$ for the entire expanded key. Next, it calls KeyExpBkwd($\langle \mathbf{w} \rangle^{\lambda..}, \langle \mathbf{k} \rangle$) to reconstruct the intermediate S-box output.

After these forward/backward steps, KeyExpCstrnts groups the subwords into blocks of four. For each subword, it extracts 8-bit chunks from $\langle \mathbf{k} \rangle$ or $\langle \bar{\mathbf{w}} \rangle$ and maps them into $\mathbb{F}_{2^{\lambda}}$

FAEST.KeyExpFwd($\langle w_0\rangle, \ldots, \langle w_{\ell_{\mathsf{ke}}-1}\rangle$; $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)

INPUT: $\langle w_i\rangle$: VOLE commitment to the i-th extended witness bit
OUTPUT: $\langle \mathbf{y}\rangle = (\langle y_0\rangle, \ldots, \langle y_{(R+1)\cdot 128-1}\rangle)$: VOLE commitments to the round keys

1 : **for** $i \in [0..\lambda)$ **do**
2 : $\quad \langle y_i\rangle \leftarrow \langle w_i\rangle$ // First N_{k} words
3 : $i_{\mathsf{wd}} \leftarrow \lambda$ // Index to read words from x
4 : **for** $j \in [N_{\mathsf{k}}..4(R+1))$ **do** // Remaining key words, $R+1$ round keys, 4 words each
5 : $\quad$ **if** $j \bmod N_{\mathsf{k}} = 0$ **or** $(N_{\mathsf{k}} > 6$ **and** $j \bmod N_{\mathsf{k}} = 4)$
6 : $\quad\quad \langle y_{32j}\rangle, \ldots, \langle y_{32j+31}\rangle \leftarrow \langle w_{i_{\mathsf{wd}}}\rangle, \ldots, \langle w_{i_{\mathsf{wd}}+31}\rangle$ // Selecting expanded key word bits
7 : $\quad\quad i_{\mathsf{wd}} \leftarrow i_{\mathsf{wd}} + 32$
8 : $\quad$ **else**
9 : $\quad\quad$ **for** $i \in [0..32)$ **do**
10 : $\quad\quad\quad \langle y_{32j+i}\rangle \leftarrow \langle y_{32(j-N_{\mathsf{k}})+i}\rangle + \langle y_{32(j-1)+i}\rangle$
11 : **return** $\langle \mathbf{y}\rangle$, where $\mathbf{y} \in \mathbb{F}_2^{(R+1)\cdot 128}$

Fig. 6.16: Transforming witness or VOLE values into AES round keys.

via ZK.ByteCombine. It also applies SquareAndCombine to obtain their squares in $\mathbb{F}_{2^\lambda}$. Finally, for each subword it generates two degree-2 VOLE constraint commitments, of the form

$$\hat{k}^2 \cdot \hat{w} - \hat{k} = 0 \quad \text{and} \quad \hat{k} \cdot \hat{w}^2 - \hat{w} = 0,$$

where $\hat{k}$ and $\hat{w}$ are the combined elements in $\mathbb{F}_{2^\lambda}$. These constraints encode that $\hat{k}$ is indeed the Hence, if all such constraints are satisfied, the final commitments are effectively commitments to zero.

6.7 Deriving Constraints for the Encryption Routine

Here, we first describe the InvNormConstraints procedure and then the main algorithm ZK.EncCstrnts.

InvNormConstraints *(Figure 6.19).* This function derives the constraints to check that a committed element y is the inverse $\mathbb{F}_{2^4}$-norm of another committed value $a \in \mathbb{F}_{2^8}$. It outputs a single degree-3 VOLE commitment $\langle z\rangle$ that must equal zero if (and only if) $y \times a^2 \times a^{16} = a$. This ensures y is indeed the inverse-norm of a, or equivalently, $y = a^{-17}$

ZK.EncCstrnts *(Figure 6.20).* This procedure generates the constraint vector $\{\langle z_k\rangle\}_{k=0}^{S_{\mathsf{enc}}-1}$ needed by the ZK.OWFProve algorithm to show correctness of the AES (or Rijndael) encryption. Below is an overview of how it works.

Inputs and Outputs. The algorithm takes as input: commitments $\langle \mathbf{in}\rangle, \langle \mathbf{out}\rangle$ to the initial and final AES/Rijndael state, each in $\{0,1\}^{N_{\text{st-bits}}}$; a commitment $\langle \mathbf{w}\rangle$ to the extended witness (length ℓ_{enc} bits); commitments $\langle \mathbf{k}_i\rangle$ to the round keys, for $i = 0, \ldots, R$. It returns a list of QuickSilver constraints $\{\langle z_0\rangle, \ldots, \langle z_{S_{\mathsf{enc}}-1}\rangle\}$, that must evaluate to zero if the witness and ciphertext are consistent with a valid encryption.

Overall Structure. First, EncCstrnts calls ZK.AddRoundKey($\langle \mathbf{in}\rangle, \langle \mathbf{k}_0\rangle$), preparing the initial state for the main AES loop. Then it iterates over $r = 0, 1, \ldots, R/2 - 1$. Each iteration handles two consecutive rounds $(2r)$ and $(2r+1)$, as follows:

1. *Conjugates of the Current State.* The algorithm computes bitwise commitments to the state and then calls $\mathbb{F}_{256}/\mathbb{F}_2$.Conjugates to obtain each byte's Frobenius conjugates in $\mathbb{F}_{2^8}$. Let $\langle \mathsf{state}_{\mathsf{conj},i,j}\rangle_\lambda$ be the commitment to $s_i^{2^j}$, where s_i is the i-th byte in the current state.

FAEST.KeyExpBkwd($\langle \mathbf{x} \rangle, \langle \mathbf{x}_{\mathsf{key}} \rangle, ; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)

INPUT: VOLE-committed extended witness bits $\{\langle \mathbf{x}[i] \rangle\}_{i=0}^{\ell_{\mathsf{ke}}-1}$ and round keys $\{\langle \mathbf{x}_{\mathsf{key}}[i] \rangle\}_{i=0}^{(R+1)\cdot 128} - 1$

OUTPUT: $\{\langle \mathbf{y}[i] \rangle\}_{i=0}^{8S_{\mathsf{ke}}-1}$: committed bits of the key schedule $\mathbb{F}_{2^8}$ inversion outputs

1 : **new** $\langle \mathbf{y} \rangle$, where $\mathbf{y} \in \mathbb{F}_2^{8S_{\mathsf{ke}}}$ // The output array
2 : **new** $\langle \tilde{\mathbf{x}} \rangle$, where $\tilde{\mathbf{x}} \in \mathbb{F}_2^8$ // Temporary storage
3 : $i_{\mathsf{wd}} := 0$ // Index to read words from x_k
4 : $\mathtt{rconEvry} := 4 \left\lfloor \frac{\lambda}{128} \right\rfloor$ // Period of round constant removal
5 : **for** $j \in [0..S_{\mathsf{ke}})$ **do** // Iterating S-box-wise
6 : **for** $i \in [0..8)$ **do** // Removing XOR-ed byte
7 : $\langle \tilde{\mathbf{x}}[i] \rangle \leftarrow \langle \mathbf{x}[8j+i] \rangle + \langle \mathbf{x}_{\mathsf{key}}[i_{\mathsf{wd}} + 8(j \bmod 4) + i] \rangle$
8 : **if** $j \bmod \mathtt{rconEvry} = 0$ **then**
9 : // $\mathtt{Rcon}[\mathtt{k}][i]$ is i-th bit of k-th round constant (cf. Table 4.1)
10 : $\langle \tilde{\mathbf{x}}[i] \rangle \leftarrow \langle \tilde{\mathbf{x}}[i] \rangle + \mathtt{Rcon}[\lfloor \mathtt{j}/\mathtt{rconEvry} \rfloor][i]$
11 : $\langle \mathbf{y}[8j..8j+7] \rangle \leftarrow$ InverseAffine$(\langle \tilde{\mathbf{x}} \rangle)$ // Storing the affine layer inverse
12 : **if** $j \bmod 4 = 3$ **then** // Move i_{wd} every 4 S-boxes
13 : **if** $\lambda = 192$ **then** $i_{\mathsf{wd}} := i_{\mathsf{wd}} + 192$
14 : **else** $i_{\mathsf{wd}} := i_{\mathsf{wd}} + 128$
15 : **return** $\langle \mathbf{y} \rangle$

Fig. 6.17: Transforming VOLE-committed extended witness for the key schedule into $\mathbb{F}_{2^8}$-inverse outputs.

2. *Inverse Norm Constraints.* From the extended witness $\langle \mathbf{w} \rangle$, the algorithm extracts the 4-bit values (InvNorm-related) corresponding to each byte's S-box input. It lifts those bits to $\mathbb{F}_{2^\lambda}$ via InvNormToConjugates. Then, for each byte i, it calls InvNormConstraints (Figure 6.19) on $\langle \mathsf{state}_{\mathsf{conj},i} \rangle, \langle y_i \rangle$ to check that the witness indeed encodes a^{-17} in $\mathbb{F}_{2^8}$. This yields a degree-3 commitment $\langle z_{2r,i} \rangle_\lambda^3$ to zero if (and only if) the InvNorm bits match the current state. These constraints enforce that the 4-bit values in $\mathbf{w}$ describe each S-box input's inverse norm in $\mathbb{F}_{2^4}$.
3. *Forward Computation of Round* $(2r) \to (2r+1)$. Using the state conjugates and the InvNorm conjugates, the algorithm forms commitments to each S-box output at round $(2r+1)$. Concretely:
   - It constructs the subkey $\mathbf{k}_{2r+1}$ at both degree 1 and degree 2 (SquareAndCombine per-byte).
   - For each $b \in \{0,1\}$, it runs both the squared and not-squared version of the S-box affine operation, ShiftRows, MixColumns, and AddRoundKey. The result is the two commitments $\langle st_b \rangle_\lambda^2$.
4. *Backward Reconstruction of Round* $(2r+1)$. After finishing the "forward" steps of round $(2r+1)$, the algorithm verifies consistency from the "backward" computation. If r is not the final iteration, the partial state after ShiftRows (round $2r+1$) is retrieved from the extended witness (or **out** if it is the final round). The code calls InverseShiftRows and InverseAffine on these bits to reconstruct the S-box outputs. Then it compares them to the forward state $\langle st_b \rangle$ by creating degree-3 constraints that tie each "inverse-output" to the "forward-output." Specifically, for each byte, it calls SquareAndCombine to form $\hat{s}$ and $\hat{s}^2$ and generates $\langle z_{(3r+1)N_{\text{st-bytes}}+2i} \rangle_\lambda^3$ and $\langle z_{(3r+1)N_{\text{st-bytes}}+2i+1} \rangle_\lambda^3$, which must be zero if the forward and backward views match.
5. *Next Round State.* Except for the last iteration, the procedure obtains the intermediate state bits $\langle \tilde{\mathbf{s}} \rangle$ after ShiftRows in round $2r+1$ and applies BitwiseMixColumns and AddRoundKey with $\langle \mathbf{k}_{2r+2} \rangle$. This produces the committed bits $\langle \mathtt{state\text{-}bits} \rangle$ for the next iteration $(2r+2)$.

FAEST.KeyExpCstrnts($\langle \mathbf{w} \rangle$; $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)

INPUT: Commitment to witness $\mathbf{w} \in \{0,1\}^{\ell_{\mathsf{ke}}}$

OUTPUT: $\langle \boldsymbol{z} \rangle^2_\lambda$: constraint vector $\boldsymbol{z} \in \mathbb{F}_{2^\lambda}^{2S_{\mathsf{ke}}}$, $\langle \mathbf{k} \rangle$: committed round keys $\mathbf{k} \in \mathbb{F}_2^{(R+1)\cdot 128}$

$\langle \mathbf{k} \rangle \leftarrow \mathsf{KeyExpFwd}\big(\langle \mathbf{w} \rangle; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}\big)$ // Expanded key bits

$\langle \bar{\mathbf{w}} \rangle \leftarrow \mathsf{KeyExpBkwd}\big(\langle \mathbf{w}[\lambda..] \rangle, \langle \mathbf{k} \rangle; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}\big)$ // Inverse output bits

// In input $\boldsymbol{w}[\lambda..]$ means: skip the first λ bits

// Setup index for reading every 4 S-boxes:

$i_{\mathsf{wd}} := 32 \cdot (N_{\mathsf{k}} - 1)$

$\mathtt{doRotWord} := \mathtt{True}$ // Used for AES256 (rotword toggling).

for $j \in [0..S_{\mathsf{ke}}/4)$ **do**

  new $\langle \hat{\boldsymbol{k}} \rangle^1_\lambda, \langle \hat{\boldsymbol{k}}_{\mathsf{sq}} \rangle^1_\lambda, \langle \hat{\boldsymbol{w}} \rangle^1_\lambda, \langle \hat{\boldsymbol{w}}_{\mathsf{sq}} \rangle^1_\lambda$, each a commitment to vector in $\mathbb{F}^4_{2^\lambda}$

  for $r \in [0..3]$ **do**

    $r' \leftarrow r$

    if $\mathtt{doRotWord}$ **then** $r' \leftarrow (r+3) \bmod 4$

    $\langle \hat{\boldsymbol{k}}[r'] \rangle^1_\lambda \leftarrow \mathsf{ZK.ByteCombine}\big(\langle \mathbf{k}[\, i_{\mathsf{wd}} + 8r \,..\, i_{\mathsf{wd}} + 8r + 7] \rangle^1_1,\ \lambda\big)$

    $\langle \hat{\boldsymbol{k}}_{\mathsf{sq}}[r'] \rangle^1_\lambda \leftarrow \mathsf{SquareAndCombine}\big(\langle \mathbf{k}[\, i_{\mathsf{wd}} + 8r \,..\, i_{\mathsf{wd}} + 8r + 7] \rangle^1_1,\ \lambda\big)$

    $\langle \hat{\boldsymbol{w}}[r] \rangle^1_\lambda \leftarrow \mathsf{ZK.ByteCombine}\big(\langle \bar{\mathbf{w}}[\, 32j + 8r \,..\, 32j + 8r + 7] \rangle^1_1,\ \lambda\big)$

    $\langle \hat{\boldsymbol{w}}_{\mathsf{sq}}[r] \rangle^1_\lambda \leftarrow \mathsf{SquareAndCombine}\big(\langle \bar{\mathbf{w}}[\, 32j + 8r \,..\, 32j + 8r + 7] \rangle^1_1,\ \lambda\big)$

  if $(\lambda = 256)$ **then** $\mathtt{doRotWord} \leftarrow \neg \mathtt{doRotWord}$

  for $r \in [0..3]$ **do**

    // z's are always commitments to zero, so signer need not compute degree-2 coefficients

    $\langle z_{8j+2r} \rangle^2_\lambda \leftarrow \langle \hat{\boldsymbol{k}}_{\mathsf{sq}}[r] \rangle^1_\lambda \cdot \langle \hat{\boldsymbol{w}}[r] \rangle^1_\lambda - \langle \hat{\mathbf{k}}[r] \rangle^1_\lambda$

    $\langle z_{8j+2r+1} \rangle^2_\lambda \leftarrow \langle \hat{\boldsymbol{k}}[r] \rangle^1_\lambda \cdot \langle \hat{\boldsymbol{w}}_{\mathsf{sq}}[r] \rangle^1_\lambda - \langle \hat{\boldsymbol{w}}[r] \rangle^1_\lambda$

  if $(\lambda = 192)$ **then** $i_{\mathsf{wd}} \leftarrow i_{\mathsf{wd}} + 192$ **else** $i_{\mathsf{wd}} \leftarrow i_{\mathsf{wd}} + 128$

return $(\langle z_0 \rangle^2_\lambda, \ldots, \langle z_{2S_{\mathsf{ke}}-1} \rangle^2_\lambda, \langle \mathbf{k} \rangle)$

Fig. 6.18: Deriving the $2S_{\mathsf{ke}}$ constraints for the AESλ key expansion routine.

InvNormConstraints($\langle \boldsymbol{a}_{\mathrm{conj},0} \rangle_\lambda, \ldots, \langle \boldsymbol{a}_{\mathrm{conj},7} \rangle_\lambda, \langle y \rangle_\lambda$)

INPUT: Commitments such that $a_{\mathrm{conj},i} = a_{\mathrm{conj},0}^{2^j}$ for $j \in [1..7]$

OUTPUT: $\langle z \rangle^3 \lambda$, commitment to zero

// Verify that $y \times a^2 \times a^{16} = a$

// These are commitments to zero and thus have degree 2

$\langle z \rangle^3_\lambda := \big(\langle y \rangle_\lambda \cdot \langle a_{\mathrm{conj},1} \rangle_\lambda \cdot \langle a_{\mathrm{conj},4} \rangle_\lambda\big) - \langle a_{\mathrm{conj},0} \rangle_\lambda$

return $\langle z \rangle^3_\lambda$

Fig. 6.19: Constraint to check that if $a \neq 0$ then c is the inverse of the $\mathbb{F}_{2^4}$ norm of a (that is, $c = a^{-17}$ in $\mathbb{F}_{2^8}$).

6.8 Complete OWF Constraints

This section describes the remaining steps needed to prove with QuickSilver ZK proof system that $\mathsf{pk} = (\mathbf{x}, \mathbf{y})$ is the image of the secret key $\mathbf{w}$ under the AES/Rjiandael-based one-way function. We have four algorithms:

1. The first one in Figure 6.21 presents some auxiliary operations used for constraints
2. The main OWFConstraints algorithm that combines both key-schedule and encryption constraints
3. The signer-side procedure OWFProve

ZK.EncCstrnts($\langle \mathbf{in} \rangle, \langle \mathbf{out} \rangle, \langle \mathbf{w} \rangle, \langle \mathbf{k}_0 \rangle, \ldots, \langle \mathbf{k}_R \rangle$; param, $\mathsf{param}_{\mathsf{OWF}}$)

INPUT: Commitments to $\mathbf{in} \in \{0,1\}^{N_{\text{st-bits}}}$, $\mathbf{out} \in \{0,1\}^{N_{\text{st-bits}}}$, $\mathbf{w} \in \mathbb{F}_2^{\ell_{\text{enc}}}$ and round keys $\mathbf{k}_i \in \mathbb{F}_2^{N_{\text{st-bits}}}$
OUTPUT: $\langle z_0 \rangle, \ldots, \langle z_{S_{\text{enc}}-1} \rangle$

$\langle$state-bits$\rangle \leftarrow$ ZK.AddRoundKey($\langle \mathbf{in} \rangle, \langle \mathbf{k}_0 \rangle$) // Input and output in bits
for $r \in [0..R/2)$ **do**
  // $\mathsf{state}_{\text{conj},i,j} = s_i^{2^j} \in \mathbb{F}_{2^8}$, where s_i is i-th byte of state, for $j \in [0..7]$
  $\langle \mathsf{state}_{\text{conj}} \rangle_\lambda :=$ $\mathbb{F}_{256}/\mathbb{F}_2$.Conjugates($\langle$state-bits$\rangle$) // Input in bits, output in bytes
  // Extract committed InvNorm's from the witness
  $\langle \mathbf{n} \rangle := (\langle \mathbf{w}[\frac{3}{2} N_{\text{st-bits}} \cdot r] \rangle, \ldots, \langle \mathbf{w}[\frac{3}{2} N_{\text{st-bits}} \cdot r + \frac{N_{\text{st-bits}}}{2} - 1] \rangle)$
  for $i \in [0..N_{\text{st-bytes}})$ **do**
    // Compute the conjugates $y_j = n_i^{2^j}$, for $j \in [0..3]$
    $(\langle y_0 \rangle_\lambda, \ldots, \langle y_3 \rangle_\lambda) :=$ InvNormToConjugates($\langle \mathbf{n}[4i..4i+3] \rangle$)
    // Run InvNormConstraints(st_i, n_i), using conjugates of st_i
    $\langle z_{3rN_{\text{st-bytes}}+i} \rangle_\lambda^3 :=$ InvNormConstraints($\langle \mathsf{state}_{\text{conj},i} \rangle_\lambda, \langle y_0 \rangle_\lambda$) // degree-3 commitments
    for $j \in [0,7]$ **do**
      // For state byte s, obtain conjugate s^{-2^j} by multiplying $s^{16 \cdot 2^j}$ with inv-norm conjugate, $s^{-17 \cdot 2^j}$
      $\langle (st'_{i,j}) \rangle_\lambda^2 := \langle \mathsf{state}_{\text{conj},i,j+4 \bmod 8} \rangle_\lambda \cdot \langle y_{j \bmod 4} \rangle_\lambda$
  // Compute linear part of the round key and its square, in $\mathbb{F}_{2^8}$
  $\langle \mathbf{k}^0 \rangle_\lambda^1 :=$ StateToBytes($\langle \mathbf{k}_{2r+1} \rangle$) // subkey for round $2r+1$
  $\langle \mathbf{k}^1 \rangle_\lambda^2 := (\langle k_0^0 \rangle_\lambda \cdot \langle k_0^0 \rangle_\lambda, \ldots, \langle k_{N_{\text{st-bytes}}}^0 \rangle_\lambda \cdot \langle k_{N_{\text{st-bytes}}}^0 \rangle_\lambda)$ // bytewise square of subkey
  for $b \in \{0,1\}$ **do**
    $\langle st_b \rangle_\lambda^2 :=$ ZK.SBoxAffine($(\langle st'_{0,0} \rangle_\lambda^2, \ldots, \langle st'_{0,7} \rangle_\lambda^2), \ldots, (\langle st'_{N_{\text{st-bytes}}-1,0} \rangle_\lambda^2, \ldots, \langle st'_{N_{\text{st-bytes}}-1,7} \rangle_\lambda^2)$; $\mathsf{sq} = b$)
    $\langle st_b \rangle_\lambda^2 :=$ ZK.ShiftRows($\langle st_b \rangle_\lambda^2$)
    $\langle st_b \rangle_\lambda^2 :=$ ZK.MixColumns($\langle st_b \rangle_\lambda^2, \mathsf{sq} = b$)
    $\langle st_b \rangle_\lambda^2 :=$ ZK.AddRoundKeyBytes($\langle st_b \rangle_\lambda^2, \langle \mathbf{k}^b \rangle$) // round $2r+1$ S-box inputs
  // Now go backwards to get S-box outputs for round $2r+1$
  if $r = \frac{R}{2} - 1$ **then**
    $\langle \tilde{\mathbf{s}} \rangle :=$ ZK.AddRoundKey($\langle \mathbf{out} \rangle, \langle \mathbf{k}_R \rangle$)
  else
    // Extract the bits corresponding to ShiftRows output from the extended witness
    $\langle \tilde{\mathbf{s}} \rangle := \langle \mathbf{w}[\frac{1}{2} N_{\text{st-bits}} + \frac{3}{2} N_{\text{st-bits}} \cdot r] \rangle, \ldots, \langle \mathbf{w}[\frac{3}{2} N_{\text{st-bits}} \cdot r + \frac{3}{2} N_{\text{st-bits}} - 1] \rangle$
  $\langle \mathbf{s}'' \rangle :=$ InverseShiftRows($\tilde{\mathbf{s}}$) (computed bit-wise)
  $\langle \mathbf{s} \rangle :=$ InverseAffine($\langle \mathbf{s}'' \rangle$) (bit-wise)
  for $i \in [0..N_{\text{st-bytes}})$ **do**
    $\langle s \rangle_\lambda^1 :=$ ZK.ByteCombine($\langle \mathbf{s}[8i] \rangle, \ldots, \langle \mathbf{s}[8i+7] \rangle$) // i-th output of S-box inverse
    $\langle s^{\mathsf{sq}} \rangle_\lambda^1 :=$ SquareAndCombine($\langle \mathbf{s}[8i] \rangle, \ldots, \langle \mathbf{s}[8i+7] \rangle$)

    // Parse $st_b = (st_{b,0}, \ldots, st_{b,N_{\text{st-bytes}}-1})$
    $\langle z_{(3r+1)N_{\text{st-bytes}}+2i} \rangle_\lambda^3 := \langle s^{\mathsf{sq}} \rangle_\lambda^1 \cdot \langle st_{0,i} \rangle_\lambda^2 - \langle s \rangle_\lambda^1$
    $\langle z_{(3r+1)N_{\text{st-bytes}}+2i+1} \rangle_\lambda^3 := \langle s \rangle_\lambda^1 \cdot \langle st_{1,i} \rangle_\lambda^2 - \langle st_{0,i} \rangle_\lambda^2$
  // Go forwards to compute the next round's state
  if $r \neq \frac{R}{2} - 1$ **then**
    $\langle$state-bits$\rangle :=$ ZK.BitwiseMixColumns($\langle \tilde{\mathbf{s}} \rangle$)
    $\langle$state-bits$\rangle :=$ ZK.AddRoundKey($\langle$state-bits$\rangle, \langle \mathbf{k}_{2r+2} \rangle$)
return $(\langle z_0 \rangle, \ldots, \langle z_{S_{\text{enc}}} - 1 \rangle)$

Fig. 6.20: Deriving the constraints for the encryption routine

Signer.ConstantToVOLE($\mathbf{x} \in \mathbb{F}_2^n$)	Verifier.ConstantToVOLE($\mathbf{x} \in \mathbb{F}_2^n$)
1 : **return** $\langle \mathbf{x} \rangle_2^1 := (\mathbf{0}, \mathbf{x}) \in \mathbb{F}_{2^\lambda}^n \times \mathbb{F}_2^n$	1 : // verifier holds $\Delta \in \mathbb{F}_{2^\lambda}$ 2 : **return** $\langle \mathbf{x} \rangle_2^1 := \mathbf{x} \cdot \Delta \in \mathbb{F}_{2^\lambda}^n$
Signer.Deg2To3($\langle \boldsymbol{z} \rangle_e^2$)	**Verifier.Deg2To3**($\langle \boldsymbol{z} \rangle_e^2$)
1 : // parse $\langle \boldsymbol{z} \rangle_e^2$ as poly $\boldsymbol{z}_0 + \boldsymbol{z}_1 x + \boldsymbol{z} x^2$, 2 : // where $\boldsymbol{z} = \mathbf{0} \in \mathbb{F}_{2^e}^n$ and $\boldsymbol{z}_0, \boldsymbol{z}_1 \in \mathbb{F}_{2^\lambda}^n$ 3 : **return** $\langle \boldsymbol{z} \rangle_e^3 := (\mathbf{0}, \boldsymbol{z}_0, \boldsymbol{z}_1, \mathbf{0}) \in (\mathbb{F}_{2^\lambda}^n)^4$	1 : // Parse $\langle \boldsymbol{z} \rangle_e^2$ as $\boldsymbol{\gamma}_x \in \mathbb{F}_{2^\lambda}^n$ 2 : **return** $\langle \boldsymbol{z} \rangle_e^3 := \boldsymbol{\gamma}_x \cdot \Delta \in \mathbb{F}_{2^\lambda}^n$

Fig. 6.21: Auxiliary operations used for constraints

4. The verifier-side procedure OWFVerify.

Auxiliary operations. In Figure 6.21, we have procedures ConstantToVOLE and Deg2To3. These are given in two versions, one for the signer and one for the verifier.

- Signer.ConstantToVOLE($\mathbf{x} \in \mathbb{F}_2^n$): Produces a degree-1 commitment $\langle \mathbf{x} \rangle_2^1$ in $\mathbb{F}_{2^\lambda}^n \times \mathbb{F}_2^n$, effectively storing the bit-string $\mathbf{x}$ in the second component.
- Conversely, Verifier.ConstantToVOLE($\mathbf{x}$) multiplies $\mathbf{x}$ by the global difference $\Delta \in \mathbb{F}_{2^\lambda}$ and so obtains an $\mathbb{F}_{2^\lambda}$-commitment from the verifier's standpoint.
- Deg2To3 extends a degree-2 commitment (resp. from the signer or verifier) to a degree-3 commitment, placing zeros in the unneeded coefficients. This is used when a procedure generates a deg ≤ 2 polynomial in $\mathbb{F}_{2^\lambda}[x]$ and we wish to embed it consistently in a deg ≤ 3 polynomial for the QuickSilver proof. The signer and verifier each have a slightly different way of "shifting" such polynomials.

ZK.OWFConstraints($\langle \mathbf{w} \rangle, \mathsf{pk}$) *(Figure 6.22).* This is the main algorithm that combines all sub-constraints, from key-schedule and encryption. Concretely:

1. It parses $\mathsf{pk} = (\mathbf{x}, \mathbf{y})$ and allocates a new $\langle \boldsymbol{z} \rangle_\lambda^3$ for the accumulated constraints.
2. It first appends a small keyspace-reduction constraint $\langle \mathbf{w}[0] \rangle \times \langle \mathbf{w}[1] \rangle$ using Deg2To3).
3. If EM = `true`, it interprets $\mathbf{x}$ as an AES/Rijandael key, runs KeyExpansion, and then sets $\langle \mathbf{k} \rangle$,$\langle \mathbf{in} \rangle$, $\langle \mathbf{out}_0 \rangle$ accordingly. Otherwise, it calls ConstantToVOLE($\mathbf{x}$) to get a bitwise-committed input block $\langle \mathbf{in} \rangle$ and similarly for $\mathbf{y}[0..127]$. For the key schedule, it calls KeyExpCstrnts$\big(\langle \mathbf{w} \rangle\big)$, obtaining both round-key commitments $\langle \mathbf{k} \rangle$ and a degree-2 constraint vector $\langle \tilde{\boldsymbol{z}} \rangle_\lambda^2$. This is appended to the main buffer by calling Deg2To3.
4. For each encryption block $b \in \{0, 1\}$ (depending on β), it extracts the portion of $\langle \mathbf{w} \rangle$ dedicated to that block's encryption bits, possibly flips the first plaintext bit if $b = 1$ (xor with 1), and calls EncCstrnts$(\langle \mathbf{in} \rangle, \langle \mathbf{out}_b \rangle, \dots)$ to produce the final encryption constraints. These are all collected and appended into the same $\langle \boldsymbol{z} \rangle_\lambda^3$.
5. Finally, the algorithm returns $\langle \boldsymbol{z} \rangle_\lambda^3$.

ZK.OWFProve, ZK.OWFVerify *(Figure 6.23).* The first procedure, ZK.OWFProve, described the signer's algorithm:

1. It converts each bit $\mathbf{w}[i]$ into a VOLE-based commitment $\langle \mathbf{w}[i] \rangle$, using $\mathbf{V}$ for the tags; it parses $\mathbf{u}$ mask elements $\{u_j^*\}$ and $\{v_j^*\}$.
2. The, it invokes OWFConstraints($\langle \mathbf{w} \rangle, \mathsf{pk}$), which returns a polynomial $\langle \boldsymbol{z} \rangle_\lambda^3$ of degree ≤ 3. This is parsed as three $\mathbb{F}_{2^\lambda}^C$ vectors $\boldsymbol{a}_0, \boldsymbol{a}_1, \boldsymbol{a}_2$.
3. Finally, it generates the QuickSilver response $\tilde{a}_0, \tilde{a}_1, \tilde{a}_2$ by hashing $\boldsymbol{a}_j \| (\text{masks})$ with chall_2. The values $\tilde{a}_0, \tilde{a}_1, \tilde{a}_2$ are then returned as the final proof message.

ZK.OWFConstraints($\langle \mathbf{w} \rangle, \mathsf{pk}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)

INPUT: Committed witness $\mathbf{w} \in \mathbb{F}_2^{\ell}$

OUTPUT: Committed constraints $\langle \boldsymbol{z} \rangle_{\lambda}^{3}$, for $\boldsymbol{z} \in \mathbb{F}_{2^\lambda}^{C}$ (will check that $\boldsymbol{z} = 0$)

Parse $\mathsf{pk} = (\mathbf{x}, \mathbf{y}) \in \{0,1\}^{N_{\text{st-bits}}} \times \{0,1\}^{N_{\text{st-bits}}\beta}$

// Allocate space for constraint commitments

new $\langle \boldsymbol{z} \rangle_{\lambda}^{3}$, $\boldsymbol{z} \in \mathbb{F}_{2^\lambda}^{\leq C}$

// constraint to reduce keyspace by 1 bit

$\langle \boldsymbol{z} \rangle_{\lambda}^{3} := \langle \boldsymbol{z} \rangle_{\lambda}^{3} \parallel \mathsf{Deg2To3}(\langle \mathbf{w}[0] \rangle \cdot \langle \mathbf{w}[1] \rangle)$

// Define committed **in**, **out** for EM compatibility

if EM = `true` **then**

  $(\mathbf{k}_0, \ldots, \mathbf{k}_R) \leftarrow \mathsf{KeyExpansion}(\mathbf{x}; \mathtt{param}_{\mathsf{OWF}})$ // $32N_{\text{st}}$-bit round key $\mathbf{k}_i$

  $(\langle \mathbf{k}_0 \rangle, \ldots, \langle \mathbf{k}_R \rangle) \leftarrow \mathsf{ConstantToVOLE}(\mathbf{k}_0, \ldots, \mathbf{k}_R)$

  $\langle \mathbf{in} \rangle \leftarrow \langle \mathbf{w}[0..\lambda) \rangle$ // Rijndael message block from secret key

  $\langle \mathbf{out}_0 \rangle \leftarrow \langle \mathbf{w}[0..\lambda) \rangle + \mathbf{y}$ // Rijndael output block from public key

else

  $\langle \mathbf{in} \rangle := \mathsf{ConstantToVOLE}(\mathbf{x})$ // convert each bit separately

  $\langle \mathbf{out}_0 \rangle \leftarrow \mathsf{ConstantToVOLE}(\mathbf{y}[0..127])$

  if $\beta = 2$ **then** $\langle \mathbf{out}_1 \rangle := \mathsf{ConstantToVOLE}(\mathbf{y}[128..255])$

  $(\langle \tilde{\boldsymbol{z}} \rangle_{\lambda}^{2}, \langle \mathbf{k} \rangle_{2}^{1}) := \mathsf{AES.KeyExpCstrnts}(\langle \boldsymbol{w}[0] \rangle, \ldots, \langle \boldsymbol{w}[0..\ell_{\mathsf{ke}}) \rangle; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$

  $\langle \boldsymbol{z} \rangle_{\lambda}^{3} := \langle \boldsymbol{z} \rangle_{\lambda}^{3} \parallel \mathsf{Deg2To3}(\langle \tilde{\boldsymbol{z}} \rangle_{\lambda}^{2})$ // Saving constraints

for $b \in [0, \beta)$ **do**

  $\langle \tilde{\mathbf{w}} \rangle := \langle \mathbf{w}[\ell_{\mathsf{ke}} + b\ell_{\mathsf{enc}}..\ell_{\mathsf{ke}} + (b+1)\ell_{\mathsf{enc}} - 1] \rangle$ // Selecting encryption bits

  if $b = 1$ **then** $\langle \mathbf{in}[0] \rangle \leftarrow \langle \mathbf{in}[0] \rangle + 1$

  $\langle \tilde{\boldsymbol{z}} \rangle_{\lambda}^{3} := \mathsf{AES.EncCstrnts}(\langle \mathbf{in} \rangle, \langle \mathbf{out}_b \rangle, \langle \tilde{\mathbf{w}} \rangle, \langle \mathbf{k}_0 \rangle, \ldots, \langle \mathbf{k}_R \rangle; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$

  $\langle \boldsymbol{z} \rangle_{\lambda}^{3} := \langle \boldsymbol{z} \rangle_{\lambda}^{3} \parallel \langle \tilde{\boldsymbol{z}} \rangle_{\lambda}^{3}$ // Appending values

return $\langle \boldsymbol{z} \rangle_{\lambda}^{3}$

Fig. 6.22: Proof of one-way function constraints

The algorithm OWFVerify is the corresponding verifier procedure. It reconstructs the commitments values $\langle \mathbf{w}[i] \rangle$ from $\mathbf{d}[i]$ and $\mathbf{Q}$. It recomputes the constraints $\langle \boldsymbol{z} \rangle_{\lambda}^{3}$ using the function OWFConstraints($\langle \mathbf{w} \rangle, \mathsf{pk}$), and checks consistency of the hashes with the prover's $\tilde{a}_1, \tilde{a}_2$. In particular, it computes $\tilde{q} = \mathsf{ZKHash}(\mathsf{chall}_2, \boldsymbol{b} \| q^*)$, then returns $\tilde{q} - \tilde{a}_1 \Delta - \tilde{a}_2 \Delta^2$. If the proof is correct, this final expression should be the same as $\tilde{a}_0$.

ZK.OWFProve($\mathbf{w}, \mathbf{u}, \mathbf{V}, \mathsf{pk}, \mathsf{chall}_2; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)

INPUT: $\mathbf{w} \in \{0,1\}^{\ell}, \mathbf{u} \in \{0,1\}^{2\lambda}, \mathbf{V} \in \{0,1\}^{(\ell+2\lambda)\times\lambda}$ ...
OUTPUT: $\tilde{a}_0, \tilde{a}_1, \tilde{a}_2 \in \mathbb{F}^3_{2^\lambda}$

1 : **for** $i \in [0..\ell)$ **do**
2 : Let $\langle \mathbf{w}[i] \rangle := (\mathbf{w}[i], \mathsf{ToField}(\mathbf{V}|_i; \lambda))$
3 : // embed VOLE masks
4 : **for** $i \in [0..2\lambda)$ **do**
5 : $\boldsymbol{v}[i] := \mathsf{ToField}(\mathbf{V}|_{\ell+i}; \lambda)$
6 : **for** $j \in [0..1]$ **do**
7 : $u^*_j := \mathsf{ToField}(\mathbf{u}[j\lambda..(j+1)\lambda); \lambda)$
8 : // $\alpha_\lambda \in \mathbb{F}_{2^\lambda}$, basis element over $\mathbb{F}_2$
9 : $v^*_j := \sum_{i\in[0..\lambda)} \boldsymbol{v}[i+\lambda j]\alpha^i_\lambda$
10 : // Compute the constraints
11 : // (poly $\boldsymbol{a}_0 + \boldsymbol{a}_1 x + \boldsymbol{a}_2 x^2 + \boldsymbol{z}x^3$; $\boldsymbol{z} = 0$)
12 : $\langle \boldsymbol{z} \rangle^3_\lambda \leftarrow \mathsf{OWFConstraints}(\langle \mathbf{w} \rangle, \mathsf{pk})$
13 : Parse $\langle \boldsymbol{z} \rangle^3_\lambda$ as $(\boldsymbol{a}_0, \boldsymbol{a}_1, \boldsymbol{a}_2) \in (\mathbb{F}^C_{2^\lambda})^3$
14 : $\tilde{a}_0 := \mathsf{ZKHash}(\mathsf{chall}_2, (\boldsymbol{a}_0 \| u^*_0))$
15 : $\tilde{a}_1 := \mathsf{ZKHash}(\mathsf{chall}_2, (\boldsymbol{a}_1 \| v^*_0 + u^*_1))$
16 : $\tilde{a}_2 := \mathsf{ZKHash}(\mathsf{chall}_2, (\boldsymbol{a}_2 \| v^*_1))$
17 : **return** $(\tilde{a}_0, \tilde{a}_1, \tilde{a}_2)$

ZK.OWFVerify($\mathbf{d}, \mathbf{Q}, \mathsf{pk}, \mathsf{chall}_2, \mathsf{chall}_3, \tilde{a}_1, \tilde{a}_2; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)

INPUT: $\mathbf{d} \in \{0,1\}^{\ell}, \mathbf{Q} \in \{0,1\}^{(\ell+2\lambda)\times\lambda}$ ...
OUTPUT: $\tilde{a}_0 \in \mathbb{F}_{2^\lambda}$

1 : $\Delta := \mathsf{ToField}(\mathsf{chall}_3; \lambda)$
2 : **for** $i \in [0..\ell)$ **do**
3 : Let $\langle \mathbf{w}[i] \rangle := \mathsf{ToField}(\mathbf{Q}|_i; \lambda) + \mathbf{d}[i] \cdot \Delta$
4 : // embed VOLE masks
5 : **for** $i \in [0..2\lambda)$ **do**
6 : $\boldsymbol{q}[i] := \mathsf{ToField}(\mathbf{Q}|_{\ell+i}; \lambda)$
7 : **for** $j \in [0..1]$ **do**
8 : // $\alpha_\lambda \in \mathbb{F}_{2^\lambda}$, basis element over $\mathbb{F}_2$
9 : $q^*_j := \sum_{i\in[0..\lambda)} \boldsymbol{q}[i+\lambda j]\alpha^i_\lambda$
10 : $q^* := q^*_0 + \Delta \cdot q^*_1$
11 : // Compute the constraints
12 : $\langle \boldsymbol{z} \rangle^3_\lambda \leftarrow \mathsf{OWFConstraints}(\langle \mathbf{w} \rangle, \mathsf{pk})$
13 : Parse $\langle \boldsymbol{z} \rangle^3_\lambda$ as $\boldsymbol{b} \in \mathbb{F}^C_{2^\lambda}$
14 : $\tilde{q} := \mathsf{ZKHash}(\mathsf{chall}_2, (\boldsymbol{b} \| q^*))$
15 : // Reproduce prover's $\tilde{a}_0$
16 : **return** $\tilde{q} - \tilde{a}_1 \cdot \Delta - \tilde{a}_2 \cdot \Delta^2$

Fig. 6.23: Signer and verifier algorithms for the ZK proof of the one-way function.

7 The FAEST Signature Scheme

In this section we present the principal algorithms of the FAEST signature scheme:

- Section 7.1: Key Generation algorithm FAEST.KeyGen in Figure 7.1
- Section 7.2: Signing algorithm FAEST.Sign in Figure 7.2

FAEST.KeyGen($\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)

1 : **do**
2 : $\quad \mathbf{k} \leftarrow \{0,1\}^{\lambda}$
3 : **while** $\mathbf{k}[0] \wedge \mathbf{k}[1] \neq 0$
4 : $\mathbf{x} \leftarrow \{0,1\}^{N_{\text{st-bits}}}$
5 : $\mathbf{y} := F_{\mathbf{k}}(\mathbf{x})$
6 : $\mathsf{sk} := \mathbf{k}$
7 : $\mathsf{pk} := (\mathbf{x}, \mathbf{y})$
8 : **return** $(\mathsf{sk}, \mathsf{pk})$

Fig. 7.1: FAEST key generation algorithm

– Section 7.3: Verification algorithm FAEST.Verify in Figure 7.3.

7.1 Key Generation

The key generation algorithm takes as input one of the parameter sets described in Table 3.2 via $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$, which fixes the security parameter λ, the output length $128 \cdot \beta$ to 128 (i.e., $\beta = 1$) if $\lambda = 128$ and to 256 if $\lambda \in \{192, 256\}$, and the OWF. The algorithm, given in Figure 7.1, is almost the same for all parameter sets.

The algorithm samples a candidate $\mathbf{k} \in \{0,1\}^{\lambda}$ until the two least significant bits of $\mathbf{k}$ (denoted $\mathbf{k}[0]$ and $\mathbf{k}[1]$) are both 1. This is a technical artifact used in the security proof, in particular this implies there exist some public keys pk for which no valid $\mathbf{k}$ can produce them. Once a valid $\mathbf{k}$ is found, the algorithm generates a uniformly random $\mathbf{x} \in \{0,1\}^{N_{\text{st-bits}}}$, then encrypts $\mathbf{x}$ under $\mathbf{k}$ to obtain $\mathbf{y} = F_{\mathbf{k}}(\mathbf{x})$. The secret key is $\mathsf{sk} = \mathbf{k}$, and the public key is $\mathsf{pk} = (\mathbf{x}, \mathbf{y})$.

7.2 Signing

The algorithm uses some procedures defined in previous sections. In particular, hash functions H_i, as described in Section 3.3. The challenges chall_j, $j \in \{1,2,3\}$, are obtained using random oracle H_2^j. The signing algorithm runs in 4 phases, defined by the generation of the challenges.

In phase 1, the signing procedure (Figure 7.2) starts by initializing an integer counter ctr to zero. It will be used later for the so-called "grind check." Then, it samples $\rho \in \{0,1\}^{\lambda}$ as signature randomness; in fully deterministic signing scenarios, ρ may simply be set to 0^{λ}. Next, it uses H_2^0 to hash together the public key pk and the message msg, producing $\mu \in \{0,1\}^{2\lambda}$. The signer then combines its secret key sk, the digest μ, and the randomness ρ as input to H_3. This call yields a λ-bit PRG seed r and a temporary 128-bit string $\mathsf{iv}^{\mathsf{pre}}$. Another hash, $\mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}})$, is then invoked to produce iv, a 128-bit IV which seeds the AES-based PRG. Using (r, iv) as input, the signer commits to all VOLE instances via a single call to FAEST.VOLECommit. This call encapsulates all the small-VOLE vectors into one large GGM-based batch vector commitment: it returns (1) a commitment com, (2) a decommitment decom sufficient to later open most seeds while hiding one per vector, (3) a set of correction vectors $\mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}$ that align each VOLE instance to the same secret, and (4) the secret $\mathbf{u}$ plus MAC tags $\mathbf{V}$. Finally, the signer derives a first challenge chall_1 from $\mu, \mathsf{com}, \{\mathbf{c}_i\}, \mathsf{iv}$, by calling H_2^1 and requesting $5\lambda + 64$ output bits.

In phase 2, the signer prepares $\widetilde{\mathbf{V}}$ and $\widetilde{\mathbf{u}}$ for the VOLE consistency check using VOLEHash with input $\mathbf{V}, \mathsf{chall}_1$ and $\mathbf{u}, \mathsf{chall}_1$, respectively. Next, it generates the extended witness by calling AES.ExtendWitness on sk and $\mathbf{in}$, producing $\mathbf{w}$. To commit to $\mathbf{w}$, the

FAEST.Sign$(\mathsf{msg}, \mathsf{sk}, \mathsf{pk}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}, \mathtt{param}_{\mathsf{VOLE}})$

$\mathsf{ctr} \leftarrow 0$ // 32-bit int
$\rho \leftarrow \{0,1\}^{\lambda}$ // Signature randomness — for deterministic signing, set $\rho := 0^{\lambda}$
$\mu := \mathsf{H}_2^0(\mathsf{pk} \| \mathsf{msg}; 2\lambda)$
$(r, \mathsf{iv}^{\mathsf{pre}}) := \mathsf{H}_3(\mathsf{sk} \| \mu \| \rho)$ // $r \in \{0,1\}^{\lambda}$, $\mathsf{iv}^{\mathsf{pre}} \in \{0,1\}^{128}$
$\mathsf{iv} := \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}})$
// Commitment and VOLE Derandomization
$(\mathsf{com}, \mathsf{decom}, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}, \mathbf{u}, \mathbf{V}) := \mathsf{FAEST.VOLECommit}(r, \mathsf{iv}, \ell + 2\lambda + B; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$
$\mathsf{chall}_1 := \mathsf{H}_2^1(\mu \| \mathsf{com} \| \mathbf{c}_1 \| \ldots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}^{\mathsf{pre}}; 5\lambda + 64)$
// VOLE consistency check and witness commitment
$\widetilde{\boldsymbol{u}} := \mathsf{VOLEHash}(\mathsf{chall}_1, \mathbf{u}) \in \{0,1\}^{\lambda+B}$
$\widetilde{\mathbf{V}} := \mathsf{VOLEHash}(\mathsf{chall}_1, \mathbf{V}) \in \{0,1\}^{(\lambda+B)\times\lambda}$ // hash column-wise
$\mathbf{w} := \mathsf{AES.ExtendWitness}(\mathsf{sk}, \mathbf{in}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}) \in \{0,1\}^{\ell}$ // Extend witness with S-Box outputs
$\mathbf{d} := \mathbf{w} \oplus \mathbf{u}_{[0..\ell)}$ // Mask extended witness
$\mathsf{chall}_2 := \mathsf{H}_2^2(\mathsf{chall}_1 \| \widetilde{\mathbf{u}} \| \widetilde{\mathbf{V}} \| \mathbf{d}; 3\lambda + 64)$
// AES proof
$\mathbf{u} := \mathbf{u}_{[\ell..\ell+2\lambda)}$
$\mathbf{V} := \mathbf{V}_{[0..\ell+2\lambda)}$ // drop last $\lambda + B$ rows (impl. may transpose to row-major order here)
$(\tilde{a}_0, \tilde{a}_1, \tilde{a}_2) := \mathsf{ZK.OWFProve}(\mathbf{w}, \mathbf{u}, \mathbf{V}, \mathsf{pk}, \mathsf{chall}_2; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
do
  $\mathsf{chall}_3 \leftarrow \mathsf{H}_2^3(\mathsf{chall}_2 \| \tilde{a}_0 \| \tilde{a}_1 \| \tilde{a}_2 \| \mathsf{ctr}; \lambda)$
  if $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}} : \lambda) \neq 0^{w_{\mathsf{grind}}}$ **then**
    $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
    continue
  $I \leftarrow \mathsf{DecodeAllChall}_3(\mathsf{chall}_3[0 : \lambda - w_{\mathsf{grind}}))$
  $\mathsf{decom}_I \leftarrow \mathsf{BAVC.Open}(\mathsf{decom}, I)$
while $\mathsf{decom}_I \neq \perp$
// Build VOLE challenge and open Vector Commitments
return $\sigma := ((\mathbf{c}_i)_{i \in [1..\tau)}, \widetilde{\mathbf{u}}, \mathbf{d}, \tilde{a}_1, \tilde{a}_2, \mathsf{decom}_I, \mathsf{chall}_3, \mathsf{iv}^{\mathsf{pre}}, \mathsf{ctr})$

Fig. 7.2: FAEST signing algorithm. For EM variant, replace AES.ExtendWitness and AES.AESProve by Rijndael-EM.ExtendWitness and Rijndael-EM.EMProve

algorithm computes $\mathbf{d} := \mathbf{w} \oplus \mathbf{u}_{[0..\ell)}$, one-time padding $\mathbf{w}$ with $\mathbf{u}$. The signer then derives a second challenge chall_2 by hashing together chall_1, $\widetilde{\mathbf{u}}$, $\widetilde{\mathbf{V}}$, and $\mathbf{d}$.

In phase 3, the signer generates the QuickSilver proof. First, it restricts $\mathbf{u}$ and $\mathbf{V}$ to the $\ell + 2\lambda$ rows that are actually needed for the check, then it calls

$$(\tilde{a}_0, \tilde{a}_1, \tilde{a}_2) \leftarrow \mathsf{ZK.OWFProve}(\mathbf{w}, \mathbf{u}, \mathbf{V}, \mathsf{pk}, \mathsf{chall}_2; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}),$$

which generates the three VOLE tags $\tilde{a}_0, \tilde{a}_1, \tilde{a}_2$.
Using these proof components, the signer creates the final Fiat–Shamir challenge chall_3. It repeatedly calls

$$\mathsf{chall}_3 \;=\; \mathsf{H}_2^3\big(\mathsf{chall}_2 \| \tilde{a}_0 \| \tilde{a}_1 \| \tilde{a}_2 \| \mathsf{ctr}; \lambda\big),$$

checking if the last w_{grind} bits of chall_3 are zero. If they are not, it increments the counter ctr and tries again. Once it has found a chall_3 whose last w_{grind} bits are $0^{w_{\mathsf{grind}}}$, the signer decodes $\mathsf{chall}_3[0 : \lambda - w_{\mathsf{grind}}]$ into an index vector I. That vector specifies which leaf is kept hidden in each VOLE vector. Using its bulk decommitment decom, the signer calls $\mathsf{decom}_I \leftarrow \mathsf{BAVC.Open}(\mathsf{decom}, I)$ to open all but one leaf in each of the τ vectors. If

FAEST.Verify($\mathsf{msg}, \mathsf{pk}, \sigma; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)

Parse $\sigma = ((\mathbf{c}_i)_{i\in[1..\tau)}, \widetilde{\mathbf{u}}, \mathbf{d}, \tilde{a}_1, \tilde{a}_2, \mathsf{decom}_I, \mathsf{chall}_3, \mathsf{iv}^{\mathsf{pre}}, \mathsf{ctr})$
$\mu \leftarrow \mathsf{H}_2^0(\mathsf{pk}\|\mathsf{msg})$
$\mathsf{iv} \leftarrow \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}})$
if $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}} : \lambda) \neq 0^{w_{\mathsf{grind}}}$ **then**
  return false
// Reconstruct VOLEs and check commitment
$(\mathsf{com}, \mathbf{Q}) \leftarrow$ FAEST.VOLEReconstruct$(\mathsf{decom}_I, \mathsf{chall}_3[0 : \lambda - w_{\mathsf{grind}}), \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$
if VOLEReconstruct returned $\perp$ **then**
  return false
$\mathsf{chall}_1 \leftarrow \mathsf{H}_2^1(\mu\|\mathsf{com}\|\mathbf{c}_1\|\ldots\|\mathbf{c}_{\tau-1}\|\mathsf{iv}^{\mathsf{pre}}; 5\lambda + 64)$
// Apply the VOLE correction values and check consistency
$\widetilde{\mathbf{Q}} \leftarrow$ VOLEHash$(\mathsf{chall}_1, \mathbf{Q}) \in \{0,1\}^{(\lambda+B)\times\lambda}$
Parse $\mathsf{chall}_3 = (\delta_0, \ldots, \delta_{\lambda-1}) \in \{0,1\}^{\lambda}$
$\mathbf{D} := \widetilde{\mathbf{Q}} \oplus [\delta_0 \cdot \widetilde{\mathbf{u}} \cdots \delta_{\lambda-1} \cdot \widetilde{\mathbf{u}}]$
$\mathsf{chall}_2 \leftarrow \mathsf{H}_2^2(\mathsf{chall}_1\|\widetilde{\mathbf{u}}\|\mathbf{D}\|\mathbf{d}; 3\lambda + 64)$ // Hash $\mathbf{D}$ in column-major order
// Compute AES consistency values
$\widetilde{a}_0 \leftarrow$ ZK.OWFVerify$(\mathbf{d}, \mathbf{Q}|_{[0..\ell+2\lambda)}, \mathsf{chall}_2, \mathsf{chall}_3, \widetilde{a}_1, \widetilde{a}_2, \mathsf{pk}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
Compute $\mathsf{chall}_3' \leftarrow \mathsf{H}_2^3(\mathsf{chall}_2\|\tilde{a}_0\|\tilde{a}_1\|\tilde{a}_2\|\mathsf{ctr}; \lambda)$
return true if $\mathsf{chall}_3 = \mathsf{chall}_3'$ **else false**

Fig. 7.3: FAEST verification algorithm. For EM variant, replace AES.AESVerify by Rijndael-EM.EMVerify

this opening algorithm returns $\perp$ (indicating the partial opening would exceed the T_{open} bound in Figure 5.4), the signer increments ctr and checks again the challenge grinding. Otherwise, it finalizes decom_I.

Finally, in phase 4, it generates the signature σ which consists of:

- The $\tau - 1$ correction vectors $\{\mathbf{c}_i\}_{i=1}^{\tau-1}$;
- The hashed VOLE secret $\widetilde{\mathbf{u}}$;
- The commitment to the extended witness $\mathbf{d}$;
- The QuickSilver proof parts $\tilde{a}_1, \tilde{a}_2$;
- The partial decommitment decom_I;
- The last challenge chall_3;
- The 128-bit string $\mathsf{iv}^{\mathsf{pre}}$;
- The final counter ctr.

7.3 Verification

The verification algorithm FAEST.Verify is shown in Figure 7.3. The verifier begins by parsing the signature σ into its components: the correction vectors $\{\mathbf{c}_i\}$, the hashed VOLE secret $\widetilde{\mathbf{u}}$, the witness commitment $\mathbf{d}$, the QuickSilver proof pieces $\tilde{a}_1, \tilde{a}_2$, the partial opening decom_I, the final challenge chall_3, the bit-string $\mathsf{iv}^{\mathsf{pre}}$, and the grind counter ctr. It then re-computes $\mu := \mathsf{H}_2^0(\mathsf{pk}\|\mathsf{msg})$ and obtains $\mathsf{iv} = \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}})$. If chall_3's last w_{grind} bits are not zero, the verifier aborts immediately, as it implies the signer did not correctly perform the grinding step.

Otherwise, the verifier calls

$$(\mathsf{com}, \mathbf{Q}) \leftarrow \text{FAEST.VOLEReconstruct}\big(\mathsf{decom}_I,\ \mathsf{chall}_3[0 : \lambda - w_{\mathsf{grind}}], \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}\big),$$

which reconstructs all leaves except those indicated by the index vector embedded in chall_3. This procedure will internally re-expand the GGM tree from the partial seeds,

check that each VOLE instance is consistent with its correction vector $\mathbf{c}_i$, and finally produce the VOLE commitment com along with a re-constructed matrix $\mathbf{Q}$ of VOLE tags. If VOLEReconstruct fails or returns $\perp$, the verifier rejects. Otherwise, it re-computes the first challenge $\mathsf{chall}_1 := \mathsf{H}_2^1(\mu \| \mathsf{com} \| \mathbf{c}_1 \| \dots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}; 5\lambda + 64)$.

Next, the verifier hashes $\mathbf{Q}$ to $\widetilde{\mathbf{Q}}$ via VOLEHash with input chall_1. It then interprets the bits of chall_3 (beyond the final w_{grind}) as $\delta_0, \dots, \delta_{\lambda-1}$ and calculate $\mathbf{D} := \widetilde{\mathbf{Q}} \oplus [\delta_0 \cdot \widetilde{\mathbf{u}} \cdots \delta_{\lambda-1} \cdot \widetilde{\mathbf{u}}]$. This permits to reconstructs

$$\mathsf{chall}_2 := \mathsf{H}_2^2(\mathsf{chall}_1 \| \widetilde{\mathbf{u}} \| \mathbf{D} \| \mathbf{d}; 3\lambda + 64).$$

In the final step, the verifier checks the QuickSilver proof. It derives

$$\tilde{a}_0 \leftarrow \mathsf{ZK.OWFVerify}\big(\mathbf{d}, \mathbf{Q}|_{[0..\ell+2\lambda)}, \mathsf{chall}_2, \mathsf{chall}_3, \tilde{a}_1, \tilde{a}_2, \mathsf{pk}\big),$$

recovering the missing piece $\tilde{a}_0$. With all three proof elements, the verifier is able to compute $\mathsf{chall}_3' \leftarrow \mathsf{H}_2^3(\mathsf{chall}_2 \| \tilde{a}_0 \| \tilde{a}_1 \| \tilde{a}_2 \| \mathsf{ctr}; \lambda)$. If chall_3' does not match the provided chall_3, the verifier rejects; otherwise, it accepts.

8 Performance Analysis

We provide two implementations of FAEST as part of the proposal. The first is a reference implementation in standard C, while the second is an architecture-specific C++ implementation with optimizations aimed at x86-64 architectures with the AVX2 and AES-NI instruction set extensions, as commonly seen in Intel and AMD CPUs. Note that the "optimized implementation" required by NIST is identical to the reference implementation. Both implementations were built using the eXtended Keccak Code Package (XKCP) and OpenSSL libraries.

We used a few techniques worth mentioning in the x86-64 implementation. To hash efficiently, especially for grinding, we used a feature of XKCP to run four Keccak permutations in parallel, with instruction level parallelism. For converting $\mathbf{V}$ and $\mathbf{Q}$ from column to row major order, we used an extension of Eklundh's matrix transposition algorithm [TE76]. Our use of AES as a PRG is limited by only generating a few blocks of output for each key, and most implementations of the AES key schedule are relatively inefficient in this setting. To improve the performance of our PRGs, we adapted the approach of Gueron et al. [GLNP15] to run four AES key schedules in parallel with vectorization. Finally, for ConvertToVOLE we adapted an efficient algorithm for parity checking Hamming codes [LLH20].[16]

Benchmarking Setup. We measured the performance of both implementations using a single core of a workstation running a AMD Zen 3 Ryzen 9 5950X processor at 3.4 GHz (with clock boosting disabled) and 128 GiB memory. The system was otherwise idle (load average 0.01), so while Simultaneous Multi-Threading was enabled it likely did not affect the results significantly. Each individual test can be run with memory usage below 19 MiB. The computer was running Linux 6.6.40, and the implementations were built with GCC 14.1.1.

Performance. Table 8.1 shows the performance of the AVX2 implementation. Runtimes of the respective algorithms are given in milliseconds and CPU cycles. Times for the AVX2 implementation were averaged over 10000 runs, while times for the reference implementation were averaged over 100 runs.

[16] The paper presents the algorithm as an encoding algorithm for Hadamard codes, the dual of Hamming codes. The transpose of this algorithm calculates the parity for a Hamming code.

Scheme	Runtimes in ms, Mcyc.						Sizes in B		
	KeyGen		Sign		Verify		sk	pk	Signature
FAEST-128s	0.002	0.005	3.761	12.787	2.877	9.783	32	32	4 506
FAEST-128f	0.002	0.005	0.507	1.722	0.415	1.413	32	32	5 924
FAEST-192s	0.003	0.011	16.084	54.687	12.438	42.290	40	48	11 260
FAEST-192f	0.003	0.011	2.072	7.045	1.788	6.079	40	48	14 948
FAEST-256s	0.004	0.013	22.450	76.330	21.925	74.546	48	48	20 696
FAEST-256f	0.004	0.013	3.256	11.071	3.012	10.241	48	48	26 548
FAEST-EM-128s	0.002	0.005	2.766	9.403	2.176	7.398	32	32	3 906
FAEST-EM-128f	0.002	0.005	0.413	1.404	0.327	1.113	32	32	5 060
FAEST-EM-192s	0.003	0.009	11.553	39.282	10.659	36.239	48	48	9 340
FAEST-EM-192f	0.003	0.009	1.523	5.177	1.372	4.665	48	48	12 380
FAEST-EM-256s	0.004	0.013	18.372	62.465	17.570	59.738	64	64	17 984
FAEST-EM-256f	0.004	0.013	2.775	9.436	2.566	8.725	64	64	23 476

Table 8.1: Benchmark results for the architecture specific implementation for x86-64 with AVX2.

Scheme	Runtimes in ms, Mcyc.						Sizes in B		
	KeyGen		Sign		Verify		sk	pk	Signature
FAEST-128s	0.001	0.005	98.843	336.067	93.749	318.747	32	32	4 506
FAEST-128f	0.001	0.005	23.873	81.169	20.433	69.474	32	32	5 924
FAEST-192s	0.002	0.005	362.406	1 232.180	324.491	1 103.269	40	48	11 260
FAEST-192f	0.002	0.005	107.859	366.721	85.746	291.537	40	48	14 948
FAEST-256s	0.002	0.005	771.523	2 623.178	742.455	2 524.347	48	48	20 696
FAEST-256f	0.002	0.005	175.249	595.847	147.233	500.592	48	48	26 548
FAEST-EM-128s	0.002	0.005	42.411	144.196	37.473	127.409	32	32	3 906
FAEST-EM-128f	0.002	0.005	15.233	51.794	11.865	40.342	32	32	5 060
FAEST-EM-192s	0.028	0.096	135.106	459.360	117.702	400.187	48	48	9 340
FAEST-EM-192f	0.028	0.096	61.360	208.624	45.368	154.252	48	48	12 380
FAEST-EM-256s	0.046	0.158	217.212	738.521	187.992	639.173	64	64	17 984
FAEST-EM-256f	0.046	0.158	105.891	360.029	78.538	267.029	64	64	23 476

Table 8.2: Benchmark results for the reference implementation.

9 Security Evaluation

In this section, we evaluate the security of FAEST. We start by describing the best known concrete attacks against the scheme, and then present formal security analysis in both the random oracle and quantum random oracle models.

9.1 Security Against Known Attacks

We now analyze the security of FAEST with respect to known attacks.

The cryptographic primitives used in FAEST are the one-way functions (OWF) based on AES or EM-AES, as well as the AES-CTR PRG and the SHA3 hash functions SHAKE-128 and SHAKE-256 (Section 3.3). While we, in fact, exploit the PRG security of these primitives, we will call them one-way functions here to distinguish them from the other PRGs used, and for consistency with MPC-in-the-head based signature constructions. In Section 9.2, we give a detailed analysis of the security of the OWF constructions. The other primitives have all been previously standardized by NIST (SP 800-90A for the AES-CTR PRG, and FIPS 202 for SHA3), have received a large amount of scrutiny and are used ubiquitously in applications today. There are no known attacks on the pseudo-randomness of AES-CTR, nor on the pseudo-randomness or collision-resistance of SHA3, that perform much better than exhaustive search. In our security proof, we also model the SHA3 hash functions we use as random oracles. This is a standard practice in security analysis of cryptography, and it is widely believed that instantiating random oracles using SHA3 — with appropriate domain separation — does not lead to any security weaknesses in typical protocols that do not make any non-black-box use of the hash function.

9.1.1 Brute Force the Public Key. The simplest attack strategy is to attempt to invert the OWF output given in the public key, in order to recover the signing key. This is essentially a key recovery attack on AES, given one or two ciphertext blocks. A couple of slight differences must be accounted for, however. Firstly, in our case, since KeyGen uses rejection sampling to sample a key such that the first two bits are not both 1, the effective key space is reduced slightly in size. Secondly, since there are only one or two blocks of AES output in the public key, it is possible that there are collisions, and the attacker could find a different key k' that is still a preimage of the OWF. As discussed in Section 9.2.4, this has only a minimal impact on security, and we estimate the attack cost for the FAEST and FAEST-EM instances to be between $2^{\lambda-2}$ and $2^{\lambda-1}$ evaluations of AES-λ. So, our OWFs lose only around 1–2 bits of security, compared with standard AES encryption.

9.1.2 Brute Force the PRG. Another way to try to recover the signing key is to attack the pseudo-random generator PRG, used in the BAVC.Commit and ConvertToVOLE procedures. As described in Section 3.3, PRG is built using AES-CTR at the λ-bit security level, with a random, per-signature IV and a tweak that varies at different places in the signature. Suppose an attacker is attempting to recover the secret key, given just one signature. If the same $(\mathsf{iv}, \mathsf{twk})$ pair is used in several PRG calls, an attacker may attempt to mount a multi-target attack, where a single key guess is tested for correctness with n possible candidates. If testing correctness with respect to all n candidates is cheaper than computing n AES-CTR outputs, then this can be cheaper than a naive brute-force attack. This is a type of time/space tradeoff attack, which has been explored in other post-quantum signature schemes like Picnic [DN19].

We argue that FAEST is not susceptible to such attacks, because of the way the tweaks are chosen. First, consider the length-doubling tweakable PRGs (tPRGs) used in the tree-based vector commitment algorithm, BAVC.Commit. At each level of the tree, a signature

reveals all-but-one of the keys corresponding to the nodes at that level. Given a candidate guess g for a missing key, the guess can be verified by expanding g into two new keys using PRG, and then comparing these keys with the corresponding known left/right keys in the tree. If there is a match, it's likely (unless we have found a collision in one half of the PRG output) that g is the correct missing key and allows recovering the missing leaf, which then leaks the secret witness. A second place[17] PRG is used is in the ConvertToVOLE algorithm, where each seed sd_i is expanded, and then the sum of these outputs forms the vector $\mathbf{u}$ used to mask the witness. Since all-but-one of the seeds are known, this tPRG may be attacked in a similar way, since given any guess g and the first block of AES-CTR under key g, one gets a candidate signing key k that can be tested for. As different tweaks are used for each of the described invocations of PRG, a multi-target attack against them is not possible. We conclude that the cost of key recovery through these uses of the PRG is the same as that of attacking a single target of a PRG based on AES-CTR, which requires e.g. $\approx 2^\lambda$ AES evaluations in the classical setting to obtain a success probability close to 1.

Given Multiple Signatures. Suppose the attacker is given up to Q_{sig} signatures. Since each signature uses an independent, random $\mathsf{iv} \in \{0,1\}^{128}$, we expect all IVs to be unique with good probability, as long as $Q_{\mathsf{sig}} < 2^{64}$. If the IVs are unique then we are not aware of any attacks that exploit having multiple signatures and perform better than the single signature attack described above. If there are a small number of collisions, say c, then the attack's complexity will be reduced by a factor of c, since there are c times as many PRG targets with the same IV to attack. However, since collisions are unlikely to happen and out of the control of the attacker, having a large number of signatures does not seem to help the attacker here. The iv and twk are separate inputs of PRG, so multi-target attacks combining different calls and multiple signatures are also prevented.

9.1.3 Attack Soundness of the ZK Protocol. Instead of directly recovering the signing key, an attacker may attempt to forge a signature by violating the soundness property of the underlying interactive proof, using an invalid witness with the public key. Soundness is covered in our EU-KO security proof in Theorem 9.24. This proof is tight, up to a small constant factor, and we now explain why the individual terms in the adversary's advantage do not lead to any effective attacks when our concrete SHA3-based hash functions are modeled as random oracles.

The first two summands in soundness bound of Theorem 9.24, which grow quadratically in the random oracle queries, correspond to finding a collision in a random oracle with at least 2λ output bits, which requires around 2^λ queries to succeed. Since we use SHAKE instead of a random oracle, this is equivalent to a generic collision search on SHAKE. The third summand is related to the leaf commitments. For FAEST, it is $\tau Q_0 \cdot 2^{2\lambda} \cdot \varepsilon_{\mathsf{uhash}} = \tau Q_0 \cdot 2^{-\lambda}$, which is the probability that a universal hash does not lead to an injective leaf commitment. For a random oracle, any attack boils down to exhaustive search requiring $2^\lambda/\tau \geq 2^{\lambda-5}$ evaluations. For FAEST-EM, we make a stronger assumption (Definition 9.13), which cannot be efficiently verified. We discuss its plausibility in Remark 9.14; in short, we are not aware of a better attack than a collision search. The last summand in the soundness advantage bound is the round-by-round soundness of the proof system. This can only be broken by forcing the output of SHAKE to lie within a set of outputs (i.e. challenges) which is a $3/2^\lambda$ fraction of all outputs. For a random oracle, any attack boils down to exhaustive search requiring $2^\lambda/3 \geq 2^{\lambda-2}$ evaluations. Note that, since SHAKE is more costly to evaluate than AES, none of these attacks are worthwhile for an attacker

[17] PRG is also used to derive the randomness r_i in VOLECommit, however, this does not seem to permit a multi-target attack, since there's no way to verify a guess for r_i without performing another PRG call.

compared with directly inverting the one-way function in the public key to recover the secret key. (The advantage term related to pseudorandomness of the one-way function in Theorem 9.24 is due to our proof technique and related to tightly secure signatures following the approach of Katz and Wang [KW03].)

9.1.4 Multi-User Attacks. We have also taken steps to ensure that FAEST remains secure in a multi-user setting, where many different signers are using the same signing algorithms but with independently generated public keys. We first consider the setting where an attacker has access to a large number of public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_N$, and wishes to recover a single secret key sk_i. Since each pk_i uses an independently sampled $\mathbf{x}$ in KeyGen, which defines the OWF $F_{(\cdot)}(\mathbf{x})$, there is no way to perform a multi-target attack (as described above in the single user setting) across all N OWF instances. Indeed, any attempt at exhaustive search must be based on a value $\mathbf{x}$ for a specific user, so having multiple instances does not help carry out the attack. Another countermeasure that helps prevent this type of attack is the random IVs sampled for each signature, which again ensure independence of the PRGs used across different signatures and public keys.

Another type of multi-user attack is a *key substitution attack* [MS04], where the attacker is given a signature σ under some public key pk, and tries to find a signature σ' that verifies under another public key pk' for the same message. In FAEST, we avoid this type of attacking by hashing the public key together with the message to obtain the hash μ. This uniquely binds the public key to each signature, preventing key substitution.

9.2 Concrete Analysis of AES as a OWF

The security proof for FAEST relies on the *PRG security* of OWF. The FAEST OWF is AES counter mode encryption of zero. As such, pseudorandom function security is well-tested and implies the PRG security we require. For FAEST-EM, the security of the Even Mansour cipher as a pseudorandom permutation in the ideal permutation model (which also holds in the quantum-accessible ideal permutation model [ABKM22]) implies the PRG security we require, in the same idealized model.

In practice, even though our proofs assume pseudorandomness of these functions, we are not aware of any attack strategy for forging a signature without actually inverting the functions to recover the secret key. Below, we therefore also include some formal analysis of the one-wayness of our two one-way function instantiations.

9.2.1 AES with Key Expansion. In [CDG+17], Chase et al. formally show that it is possible to use a block cipher, with key size equal to the block size and viewed as a PRF, to instantiate a OWF. Similarly, we show that our F is a OWF based on it being the concatenation of 1 or more calls to a PRP.

Lemma 9.1. *If $P\colon \mathcal{K} \times \mathcal{M} \to \mathcal{M}$ is a PRP, then for all sufficiently high $\beta < |\mathcal{M}|/2$, $F((x_0, \ldots, x_{\beta-1}), k) = (P(k, x_0), \ldots, P(k, x_{\beta-1}))$ is a OWF. Concretely, any adversary $\mathcal{A}$ against the OWF F can be turned into an adversary $\mathcal{A}'$ against the PRP P, such that*

$$\mathsf{AdvOWF}^{F}_{\mathcal{A}} \leq \left(1 + B_\beta \frac{|\mathcal{K}|}{|\mathcal{M}|^\beta}\right) \mathsf{AdvPRP}^{P}_{\mathcal{A}'} + B_\beta \frac{|\mathcal{K}|}{|\mathcal{M}|^\beta \, (|\mathcal{M}| - \beta)_\beta},$$

where B_β is the βth Bell number and $(x)_y = x(x-1)\cdots(x-y+1)$ is the falling factorial. $\mathcal{A}'$ makes at most 2β queries to the PRP oracle, and takes computation similar to $\mathcal{A}$ plus an extra β PRP evaluations.

Proof. We present a sequence of games, and bound how much the advantage can decrease from one game to the next. Each game is a randomized algorithm that interacts with the adversary, then outputs a rational number representing the adversary's payoff. The advantage is the expected value of the payoff.[18] We will then complete the hybrid proof by bounding the advantage of the final game.

G_0: This is the OWF game: $k \leftarrow \mathcal{K}$ and $x_i \leftarrow \mathcal{M}$ are all sampled independently, and $\mathcal{A}$ is given x_i and $y_i = P(k, x_i)$ for all i. The game outputs 1 if $\mathcal{A}$ returns k' such that $P(k', x_i) = y_i$ for all i, and 0 otherwise.

G_1: Instead of evaluating $y_i = P(k, x_i)$, sample all $y_i \leftarrow \mathcal{M}$ uniformly, subject to $y_i = y_j \iff x_i = x_j$ for all i, j. This change reduces directly to the security of the PRP, because k is only used for computing the y_i.

G_2: Reintroduce the variable k, and again sample it uniformly as $k \leftarrow \mathcal{K}$. In addition to checking $P(k', x_i) = y_i$, also require that $k' = k$. However, have the game output $|\mathcal{K}|$ instead of 1 when this check succeeds. This change divides the chance of the adversary succeeding by $|\mathcal{K}|$, but multiplies the payoff on success by the same. Therefore, the advantage is unchanged.

G_3: Refactor the game by checking $P(k, x_i) = y_i$ instead of $P(k', x_i) = y_i$. If $k \neq k'$ then the game outputs 0 anyway, so this game behaves identically to the previous.

G_4: Again refactor, this time by replacing y_i with $P(k, x_i)$ in $\mathcal{A}$'s input. Again, the game will output 0 anyway if they are not equal.

G_5: Notice that y_i is only used in the equality checks $P(k, x_i) = y_i$ for $i \in [0, \beta)$. Therefore, these checks all succeeding has probability exactly

$$\frac{1}{|\mathcal{M}|\,(|\mathcal{M}|-1)\cdots(|\mathcal{M}|-\beta'+1)} = \frac{1}{(|\mathcal{M}|)_{\beta'}},$$

where β' is the number of distinct x_i. Remove these checks and all the y_i, and instead reduce the payoff on success from $|\mathcal{K}|$ to $\frac{|\mathcal{K}|}{(|\mathcal{M}|)_{\beta'}}$. The advantage for this game is identical to the previous, because the success probability was multiplied by $(|\mathcal{M}|)_{\beta'}$, but the payoff on success was divided by the same.

G_6: So far, the x_i have all been sampled uniformly. This implies a distribution for the partition defined by which x_i are equal to each other. In particular, there are $(|\mathcal{M}|)_{\beta'}$ ways to assign β' distinct values from $\mathcal{M}$ to the x_i, so the probability of each partition with β' classes (i.e., β' distinct values of x_i) is $\frac{(|\mathcal{M}|)_{\beta'}}{|\mathcal{M}|^\beta}$.

Instead, first sample a partition uniformly at random, then sample the x_i to be identical within each class, and distinct between classes. Additionally, change the payoff on success from $\frac{|\mathcal{K}|}{(|\mathcal{M}|)_{\beta'}}$ to $B_\beta \frac{|\mathcal{K}|}{|\mathcal{M}|^\beta}$, since B_β is the number of partitions of a set of size β. This does not change the advantage, because while we have multiplied the chance of selecting each partition with β' classes by $\frac{|\mathcal{M}|^\beta}{(|\mathcal{M}|)_{\beta'} B_\beta}$, we divided the success payoff by the same.

To summarize the current game, we are now almost back at G_0. However, there are two changes: $\mathcal{A}$ now has to output *the correct* key $k' = k$ to win, not just find another preimage to $P(k', x_i) = y_i$, and the payoff of winning the game is now $B_\beta \frac{|\mathcal{K}|}{|\mathcal{M}|^\beta}$ instead of 1.

G_7: Let $u_0, \ldots, u_{\beta-1} \in \mathcal{M}$ be distinct, and distinct from all x_i. Instead of checking that $k' = k$, check that $P(k', u_i) = P(k, u_i)$ for all i. This can only increase $\mathcal{A}$'s advantage, because if $k' = k$ then $P(k', u_i) = P(k, u_i)$.

[18] This is a generalization of the usual notion of cryptographic games, which typically only output in $\{0, 1\}$ and define advantage to be the probability of outputting 1.

G_8: Like in G_1, replace the evaluations of $P(k, x_i)$ with randomly sampled $y_i \leftarrow \mathcal{M}$, subject to $y_i = y_j \iff x_i = x_j$. Additionally, replace the evaluations of $P(k, u_i)$ with v_i, which are sampled as distinct elements of $\mathcal{M} \setminus (y_0, \ldots, y_{\beta-1})$. This change again reduces to the security of the PRP. The difference in advantage from the previous hybrid is at most $B_\beta \frac{|\mathcal{K}|}{|\mathcal{M}|^\beta}$ times the advantage of the reduction to PRP security, because the payoff of winning has been multiplied by $B_\beta \frac{|\mathcal{K}|}{|\mathcal{M}|^\beta}$.

Finally, we bound the advantage of G_8. Notice that the adversary can win only when the freshly random values v_i satisfy $P(k', u_i) = v_i$. There are at least $(|\mathcal{M}| - \beta)(|\mathcal{M}| - \beta - 1) \cdots (|\mathcal{M}| - 2\beta + 1)$ possibilities for $(v_0, v_1, \ldots, v_{\beta-1})$, so $\mathcal{A}$ can succeed with probability at most $\frac{1}{(|\mathcal{M}|-\beta)_\beta}$. The payoff on success is $B_\beta \frac{|\mathcal{K}|}{|\mathcal{M}|^\beta}$, so the advantage of the G_8 can be at most $B_\beta \frac{|\mathcal{K}|}{|\mathcal{M}|^\beta (|\mathcal{M}|-\beta)_\beta}$.

Totalling the advantage across all steps of the proof gives the stated bound. To construct $\mathcal{A}'$, randomly select which of the two reductions to run. I.e., with probability $\frac{|\mathcal{M}|^\beta}{|\mathcal{M}|^\beta + B_\beta |\mathcal{K}|}$ run the reduction for the change from G_0 to G_1, and otherwise use the one for the change from G_7 to G_8. □

To provide some intuition for why the AdvOWF can be greater than the AdvPRP, note that there are two ways the adversary can win the OWF game. $\mathcal{A}$ can either find the correct key, or find another key consistent with the OWF output. It is only in the former case that the PRP gets broken.

9.2.2 AES Without Key Expansion.

The single-key Even–Mansour scheme is a way to construct a block cipher F from a cryptographic permutation π [DKS12, EM97]. It works by adding a key k to the input z (in our case set to 0) and to the output of the permutation, i.e.,

$$F_k^\pi(z) = k + \pi(k + z)\,. \tag{2}$$

In FAEST we instantiate π with a block cipher such as AES with x as the key, as described in Section 3.3. As in the previous case, we need it to be a one-way function: given some (x, y) such that $y = F_k^\pi(0)$, it should be difficult to find a k' such that $F_{k'}^\pi(0) = y$. More formally, the adversary's advantage in breaking the OWF is the following probability:

$$\Pr[z \leftarrow \{0,1\}^\lambda, k \leftarrow \{0,1\}^\lambda, k' \leftarrow A^\pi(z, F_k^\pi(z)) : F_{k'}^\pi(0) = F_k^\pi(0)]. \tag{3}$$

As done in previous work, we consider the single-key variant and model π as an ideal permutation and consider attackers with oracle access to it. The following proof is adapted from a proof of Dobrauning et al. [DKR+22]

Theorem 9.2. *The single-key Even–Mansour construction (Equation (2)) is a secure one-way function, when the permutation π is an ideal random permutation.*

Proof. The attacker $\mathcal{A}$ is initialized with y and has oracle access to π. We must show that the probability in Equation (3) is negligible in λ (the key size and block size in bits).

Just before producing an output, $\mathcal{A}$ has made q queries to π, and has pairs (k_i, y_i) where $y_i = \pi(k_i)$ for $i \in [0, q)$. W.l.o.g., we assume that inputs k_i to π are distinct.

Each query is in exactly one of two cases. In the "consistent case", $\mathcal{A}$ learns a consistent key k_i, which means that $k_i = y - y_i$. In the "inconsistent case", $\mathcal{A}$ does not learn a consistent key, but it does learn that both k_i and $k_i' = y - y_i$ cannot be consistent keys. To see why also k_i' cannot be a consistent key in the "inconsistent case", notice that if it

were consistent then

$$\begin{aligned} y &= \pi(k'_i) + k'_i \\ y &= \pi(k'_i) + y - y_i \\ \pi(k_i) &= \pi(k'_i)\ , \end{aligned}$$

which is a contradiction since π is a permutation and $k_i \neq k'_i$.

We need to bound the probability of the "consistent case" occurring. Assume that all past queries have been in the "inconsistent case". If $\pi(k_i)$ is queried in the forward direction, then there are two possibilities: either $k_i = k$, which has probability at most $1/(2^\lambda - 2i)$ because only thing the adversary knows about k is that $2i$ possibilities have been eliminated, or $k_i \neq k$, in which case $y_i = \pi(k_i)$ is uniform from a set of size $2^\lambda - i - 1$, and so has probability at most $1/(2^\lambda - i - 1)$ of outputting $y_i = y - k_i$. A similar analysis holds when $\pi^{-1}(y_i)$ queried, where either $y_i = y - k$, and so the adversary has guessed k, or otherwise $k_i = \pi^{-1}(y_i)$ has probability at most $1/(2^\lambda - i - 1)$ of equaling $y - y_i$. Combine these using a union bound to get

$$\frac{1}{2^\lambda - 2i} + \frac{1}{2^\lambda - i - 1} < \frac{2}{2^\lambda - 2i - 1}$$

By possibly adding an extra query, we can makes that $\mathcal{A}$ always queries π on its output k'. The adversary then wins only when it hits the "correct case" defined above, for some query. Therefore, the adversary wins with probability at most

$$\begin{aligned} 1 - \prod_{i=0}^{q}(1 - 2/(2^\lambda - 2i - 1)) &= 1 - \prod_{i=0}^{q}(2^\lambda - 2(i+1) - 1)/(2^\lambda - 2i - 1) \\ &= 1 - (2^\lambda - 2(q+1) - 1)/(2^\lambda - 1) \\ &= 2(q+1)/(2^\lambda - 1) \end{aligned}$$

□

The factor of 2 in the bound may be unexpected. Why can the adversary do better than guessing the correct key k? To break the OWF, the adversary can either find the correct key k, or another key k' that is consistent with y. For a random key guess k_i, each of these has probability roughly $2^{-\lambda}$, so the guess is right with probability roughly $2 \cdot 2^{-\lambda}$. This is where the factor of 2 comes from.

9.2.3 Post-quantum security of the single-key Even-Mansour OWF. It was shown in [ABKM22] that the single-key Even-Mansour construction is a post-quantum-secure block cipher (i.e., a pseudo-random permutation, PRP) when the permutation π is an ideal random permutation. Here, we quote the result from [ABKM22], and use Lemma 9.1 to prove a simple corollary showing that single-key Even-Mansour is a post-quantum-secure one-way function in the same setting. The security bound we prove essentially matches the straight-forward Grover search attack (up to a square root in success probability).

The theorem from [ABKM22], specialized to single-key Even-Mansour, is

Theorem 9.3 (Special case of Theorem 3 of [ABKM22]). *Let $\mathcal{A}$ be an adversary making q_F classical queries to its first oracle and q quantum queries to its second oracle. Let π and σ be uniformly random λ-bit permutations, and $k \leftarrow \{0,1\}^\lambda$. Then*

$$\begin{aligned} &\left|\Pr\left[\mathcal{A}^{F_k^\pi,\pi}(1^\lambda) = 1\right] - \Pr\left[\mathcal{A}^{\sigma,\pi}(1^\lambda) = 1\right]\right| \\ &\qquad \leq 10 \cdot 2^{-\lambda/2}\left(q_F\sqrt{q} + q\sqrt{q_F}\right), \end{aligned}$$

where $F_k^\pi(z) = k + \pi(k + z)$ denotes the single-key Even-Mansour construction using permutation π and key k, and it is understood that $\mathcal{A}$ has forward and inverse access to its oracles.

Plug this bound into Lemma 9.1 with $m = \lambda$ and $\beta = 1$.

Corollary 9.4. *Let $\mathcal{A}$ be an adversary making q quantum queries to its uniformly random λ-bit permutation π. Each query can be either forward or inverse. Then,*

$$\mathsf{AdvOWF}_{\mathcal{A}^\pi} \leq 20 \cdot 2^{-(\lambda-1)/2} \left(\sqrt{2q+2} + q + 1\right) + \frac{1}{2^\lambda - 1}.$$

This shows that $\Omega(2^{\lambda/2})$ quantum queries to π are necessary to find an inverse to the single-key Even-Mansour OWF, matching the mentioned Grover search attack.

9.2.4 Accounting for the Reduced Key Space. One slight difference between our OWF instantiations and standard uses of AES is that our key space is $\frac{3}{4}$ the size of the standard, λ-bit key space. Recall that this restriction is needed for the security proof. In FAEST v1, key generation had a stronger restriction on the inputs to each S-box, which further reduced the key space, and was analyzed to reduce the concrete attack costs by 1–2 bits. In FAEST v2, we only lose 0.5 bits of security: an attacker can easily discard the 1/4 of the key space that does not satisfy the constraint, and the remaining 3/4 of the key space is uniformly distributed.

9.2.5 Security margin of OWF instantiations. We argued earlier that choosing an AES-based OWF is conservative. Here, we study the security margin of our two OWF instantiations in more detail.

When cryptanalysts cannot break a full version of a block cipher such as AES, variants with a reduced number of rounds are considered. The gap between the number of rounds that can be attacked and the number of rounds of the full version is considered a security margin, a buffer that may help to defend against yet unknown attack vectors.

Key recovery attacks on AES. There are no known key-recovery shortcut attacks that work with a single (plaintext, ciphertext) pair except variants of brute-force key search. The best known attack on round-reduced variants of AES in this class is from more than 10 years ago, given by Bouillaguet, Derbez and Fouque [BDF11], applied to 4-round AES and is marginal, costing 2^{120} time and 2^{80} memory.

Key recovery attacks on EM-AES. There are no known key-recovery shortcut attacks that work with a single (plaintext, ciphertext) pair except variants of brute-force key search, also not shortcut attacks on variants with less rounds. In absence of key recovery attacks, it is instructive to look at cryptanalytic results on AES that allow to distinguish the fixed-key AES permutation from random. For 5-round AES, a distinguisher [GRR17] that requires 2^{32} (plaintext, ciphertext) pairs and works for any key is known. Subsequently, for a setting giving more possibilities to an attacker, (adaptively chosen ciphertexts) this got improved in [BR19b] and to 6-rounds AES in [BR19a]).

In conclusion, the issue of how the EM and non-EM one-way functions compare for AES in terms of security margin is an interesting open question. Removing the constraint imposed by the OWF (only a single input/output pair available to an attacker) gives an upper bound on the security margin. Using the example of AES-128 with 10 rounds, the best key recovery attacks are on 7 rounds [DFJ13, BR22] (or 8 rounds if time-complexity close to brute-force is included [BKR11]), and the best permutation distinguishers reach up to 6 rounds [GRR17, BR19a].

9.3 Preliminaries for Security Reductions

In this subsection, we gather some notation, definitions and helpful lemmata.

9.3.1 Notation For a function $F\colon \mathcal{X} \to \mathcal{Y}$, we write $\mathsf{dom}(F) = \mathcal{X}$ for the *domain* of F, $\mathsf{cod}(F) = \mathcal{Y}$ for the codomain of F, and $\mathsf{im}(F) = \{F(x) \mid x \in \mathcal{X}\} \subseteq \mathcal{Y}$ for the *image* of F.

9.3.2 Divergence By construction, AES is a pseudorandom *permutation* (PRP), yet we want to treat it as a pseudorandom *function* (PRF). Naively, switching from a random permutation to a random function is distinguishable with advantage about Q^2/D for domain size D. Since our domain would be $\{0,1\}^{128}$, i.e., 128bit block (as specified by AES), we could not prove higher than 128bit security, even when limited to 2^{64} signatures overall. Fortunately, this is just a proof artefact and easily resolved by using suitable proof technique, such as the "H-coefficient" method or "Renyi divergence". Namely, we have the following facts.

Lemma 9.5 (∞-divergence). *Let X and Y be random variables with values in a set Ω and countable support. Let $\rho := \sup_{\omega\in\Omega} \frac{\Pr[X=\omega]}{\Pr[Y=\omega]}$, where $0/0 := 0$ and $t/0 := \infty$ for any $t > 0$. For any function $f\colon \Omega \to \mathbb{R}_{\geq 0}$ we have*

$$\mathbb{E}[f(X)] \leq \rho \cdot \mathbb{E}[f(Y)]$$

In particular, for any event $E \subseteq \Omega$, we have

$$\Pr[X \in E] \leq \rho \cdot \Pr[Y \in E]$$

Note that $D_\infty(X\|Y) := \log(\rho)$ is called ∞-divergence or Rényi divergence of order ∞.

Proof. We have by definition of ρ and since $f \geq 0$ that

$$\begin{aligned}
\mathbb{E}[f(X)] &= \sum_{\omega\in\Omega} \Pr[X=\omega] \cdot f(\omega) \\
&= \sum_{\omega\in\Omega} \frac{\Pr[X=\omega]}{\Pr[Y=\omega]} \cdot \Pr[Y=\omega] \cdot f(\omega) \\
&\leq \sum_{\omega\in\Omega} \rho \cdot \Pr[Y=\omega] \cdot f(\omega) \\
&= \rho \cdot \mathbb{E}[f(Y)]
\end{aligned}$$

which proves the first claim. The second follows since $\Pr[X \in E] = \mathbb{E}[f(X)]$ for the predicate $f(\omega) = [\omega \in E] \in \{0,1\}$. □

Corollary 9.6. *Let $\mathcal{X}$ be a set of size N and let $n \in \mathbb{N}$ with $n < N$. Let $\mathcal{A}\colon \mathcal{X}^n \to \{0,1\}$ be any function. Let R be the uniform distribution on $\mathcal{X}^n$ and let P be the distribution of n uniform samples without replacement from $\mathcal{X}$. Then we have*

$$\Pr[\mathcal{A}(P)=1] \leq e^{\frac{n(n-1)}{N}} \Pr[\mathcal{A}(R)=1]$$

Proof. By Lemma 9.5, it suffices to show $\frac{\Pr[P=x]}{\Pr[R=x]} \leq e^{\frac{n(n-1)}{N}}$. This follows since $\Pr[P = x] \leq \frac{1}{N\cdot(N-1)\cdot\ldots\cdot(N-n+1)}$ and $\Pr[R=x] = \frac{1}{N^n}$, and therefore

$$\begin{aligned}
\frac{1/(N\cdot(N-1)\cdot\ldots\cdot(N-n+1))}{1/N^n} &= \frac{N^n}{(N\cdot(N-1)\cdot(N-n+1))} \\
&= \Big(\prod_{i=1}^{n-1}(1-\frac{i}{N})\Big)^{-1} \leq e^{\frac{n(n-1)}{N}}
\end{aligned}$$

because $e^{-x} \leq 1 - x/2$ for $x \geq 0$. □

In particular, we can use Corollary 9.6 to switch outputs of PRGs in the GGM tree to truly random outputs. Then, they can serve as random seeds for the next GGM application.

Remark 9.7 (Hybrids with ∞-divergence). For a sequence of hybrids $\mathsf{H}_0, \mathsf{H}_1, \ldots, \mathsf{H}_n$, where $\Pr[\mathsf{H}_{i-1} = 1] \leq \rho_i \cdot \Pr[\mathsf{H}_i = 1] + \varepsilon_i$, with $\rho_i \geq 1$ and $\varepsilon_i \geq 0$, we find

$$\begin{aligned}\Pr[\mathsf{H}_0 = 1] &\leq \Big(\prod_{j=1}^{n} \rho_j\Big) \cdot \Pr[\mathsf{H}_n = 1] + \sum_{i=1}^{n} \Big(\prod_{j=1}^{i-1} \rho_j\Big) \cdot \varepsilon_i \\ &\leq \Big(\prod_{j=1}^{n} \rho_j\Big) \cdot \Big(\Pr[\mathsf{H}_n = 1] + \sum_{i=1}^{n} \varepsilon_i\Big)\end{aligned}$$

This is the analogue to the usual hybrid argument.

9.3.3 Standard Hardness Assumptions Most of our hardness assumptions are related to AES, in particular indistinguishability of AES from a pseudorandom permutation. We do not make explicit that hardness assumptions on the hashes $\mathsf{H}_0, \ldots, \mathsf{H}_4$; these are idealized as random oracles in the security proofs.

Definition 9.8 (Indistinguishability). *Let $\mathcal{D}_0, \mathcal{D}_1$ be two distributions. Let $\mathcal{A}$ be an adversary in the following experiment $\mathsf{ExpDist}_{\mathcal{A}}$:*

- $x_0 \leftarrow \mathcal{D}_0$, $x_1 \leftarrow \mathcal{D}_1$, $b \leftarrow \{0,1\}$,
- $b' = \mathcal{A}(1^\lambda, x_b)$
- *Return* $b' = b$.

The distinguishing advantage of $\mathcal{A}$ is

$$\mathsf{AdvDist}_{\mathcal{A}}^{\mathcal{D}_0,\mathcal{D}_1} = 2 \cdot \left|\Pr[\mathsf{ExpDist}_{\mathcal{A}}] - \frac{1}{2}\right| = \left|\Pr[b' = 1 \mid b = 0] - \Pr[b' = 1 \mid b = 1]\right|$$

If $\mathcal{A}$ is given access to Q independent samples of $\mathcal{D}_0$ or $\mathcal{D}_1$ instead, we write $\mathsf{AdvDist}_{\mathcal{A}}^{\mathcal{D}_0,\mathcal{D}_1}[Q]$ for the advantage in the respective game. The definition for indistinguishability of oracles (instead of distributions) is analogous.

An efficiently computable function family $\mathsf{PRF}\colon \mathcal{K}_\lambda \times \mathcal{X}_\lambda \to \mathcal{Y}_\lambda$ where $\mathcal{K}_\lambda$ is the key space is a *pseudorandom function (PRF)* if for a uniform secret key $\mathbf{k} \leftarrow \mathcal{K}_\lambda$, oracles-access to $\mathsf{PRF}(\mathbf{k}, \cdot)$ is indistinguishable from access to truly random function $\mathsf{H}\colon \mathcal{X}_\lambda \to \mathcal{Y}_\lambda$ for any efficient adversary. A *pseudorandom permutation (PRP)* $\mathsf{PRP}\colon \mathcal{K}_\lambda \times \mathcal{X}_\lambda \to \mathcal{X}_\lambda$ is a PRF such that $\mathsf{PRP}(\mathbf{k}, \cdot)$ is a permutation for any $\mathbf{k} \in \mathcal{K}_\lambda$.

9.3.4 IV-based tweakable PRGs For analysis reasons, we define IV-based tweakable PRG tailored specifically to our construction PRG in Figure 3.6, with block sizes of 128. Then we define and analyze the security of PRG in two aspects: A pseudorandomness notion which is used to prove that VOLECommit is hiding, and a injectivity-type notion which is needed only in FAEST-EM and used to prove that VOLECommit is statistically binding.

Definition 9.9 (IV-based tweakable PRG). *An IV-based tweakable PRG is an efficient map $\mathsf{PRG}_m\colon \{0,1\}^\lambda \times \{0,1\}^{128} \times \{0,1\}^{32} \to \{0,1\}^{\alpha \cdot 128}$, where $\alpha \in \mathbb{N}$ and output length $m = \alpha \cdot 128$.*

Pseudorandomness notions. Since our notion of IV-based tweakable PRG is so closely tied to PRG from Figure 3.6, we define two types of security, namely PRP-type and PRF-type security in Definition 9.10. The underlying reason is, that as a pseudorandom function, the fixed block size of 128 of AES would be susceptible to birthday attacks. However, by using a divergence-based switching lemma (Corollary 9.6), we can still show security in our setting, where signature queries will be limited to $Q_{\mathsf{sig}} \leq 2^{64}$.

Definition 9.10 (PRP-type and PRF-type security of PRG). *Let $\alpha \in \mathbb{N}$ and let $\mathsf{PRG}_m \colon \{0,1\}^\lambda \times \{0,1\}^{128} \times \{0,1\}^{32} \to \{0,1\}^{\alpha \cdot 128}$ be a deterministic polynomial-time algorithm with output length $m = \alpha \cdot 128$ (not necessarily PRG from Figure 3.6). Let $\mathcal{A}$ be a T-query adversary in the following PRP-type game:*

- *For $k \leftarrow \{0,1\}^\lambda$, $\mathsf{iv} \leftarrow \{0,1\}^{128}$.*
- *Sample $b \leftarrow \{0,1\}$ and set $\mathsf{RP} = \{\}$ as an empty associative array and let $B = \emptyset$.*
- *Run $b' \leftarrow \mathcal{A}^{\mathsf{PRG}^b}(\mathsf{iv})$ where $\mathcal{A}$ is limited to T oracle queries and*
  - *$\mathsf{PRG}^0(k, \mathsf{iv}, \mathsf{twk})$: Output $\mathsf{PRG}_m(k, iv, \mathsf{twk})$*
  - *$\mathsf{PRG}^1(k, \mathsf{iv}, \mathsf{twk})$: If $\mathsf{RP}[\mathsf{twk}] = \bot$: For $i = 1, \ldots, \alpha$:*
    - *Sample $y_i \leftarrow \{0,1\}^{128} \setminus B$*
    - *Set $B = B \cup \{y_i\}$.*

    Set $\mathsf{RP}[\mathsf{twk}] = (y_1 \| \ldots \| y_\alpha)$ and output $\mathsf{RP}[\mathsf{twk}]$.
- *Output 1 (win) if $b = b'$, else 0 (lose).*

Let p be the probability that $\mathcal{A}$ wins the game. The advantage $\mathsf{AdvPRP}_{\mathcal{A}}^{\mathsf{PRG}_m}[T]$ of T-query adversary against the PRP-type security of the (IV-based tweakable) PRG PRG is defined as

$$\mathsf{AdvPRP}_{\mathcal{A}}^{\mathsf{PRG}_m}[T] = |2p - 1| \,.$$

Define the PRF-type game which is identical to the above, except that we implement a (lazy sampled) random function RF instead of the random permutation RF. We write $\mathsf{AdvPRG}_{\mathcal{A}}^{\mathsf{PRG}_m}[T] = |2p - 1|$ to denote the respective advantage for PRF-type security.

We note that Definition 9.10 imposes a stronger requirement on PRG than strictly necessary for our proofs. Namely, we let the adversary choose the tweaks freely, while in our construction, the tweaks are dependent on the GGM layer and fixed.

Remark 9.11 (PRP-type security). It is easy to see that our PRG constructed in Figure 3.6 satisfies PRP-type security of Definition 9.10 by a straightforward reduction to PRP security of AES with αT queries. (Observe that (by definition of AddToUpperWord and AES-CTR) we never query AES on the same input, hence we get αT distinct queries and thus random blocks.)

To deal with PRF-type security, we argue with ∞-divergence (Lemma 9.5 and Corollary 9.6).

Lemma 9.12. *Let $\beta \in \mathbb{N}$ and let $\mathsf{PRG}_m \colon \{0,1\}^\lambda \times \{0,1\}^{128} \times \{0,1\}^{32} \to \{0,1\}^{\beta \cdot 128}$ be a deterministic polynomial-time algorithm with $m = \beta \cdot 128$. Let $\mathcal{A}$ be a T-query adversary in for PRF-type security of PRG (Definition 9.10). Let $\mathsf{G}_{\mathsf{PRP}}$ denote the output of $\mathcal{A}$ real PRP-type game and G_{RP} (resp. G_{RF}) be the output in the ideal PRP-type game (resp. PRF-type). Then it holds that*

$$\Pr[\mathsf{G}_{\mathsf{PRP}} = 1] \leq e^{\beta^2 T^2 / 2^{128}} \cdot \Pr[\mathsf{G}_{\mathsf{RF}} = 1] + \mathsf{AdvPRP}_{\mathcal{A}}^{\mathsf{PRG}_m}[T]$$

Proof. Let G_{RP} denote the ideal PRP-type game. Then

$$\Pr[\mathsf{G}_{\mathsf{PRP}} = 1] \leq \Pr[\mathsf{G}_{\mathsf{RP}} = 1] + \mathsf{AdvPRP}_{\mathcal{A}}^{\mathsf{PRG}_m}[T]$$

follows by definition. By an application of Corollary 9.6, switching the sampling of blocks without replacement to sampling of blocks with replacement (i.e., truly at random) and therefore to G_1, leads to

$$\Pr[\mathsf{G}_{\mathsf{RP}} = 1] \leq e^{\beta^2 T^2 / 2^{128}} \cdot \Pr[\mathsf{G}_{\mathsf{RF}} = 1]$$

□

Non-standard collision resistance. While we rely on injectivity of universal hashing in FAEST under parameter choices which statistically guarantee this, we make a more aggressive assumption for FAEST-EM. The assumption is a strengthening of collision resistance, which asserts that it is hard to even *find an image* for which a *collision exists*, i.e. for which two or more preimages exist. Note that the adversary does not need to output the *preimages*, their existence suffices.

Definition 9.13 (Almost injective PRG). *Let* $\mathsf{PRG}(k, \mathsf{iv}, \mathsf{twk})$ *be a (IV-based tweakable) PRG, where* $k \in \mathcal{K}_\lambda = \{0,1\}^\lambda$, $\mathsf{iv} \in \mathcal{I}_\lambda$, *and* $\mathsf{twk} \in \mathcal{T}_\lambda$. *Consider the following game, parameterized by* Q_{out}*:*

1. *The adversary outputs outputs* $\mathsf{iv} \in \mathcal{I}_\lambda$ *together with* Q_{out} *tuples* $(\mathsf{twk}_j, y_j)_{j \in [1..Q_{\mathsf{out}}]}$.
2. *If for any* j, *there exists* $\mathbf{k} \neq \mathbf{k}'$ *in* $\mathcal{K}_\lambda$ *such that* $\mathsf{PRG}(\mathbf{k}, \mathsf{iv}, \mathsf{twk}_j) = \mathbf{y}_j = \mathsf{PRG}(\mathbf{k}', \mathsf{iv}, \mathsf{twk}_j)$, *output* 1 *(success). Else output* 0.

We define the advantage of $\mathcal{A}$ *in the game as*

$$\mathsf{AdvInj}^{\mathsf{PRG}}_{\mathcal{A}}[Q_{\mathsf{out}}] = \Pr[\textit{The game outputs } 1].$$

We say that PRG *is* almost injective *if for any polynomial-time adversary and polynomial* Q_{out}, *the advantage* $\mathsf{AdvInj}^{\mathsf{PRG}}_{\mathcal{A}}[Q_{\mathsf{out}}]$ *is negligible.*

Observe that the adversary must produce an element y_j for which a collision exists to win in Definition 9.13. Since it seems hard to tell if an image y_j has at more than one preimage, without *knowing* more than one preimages of it (i.e. a collision), the assumption seems plausible.

As a sanity check, we consider a simple attacks on Definition 9.13 for the PRG constructed in Figure 3.6 under heuristic simplifications.

Remark 9.14 (Heuristic). We model AES as an ideal cipher and assume a bound Q_{AES} on ideal cipher queries. The map PRG: $\{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ with $\mathsf{PRG}(\mathbf{k}, \mathsf{iv}) = (\mathsf{AES.Enc}(\mathbf{k}, \mathsf{iv}), \ldots, \mathsf{AES.Enc}(\mathbf{k}, \mathsf{AddToLowerWord}(\mathsf{iv}, h)))$ is a random oracle in $\mathbf{k}$ for fixed iv (where $h = 2\lambda/128$, see Figure 3.6). We heuristically ignore the correlations caused by tweak twk and counter i acting on the same IV space via AddToUpperWord and AddToLowerWord respectively. Moreover, we heuristically treat ideal cipher queries and PRG queries as the same. This heuristic is partly justified by the PRG construction; ideal cipher encryption queries can be recovered by running PRG. We note however, that it is not clear if *decryption queries* could somehow be exploited for improved attacks; we ignore them.

Next, we focus on a naive strategies an adversary could follow: Let $m = 2^\lambda$, $n = 2^{2\lambda}$. Sticking to single IV chosen in advance, and mounting a mix of birthday attack and guessing via Q_{out} images $\mathbf{y}_j$. The birthday attack part is straightforward. By focusing on a single iv, the probability of collision with Q_{AES} PRG queries is less than $\frac{Q^2_{\mathsf{AES}}}{n}$. Note that, importantly, in this case the adversary *knows* for certain that it found a $\mathbf{y}$ with multiple preimages. On the other hand, as long as no collisions occurred, all images are the "same", namely, random distinct elements. Thus, it may as well output a random subset of Q_{out} of these $\mathbf{y}$s. The probability of success in that case is the same for each subset, and is given by $1 - (1 - \frac{Q_{\mathsf{out}}}{n})^{m - Q_{\mathsf{AES}}} \leq \frac{Q_{\mathsf{out}}(m - Q_{\mathsf{AES}})}{n}$. In summary, the success probability of this strategy is upper-bounded by $\frac{Q^2_{\mathsf{AES}}}{n} + \frac{Q_{\mathsf{out}}(m - Q_{\mathsf{AES}})}{n} = \frac{Q_{\mathsf{AES}}(Q_{\mathsf{AES}} - Q_{\mathsf{out}})}{n} + \frac{Q_{\mathsf{out}} \cdot m}{n}$.

9.4 Properties of VOLECommit

9.4.1 Binding We first argue that commitments are binding, by showing that there is an extractor algorithm that is allowed to observe random oracle queries, and will successfully extract the adversary's committed message with high probability. The analysis is similar to [BBD+23b, Sec. 3.1], with the main differences being that: (1) Instead of analyzing the vector commitments BAVC directly, we analyze the VOLECommit and VOLEReconstruct algorithms which use them. Thus, we show the binding property on the signer's VOLE matrices, rather than the original vectors committed using VC (even though these are also still binding).[19] By arguing security for our specific construction, we also avoid the security loss incurred by generic reductions in [BBD+23b]. (2) We allow for the extractor to be *unbounded*. This is possible since we will reduce to *soundness* instead of knowledge for the Fiat–Shamir transformation. In other words, we only rely on statistically binding commitments, not efficiently extractable ones.

Lemma 9.15 (Extractable Binding of VOLE Commitment). *Consider the following extractable binding game for* (VOLECommit, VOLEReconstruct)*, a stateful adversary* $\mathcal{A}$ *and a straightline extractor* Ext*:*

1. *Commit-Phase: Let* $\mathcal{C} = \{\}$ *be an empty associative array.*
   (a) $\mathbf{C} = (\mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}^{\mathsf{pre}}) \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_4}(\mathtt{commit})$.
   (b) $\mathbf{D} = (\boldsymbol{u}_i^*, \mathbf{V}_i^*)_{i\in[0,\tau)} = \mathsf{Ext}^{\mathsf{H}_4}(\mathcal{L}_0, \mathcal{L}_1, \mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}^{\mathsf{pre}})$*, where* $\mathcal{L}_b$ *is a set* $\{(x_i, \mathsf{H}_b(x_i))\}$ *of query-response pairs from queries* $\mathcal{A}$ *made to the random oracle* H_b*.*
   (c) *Set* $\mathcal{C}[\mathbf{C}] = \mathbf{D}$*. Note that* $\mathbf{D} = \perp$ *is allowed.*
   (d) $\mathcal{A}$ *can choose to go to Step 1a or it outputs* $\mathbf{C}$ *in* $\mathcal{C}$ *and the game continues below.*
2. *Decommitment-Phase: Parse* $\mathbf{C} = (\mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}^{\mathsf{pre}})$.
3. $(\mathsf{chall}_3, \mathsf{decom}_I) \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_4}(\mathtt{open})$ *where* $I = (\Delta_0, \ldots, \Delta_{\tau-1}) \leftarrow \mathsf{DecodeAllChall}_3(\mathsf{chall}_3)$.
4. *If* $I = \perp$ *then output* 0 *(failure).*
5. *Let* $\mathsf{iv} = \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}})$.
6. *For* $i \in [0..\tau)$*:* $(\delta_{i,0}, \ldots, \delta_{i,k_i}) := \mathsf{BitDec}(\Delta_i, k_i)$
7. $(\overline{\mathsf{com}}, \mathbf{Q}) := \mathsf{VOLEReconstruct}^{\mathsf{H}_0,\mathsf{H}_1}(\mathsf{chall}_3, \mathsf{decom}_I, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$.
8. *Parse* $[\mathbf{Q}_0 \ldots \mathbf{Q}_{\tau-1}\mathbf{0}_{\hat{\ell},w}] = \mathbf{Q}$ *where* $\mathbf{Q}_i \in \mathbb{F}_2^{\hat{\ell}\times k_i}$
9. *Output* 0 *(failure) if*
   (a) $\overline{\mathsf{com}} = \perp$ *or* $\mathsf{com} \neq \overline{\mathsf{com}}$ *(i.e., verifier rejects), or*
   (b) $\mathbf{Q}_i = \mathbf{V}_i^* \oplus [\delta_{i,0} \cdot (\boldsymbol{u}_i^* \oplus \boldsymbol{c}_i), \cdots, \delta_{i,k_b} \cdot (\boldsymbol{u}_i^* \oplus \boldsymbol{c}_i)]$*, for all* $i \in [0, \ldots, \tau)$ *(i.e., extraction correct).*
10. *Output* 1 *(success) otherwise.*

Suppose $\mathcal{A}$ *makes at most* Q_0, Q_1 *queries to the oracles* $\mathsf{H}_0, \mathsf{H}_1$*, respectively. Moreover, suppose that* $\mathcal{A}$ *checks if* VOLEReconstruct *returns* $\perp$ *for the purported opening* decom_I*, and outputs* $\perp$ *if that happens. There is a deterministic (unbounded) straightline extractor* Ext *(defined in the security proof for* FAEST*) such that the advantage of* $\mathcal{A}$ *in the above game is at most*

$$\frac{(Q_1+1)^2}{2^{2\lambda}} + \tau Q_0 \cdot 2^{-\lambda}.$$

There is a deterministic (unbounded) straightline extractor Ext *which is allowed to program* H_4 *(defined in the security proof for* FAEST-EM*), such that the advantage of* $\mathcal{A}$ *in the above game is at most*

$$\frac{(Q_1+1)^2}{2^{2\lambda}} + \mathsf{AdvInj}_{\mathcal{B}}^{\mathsf{PRG}_{2\lambda}}[Q_4, L].$$

where the running time of $\mathcal{B}$ *is roughly that of* $\mathcal{A}$*. The IV-based tweakable PRG* $\mathsf{PRG}_{2\lambda}$ *is defined in Figure 3.6, where we restrict the tweak domain to* $[L-1, L+\tau-1]$.

[19] Note here that binding does not imply that the extracted base VOLEs consistently have the same $\mathbf{u}$, i.e., that the correction terms $\mathbf{c}_i$ were computed honestly. This is established by additional checks.

Intuitively, Lemma 9.15 asserts that, (after switching iv to injective mode), extraction succeeds unless $\mathcal{A}$ cheats by breaking collision or preimage resistance of the random oracle. This gives the advantage bounds in the lemma. The extractor will brute-force some preimage and thus be inefficient. This will be unproblematic since the binding property will be (switched to) statistical and the proof system will be statistically sound.

Note that $\mathcal{A}$ chooses the challenges Δ_i, hence it can cheat in the sense that only an opening for Δ_i is "known" to $\mathcal{A}$, i.e. the (extracted) commitment cannot be opened for any other challenge than Δ_i. However, as long as the extractor extracts that VOLE correlation, this does not constitute a success for the adversary — the adversary only wins if the extracted VOLE correlation disagrees with the opened VOLE correlation.

Remark 9.16. The requirement that $\mathcal{A}$ checks its purported decommitment can be removed by replacing a non-conforming adversary $\mathcal{A}'$ with $\mathcal{A}$ where this check is added. Due to this change, the maximal number of queries Q_0 (resp. Q_1) to H_0 (resp. H_1) increases to at most Q_0+1 (resp. $Q_1+\tau+1$), which for concrete security makes almost no difference, since $Q_0, Q_1 \gg \tau$.

Remark 9.17. Lemma 9.15 is a multi-challenge setting, i.e., the adversary can try many commitments during the Commitment-Phase. This allows us to avoid losses caused by hybrid arguments: A reduction can simply extract all openings, and should a decommitment ever be inconsistent, it can choose it to for the Decommitment-Phase to win the game. Running the extractor in the reduction instead of the game leads to a subtle problem, namely, a commitment can fail to win the extractable binding experiment even though the reductions failed to extract it (using Ext). This case can occur if collisions or preimages of (purported) hash value are produced after the reduction's extractions failed, but before it received the opening information from the adversary. As we bound the probability of this event, it will however not complicate the use in reductions.

We use following lemma to simplify the proof of Lemma 9.15.

Lemma 9.18 (Random oracle graph game). *Let $H\colon \{0,1\}^* \to \mathcal{Y}$ be a random oracle and consider following game with an adversary $\mathcal{A}$. The game keeps track of a directed graph $G=(V,E)$, initially empty, and proceeds as follows: The adversary can (repeatedly) query H at some x. Let $H(x)=y$.*

- *If $y \in V$ but there is no edge $(x',y) \in E$, then the adversary wins. (Preimage)*
- *If an edge $(x',y) \in E$ exists with $x' \neq x$, then the adversary wins. (Collision)*
- *Else, add nodes x and y to V and add an edge $e=(x,y)$ from x to y to E.*

Let $\mathcal{A}$ be an adversary which makes at most Q queries to H. Then the probability that $\mathcal{A}$ wins is bounded by

$$Q^2/\left|\mathcal{Y}\right|.$$

Proof of Lemma 9.18. The chances for $\mathcal{A}$ to win with a query to H (if it has not yet won) are at most $\frac{|V|}{|\mathcal{Y}|}$. Namely, the conditions on no edge to y or an existing edge to y separate V into two disjoint sets: Let V' be the set of nodes with no edges pointing to them. Let V'' be the set of nodes with edges pointing to them (that is, images under H). The adversary wins if a fresh query x yields an edge to a node y in V' (the case $\nexists x'\colon (x',y) \in E$), or if x yields another edge to a node in V'' (the case $\exists x'\colon (x',y) \in E \wedge x \neq x'$). (The former is a preimage attack, the latter is a collision attack.) Clearly, the chances that the former happens are $\frac{|V'|}{|\mathcal{Y}|}$ and the chances that the latter happens are $\frac{|V''|}{|\mathcal{Y}|}$, and since V' and V'' are disjoint, the chances that either happens is $\frac{|V|}{|\mathcal{Y}|}$.

Since the set V grows by at most 2 nodes per query, we can bound the success probability of $\mathcal{A}$ by

$$\sum_{i=1}^{Q} \frac{2 \cdot (i-1)}{|\mathcal{Y}|} \leq \frac{Q^2}{|\mathcal{Y}|}.$$

□

Proof of Lemma 9.15. We first analyze the straightline extractor for FAEST. At the end of the proof, we explain the modifications required for FAEST-EM. The core of the proof is the extraction of the BAVC commitment com, i.e., the seeds $\mathsf{sd}_{i,j}$. Transforming the seeds, given $(\mathsf{chall}_3, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv})$ to the VOLE output $\mathbf{Q}$ is simply post-processing of the opening. For convenience, the extractor also applies the post-processing to compute the VOLE relation (or $\perp$ if too many seeds failed to extract).

We define the straightline extractor Ext, by looking through the lists of queries $\mathcal{L}_1$ as follows.

1. Find a preimage $(h_0 \| \ldots \| h_{\tau-1})$ of com under H_1. If there is no unique preimage, i.e., there are none or multiple preimages, immediately output $\perp$.
2. For each $i \in [0..\tau)$:
   (a) Find a preimage $(\mathsf{com}_{i,1}, \ldots, \mathsf{com}_{i,N_i})$ of h_i under H_1, where $N_i = 2^k$ if $i < \tau_1$ and 2^{k-1} otherwise (cf. Table 5.1). If there is no unique preimage, i.e., there are none or multiple preimages, immediately output $\perp$.
   (b) For each $j \in [0..N_i)$, brute-force a preimage r for $\mathsf{com}_{i,j}$ under $r \mapsto \mathsf{LeafCommit}(r, \mathsf{iv}, i+L-1, \mathsf{uhash}_i)$, where uhash_i is set as $(\mathsf{uhash}_0, \ldots, \mathsf{uhash}_{\tau-1}) = \mathsf{H}_0(\mathsf{iv}; 3\lambda\tau)$ and iv is the IV used in the commit. That is, find r with $(\mathsf{sd}_{i,j}, \mathsf{com}_{i,j}) = \mathsf{LeafCommit}(r, \mathsf{iv}, i+L-1, \mathsf{uhash}_i)$. If there is no such preimage, set $\mathsf{sd}_{i,j} = \perp$. If there are multiple preimages, immediately output $\perp$. Otherwise, remember $\mathsf{sd}_{i,j}$ corresponding to $\mathsf{com}_{i,j}$.

Then, for each of the $i \in [0, \tau)$ VOLEs, Ext distinguishes following cases:

- Case 1 (All seeds found): If all $(\mathsf{sd}_{i,j})_{j \in [0,N)}$ are not $\perp$ for i, then Ext computes $\mathbf{V}_i^*$ and $\widetilde{\mathbf{u}}_i$ honestly (i.e., via $\mathsf{ConvertToVOLE}$) and sets $\Delta_i^* = \perp$.
- Case 2 (Missing one seed): If a single seed is missing, say $\mathsf{sd}_{i,j^*} = \perp$, then Ext sets $\Delta_i^* = j^*$ and, as in $\mathsf{VOLEReconstruct}$, computes $(\widetilde{\mathbf{u}}_i, \mathbf{q}'_{i,0}, \ldots, \mathbf{q}'_{i,k_b-1}) = \mathsf{ConvertToVOLE}(\perp, \mathsf{sd}_{i,1 \oplus \Delta_i^*}, \ldots, \mathsf{sd}_{i,(N_i-1) \oplus \Delta_i^*}, \mathsf{iv}; \hat{\ell})$ and lets $\mathbf{Q}_0 = [\mathbf{q}'_{0,0} \cdots \mathbf{q}'_{0,k_i-1}]$ and $\mathbf{u}_0^* = \widetilde{\mathbf{u}}_0$ as well as $\mathbf{Q}_i = [\mathbf{q}'_{i,0} \cdots \mathbf{q}'_{i,k_i-1}] + [\delta_{i,0} \cdot \mathbf{c}_i \cdots \delta_{i,k_i-1} \cdot \mathbf{c}_i]$ and $\mathbf{u}_i^* = \widetilde{\mathbf{u}}_i \oplus \mathbf{c}_i$ for $i > 0$. With that, Ext defines $\mathbf{V}_i^* = \mathbf{Q}_i + [\delta_{i,0} \cdot \mathbf{u}_i^*, \cdots, \delta_{i,k_b} \cdot \mathbf{u}_i^*]$, for $i \in [0, \tau)$.
- Case 3 (Missing multiple seeds): If two or more seeds are $\perp$ (due to missing or non-unique preimages), say $\mathsf{sd}_{i,j_1^*} = \perp = \mathsf{sd}_{i,j_2^*}$ for $j_1^* \neq j_2^*$, then Ext outputs $\perp$, i.e., extraction fails.

This yields the output $(\boldsymbol{u}_i^*, \mathbf{V}_i^*)_{i \in [0,\tau)}$ of Ext. While not part of its output, we make use of the bad challenges $(\Delta_i^*)_{i \in [0,\tau)}$ found during extraction later. Finally, to invoke Lemma 9.18 later, we add the following actions to Ext if extraction failed:

- If extraction failed since the preimage for com (resp., some preimage for h_i) was missing, in step 1 (resp. step 2), query $\mathsf{H}_1(h_i)$ (resp., pick the smallest h_i without preimage and query $\mathsf{H}_1(h_i)$). (Lemma 9.18 will then rule out that the adversary finds a preimage after extraction failure.)

Note that Ext will make at most one query to H_1 (per extraction).

To analyse Ext, we first consider the (graph from) the game from Lemma 9.18 and define by Fail_1 the (failure) event that, during the extractable binding game, a preimage attack or a collision attack succeeded in the random oracle graph for H_1.

Consider the whole binding game as an adversary $\mathcal{A}$ to the random oracle graph game in Lemma 9.18, where only the subroutine $\mathcal{A}'$ of $\mathcal{A}$ makes fresh random oracle queries. Observe that if the bad event Fail_1 occurs, then $\mathcal{A}'$ wins Lemma 9.18. This holds because, either the adversary finds a random oracle collision, or it finds a preimage for a value x which Ext could not extract. To cover the latter case in the random oracle game of Lemma 9.18, Ext makes a additional query $\mathsf{H}_1(x)$ which adds x as a node to the random oracle game, so that if the adversary finds the missing preimage of x, then the game aborts.

As a consequence, failing to look up com or h_i can trigger an extra query to H_1 by Ext. By Lemma 9.18 and a union bound, we therefore find

$$\Pr[\mathsf{Fail}_1] \leq (Q_1+1)^2/2^{2\lambda}$$

As a second step, consider the failure event $\mathsf{Fail}_{\mathsf{uhash}}$ which by definition occurs if brute-forcing $\mathsf{sd}_{i,j}$ finds multiple preimages. Observe that hash LeafHash (used in LeafCommit to hash $(\mathsf{sd}, \mathbf{x}_1)$) is clearly $\varepsilon_{\mathsf{uhash}}$-almost universal for $\varepsilon_{\mathsf{uhash}} = 2^{-3\lambda}$. Moreover, the map $r \mapsto \mathsf{LeafCommit}(r, \mathsf{iv}, i+L-1, \mathsf{uhash}_i)$ has input domain $\{0,1\}^\lambda$ of size 2^λ for each fixed $i \in [0..\tau)$ and iv. Hence, at most 2^λ different values $(\mathsf{sd}, \mathbf{x}_1)$ can be hashed through $\mathsf{LeafHash}(\mathsf{uhash}_i, \cdot)$ for fixed (i, iv). Since uhash_i is a random oracle output and thus is independent of the hashed inputs, we find that for fixed i the map $r \mapsto \mathsf{LeafCommit}(r, \mathsf{iv}, i+L-1, \mathsf{uhash}_i)$ is injective except with probability at most $(2^\lambda)^2 \cdot \varepsilon_{\mathsf{uhash}} \leq 2^{-\lambda}$. Namely, by $\varepsilon_{\mathsf{uhash}}$-almost universality, for any $\mathbf{x} \neq \mathbf{y} \in \{0,1\}^{4\lambda}$ the probability that $\mathsf{LeafHash}(\mathsf{uhash}_i, \mathbf{x}) = \mathsf{LeafHash}(\mathsf{uhash}_i, \mathbf{y})$ is at most $\varepsilon_{\mathsf{uhash}}$ (over uniform uhash_i). Hence, for any set $\mathcal{S}$ of size $S = \mathcal{S}$, by a union bound over $\mathbf{x} \neq \mathbf{y} \in \mathcal{S}$, we have a collision with probability at most $S^2/2 \cdot \varepsilon_{\mathsf{uhash}}$. Plugging in the size $S = 2^\lambda$ which is hashed by LeafCommit for fixed (i, iv), and $\varepsilon_{\mathsf{uhash}} \leq 2^{-3\lambda}$, the hash LeafHash is injective with probability at least $2^{-\lambda}$ (for uniform uhash_i). As we assume that the adversary runs VOLEReconstruct, hence queried H_0 to learn $(\mathsf{uhash}_0, \ldots, \mathsf{uhash}_{\tau-1})$ for iv, we find (by a union bound over all τ choices of $i \in [0..\tau)$ and Q_0 queries to H_0) that

$$\Pr[\mathsf{Fail}_{\mathsf{uhash}}] \leq \tau Q_0 \cdot 2^{-\lambda}$$

In the following, we analyze the extraction conditioned on $\neg\mathsf{Fail}$ where $\mathsf{Fail} = \mathsf{Fail}_1 \vee \mathsf{Fail}_{\mathsf{uhash}}$. Consequently, whenever the extractor does not know a preimage the game will not encounter one (either due to a query of $\mathcal{A}$ or during VOLEReconstruct), and likewise, there will never be multiple preimages. We observe the following:

- com uniquely defines $(h_0 \| \ldots \| h_{\tau-1})$.
- h_i uniquely defines $(\mathsf{com}_{i,0}, \ldots, \mathsf{com}_{i,N_i-1})$ for $i \in [0,\tau)$.
- $\mathsf{com}_{i,j}$ uniquely defines $\mathsf{sd}_{i,j}$.[20]

Thus, it is only possible for $\mathcal{A}$ to open com successfully (i.e., $\mathsf{com} = \overline{\mathsf{com}} \neq \bot$) if

- Whenever $\Delta_i^* = \bot$, i.e., h_i could be fully extracted, then

$$\mathbf{Q}_i = \mathbf{V}_i^* \oplus \left[\delta_{i,0} \cdot \boldsymbol{u}_i^* \cdots \delta_{i,k_b} \cdot \boldsymbol{u}_i^*\right].$$

  This can be seen as follows: By the condition $\neg\mathsf{Fail}$, the reconstructed seeds $\mathsf{sd}'_{i,j}$ in VOLEReconstruct coincide with the (extracted) seeds $\mathsf{sd}_{i,j\oplus\Delta_i}$ for $j \in [0,N)$, except that $\mathsf{sd}'_{i,j} = \bot$. Proposition 5.2 asserts that in this case, the equality $\mathbf{Q}_i = \mathbf{V}_i^* \oplus \left[\delta_{i,0} \cdot \boldsymbol{u}_i^* \cdots \delta_{i,k_b} \cdot \boldsymbol{u}_i^*\right]$ holds, as required.[21]

[20] Note that attacks on the GGM tree, i.e., on PRG, are not possible here, as the GGM tree is only used to compress the decommitment for BAVC.Reconstruct. The checks and extraction rely only on $(\mathsf{sd}_{i,j}, \mathsf{com}_{i,j}) = \mathsf{LeafCommit}(\ldots)$, in the notation of BAVC.Reconstruct.

[21] More explicitly, it follows from Proposition 5.2 and $\mathbf{Q}_0 = \mathbf{V}_0^* + [\delta_{0,0} \cdot \mathbf{u}'_0 \cdots \delta_{0,k_0-1} \cdot \mathbf{u}'_0]$ as well as $\mathbf{Q}_i = \mathbf{Q}'_i + [\delta_{i,0} \cdot \mathbf{c}_i \cdots \delta_{i,k_i-1} \cdot \mathbf{c}_i] = \mathbf{V}_i^* + [\delta_{i,0} \cdot \mathbf{u}'_i \cdots \delta_{i,k_i-1} \cdot \mathbf{u}'_i] + [\delta_{i,0} \cdot \mathbf{c}_i \cdots \delta_{i,k_i-1} \cdot \mathbf{c}_i]$ for $i > 0$.

– Whenever $\Delta_i^* \neq \bot$, the choice Δ_i of $\mathcal{A}$ is Δ_i^*. Otherwise, $\mathcal{A}$ would lose: If $\Delta_i^* \neq \bot$, then at least one commitment $\mathsf{com}_{i,j}$ could not be extracted, hence this challenge must be used by $\mathcal{A}$ otherwise it could not successfully decommit (due to condition $\neg\mathsf{Fail}$). If challenge Δ_i^* is used, then by condition $\neg\mathsf{Fail}$ and Proposition 5.2, we again find that the equality $\mathbf{Q}_i = \mathbf{V}_i^* \oplus \left[\delta_{i,0} \cdot \boldsymbol{u}_i^* \cdots \delta_{i,k_b} \cdot \boldsymbol{u}_i^*\right]$ holds, as required.

Thus, we have shown that if Fail does not occur, extraction always succeeds and coincides with the opening in the Decommitment-Phase. This concludes the proof for FAEST.

Now, we explain the changes required for FAEST-EM. Essentially, we can apply the exact same argument, except that we reduce to almost injectivity (Definition 9.13) to ensure that extraction of the challenge commitments succeeds. The only difference, due to the different LeafHash, which occurs in the proof is an extraction failure when $\mathsf{com}_{i,j}$ has multiple preimages. To exclude this, we reduce to almost injectivity by outputting the challenge iv and $Q_{\mathsf{out}} = L$ commitments, namely, all the $\mathsf{com}_{i,j}$ which belong to the challenge commitment. Observe that an extraction failure due to multiple preimages for $\mathsf{com}_{i,j}$ now implies that the reduction wins the almost injectivity game. Thus, we constructed an adversary $\mathcal{B}$ with advantage $\mathsf{AdvInj}_{\mathcal{B}}^{\mathsf{PRG}_{2\lambda}}[L]$ and running time roughly that of $\mathcal{A}$. □

9.4.2 Hiding Next, we show that VOLECommit is hiding. To do this, we model our use of random IVs in PRGs with the following multi-target security game. The batch size L denotes the number of times the PRG is used with the same IV, while the number of queries denotes the number of independent IVs used. In the AES-CTR construction we use, increasing L degrades security by a factor of up to L, while increasing Q does not lead to any practical loss.

*Remark 9.19 (*BAVC *opening aborts).* Observe that BAVC.Open aborts for certain challenges $I = (\Delta_0, \dots, \Delta_{\tau-1})$. These aborts depend solely on I, and are independent from decom. In particular, it is efficiently verifiable if a choice I causes an abort.

Lemma 9.20 (VOLECommit is multi-hiding). *Let $\mathsf{H}_0, \mathsf{H}_1$ be random oracles. Consider the following multi-hiding experiment for $\mathsf{VOLECommit}^{\mathsf{H}_0,\mathsf{H}_1}$ and a Q-query adversary $\mathcal{A}$, defined for some arbitrary $\hat{\ell}$,* $\mathtt{param}$.

1. *Sample $b^* \leftarrow \{0,1\}$*
2. *For $j \in [1, Q]$:*
   - *$r^j \leftarrow \{0,1\}^\lambda$, $\mathsf{iv}^j \leftarrow \{0,1\}^{128}$*
   - *$(\mathsf{com}^j, \mathsf{decom}^j, (\mathbf{c}_i^j)_{i\in[1,\tau)}, \mathbf{u}^j, \mathbf{V}^j) := \mathsf{VOLECommit}(r^j, \mathsf{iv}^j, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$*
   - *Sample $\mathsf{chall}_3^j \leftarrow \{0,1\}^{\lambda - w_{\mathsf{grind}}} \times 0^{w_{\mathsf{grind}}}$ uniformly, conditioned on $\mathsf{BAVC.Open}$ not aborting for $I^j = (\Delta_0^j, \dots, \Delta_{\tau-1}^j) := \mathsf{DecodeChall}_3(\mathsf{chall}_3^j; \mathtt{param})$ (cf. Remark 9.19).*
   - *$\mathsf{decom}_{I^j} := \mathsf{BAVC.Open}(\mathsf{decom}^j, I^j)$*
   - *Let $\mathbf{u}_{\$}^j \leftarrow \mathbb{F}_2^{\hat{\ell}}$ and define*

$$(\overline{\mathbf{u}}^j, \overline{\mathbf{c}}_1^j, \dots, \overline{\mathbf{c}}_{\tau-1}^j) = \begin{cases} (\mathbf{u}^j, \mathbf{c}_1^j, \dots, \mathbf{c}_{\tau-1}^j) & \text{if } b^* = 0 \\ \text{random from } (\mathbb{F}_2^{\hat{\ell}})^\tau & \text{if } b^* = 1 \end{cases}$$

3. *$b \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1}((\overline{\mathbf{u}}^j, (\overline{\mathbf{c}}_i^j)_{i\in[1,\tau)}, (\mathsf{decom}^j)_{i\in[0..\tau)}, \mathsf{chall}_3^j, \mathsf{com}^j)_{j\in[1,N]})$*
4. *Output 1 (success) if $b = b^*$. Else output 0 (failure).*

Write $\mathsf{PRG}_{\mathsf{len}(s)} = \mathsf{PRG}(s; \mathsf{len})$ and $\beta = \frac{\lambda}{128}$. Let G_i^* be the hiding game with $b^* = i$. Then we have

$$\Pr[\mathsf{G}_0^* = 1] \leq e^{Q\tau\left(4\lceil\log(L)\rceil\beta^2 + 16\beta^2 N_0^2 + (\lceil\hat{\ell}/128\rceil)^2\right)/2^{128}} \cdot \Big(\Pr[\mathsf{G}_1^* = 1] + Q\tau \cdot \Big(\lceil\log(L)\rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}_1}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRG}_{\mathcal{B}_2}^{\mathsf{LeafCommit}}[N_0] + \mathsf{AdvPRP}_{\mathcal{B}_3}^{\mathsf{PRG}_{\hat{\ell}}}[1]\Big)\Big)$$

where by abuse of notation, we write $\mathsf{AdvPRG}^{\mathsf{LeafCommit}}$ *which is*

- $\mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}$ *in* FAEST
- $\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}$ *in* FAEST-EM

as the advantage pertains to the respective PRG *evaluation in* LeafCommit*. The same holds against quantum adversaries, in which case the advantage terms are with respect to quantum adversaries.*

Note that we can sample the challenges $(\mathsf{chall}_3^j)_{j\in[1,Q]}$ before committing without changing the adversary's view (because whether or not BAVC.Open rejects a challenge is independent from the actual commitment). Hence, all decommitment indices for all VC commitments are known at commitment time. That is, we only assert selective security.

Proof. As remarked above, we assume that $(\mathsf{chall}_3^j)_{j\in[1,Q]}$ is sampled before the commitments are generated, so that we are obviously in a selective security setting for BAVC openings. We argue in two main steps: First, we replace the hidden outputs of the BAVC commitments with true randomness. Then, we replace the output of ConvertToVOLE. In the end, each $\mathbf{u}_i^j$ will be uniformly random, as required. (This follows since $\mathbf{u} = \sum_{i\in[0,\tau)} \mathsf{PRG}(\mathsf{sd}_i, \mathsf{iv}, i; \hat{l})$ in ConvertToVOLE.. Thus, if a single summand is truly random, so is $\mathbf{u}$ output by ConvertToVOLE.)

Game G_0 (Real hiding experiment with $b = 0$). By definition, $\mathsf{G}_0 = \mathsf{G}_0^*$ is the hiding experiment with the challenge bit $b^* = 0$, i.e., the outputs $(\overline{\mathbf{u}}^j, \overline{\mathbf{c}}_1^j, \dots, \overline{\mathbf{c}}_{\tau-1}^j)$ corresponds to the generated commitment.

For simplicity, we describe the reductions first for $Q = 1$ instead of the multi-challenge setting and omit the superscript j. Since all VOLECommit instances are independent, it is straightforward to generalize it to $Q > 1$, as we will show in the final step.

Game G_1 (Randomize hidden GGM tree seeds). In game G_1, we replace all hidden seeds k_{α_i} in the (de)commitment with truly random seeds.

We make use of the proof that the GGM construction is a selectively secure puncturable PRF from a length-doubling PRG, and use the hybrids to (eventually) randomize the hidden seeds. Following that proof, we successively replace PRG calls along the path from the root to the leaf $\alpha_i = \mathsf{BAVC.PosInTree}(i, \Delta_i)$ (for $i \in [0, \tau)$) by truly random outputs. After at most $\tau\lceil\log(L)\rceil$ hybrid steps, the leaf seeds k_{α_i} are replaced with random values. By applying Corollary 9.6 and Remark 9.7 we arrive at

$$\begin{aligned}\Pr[\mathsf{G}_0 = 1] &\le (e^{(2\beta)^2/2^{128}})^{\tau\lceil\log(L)\rceil}(\Pr[\mathsf{G}_1 = 1] + \sum_{i=1} \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_{1.i}}[1]) \\ &= e^{4\tau\lceil\log(L)\rceil\beta^2/2^{128}}(\Pr[\mathsf{G}_1 = 1] + \tau\lceil\log(L)\rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1])\end{aligned}$$

since each call to $\mathsf{PRG}(k_\alpha, \mathsf{iv}, \alpha; 2\lambda)$ in line (7) of BAVC.Commit outputs $2\beta = \frac{2\lambda}{128}$ blocks for $T = 1$ queries and we have at most $\tau\lceil\log(L)\rceil$ hybrid steps.

Game G_2 (Randomize hidden LeafCommit outputs). In game G_2 for FAEST, we replace the outputs $(\mathsf{sd}_{i,\Delta_i}, \mathsf{com}_{i,\Delta_i})$ corresponding to the hidden leaves $\alpha_0, \dots, \alpha_{\tau-1}$ in line XXX(10) of BAVC.Commit by the image $\mathsf{LeafHash}(\mathsf{uhash}, \mathsf{sd}_{i,\Delta_i} \| s_0 \| s_1 \| s_2)$ for random $(\mathsf{sd}_{i,\Delta_i}, s_0, s_1, s_3) \leftarrow \{0,1\}^{4\lambda}$. Note that $\{0,1\}^{4\lambda} \ni \mathbf{x} \mapsto \mathsf{LeafHash}(\mathsf{uhash}, \mathbf{x}) \in \{0,1\}^{3\lambda}$ maps uniform distributions to uniform distributions by construction. Hence, the output $(\mathsf{sd}_{i,\Delta_i}, \mathsf{com}_{i,\Delta_i})$ will be uniform.

In game G_2 for FAEST-EM, we instead sample $\mathsf{com}_{i,\Delta_i} \leftarrow \{0,1\}^{2\lambda}$. (Since we consider hidden sd_{i,Δ_i}, we do not need to sample it in FAEST-EM.) For simplicity, we focus on FAEST.LeafCommit below.

Since in game G_1, all seeds k_{α_i} are truly random, we can apply pseudorandomness of LeafCommit to replace the outputs $(\mathsf{sd}_{i,\Delta_i}, \mathsf{com}_{i,\Delta_i})$ corresponding to leaves $\alpha_0, \dots, \alpha_{\tau-1}$.

Again, we apply Corollary 9.6 and Remark 9.7 to arrive at

$$\begin{aligned}\Pr[\mathsf{G}_1 = 1] &= (e^{(4\beta)^2 N_0^2/2^{128}})^\tau (\Pr[\mathsf{G}_2 = 1] + \sum_{i=1}^{\tau} \mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}_{\mathcal{B}_{2.i}}[N_i]) \\ &\leq e^{16\tau\beta^2 N_0^2/2^{128}} (\Pr[\mathsf{G}_2 = 1] + \tau \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}_{\mathcal{B}_2}[N_0])\end{aligned}$$

since each call $\mathsf{PRG}(k_{\alpha_i}, \mathsf{iv}, \alpha; 4\lambda)$ outputs $4\beta = \frac{4\lambda}{128}$ blocks for $T = N_i$ queries, and $N_i \leq N_0$ by definition of N_i (see Table 5.1).

Coming back to FAEST-EM, the only difference is that instead of $\mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}_{\mathcal{B}_{2.i}}[N_i]$, we reduce to a PRG with 2λ output length, i.e., we have $\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_{2.i}}[N_i]$ in the above and the factor 16 becomes 4.

Game G_3 (Switch ConvertToVOLE to random). In game G_3, we replace the output of $\mathbf{r}_{0,0} = \mathsf{PRG}(\mathsf{sd}_{i,\Delta_i}, \mathsf{iv}, i; \hat{l})$ in line (2) of that ConvertToVOLE computation called from VOLECommit line (4) by true randomness. In the output of ConvertToVOLE, $\mathbf{u}$ equals $\mathbf{u} = \sum_{i\in[0,\tau)} \mathsf{PRG}(\mathsf{sd}_i, \mathsf{iv}, i; \hat{l})$ and thus if $\mathbf{r}_{i,0}$ is truly random, so is $\mathbf{u}$. Hence, all $\mathbf{u}_i$ in VOLECommit line (4) are truly random. Consequently, the outputs $\mathbf{u}$ and $\mathbf{c}_i = \mathbf{u}_i \oplus \mathbf{u}$ (from lines (7) and (9)) are truly random in G_3.

By the change in game G_2, we know that the seed sd_{i,Δ_i} which is input into ConvertToVOLE in line (4) of VOLECommit is truly random, so we can reduce to PRG security as before. We get

$$\begin{aligned}\Pr[\mathsf{G}_2 = 1] &= (e^{(\lceil \hat{\ell}/128 \rceil)^2/2^{128}})^\tau (\Pr[\mathsf{G}_3 = 1] + \sum_{i=1}^{\tau} \mathsf{AdvPRP}^{\mathsf{PRG}_{\hat{\ell}}}_{\mathcal{B}_{3.i}}[1]) \\ &\leq e^{\tau(\lceil \hat{\ell}/128 \rceil)^2/2^{128}} (\Pr[\mathsf{G}_3 = 1] + \tau \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{\hat{\ell}}}_{\mathcal{B}_3}[1])\end{aligned}$$

since we have $\frac{\lceil \hat{\ell} \rceil}{128}$ blocks of output in $\mathsf{PRG}(\mathsf{sd}_i, \mathsf{iv}, i; \hat{\ell})$ in for the i-th run of ConvertToVOLE (for $i \in [0, \tau - 1]$), where each only has $T = 1$ queries. All in all, we have shown that modifying the single-challenge experiment from $b^* = 0$ to $b^* = 1$ is indistinguishable.

Final step: Handling $Q > 1$. Let G_4 the multi-challenge game with $b^* = 1$. By plugging the above inequalities for games 1 to 3 in another, we see that game G_3 (for $Q = 1$) satisfies

$$\begin{aligned}\Pr[\mathsf{G}_0 = 1] \leq e^{(4\tau\lceil \log(L) \rceil \beta^2 + 16\tau\beta^2 N_0^2 + \tau(\lceil \hat{\ell}/128 \rceil)^2)/2^{128}} \cdot \Big(&\Pr[\mathsf{G}_3 = 1] + \\ &\tau\lceil \log(L) \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1] + \tau \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}_{\mathcal{B}_2}[N_0] + \tau \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{\hat{\ell}}}_{\mathcal{B}_3}[1]\Big)\end{aligned}$$

Applying the same reasoning for a hybrid with Q commitments, we therefore arrive at

$$\begin{aligned}\Pr[\mathsf{G}_0 = 1] &\leq (e^{(4\tau\lceil \log(L) \rceil \beta^2 + 16\tau\beta^2 N_0^2 + \tau(\lceil \hat{\ell}/128 \rceil)^2)/2^{128}})^Q \cdot \Big(\Pr[\mathsf{G}_4 = 1] + \\ &\qquad Q\tau\lceil \log(L) \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1] + Q\tau \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}_{\mathcal{B}_2}[N_0] + Q\tau \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{\hat{\ell}}}_{\mathcal{B}_3}[1]\Big) \\ &\leq e^{Q\tau\left(4\lceil \log(L) \rceil \beta^2 + 16\beta^2 N_0^2 + (\lceil \hat{\ell}/128 \rceil)^2\right)/2^{128}} \cdot \Big(\Pr[\mathsf{G}_4 = 1] + \\ &\qquad Q\tau \cdot \Big(\lceil \log(L) \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1] + \mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}_{\mathcal{B}_2}[N_0] + \mathsf{AdvPRP}^{\mathsf{PRG}_{\hat{\ell}}}_{\mathcal{B}_3}[1]\Big)\Big)\end{aligned}$$

This concludes the claim for the multi-challenge setting. Note that the contribution of $\mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}$ changes to $\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}$ for FAEST-EM (and the term $16\beta^2 N_0^2$ could be reduced to $4\beta^2 N_0^2$).

Observing that we did not make use of $\mathcal{A}$'s structure in any way yields the post-quantum security statement. □

9.4.3 Unpredictability We define unpredictability in a slightly non-standard manner, namely we let the adversary $\mathcal{A}$ output a bit and replace that bit with 0 if it succeeded in guessing. This is tailored to our application of unpredictability, where $\mathcal{A}$ will actually be a (hybrid) game which outputs 1 if the EUF-CMA adversary succeeds. Hence, replacing the output 1 with 0 means that the EUF-CMA adversary loses the modified EUF-CMA game if it predicts a commitment.

Lemma 9.21 (VOLECommit is unpredictable). *Fix some $\hat{\ell} \in \mathbb{N}$ and admissible* $\mathtt{param}$, $\mathtt{param}_{\mathsf{VOLE}}$. *Let $\mathcal{A}$ be an adversary in the following game* $\mathsf{ExpUnpred}_{b,\mathcal{A}}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}}(\lambda)$*:*

1. $(\mathcal{L}_{\mathsf{com}}, \mathsf{state}) \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1}(1^\lambda)$. *Here $\mathcal{L}_{\mathsf{com}}$ is list of commitment guesses.*
2. $r \leftarrow \{0,1\}^\lambda$, $\mathsf{iv} \leftarrow \{0,1\}^{128}$
3. $C = (\mathsf{com}, \mathsf{decom}, \mathbf{c}_1, \dots, \mathbf{c}_{\tau-1}, \mathbf{u}, \mathbf{V}) \leftarrow \mathsf{VOLECommit}(r, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$
4. *Let* $b' \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1}(\mathsf{state}, C)$.
5. *If* $(\mathsf{com}, \mathbf{c}_1, \dots, \mathbf{c}_{\tau-1}) \in \mathcal{L}_{\mathsf{com}}$ *and* $b = 1$ *output* 0. *Else output* b'.

For brevity, let G_b *denote* $\mathsf{ExpUnpred}_{b,\mathcal{A}}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}}$ *for* $b \in \{0,1\}$. *Also let* $\beta = \frac{\lambda}{128} \in \mathbb{N}$. *For any adversary $\mathcal{A}$ which makes at most Q_1 queries to* H_1, *and outputs lists $\mathcal{L}$ of size at most Q_{com} we have*

$$\begin{aligned}\Pr[\mathsf{G}_0 = 1] \leq & e^{(4\lceil \log(L)\rceil \beta^2 + 16\beta^2)/2^{128}} \cdot \Big(\Pr[\mathsf{G}_1 = 1] \\ & + \lceil \log(L)\rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1] + (2Q_1 + Q_{\mathsf{com}}) \cdot 2^{-2\lambda}\Big)\end{aligned}$$

where by abuse of notation, we write $\mathsf{AdvPRP}^{\mathsf{LeafCommit}}$ *for* $\mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}$ *in* FAEST, *and* $\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}$ *in* FAEST-EM.

We note that, intuitively, unpredictability follows immediately from hiding (by replacing the outputs $\mathbf{c}_i$ with true randomness). However, we claim a concretely better bound on the adversary success, which we argue below.

Proof. The proof is a simpler version of hiding: Apply the same game hops as in games G_0 to G_2 the proof of Lemma 9.20 but only to randomize a single leaf, say $(i,j) = (0,0)$ and denote the resulting game as G_2'. (Do not continue to G_3, as we need not randomize $\mathbf{u}, \mathbf{c}_1, \dots, \mathbf{c}_{\tau-1}$.) Then, the leaf commit $\mathsf{com}_{0,0} \in \{0,1\}^{2\lambda}$ is truly random and we have

$$\begin{aligned}\Pr[\mathsf{G}_0 = 1] \leq & e^{(4\lceil \log(L)\rceil \beta^2 + 16\beta^2)/2^{128}} \cdot \Big(\Pr[\mathsf{G}_2' = 1] \\ & + \lceil \log(L)\rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1]\Big) \qquad (4)\end{aligned}$$

Further, h_0 is uniformly random, unless H_1 was queried before on the random $\mathsf{com}_{0,1} \in \{0,1\}^{2\lambda}$, which happens with probability at most $Q_1/2^{2\lambda}$. Further, com is uniformly random, unless H_1 was queried before on an input starting with the random $h_0 \in \{0,1\}^{2\lambda}$, which happens with probability at most $Q_1/2^{2\lambda}$. Finally, note that since com is uniformly random, also the probability that $(\mathsf{com}, \mathsf{decom}, \mathbf{c}_1, \dots, \mathbf{c}_{\tau-1}, \mathbf{u}, \mathbf{V}) \in \mathcal{L}_{\mathsf{com}}$ is at most $Q_{\mathsf{com}}/2^{2\lambda}$. Hence

$$\Pr[\mathsf{G}_2' = 1] \leq \Pr[\mathsf{G}_1] + (2Q_1 + Q_{\mathsf{com}}) \cdot 2^{-2\lambda}.$$

We arrive at the claimed advantage. □

9.5 EUF-KO

Building on the previous analysis, we now argue security of the overall signature scheme. We first consider EUF-KO security, that is, a key-only attack where the adversary must output a forgery given only the public key. Then, we extend this to EUF-CMA security, where the adversary also has access to a signing oracle.

Definition 9.22 (EUF-CMA and EUF-KO Security). *A signature scheme* $\mathsf{SIG} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *is* existentially unforgeable against chosen message attacks (EUF-CMA) *(resp.* existentially unforgeable against key-only attacks (EUF-KO)*) in the random oracle model if any PPT adversary has at most negligible advantage of winning the EUF-CMA (resp. EUF-KO) security game (Figure 9.1). We denote by* $\mathsf{AdvEUFCMA}^{\mathsf{SIG}}_{\mathcal{A}} = \Pr[\mathsf{ExpEUFKO}^{\mathsf{SIG}}_{\mathcal{A}} = 1]$ *(resp.* $\mathsf{AdvEUFKO}^{\mathsf{SIG}}_{\mathcal{A}} = \Pr[\mathsf{ExpEUFKO}^{\mathsf{SIG}}_{\mathcal{A}} = 1]$*) the advantage of an adversary* $\mathcal{A}$*.*

$\boxed{\mathsf{ExpEUFCMA}}$ $\mathsf{ExpEUFKO}$	$\mathsf{OSign}(\mathsf{msg})$
$1:\ \mathcal{M} := \emptyset$	$1:\ \mathcal{M} := \mathcal{M} \cup \{\mathsf{msg}\}$
$2:\ (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}^{\mathsf{H}}(\mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$	$2:\ \textbf{return}\ \mathsf{SIG.Sign}(\mathsf{sk}, \mathsf{msg})$
$3:\ (\mathsf{msg}, \sigma) \leftarrow \mathcal{A}^{\mathsf{H}, \boxed{,\mathsf{OSign}}}(\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}, \mathsf{pk})$	
$4:\ \textbf{return}\ \mathsf{Verify}^{\mathsf{H}}(\mathsf{msg}, \mathsf{pk}, \sigma; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}) \wedge \mathsf{msg} \notin \mathcal{M}$	

Fig. 9.1: The $\boxed{\mathsf{EUF\text{-}CMA}}$ and EUF-KO security experiments in the random oracle model; H is a short-hand for any number of random oracles. Only for EUF-CMA does $\mathcal{A}$ get access to OSign.

For the proof of EUF-KO security, we first need the following technical fact.

Lemma 9.23. *Let* $F\colon \mathcal{K} \to \mathcal{Y}$ *be a PRG and let* $\mathcal{A}$ *be any (potentially unbounded) distinguisher, i.e., some predicate* $\mathcal{A}\colon \mathcal{Y} \to \{0,1\}$*. Then*

$$\Pr_{y \leftarrow \mathsf{im}(F)}[\mathcal{A}(y) = 1] \leq \frac{\mathsf{dom}(F)}{\mathsf{im}(F)} \cdot \Pr_{x \leftarrow \mathsf{dom}(F)}[\mathcal{A}(F(x)) = 1]$$

At a high level, Lemma 9.23 relates the uniform distribution over the image of the PRG F and its image distribution, and allows us to switch from the former to the latter.

Proof. Since any predicate $\mathcal{A}$ defines a set $E = \{\mathcal{A}(y) = 1\} \subseteq \mathcal{Y}$, the claim will follow from ∞-divergence (Lemma 9.5) of the probability distributions I and J over $\mathcal{Y}$, where I is the uniform distribution over $\mathsf{im}(F)$ and J is the distribution given by $F(x)$ for uniform $x \leftarrow \mathcal{X}$. Clearly, if $\Pr[I = y] = 0$ then also $\Pr[J = y] = 0$, hence $\Pr[I = y]/\Pr[J = y] \leq \infty$. Moreover, for $y \in \mathsf{supp}(I)$, we have $\Pr[I = y] = \frac{1}{|\mathsf{im}(F)|}$ and $\Pr[J = y] \geq \frac{1}{|\mathsf{dom}(X)|}$. Thus

$$\frac{\Pr[I = y]}{\Pr[J = y]} \leq \frac{|\mathsf{dom}(X)|}{|\mathsf{im}(X)|}$$

and the claim follows by Lemma 9.5. □

We are now ready to prove EUF-KO of FAEST.

Theorem 9.24 (FAEST is EUF-KO). *Let* H_0, H_1 *and* $\mathsf{H}^0_2, \mathsf{H}^1_2, \mathsf{H}^2_2, \mathsf{H}^3_2$ *and* H_4 *be modeled as random oracles and* $(\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}, \mathtt{param}_{\mathsf{VOLE}})$ *be parameters of the* FAEST *signature. Let* $\mathcal{A}$ *be an adversary which, for simplicity, runs* FAEST.Verify *on its purported forgery and outputs* $\perp$ *if verification fails. Let* Q_0*,* Q_1*,* $Q_{2,i}$*,* Q_3*,* Q_4*, and* Q_5 *, denote upper*

bounds on the number of queries of $\mathcal{A}$ *to* H_0, H_1, H_2^i, H_3, H_4, *respectively, and let* $Q_2 = Q_{2,1} + Q_{2,2} + Q_{2,3}$. *Suppose that* $\hat{\ell} \leq 2^{13}, \ell' \leq 2^{13}$ *and* $2\log_2(\tau) \leq \min\{B, 50\}$. *Then*

$$\mathbf{Adv}^{\mathsf{EUF\text{-}KO}}_{\mathcal{A}} \leq \frac{1}{1 - \frac{\mathsf{dom}(\mathsf{OWF})}{\mathsf{cod}(\mathsf{OWF})}} \cdot \left(\mathsf{AdvSnd}^{\mathsf{FAEST}}_{\mathcal{B}_2} + \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1}\right)$$

where OWF *is the OWF (or more precisely, PRG) defined by* $\mathtt{param}_{\mathsf{OWF}}$, *and*

$$\begin{aligned}\mathsf{AdvSnd}^{\mathsf{FAEST}}_{\mathcal{B}_2} \leq & (Q_{2,0} + Q_{2,1} + Q_{2,1} + Q_{2,3} + 4)^2 / 2^{2\cdot\lambda} \\ & + \frac{(Q_1+1)^2}{2^{2\lambda}} + \mathsf{AdvLeafColl}^{\mathsf{LeafCommit}} \\ & + (Q_{2,1} + Q_{2,1} + Q_{2,3} + 9) \cdot 3/2^{\lambda}\end{aligned}$$

with

- $\mathsf{AdvLeafColl}^{\mathsf{LeafCommit}}_{\mathcal{B}'} \leq \tau Q_0 \cdot 2^{2\lambda} \cdot \varepsilon_{\mathsf{uhash}} \leq \tau Q_0 \cdot 2^{-\lambda}$ *for* FAEST.
- $\mathsf{AdvLeafColl}^{\mathsf{LeafCommit}}_{\mathcal{B}'} \leq (Q_{\mathsf{AES}} + 1)^2 \cdot 2^{-2\lambda}$ *for* FAEST-EM.

For FAEST and FAEST-EM, the key generation algorithm KeyGen excludes keys with bits $\mathbf{k}[0] = \mathbf{k}[1] = 1$. Thus, $\frac{\mathsf{dom}(\mathsf{OWF})}{\mathsf{cod}(\mathsf{OWF})} = \frac{3}{4}$ and $\frac{1}{1-\frac{\mathsf{dom}(\mathsf{OWF})}{\mathsf{cod}(\mathsf{OWF})}} = 4$.

Remark 9.25. As in Lemma 9.15, the assumption that $\mathcal{A}$ checks that the forgery verifies, ensures that the game does not query undefined values of any random oracle for the forgery verification. For a general $\mathcal{A}$, we can always replace it by an $\mathcal{A}'$ where the check is added. This modification increases the number of random oracle queries to at most $Q_0 + 1$, $Q_1 + \tau + 1$, $Q_{2,1} + 1$, $Q_{2,2} + 1$, $Q_{2,3} + 1$ for H_0, H_1, H_2^1, H_2^2, H_2^3 respectively. Since $Q_0, Q_1, Q_2 \gg \tau$, the effect on concrete security is minimal.

Before we argue security, we provide a high-level outline of our strategy. Observe that the signature is a Fiat–Shamir transformed VOLE-in-the-head proof of knowledge of a secret key $\mathbf{k}$ to the public key $\mathsf{pk} = (\mathbf{x}, \mathbf{y})$. The proven relation asserts $\mathsf{OWF}_{\mathbf{x}}(\mathbf{k}) = \mathbf{y} \in \{0,1\}^{\lambda}$. Observe also that we ensure in KeyGen that $\mathbf{k}[0] = 1 = \mathbf{k}[1]$ does not occur in a secret key. In particular, the effective domain of $\mathsf{OWF}_{\mathbf{x}}$ is a $\frac{3}{4}$th fraction $\mathcal{K}_\lambda$ of $\{0,1\}^{\lambda}$. Thus, at least $\frac{3}{4}$th of all $\mathbf{y} \in \{0,1\}^{\lambda}$ have no preimage. Hence, for such pk, there exists no secret key and the statement is false. This allows us to reduce to *soundness instead of knowledge* soundness.

Our security reduction for EUF-KO goes through several steps to eventually reduce security of the Fiat–Shamir transformed VOLE-in-the-head proof system to the round-by-round soundness of a derived interactive oracle proof. The first step switches the public key $\mathsf{pk} = (\mathbf{x}, \mathbf{y})$ to truly random. In particular, with probability at least $\frac{1}{4}$, the public key is not in the relation, and thus a forgery breaks soundness of the proof system. This is unlike in [BBD+23b], and follows the approach of Katz and Wang [KW03], which allows us to rely on *soundness* of the derived interactive oracle proof.

Next, we make explicit the interactive proof underlying FAEST, and reduce EUF-KO security to state-restoration security of said interactive proof. By (inefficiently) extracting the commitments in the proofs, we turn an adversary against the interactive proof into an (inefficient) adversary against an interactive oracle proof (IOP) which captures FAEST with *idealized* VOLECommit. For this IOP, we can apply the theorem that round-by-round (RBR) soundness implies state-restoration soundness, and thus it suffices to prove RBR soundness of the IOP induced by FAEST. This then follows from information-theoretic arguments for the VOLE consistency check of [Roy22], universal hashing, and the QuickSilver check [YSWW21], similar to [BBD+23b].

What remains is to upgrade EUF-KO security to EUF-CMA security. To do this, we need to implement the signing oracle without the secret key. We do this as usual: We run an honest-verifier zero-knowledge simulator and we program the simulated challenges into the Fiat–Shamir random oracles $\mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$. The unpredictability of BAVC commitments ensures that it is unlikely that programming the RO fails (because the adversary will not have queried the input belonging to the programmed output beforehand). To ensure that simulated and real transcripts are indistinguishable, we rely on the hiding property of the BAVC commitment scheme. We note here a technical detail in our proof, namely, that the fixed blocksize of 128 bits of AES necessitates the "divergence based" approach for hiding, and simulatability and relies on the upper bound of 2^{64} signature queries, independent of the security parameter. Aside from this, the upgrade from EUF-KO to EUF-CMA is completely standard.

The difference in FAEST and FAEST-EM lies in the extractability assumption needed for (leaf) commitments (and the choice of OWF). For FAEST, our construction is statistically secure (in the random oracle model), but this requires a universal hash with 3λ output length per leaf commit. For FAEST-EM, we make a stronger assumption (Definition 9.13) on $\mathsf{PRG}_{2\lambda}$, which allows us to keep the 2λ output length, which is minimal by the birthday bound. Thus, the proofs for FAEST and FAEST-EM are completely analogous, except for this difference in extraction of leaf commitments.

Proof of Theorem 9.24. We argue via game hops. We prove the result for FAEST first, and then explain what changes for FAEST-EM We write G_i for the output of the i-th game.

Game G_0: This is the real EUF-KO security game. As in the EUF-KO game, we let sk, pk be the generated challenge secret and public key, respectively. We have

$$\Pr[\mathsf{G}_0 = 1] = \mathsf{AdvEUFKO}_{\mathcal{A}}^{\mathsf{FAEST}}$$

Game G_1: In this game, we change $\mathsf{pk} := (\mathbf{x}, \mathbf{y})$ from honestly generated to truly random, i.e., to $\mathbf{x} \leftarrow \{0,1\}^{N_{\text{st-bits}}}$, $\mathbf{y} \leftarrow \{0,1\}^{N_{\text{st-bits}}}$. By a straightforward reduction to PRG security, we obtain an adversary whose running time is roughly that of the EUF-KO game. We have

$$|\Pr[\mathsf{G}_1 = 1] - \Pr[\mathsf{G}_0 = 1]| \leq \mathsf{AdvPRG}_{\mathcal{B}_1}^{\mathsf{OWF}}[1]$$

as there is a straightforward reduction to a distinguisher $\mathcal{B}_1$ which gets a single sample $\mathbf{y}$ that is either the OWF-output $\mathbf{y} = \mathsf{OWF}_{\mathbf{x}}(\mathbf{k}) := F_{\mathbf{k}}(\mathbf{x})$ (for secret $\mathbf{k} \leftarrow \{0,1\}^{\lambda}$ and public parameters $\mathbf{x} \leftarrow \{0,1\}^{N_{\text{st-bits}}}$) or a random $\mathbf{y} \leftarrow \{0,1\}^{N_{\text{st-bits}}}$.

Splitting Game G_1: Now, we derive a bound on the probability $\Pr[\mathsf{G}_1 = 1]$ by distinguishing two events: The event I, that $\mathbf{y} \in \mathsf{im}(\mathsf{OWF}_{\mathbf{x}})$, i.e., $(\mathbf{x}, \mathbf{y})$ is a valid public key, and the negated event $\neg I$, where there exists no key such that $F_{\mathbf{k}}(\mathbf{x}) = \mathbf{y}$. Observe that we can apply Corollary 9.35 to bound the advantage that $\mathcal{A}$ wins G_1 *conditioned on* $\neg I$; in this case, the EUF-KO adversary $\mathcal{A}$ breaks the soundness of the proof system. For the advantage p in Corollary 9.35, we write AdvSnd below, to indicate the soundness of the Fiat–Shamir transformed proof system of FAEST. We have that

$$\Pr[\mathsf{G}_1 = 1 \mid \neg I] \leq \mathsf{AdvSnd}_{\mathcal{B}_2}^{\mathsf{FAEST}}.$$

On the other hand, we have from Lemma 9.23

$$\Pr[\mathsf{G}_1 = 1 \mid I] \leq \frac{|\mathsf{dom}(\mathsf{OWF})|}{|\mathsf{im}(\mathsf{OWF}_{\mathbf{x}})|} \cdot \Pr[\mathsf{G}_0 = 1]$$

the left hand side is G_1 with a uniformly random image $\mathbf{y}$ of $\mathsf{OWF}_{\mathbf{x}}$ and the right hand side is the distribution of $\mathbf{y}$ under $\mathsf{OWF}_{\mathbf{x}}$. Note that we omit $\mathbf{x}$ from domain and codomain of OWF, since these are independent of the parameter $\mathbf{x}$. Thus, we derive

$$\begin{aligned}\Pr[\mathsf{G}_1 = 1] &= \Pr[I] \cdot \Pr[\mathsf{G}_1 = 1 \mid I] + \Pr[\neg I] \cdot \Pr[\mathsf{G}_1 = 1 \mid \neg I] \\ &\leq \Pr[I] \cdot \frac{|\mathsf{dom}(\mathsf{OWF})|}{|\mathsf{im}(\mathsf{OWF}_{\mathbf{x}})|} \Pr[\mathsf{G}_0 = 1] + \Pr[\neg I] \cdot \mathsf{AdvSnd}^{\mathsf{FAEST}}_{\mathcal{B}_2} \\ &\leq \frac{|\mathsf{dom}(\mathsf{OWF})|}{|\mathsf{cod}(\mathsf{OWF})|} \Pr[\mathsf{G}_0 = 1] + \mathsf{AdvSnd}^{\mathsf{FAEST}}_{\mathcal{B}_2}\end{aligned}$$

where the last step uses $\Pr[I] = \frac{|\mathsf{im}(\mathsf{OWF}_{\mathbf{x}})|}{|\mathsf{cod}(\mathsf{OWF})|}$ and $\Pr[\neg I] \leq 1$. The latter holds, since a uniformly random $\mathsf{pk} \leftarrow \mathsf{cod}(\mathsf{OWF})$ lies in $\mathsf{im}(\mathsf{OWF})$ with probability $\frac{\mathsf{im}(\mathsf{OWF}_{\mathbf{x}})}{\mathsf{cod}(\mathsf{OWF})}$. By the first game hop, we have

$$\Pr[\mathsf{G}_0 = 1] \leq \Pr[\mathsf{G}_1 = 1] + \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1} \leq \frac{\mathsf{dom}(\mathsf{OWF})}{\mathsf{cod}(\mathsf{OWF})} \Pr[\mathsf{G}_0 = 1] + \mathsf{AdvSnd}^{\mathsf{FAEST}}_{\mathcal{B}_2} + \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1}$$

and resolving for $\Pr[\mathsf{G}_0 = 1]$ yields

$$\Pr[\mathsf{G}_0 = 1] \leq \frac{1}{1 - \frac{\mathsf{dom}(\mathsf{OWF})}{\mathsf{cod}(\mathsf{OWF})}} \cdot \left(\mathsf{AdvSnd}^{\mathsf{FAEST}}_{\mathcal{B}_2} + \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1}\right) \tag{5}$$

which yields the claim after plugging in the advantage from Corollary 9.35.

□

9.5.1 Interactive proof induced by FAEST We define interactive proofs (IP) and present the one derived from the FAEST signature scheme.

Definition 9.26 (Interactive proof system). *(Extracted from [CY24, Section 13.1].) An* interactive proof (IP) *for relation $\mathcal{R}$ in the random oracle model is a tuple of algorithms $\mathrm{IP} = (\mathcal{P}_{\mathrm{IP}}, \mathcal{V}_{\mathrm{IP}})$ where the prover and the verifier interact over $\mathsf{k} \in \mathbb{N}$ rounds as follows, with access to one or more random oracles.*

- *The IP prover $\mathcal{P}_{\mathrm{IP}}$ receives as input the instance $\mathbb{x}$ and a witness $\mathbb{w}$ where $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ and the IP verifier $\mathcal{V}_{\mathrm{IP}}$ receives as input only the instance $\mathbb{x}$.*
- *In each round $i \in [\mathsf{k}]$, the prover $\mathcal{P}_{\mathrm{IP}}$, with access to one or more random oracles, sends a message α_i and then the verifier $\mathcal{V}_{\mathrm{IP}}$ responds with a (possibly empty) message ρ_i.*
- *After the interaction, $\mathcal{V}_{\mathrm{IP}}$ outputs a bit, where 1 denotes acceptance and 0 denotes rejection, computed based on the instance $\mathbb{x}$, the random oracle queries, the prover messages $(\alpha_i)_{i\in[\mathsf{k}]}$ and the verifier messages $(\rho_i)_{i\in[\mathsf{k}]}$.*

Definition 9.27 ([CY24, Definition 13.2.1]). *The* state-restoration experiment *for $\mathrm{IP} = (\mathcal{P}_{\mathrm{IP}}, \mathcal{V}_{\mathrm{IP}})$, in the random oracle model, with salt sizes $s_i \in \mathbb{N}$, randomness functions $\mathsf{rnd} = (\mathsf{rnd}_i)_{i\in[\mathsf{k}]} \in \mathcal{U}((\mathsf{r}_i)_{i\in[\mathsf{k}]})$, and IP state-restoration prover $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ is defined in Figure 9.2. We say that $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ is t-*move *if it exits the* **while** *loop after at most t iterations.*

Definition 9.28 ([CY24, Definition 13.2.2]). *$\mathrm{IP} = (\mathcal{P}_{\mathrm{IP}}, \mathcal{V}_{\mathrm{IP}})$ has* state-restoration soundness error *$\epsilon^{\mathrm{sr}}_{\mathrm{IP}}$ if for every salt sizes $s_i \in \mathbb{N}$, move budget $\mathsf{k} \in \mathbb{N}$, k-round malicious IP state-restoration prover $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$,*

$$\Pr\left[\begin{array}{l|c} \mathbb{x} \notin \mathcal{L}(\mathcal{R}) & \mathsf{rnd} = (\mathsf{rnd}_i)_{i\in[\mathsf{k}]} \leftarrow \mathcal{U}((\mathsf{r}_i)_{i\in[\mathsf{k}]}) \\ \wedge\, \mathcal{V}_{\mathrm{IP}}(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}) = 1 & (\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\mathsf{st}_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}) \leftarrow \\ & \mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}((s_i)_{i\in[\mathsf{k}]}, \mathsf{r}, \tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}) \end{array}\right] \leq \epsilon^{\mathrm{sr}}_{\mathrm{IP}}((s_i)_{i\in[\mathsf{k}]}, \mathsf{k})$$

$\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}((s_i)_{i\in[\mathsf{k}]}, \mathsf{rnd}, \tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}})$

$\mathsf{H} \leftarrow \mathcal{U}(2\lambda)$
while $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ doesn't exit:
  $\tilde{\mathcal{P}}^{\mathrm{sr}\,\mathsf{H}}_{\mathrm{IP}}$ outputs $(\mathbb{x}, (\alpha_1, \ldots, \alpha_i), (\mathsf{st}_1, \ldots, \mathsf{st}_i))$ where $\mathsf{st}_i \in \{0,1\}^{s_i}$
  Set $\rho_i \leftarrow \mathsf{rnd}_i(\mathbb{x}, (\alpha_1, \ldots, \alpha_i), (\mathsf{st}_1, \ldots, \mathsf{st}_i))$
  Give ρ_i to $\tilde{\mathcal{P}}^{\mathrm{sr}\,\mathsf{H}}_{\mathrm{IP}}$
$\tilde{\mathcal{P}}^{\mathrm{sr}\,\mathsf{H}}_{\mathrm{IP}}$ exits with output $(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\mathsf{st}_i)_{i\in[\mathsf{k}]})$
for $i \in [\mathsf{k}]$, set $\rho_i \leftarrow \mathsf{rnd}_i(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\mathsf{st}_i)_{i\in[\mathsf{k}]})$
return $(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\mathsf{st}_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]})$

Fig. 9.2: The state-restoration game for IPs.

$\mathcal{V}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_4}_{\mathrm{IP}}(\mathbb{x}, (\alpha_1, \ldots, \alpha_4), (\rho_1, \ldots, \rho_4))$

// Decode input
Write $\mathbb{x} = (\mathsf{pk}, \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$, $\alpha_1 = (\mathsf{com} \| \mathbf{c}_1 \| \ldots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}^{\mathsf{pre}})$, $\alpha_2 = (\tilde{\mathbf{u}} \| \mathbf{D} \| \mathbf{d})$, $\alpha_3 = (\tilde{a}_0 \| \tilde{a}_1 \| \tilde{a}_2)$, $\alpha_4 = (\mathsf{decom}_I)$
Set $\mathsf{chall}_1 := \rho_1, \mathsf{chall}_2 := \rho_2, \mathsf{chall}_3 := \rho_3$
Set $(\delta_0, \ldots, \delta_{\lambda-1}) := \mathsf{chall}_3$
Set $\mathsf{iv} \leftarrow \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}})$
// Reconstruct VOLEs and check commitment
$(\widetilde{\mathsf{com}}, \mathbf{Q}) \leftarrow \mathsf{FAEST.VOLEReconstruct}^{\mathsf{H}_0,\mathsf{H}_1}(\mathsf{decom}_I, \mathsf{chall}_3, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}; \mathtt{param})$
if $\widetilde{\mathsf{com}} = \bot$ or $\widetilde{\mathsf{com}} \neq \mathsf{com}$ or $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}}, \lambda) \neq 0$ **then return false**
// Apply the VOLE correction values and check consistency
$\widetilde{\mathbf{Q}} \leftarrow \mathsf{VOLEHash}(\mathsf{chall}_1, \mathbf{Q}) \in \{0,1\}^{(\lambda+B)\times\lambda}$
if $\mathbf{D} \neq \widetilde{\mathbf{Q}} \oplus [\delta_0 \cdot \tilde{\mathbf{u}} \cdots \delta_{\lambda-1} \cdot \tilde{\mathbf{u}}]$ **then return false**
// Compute AES consistency values
$\widehat{a}_0 \leftarrow \mathsf{ZK.OWFVerify}(\mathbf{d}, \mathbf{Q}|_{[0..\ell+\lambda)}, \mathsf{chall}_2, \mathsf{chall}_3, \tilde{a}_1, \tilde{a}_2, \mathsf{pk}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
return true if $\widehat{a}_0 = \tilde{a}_0$ **else false**

Fig. 9.3: The FEAST interactive proof verifier.

In Figure 9.3 we define the IP verifier $\mathcal{V}_{\mathrm{IP}}$ with $\mathsf{k} = 4$ for FAEST explicitly. Note that, when interacting with any honest prover, the protocol running with $\mathcal{V}_{\mathrm{IP}}$ has a very large completeness loss. This is because chall_3 must be zero in the last w_{grind} bits, which is only true with small probability in the honest SRS experiment from Definition 9.27. This is, however, of no importance for the proof, which only considers a reduction to the state-restoration soundness of the FAEST IP.

9.5.2 From EUF-KO to SR soundness We begin with a proposition that we need in the construction of the reduction:

Proposition 9.29 (Random oracle list game). *Let $n \in \mathbb{N}$ and for $i \in \{0, \ldots, n+1\}$ let $\mathcal{Y}_i, \mathcal{Y}'_i \subset \{0,1\}^*$ be finite sets. For each $i \in \{0, \ldots, n\}$ let $H_i \colon \mathcal{Y}_i \times \mathcal{Y}'_i \to \mathcal{Y}_{i+1}$ be a random oracle. Define $Y = \min_{i=1}^{n} |\mathcal{Y}_i|$.*

Consider the following game with an adversary $\mathcal{B}$: For $i \in [1..n]$ the game keeps track of initially empty lists $L_i \subset \mathcal{Y}_i$ and proceeds as follows: For $i \in \{0, \ldots, n\}$ the adversary can (repeatedly) query each H_i on an input x, x', producing an output y:

1. *When querying $y = H_i(x, x')$, if $y \in L_{i+1}$ then the adversary wins.*
2. *Thereafter if $i \geq 1$ then add x to L_i.*

Let $\mathcal{B}$ be an adversary which makes at most Q queries to the oracles $H_0, \ldots, H_n$. Then the probability that $\mathcal{B}$ wins is bounded by

$$Q^2/(2 \cdot Y).$$

Proof of Proposition 9.29. Clearly, $\mathcal{B}$ can only win by querying $H_0, \ldots, H_{n-1}$. At the first query, the adversary cannot win because each L_i is empty. At the second query, it can win with probability at most $\max_{i=1}^{n} \frac{1}{|\mathcal{Y}_i|} \leq 1/Y$ if the previous query was to $H_1, \ldots, H_n$. This is because the output of the oracle to which the query is now made is uniformly random in the respective $\mathcal{Y}_i$ and will match the single element on a list (if it exists) with probability $1/|\mathcal{Y}_i|$. At the third query, this probability is at most $\max_{i=1}^{n} \frac{2}{|\mathcal{Y}_i|} \leq 2/Y$ as the chance is maximal if both previous queries where to the same H_i. By a union bound, $\mathcal{B}$ will win during any of the queries with probability at most

$$\sum_{i=1}^{Q} \frac{i-1}{Y} \leq \frac{Q^2}{2 \cdot Y}.$$

□

We now prove that any attacker against the EUF-KO security of FAEST can be used to construct an attacker against an SRS IP with verifier $\mathcal{V}_{\mathrm{IP}}$.

Lemma 9.30. *Let $\mathcal{R}$ be the relation such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ if $\mathbb{x}$ is a public key and $\mathbb{w}$ the corresponding signing key for the FAEST signature scheme (with parameters $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}, \mathtt{param}_{\mathsf{VOLE}}$ which will be left implicit). Let $\mathbb{x} = \mathsf{pk} = (\mathbf{x}, \mathbf{y}) \in \{0,1\}^{4\lambda}$ be a purported public key, and let $\mathcal{A}$ be a (non-uniform) algorithm with access to random oracles $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_3, \mathsf{H}_4$ playing the EUF-KO game of Definition 9.22 with public key pk. Suppose $\mathcal{A}$ wins the game with probability at least p.[22] Suppose further that $\mathcal{A}$ makes at most $Q_{2,0}, Q_{2,1}, Q_{2,2}, Q_{2,3}$ queries to each of the random oracles $\mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$, respectively.*

Then there exists a 4-round probabilistic algorithm $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ for the relation $\mathcal{R}$ with access to the random oracles $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_3, \mathsf{H}_4$ that wins the state-restoration game of Definition 9.27 with probability at least $p - (Q_{2,0} + Q_{2,1} + Q_{2,1} + Q_{2,3} + 4)^2/2^{2\cdot\lambda}$ against $\mathcal{V}_{\mathrm{IP}}$. Moreover, $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ makes at most $(Q_{2,1} + Q_{2,1} + Q_{2,3} + 4)$ queries to the SRS game and its running time is roughly that of $\mathcal{A}$.

Proof. We define a malicious prover $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ against the state-restoration soundness based on $\mathcal{A}$. Note that $\mathcal{A}$ is non-interactive and interacts with the random oracles $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_3, \mathsf{H}_4$ during the EUF-KO game. $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ is defined for an interactive game, where the messages from the SRS experiment will replace the responses from $\mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$.

$\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ starts $\mathcal{A}$ with the pk as well as $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}, \mathtt{param}_{\mathsf{VOLE}}$ as inputs. $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ runs the EUF-KO game with $\mathcal{A}$ and simulates access to the random oracles. Whenever $\mathcal{A}$ makes a query to any of $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_3, \mathsf{H}_4$, $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ observes it and forwards it to a random oracles it has access to itself, passing the response back to $\mathcal{A}$. For queries to the random oracles $\mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$ $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ runs the following *RO simulation*:

- When $\mathcal{A}$ makes a query $\mu \| \mathsf{com} \| \mathbf{c}_1 \| \ldots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}^{\mathsf{pre}}$ to H_2^1,
  1. $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ checks if μ was previously returned to $\mathcal{A}$ as a response to a query to H_2^0; if not, $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ returns a randomly sampled chall_1 to $\mathcal{A}$ and puts μ on the bad list.
  2. Otherwise, $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ recovers the pk and msg that were queried to H_2^0 to create an instance $\mathbb{x}$, and outputs $(\mathbb{x}, (\alpha_1), (\mathsf{st}_1))$ to $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$, where $\alpha_1 = \mathsf{com} \| \mathbf{c}_1 \| \ldots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}^{\mathsf{pre}}$ and $\mathsf{st}_1 = \mu$ serves as salt. It receives ρ_1 from $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$ and passes $\mathsf{chall}_1 = \rho_1$ to $\mathcal{A}$ as the response from H_2^1.

[22] In our setting of interest, pk will be an *invalid* public key, i.e., there exists no secret key. Hence, despite non-uniform dependency of $\mathcal{A}$ on pk, there is no trivial winning strategy. Indeed, we show it to be hard.

– When $\mathcal{A}$ makes a query $\mathsf{chall}_1\|\tilde{\mathbf{u}}\|\mathbf{D}\|\mathbf{d}$ to H_2^2,
  1. $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ checks if chall_1 was previously returned to $\mathcal{A}$ as a response from H_2^1; if not, $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ returns a randomly sampled chall_2 to $\mathcal{A}$ and puts chall_1 on the bad list.
  2. Otherwise, $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ recovers the partial transcript $(\mathbb{x}, (\alpha_1), (\mathsf{st}_1))$ that received $\rho_1 = \mathsf{chall}_1$ as response, and it outputs $(\mathbb{x}, (\alpha_1, \alpha_2), (\mathsf{st}_1, \mathsf{st}_2))$ to $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$, where $\alpha_2 = \tilde{\mathbf{u}}\|\mathbf{D}\|\mathbf{d}$ and $\mathsf{st}_2 = \mathsf{chall}_1$ serves as salt. It receives ρ_2 from $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$ and returns $\mathsf{chall}_2 = \rho_2$ to $\mathcal{A}$ as the response from H_2^2.
– When $\mathcal{A}$ makes a query $\mathsf{chall}_2\|\tilde{a}_0\|\tilde{a}_1\|\tilde{a}_2\|\mathsf{ctr}$ to H_2^3,
  1. $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ checks if chall_2 was previously returned to $\mathcal{A}$ as a response from H_2^2; if not, $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ returns a randomly sampled chall_3 to $\mathcal{A}$ and puts chall_2 on the bad list.
  2. Otherwise, $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ recovers the partial transcript $(\mathbb{x}, (\alpha_1, \alpha_2), (\mathsf{st}_1, \mathsf{st}_2))$ that received $\rho_2 = \mathsf{chall}_2$ as response, and it outputs $(\mathbb{x}, (\alpha_1, \alpha_2, \alpha_3), (\mathsf{st}_1, \mathsf{st}_2, \mathsf{st}_3))$, where $\alpha_3 = \tilde{a}_0\|\tilde{a}_1\|\tilde{a}_2$ and $\mathsf{st}_3 = (\mathsf{chall}_2, \mathsf{ctr})$. It receives $\rho_3 \in \{0,1\}^\lambda$ from $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$ returns $\mathsf{chall}_3 = \rho_3$ to $\mathcal{A}$ as the response from H_2^3.

At the end of the EUF-KO game, $\mathcal{A}$ terminates with $\mathsf{msg}, \sigma = ((\mathbf{c}_i)_{i\in[1..\tau)}, \tilde{\mathbf{u}}, \mathbf{d}, \tilde{a}_1, \tilde{a}_2, \mathsf{decom}_I, \mathsf{chall}_3, \mathsf{iv}^{\mathsf{pre}}, \mathsf{ctr})$. $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$, upon receiving this, runs $\mathsf{FAEST.Verify}(\mathsf{msg}, \mathsf{pk}, \sigma; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$ to check if the signature σ is valid. As $\mathsf{FAEST.Verify}$ interacts again with the random oracles $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_3, \mathsf{H}_4$ the algorithm $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ will, as before, forward queries to $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_3, \mathsf{H}_4$ to the random oracles that it has access to. For queries to $\mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$ it does the following:

– When Verify makes the query $\mu\|\mathsf{com}\|\mathbf{c}_1\|\ldots\|\mathbf{c}_{\tau-1}\|\mathsf{iv}^{\mathsf{pre}}$ to H_2^1, $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ checks if μ was put on the bad list during *RO simulation*. If yes, then $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ aborts. Otherwise it outputs $(\mathbb{x}, (\alpha_1), (\mathsf{st}_1))$ to $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$, where $\alpha_1 = \mathsf{com}\|\mathbf{c}_1\|\ldots\|\mathbf{c}_{\tau-1}\|\mathsf{iv}^{\mathsf{pre}}$ and $\mathsf{st}_1 = \mu$ serves as salt. It receives ρ_1 from $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$ and passes $\mathsf{chall}_1 = \rho_1$ to Verify as the response from H_2^1.
– When Verify makes the query $\mathsf{chall}_1\|\tilde{\mathbf{u}}\|\mathbf{D}\|\mathbf{d}$ to H_2^2, $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ checks if chall_1 was put on the bad list during *RO simulation*. If yes, then $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ aborts. Otherwise it outputs $(\mathbb{x}, (\alpha_1, \alpha_2), (\mathsf{st}_1, \mathsf{st}_2))$ to $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$, where $\alpha_2 = \tilde{\mathbf{u}}\|\mathbf{D}\|\mathbf{d}$ and $\mathsf{st}_2 = \mathsf{chall}_1$ serves as salt. It receives ρ_2 from $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$ and returns $\mathsf{chall}_2 = \rho_2$ to Verify as the response from H_2^2.
– When Verify makes the query $\mathsf{chall}_2\|\tilde{a}_0\|\tilde{a}_1\|\tilde{a}_2\|\mathsf{ctr}$ to H_2^3, $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ checks if chall_2 was put on the bad list during *RO simulation*. If yes, then $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ aborts. Otherwise it outputs $(\mathbb{x}, (\alpha_1, \alpha_2, \alpha_3), (\mathsf{st}_1, \mathsf{st}_2, \mathsf{st}_3))$, where $\alpha_3 = \tilde{a}_0\|\tilde{a}_1\|\tilde{a}_2$ and $\mathsf{st}_3 = (\mathsf{chall}_2, \mathsf{ctr})$. It receives $\rho_3 \in \{0,1\}^\lambda$ from $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$ returns $\mathsf{chall}_3 = \rho_3$ to Verify as the response from H_2^3.

By construction, the outputs of the simulated ROs during Verify are consistent with those of the simulated ROs during the EUF-KO game with $\mathcal{A}$.

If $\mathsf{FAEST.Verify}$ rejects the signature, then $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ aborts. Else, define $\alpha_1 = (\mathsf{com}\|\mathbf{c}_1\|\ldots\|\mathbf{c}_{\tau-1}\|\mathsf{iv}^{\mathsf{pre}})$, $\alpha_2 = (\tilde{\mathbf{u}}\|\mathbf{D}\|\mathbf{d})$, $\alpha_3 = (\tilde{a}_0\|\tilde{a}_1\|\tilde{a}_2)$ and $\alpha_4 = (\mathsf{decom}_I)$. Then exit the while loop in $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$ and output $(\mathsf{pk}, (\alpha_1, \alpha_2, \alpha_3, \alpha_4), (\mu, \mathsf{chall}_1, \mathsf{chall}_2\|\mathsf{ctr}, \bot))$ to the experiment.

Note that the outputs of the ROs $\mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$ which $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ simulates towards $\mathcal{A}$ are identically distributed to those of the outputs of the random oracles in the EUF-KO game. This follows from Definition 9.27, which replaces the random oracles with outputs of randomness functions (that map each input to a uniformly random output, just like a random oracle) and because RO outputs are chosen uniformly at random in the bad list case as well. Thus, running $\mathcal{A}$ inside $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ does not decrease the probability of $\mathcal{A}$ outputting a valid forgery when compared with the EUF-KO security experiment. Also, note that $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ never aborts during the interaction with $\mathcal{A}$ that creates the forgery but potentially after obtaining the forgery attempt from $\mathcal{A}$.

Assume that $\tilde{\mathcal{P}}_{\mathrm{IP}}^{\mathrm{sr}}$ outputs a value to the experiment $\mathsf{Exp}_{\mathrm{IP}}^{\mathrm{sr}}$ without aborting. By construction, Verify must have accepted the signature generated by $\mathcal{A}$. $\mathcal{V}_{\mathrm{IP}}$ runs the same algorithms to verify the messages as $\mathsf{FAEST.Verify}$ and equally checks that the same w_{grind}

bits of chall_3 are 0. It also uses the exact same random challenges as Verify due to the simulation of the RO responses by $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ during Verify.

The differences in verification between both algorithms are the following:

$\mathcal{V}_{\mathrm{IP}}$ compares the output $\widehat{a}_0$ of OWFVerify with the $\tilde{a}_0$ contained in the input message α_3. It also compares the output $\widetilde{\mathsf{com}}$ of VOLEReconstruct with the value com contained in the input α_1; while

Verify hashes the output $\tilde{a}_0$ of OWFVerify (and other messages) using H^2_3 to generate chall'_3, which it compares with chall_3. It also hashes the output com of VOLEReconstruct using H^1_2 to generate chall_1.

All the inputs $\mathbf{d}, \mathbf{Q}, \tilde{a}_1, \tilde{a}_2, \mathsf{pk}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$ to OWFVerify are identical in Verify, $\mathcal{V}_{\mathrm{IP}}$ as both use the same challenges. Since Verify accepts, we have that the value $\tilde{a}_0$ which $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ generates from OWFVerify (using the challenges $\mathsf{chall}_2, \mathsf{chall}_3$) and outputs to the experiment $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$ is identical to the RO input to H^2_3 that Verify makes to generate $\mathsf{chall}'_3 = \mathsf{chall}_3$. As $\mathcal{V}_{\mathrm{IP}}$ uses the same challenges as Verify, the output $\widehat{a}_0$ of OWFVerify in $\mathcal{V}_{\mathrm{IP}}$ must therefore be identical to the $\tilde{a}_0$ it obtained as input. A similar argument can be made for $\widetilde{\mathsf{com}}$: there, all inputs except chall_3 to VOLEReconstruct are fixed. As the signature verifies, Verify used the challenge chall_3 for verification, which made it output com that is placed in α_1. But since $\mathcal{V}_{\mathrm{IP}}$ uses the same inputs to VOLEReconstruct, its output $\widetilde{\mathsf{com}}$ will be identical to com. We conclude that, if $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ does not abort the experiment, then it will win the experiment with $\mathcal{V}_{\mathrm{IP}}$ due to consistency of RO simulation with the outputs of the random functions of the SRS experiment.

$\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ aborts the experiment if it previously marked an input to a RO as bad during the simulation of ROs towards $\mathcal{A}$, or if Verify aborts. Aborting in the latter case does not decrease the success probability of $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ so we only analyze the former. We invoke Proposition 9.29: if an input query $x = \mu \| \mathsf{com} \| \mathbf{c}_1 \| \ldots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}^{\mathsf{pre}}$ to H^1_2 matches a μ that was added to the bad list, then μ was not output by H^0_2 before (the same argument follows identically for $\mathsf{H}^2_2, \mathsf{H}^3_2$). If $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ aborts due to the bad list event for H^1_2 during Verify, then $\mathcal{A}$ must have output a forgery tuple msg, σ (for a predefined pk) such that $\mathsf{H}^0_2(\mathsf{pk} \| \mathsf{msg}) = \mu$ and where $x = \mu \| \ldots$ was queried to H^1_2 before $\mathsf{pk} \| \mathsf{msg}$ was queried to H^0_2 (if it was not queried by $\mathcal{A}$ during the EUF-KO game it will be queried by Verify). By modeling $\mathsf{H}^0_2, \ldots, \mathsf{H}^3_2$ as the random oracles in Proposition 9.29, the abort probability due to this event can be at most $(Q_{2,0} + Q_{2,1} + Q_{2,2} + Q_{2,3} + 4)^2 / 2^{2\lambda}$. Here we count the RO queries made by $\mathcal{A}$ and the additional 4 that are possibly made during Verify. Since Y is set to be the size of the minimum of the output spaces of $\mathsf{H}^0_2, \mathsf{H}^1_2, \mathsf{H}^2_2$ we have that $Y = 2^{2\lambda}$ as $\mu \in \{0,1\}^{2\cdot\lambda}$ is the smallest response of these ROs.

□

Note that the constructed $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ runs $\mathcal{A}$ once and forwards requests to random oracles. In particular, if $\mathcal{A}$ is a PPT algorithm, then so is $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$. $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ also ignores potential preimage and collision attacks on H_4 that allow $\mathcal{A}$ to find signatures that are identical except for differing $\mathsf{iv}^{\mathsf{pre}}$. This is out of scope as we do not claim to achieve strong unforgeability.

9.5.3 From SRS IP to SRS IOP We now define a simplified notion of an IOP (following [BBD+23a]), and will present a version of $\mathcal{V}_{\mathrm{IP}}$ that works as an IOP by idealizing the BAVC. We then show a reduction from SRS IP to SRS IOP.

Definition 9.31 (Interactive Oracle Proof (IOP)). *A (public-coin)* interactive oracle proof *(IOP) [BCS16, RRR16] for NP-relation $\mathcal{R}$ is a pair of PPT ITMs $\Pi = (\mathcal{P}, \mathcal{V})$, which are defined as follows: The* prover $\mathcal{P}$ *throughout the protocol sends strings $\overline{m}_i = (m_i, g_i)$ in each round, where $m_i \in \{0,1\}^*$ is called* message *and $g_i \in \{0,1\}^*$ is called* oracle. *The* verifier $\mathcal{V}$ *learns m_i in round i, but not g_i. It sends random challenges*

$\rho_i \in \mathcal{C}_i$ *in round* i *in response. In a final round, the receiver learns all* g_i *and outputs* $b = \mathcal{V}_{\mathrm{IOP}}(\mathbb{x}, ((m_1, \ldots, m_{\mathsf{k}}), (g_1, \ldots, g_{\mathsf{k}}), (\rho_1, \ldots, \rho_{\mathsf{k}-1})))$.

Definition 9.31 can easily be reconciled with state-restoration soundness. Namely, for any IOP protocol we simply let the message α_i be defined as $\alpha_i := (m_i, g_i)$. Then the resulting construction is still an IOP, as the state-restoration verifier, as defined in the state-restoration soundness error Definition 9.28, is only ever called after all messages $\alpha_1, \ldots, \alpha_{\mathsf{k}}$ (and the salts $\mathsf{st}_1, \ldots, \mathsf{st}_{\mathsf{k}}$) have been fixed in the experiment. This fulfills the requirements from Definition 9.31.

We now rewrite $\mathcal{V}_{\mathrm{IP}}$ as an IOP which we call $\mathcal{V}_{\mathrm{IOP}}$. The algorithm is defined below:

$\mathcal{V}_{\mathrm{IOP}}(\mathbb{x}, (\alpha_1, \ldots, \alpha_4), (\rho_1, \ldots, \rho_4))$

1 : // Decode input
2 : Write $\mathbb{x} = (\mathsf{pk}, \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$, $\alpha_1 = ((\mathbf{c}_1 \| \ldots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}^{\mathsf{pre}}) \| g_1)$, $\alpha_2 = (\tilde{\mathbf{u}} \| \mathbf{D} \| \mathbf{d})$, $\alpha_3 = (\tilde{a}_0 \| \tilde{a}_1 \| \tilde{a}_2)$
3 : Set $\mathsf{chall}_1 := \rho_1, \mathsf{chall}_2 := \rho_2, \mathsf{chall}_3 := \rho_3$
4 : Set $(\delta_0, \ldots, \delta_{\lambda-1}) := \mathsf{chall}_3$
5 : // Reconstruct VOLEs from IOP oracle
6 : $\mathbf{Q} \leftarrow \mathsf{VOLEIOPReconstruct}(g_1, \mathsf{chall}_3, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$
7 : **if** $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}}, \lambda) \neq 0$ **then return false**
8 : // Apply the VOLE correction values and check consistency
9 : $\widetilde{\mathbf{Q}} \leftarrow \mathsf{VOLEHash}(\mathsf{chall}_1, \mathbf{Q}) \in \{0,1\}^{(\lambda+B)\times\lambda}$
10 : **if** $\mathbf{D} \neq \widetilde{\mathbf{Q}} \oplus [\delta_0 \cdot \tilde{\mathbf{u}} \cdots \delta_{\lambda-1} \cdot \tilde{\mathbf{u}}]$ **return false**
11 : // Compute AES consistency values
12 : $\widehat{a}_0 \leftarrow \mathsf{ZK.OWFVerify}(\mathbf{d}, \mathbf{Q}|_{[0..\ell+\lambda)}, \mathsf{chall}_2, \mathsf{chall}_3, \tilde{a}_1, \tilde{a}_2, \mathsf{pk}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
13 : **return true if** $\widehat{a}_0 = \tilde{a}_0$ **else false**

The main difference to $\mathcal{V}_{\mathrm{IP}}$ is that the commitment com contained in α_1 is now replaced by the IOP oracle string g_1 that contains all the $\mathbf{u}_i, \mathbf{V}_i$ pairs corresponding to the committed seeds as supposedly generated by VOLECommit. Therefore, no opening decom_I for the commitments must be sent in α_4. Moreover, this also means that we use a new algorithm that reconstructs the VOLEs from the oracle string g_1 instead of the commitment and openings. We call this algorithm VOLEIOPReconstruct and it is defined in Figure 9.4.

VOLEIOPReconstruct differs from VOLEReconstruct as follows: Based on the BAVC commitment, the original algorithm uses BAVC.Reconstruct to obtain the $\mathsf{sd}_{i,j}$, except for those at the unopened positions defined in $\Delta_0, \ldots, \Delta_{\tau-1}$, and uses these, together with ConvertToVOLE, to compute the matrix $\mathbf{Q}$. This is an algorithm that can fail. In comparison, VOLEIOPReconstruct simply takes $\mathbf{u}_i, \mathbf{V}_i$ as input and computes $\mathbf{Q}$ based on the relation that ConvertToVOLE generates according to Proposition 5.2. Therefore, VOLEIOPReconstruct will succeed in computing $\mathbf{Q}$ as long as it obtains a valid oracle string g_1 as input.

Lemma 9.32. *Let* $\mathcal{R}$ *be the relation such that* $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ *if* $\mathbb{x}$ *is a public key and* $\mathbb{w}$ *the corresponding signing key for the FAEST signature scheme (with parameters* $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}, \mathtt{param}_{\mathsf{VOLE}}$ *which will be left implicit). Let* $\mathcal{A}$ *be a probabilistic polynomial-time algorithm with access to random oracles* $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_4$ *which, on input an invalid public key* $\mathbb{x} = \mathsf{pk}$*, wins the SRS soundness game defined in Definition 9.28 against* $\mathcal{V}_{\mathrm{IP}}$ *with probability at least* p*.* $\mathcal{A}$ *makes at most* Q_0 *queries to the random oracle* H_0 *and* Q_1 *to the random oracle* H_1*.*

Then there exists a 4-round probabilistic algorithm $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ *that wins the state-restoration game in Definition 9.27 with probability at least*

$$p - \left(\frac{(Q_1+1)^2}{2^{2\lambda}} + \tau Q_0 \cdot 2^{-\lambda}\right)$$

$\mathsf{VOLEIOPReconstruct}(g_1, \mathsf{chall}_3, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$

INPUT: $g_1 \in \{0,1\}^{\hat{\ell} \times \tau} \times \{0,1\}^{\hat{\ell} \times k_0} \times \cdots \times \{0,1\}^{\hat{\ell} \times k_{\tau-1}}, \mathsf{chall}_3 \in \{0,1\}^{\lambda}$

OUTPUT: $\mathbf{Q} \in \{0,1\}^{\hat{\ell} \times \lambda}$

$I = (\Delta_0, \ldots, \Delta_{\tau-1}) \leftarrow \mathsf{DecodeAllChall}_3(\mathsf{chall}_3)$
if $I = \bot$ **then return** $\bot$
else do
  Write $(\mathbf{u}_0, \ldots, \mathbf{u}_{\tau-1}, \mathbf{V}_0, \ldots, \mathbf{V}_{\tau-1}) := g_1$
  for $i \in [0 \ldots \tau)$ **do**
    $(\delta_{i,0}, \ldots, \delta_{i,k_i}) \leftarrow \mathsf{BitDec}(\Delta_i, k_i)$
    Consider $\mathbf{V}_i$ column-wise as $(\mathbf{v}_{i,0}, \ldots, \mathbf{v}_{i,k_i-1}) := \mathbf{V}_i$
    for $j \in [0 \ldots k_i)$ **do**
      $\mathbf{q}_{i,j} := \mathbf{v}_{i,j} \oplus \delta_{i,j} \cdot \mathbf{u}_i$
    if $i = 0$ **then**
      $\mathbf{Q}_i := [\mathbf{q}_{i,0} \cdots \mathbf{q}_{i,k_i-1}] \in \mathbb{F}_2^{\hat{\ell} \times k_i}$ // stored in column major representation
    else
      $\mathbf{Q}_i \leftarrow [\mathbf{q}_{i,0} \cdots \mathbf{q}_{i,k_i-1}] + [\delta_{i,0} \cdot \mathbf{c}_i \cdots \delta_{i,k_i-1} \cdot \mathbf{c}_i] \in \mathbb{F}_2^{\hat{\ell} \times k_i}$
  $\mathbf{Q} := [\mathbf{Q}_0 \cdots \mathbf{Q}_{\tau-1}\ \mathbf{0}_{\hat{\ell},w}] \in \mathbb{F}_2^{\hat{\ell} \times \lambda}$ // $\mathbf{0}$ is $(\hat{\ell} \times w)$ bits of zero-padding
  return $\mathbf{Q}$

Fig. 9.4: VOLE reconstruction for the FAEST IOP.

against $\mathcal{V}_{\mathrm{IOP}}$. *For* FAEST-EM *, the success probability is*

$$\frac{(Q_1+1)^2}{2^{2\lambda}} + \mathsf{AdvInj}_{\mathcal{B}}^{\mathsf{PRG}_{2\lambda}}[Q_4, L]$$

using the extractor from 9.15 that reprograms H_4*. If* $\mathcal{A}$ *makes* β *queries to the SRS game, then* $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ *makes* $\beta + 1$ *queries.*

Note that we do not require $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ to be polynomial-time and in fact it won't be. This is however not a problem, since $\mathbb{x}$ is not a valid instance anyways so the reduction cannot trivially win the security game by finding a witness.

Proof of Lemma 9.32. We now construct the algorithm $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ and prove its success probability. The intuitive idea is that $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ will forward most of the messages from $\mathcal{A}$ directly to the experiment, with the difference that it will extract all inputs from the BAVC com and convert them into an oracle string.

Let $\mathcal{A}$ be the algorithm described in the statement, then $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ works as follows: It runs an instance of $\mathcal{A}$ towards which it simulates the SRS IP experiment as defined in Figure 9.2. $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ itself runs within an analogous SRS IOP experiment.

- For the hash function H_0, $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ simulates a random oracle using lazy sampling and records all query-response pairs on the list $\mathcal{L}_0 := (x_i, \mathsf{H}(x_i))$. Similarly, responses for queries to H_1 will be simulated with lazy sampling and stored on $\mathcal{L}_1$ by $\tilde{\mathcal{P}}_{\mathrm{IOP}}$. $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ also simulates H_4 using lazy sampling.
- Whenever $\mathcal{A}$ outputs $(\mathbb{x}, (\alpha_1), (\mathsf{st}_1'))$, with $\alpha_1 := (\mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}^{\mathsf{pre}})$, $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ runs the extractor algorithm $\mathsf{Ext}(\mathcal{L}_0, \mathcal{L}_1, \mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}^{\mathsf{pre}})$ from Lemma 9.15 to obtain $\mathbf{D} = (\boldsymbol{u}_i^*, \mathbf{V}_i^*)_{i \in [0,\tau)}$. $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ sets $g_1 := \mathbf{D}$ regardless of whether extraction was successful. It then sets $m_1 := ((\mathbf{c}_1 \| \ldots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}^{\mathsf{pre}}) \| g_1)$, $\mathsf{st}_1 := \mathsf{st}_1' \| \mathsf{com}$ and outputs $(\mathbb{x}, (m_1), (\mathsf{st}_1))$ to the IOP SRS experiment and returns to $\mathcal{A}$ what the experiment returns.

– Whenever $\mathcal{A}$ outputs $(\mathbb{x}, (\alpha_1, \alpha_2), (\mathsf{st}'_1, \mathsf{st}_2))$ we compute m_1, st_1 as above and output $(\mathbb{x}, (m_1, (\alpha_2 \| \emptyset)), (\mathsf{st}_1, \mathsf{st}_2))$ to the experiment and return to $\mathcal{A}$ what the experiment returns.
– Similarly, whenever $\mathcal{A}$ outputs $(\mathbb{x}, (\alpha_1, \alpha_2, \alpha_3), (\mathsf{st}'_1, \mathsf{st}_2, \mathsf{st}_3))$ we compute m_1, st_1 as above and output $(\mathbb{x}, (m_1, (\alpha_2 \| \emptyset), (\alpha_3 \| \emptyset)), (\mathsf{st}_1, \mathsf{st}_2, \mathsf{st}_3))$ to the experiment and return to $\mathcal{A}$ what the experiment returns.
– Similarly, whenever $\mathcal{A}$ outputs $(\mathbb{x}, (\alpha_1, \alpha_2, \alpha_3, \alpha_4), (\mathsf{st}'_1, \mathsf{st}_2, \mathsf{st}_3, \mathsf{st}_4))$ we compute m_1, st_1 as above and output $(\mathbb{x}, (m_1, (\alpha_2 \| \emptyset), (\alpha_3 \| \emptyset), (\alpha_4 \| \emptyset)), (\mathsf{st}_1, \mathsf{st}_2, \mathsf{st}_3, \mathsf{st}_4))$ to the experiment and return $\emptyset$ to $\mathcal{A}$.
– Whenever $\mathcal{A}$ exits the loop and outputs $(\mathbb{x}, (\alpha_1, \alpha_2, \alpha_3, \alpha_4), (\mathsf{st}'_1, \mathsf{st}_2, \mathsf{st}_3, \mathsf{st}_4))$, $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ first computes g_1, m_1, st_1 as above. Then we first output $(\mathbb{x}, (m_1, (\alpha_2 \| \emptyset), (\alpha_3 \| \emptyset)), (\mathsf{st}_1, \mathsf{st}_2, \mathsf{st}_3))$ to the IOP SRS experiment to obtain ρ_3.
Then, let $\rho_3 := \mathsf{chall}_3$, $\alpha_4 := \mathsf{decom}_I$ and $\alpha_1 := (\mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}^{\mathsf{pre}})$. Additionally, compute

$$(\widehat{\mathsf{com}}, \widehat{\mathbf{Q}}) := \mathsf{FAEST.VOLEReconstruct}(\mathsf{decom}_I, \mathsf{chall}_3, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}^{\mathsf{pre}}; \mathtt{param}).$$

If $\widehat{\mathsf{com}} = \bot$ or $\widehat{\mathsf{com}} \neq \mathsf{com}$ or $g_1 = \bot$ while $\widehat{\mathsf{com}} = \mathsf{com}$ then abort. In case no abort happens, exit the loop with the IOP SRS game and output $(\mathbb{x}, (m_1, (\alpha_2 \| \emptyset), (\alpha_3 \| \emptyset), (\alpha_4 \| \emptyset)), (\mathsf{st}_1, \mathsf{st}_2, \mathsf{st}_3, \mathsf{st}_4))$ to the experiment.
The distribution of ρ_i and random oracle responses that $\mathcal{A}$ obtains from $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ is perfectly indistinguishable from those sent to $\mathcal{A}$ in the original SRS IP experiment. This is achieved by making com part of st_1 used to generate the random function response ρ_1, which is necessary as an attacker might use different com with identical $\mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathsf{iv}^{\mathsf{pre}}$ as part of some attack strategy. Thus, the behaviour of $\mathcal{A}$ towards $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ is perfectly indistinguishable from the behaviour it shows towards the regular experiment, in particular with regards to generating accepting transcripts.
The constructed reduction aborts whenever $\mathcal{V}_{\mathrm{IP}}$ would trivially abort, namely if the reconstructed $\widehat{\mathsf{com}}$ is incorrect and the commitment therefore did not open correctly. The reduction also aborts when the extractor is not successful and returns $\bot$ but VOLEReconstruct outputs $\mathbf{Q}$. Otherwise, the behaviour of $\mathcal{V}_{\mathrm{IP}}$ and $\mathcal{V}_{\mathrm{IOP}}$ only differs when $\mathcal{A}$ outputs a commitment com that it can successfully open with decom_I but where Ext extracted the wrong $(\boldsymbol{u}_i^*, \mathbf{V}_i^*)_{i \in [0,\tau)}$. However, observe that $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ plays exactly the security game of Lemma 9.15 with $\mathcal{A}$. There, the winning condition of the attacker is either that the extracted $(\boldsymbol{u}_i^*, \mathbf{V}_i^*)_{i \in [0,\tau)}$ are inconsistent with the output of VOLEReconstruct or that the extractor failed to generate $(\boldsymbol{u}_i^*, \mathbf{V}_i^*)_{i \in [0,\tau)}$ but VOLEReconstruct returns $\mathbf{Q}$. By Lemma 9.15 the probability of the attacker winning the game is at most $\frac{(Q_1+1)^2}{2^{2\lambda}} + \tau Q_0 \cdot 2^{-\lambda}$, as claimed above. Note that $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ makes only one extra query to the SRS game it interacts with in order to recover ρ_3.

□

9.5.4 FAEST IOP is RBR-sound. Looking at the Chiesa–Yogev book [CY24];[23] round-by-round (knowledge) implies state-restoration (knowledge) soundness with a $t + k$ multiplicative loss, where k is the round complexity of the proof system, and t is the "move budget" of the malicious state-restoration prover (we should think of $t = O(2^\lambda)$ because it's the number of times the malicious prover can restore the state of the verifier, i.e. query the hash function). See [CY24], Theorems 31.2.1 and 31.3.1.

To handle the QROM as well, we define an augmented notion of round-by-round soundness.

[23] https://github.com/hash-based-snargs-book/hash-based-snargs-book/blob/main/snargs-book.pdf

Definition 9.33 (Round-by-round soundness, based on [CCH+18]). *For a IP Π for a language $\mathcal{L}$, let* $\mathsf{RBRState}$ *be a state function mapping an instance and a transcript prefix to a Boolean value, with the following properties. For* $\mathsf{inst} \notin \mathcal{L}$,

1. $\mathsf{RBRState}(\mathsf{inst}, \emptyset) = 0$, *where* $\emptyset$ *denotes the empty transcript prefix.*
2. *If for some instance* inst *and a transcript prefix* trc *that is empty (i=0) or ends in the i-th challenge with* $\mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}) = 0$, *then for any prover message* pmsg_{i+1} *it holds that*
$$\Pr_{\mathsf{chall}_{i+1}}[\mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}|\mathsf{pmsg}_{i+1}|\mathsf{chall}_{i+1}) = 1] \leq \epsilon,$$
where chall_{i+1} *is sampled uniformly from the appropriate challenge space, and the* round-by-round soundness error ε *depends on the security parameter only.*
3. *For any complete transcript* trc, *if* $\mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}) = 0$ *then the verifier rejects.*

We say Π has round-by-round (RBR) soundness *if for all no-instances $x \notin \mathcal{L}$ and all transcripts* trc, ϵ *is upper bounded by a negligible function of the security parameter.*

We say that a challenge chall_{i+1} *is* isolated *if given* $\mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}) = 0$, *the event* $\mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}|\mathsf{pmsg}_{i+1}|\mathsf{chall}_{i+1}) = 1$ *depends on* chall_{i+1} *only.*

We say Π is RBR-sound with passive prefix *up to* chall_i *if for a partial transcript* trc *ending in* chall_{j-1}, *given* $\mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}) = 0$, *the event* $\mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}|\mathsf{pmsg}_{i+1}|\mathsf{chall}_{i+1}) = 1$ *is independent of the prefix of* trc *ending in* chall_m *for* $m = \min(i, j-1)$.

Lemma 9.34. *Let $\mathcal{R}$ be the relation such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ if $\mathbb{x}$ is a public key and $\mathbb{w}$ the corresponding signing key for the FAEST signature scheme (with parameters* $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}, \mathtt{param}_{\mathsf{VOLE}}$ *which will be left implicit).*

Let $\hat{\ell} \leq 2^{13}, \ell' \leq 2^{13}$, $\varepsilon_{\mathsf{v}} = 2^{-\lambda-B}(1 + 2^{B-50})$, $\varepsilon_{\mathsf{zk}} = 2^{-\lambda}(1 + 2^{-38})$ and $2\log_2(\tau) \leq \min\{B, 50\}$. The FAEST IOP with verifier $\mathcal{V}_{\mathrm{IOP}}$ has round-by-round soundness with soundness error $3/2^{\lambda}$, where $\mathbb{x}$ is not a valid instance of $\mathcal{R}$

Proof of Lemma 9.34. For the first challenge we can directly bound the probability by Lemma 4.10, for the second by Lemma 4.11 and for the last by Lemma 6.3.

The FAEST IOP is a 3-round IOP; in the first round, $\mathcal{P}_{\mathrm{IOP}}$ sends

$$m_1 = (\mathbf{c}_1 \| \dots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}^{\mathsf{pre}}) \| g_1 = (\mathbf{u}_i, \mathbf{V}_i)_i.$$

From this message, $\mathsf{RBRState}$ can reconstruct $\mathbf{u}, \mathbf{V}$ and apply $\mathbf{c}_1, \dots, \mathbf{c}_{\tau-1}$ to correct all "small" VOLE instances $\mathbf{u}_i, \mathbf{V}_i$ to the same secret as $\mathbf{u}_0$. If the secret of any small VOLE instances differs from $\mathbf{u}_0$ then the verifier should later reject.

After this, $\mathcal{V}_{\mathrm{IOP}}$ responds with $\mathsf{chall}_1 = \rho_1$. We now analyze how likely chall_1 will break soundness in this round by flipping $\mathsf{RBRState}$ from 0 to 1. This means that the hashes indicate that the secret committed in each VOLE instance is identical, when in fact they are different. We import following claim from [BBD+23b, Theorem 2].

Claim 1. For all $(\mathbf{u}_i^*, \mathbf{V}_i^*)_{i \in [0..\tau)}$, $\mathbf{C} = [\mathbf{c}_1 \cdots \mathbf{c}_{\tau-1}] \in \mathbb{F}_2^{\hat{\ell} \times (\tau-1)}$, the probability that the consistency check fails for independently sampled VOLEHash challenges chall_1 is at most $\varepsilon_{\mathsf{v}}\binom{\tau}{2}$.

Proof. See proof of [BBD+23b, Theorem 2]. □

Upon receiving chall_1, the prover responds with $m_2 = (\widetilde{\mathbf{u}} \| \mathbf{D} \| \mathbf{d})$. Using $\mathbf{d}$, we can reconstruct the witness $\mathbf{w}$ and run ZK.OWFProve inside $\mathsf{RBRState}$ (i.e. apply the computation to the committed witness). The commitments that are opened are those collected in $\langle \mathbf{z} \rangle_{\lambda}^{3}$ where z is supposedly 0. If any value committed in the commitments is non-zero then the verifier should later abort.

After the second round, $\mathcal{V}_{\mathrm{IOP}}$ responds with $\mathsf{chall}_2 = \rho_2$. We now analyze how likely non-zero commitments are hashed to a commitment of zero, thus breaking soundness and flipping $\mathsf{RBRState}$ from 0 to 1.

Claim 2 (ZK Hash check failure). Let $\mathbf{h} = \mathsf{ZKHash}(\mathsf{chall}_2, (\mathbf{x}_0, x_1)) = \mathbf{r}^\top \mathbf{x}_0 + x_1$. Let $\mathbf{e} \in \mathbb{F}_{2^\lambda}$ be the degree 3 coefficients of $\langle \mathbf{z} \rangle_\lambda^3$.

Define the bad event ("ZK Hash check fails") as $\mathbf{e} \neq 0$ but $\mathbf{r} \cdot \mathbf{e} = 0$, i.e. the event where ZK Hash corrects an error into a valid VOLE correlation. The probability that the ZK Hash check fails for an independent chosen hash chall_2 is at most $\varepsilon_{\mathsf{zk}}$.

Proof. This follows from Lemma 4.11 showing the $\varepsilon_{\mathsf{zk}}$-universality of ZKHash. □

In the third round, $\mathcal{P}_{\mathrm{IOP}}$ sends

$$\alpha_3 = (\tilde{a}_0 \| \tilde{a}_1 \| \tilde{a}_2).$$

These define a degree 2 polynomial that is supposedly an opening of the ZKHash of $\langle \mathbf{z} \rangle_\lambda^3$ as computed in OWFProve. Assume that the output of ZKHash is a degree-3 polynomial. To this, $\mathcal{V}_{\mathrm{IOP}}$ responds with $\mathsf{chall}_3 = \rho_3$ (which is the random sampling of the QuickSilver challenge Δ). By Lemma 6.3 the soundness error of this step is exactly $3/2^\lambda$. Again, RBRState flips from 0 to 1 for the 3 challenges which breaks soundness.

Since $\varepsilon_{\mathsf{zk}} = 2^{-\lambda}(1 + 2^{-38}) < 2^{-\lambda+1}$, we have that $3/2^\lambda$ is always larger than the soundness error of the last round is always larger than the soundness error of the second round. Moreover, we have that $\binom{\tau}{2} < \tau^2$, so as long as $2\log_2(\tau) \leq \min\{B, 50\}$ we also have that

$$\binom{\tau}{2} 2^{-\lambda-B}(1 + 2^{B-50}) < 2^{2\log_2(\tau)} 2^{-\lambda-B}(1 + 2^{B-50}) < 2 \cdot 2^{-\lambda}$$

and the soundness error of the first round is always smaller than the soundness error of the last round.

□

Finally, let us gather the chain from SRS security to RBR security in one corollary.

Corollary 9.35. *Consider the setting of Lemma 9.30 with invalid public key, i.e.,* $\mathsf{pk} \notin \mathsf{im}(\mathsf{KeyGen})$. *In particular,* $\mathcal{A}$ *makes at most* $Q_0, Q_1, Q_{2,0}, Q_{2,1}, Q_{2,2}, Q_{2,3}, Q_4$ *queries to each of the random oracles* $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_4$ *respectively. Suppose that* $\hat{\ell} \leq 2^{13}, \ell' \leq 2^{13}$ *and* $2\log_2(\tau) \leq \min\{B, 50\}$. *Then we have*

$$\begin{aligned} p \leq & (Q_{2,0} + Q_{2,1} + Q_{2,1} + Q_{2,3} + 4)^2 / 2^{2\cdot\lambda} \\ & + \frac{(Q_1 + 1)^2}{2^{2\lambda}} + \mathsf{AdvLeafColl}^{\mathsf{LeafCommit}} \\ & + (Q_{2,1} + Q_{2,1} + Q_{2,3} + 9) \cdot 3/2^\lambda \end{aligned}$$

where OWF *is the OWF (or more precisely, PRG) defined by* $\mathtt{param}_{\mathsf{OWF}}$, *and*

- $\mathsf{AdvLeafColl}^{\mathsf{LeafCommit}}_{\mathcal{B}'} \leq Q_0 \cdot 2^{2\lambda} \cdot \varepsilon_{\mathsf{uhash}} \leq Q_0 \cdot 2^{-\lambda}$ *for* FAEST.
- $\mathsf{AdvLeafColl}^{\mathsf{LeafCommit}}_{\mathcal{B}'} \leq \mathsf{AdvInj}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}}[Q_4, L]$ *for* FAEST-EM.

Proof. First, we apply Lemma 9.30 to turn $\mathcal{A}$ into an SRS adversary $\mathcal{A}_1$ with $(Q_{2,1}+Q_{2,1}+Q_{2,3}+4)$ as an interactive proof (IP) system; success is reduced by at most $(Q_{2,0}+Q_{2,1}+Q_{2,1}+Q_{2,3}+4)^2/2^{2\cdot\lambda}$. Secondly, we apply Lemma 9.32 to turn the SRS IP adversary into an SRS IOP adversary $\mathcal{A}_2$ which requires one additional query to the SRS IOP game; success is reduced by at most $\frac{(Q_1+1)^2}{2^{2\lambda}} + \mathsf{AdvLeafColl}^{\mathsf{LeafCommit}}$. Finally, we use [CY24, Theorem 31.2.1] to reduce the success of adversary $\mathcal{A}_2$ against SRS which makes at most $(Q_{2,1} + Q_{2,1} + Q_{2,3} + 5)$ queries, to RBR soundness. From Lemma 9.34 and the assumed bound on $\hat{\ell}, \ell', \tau, B$, we know that $\varepsilon^{\mathsf{rbr}}_{\mathsf{IOP}} \leq 3/2^\lambda$. Moreover, the IOP has 4 moves, and therefore we can bound success of $\mathcal{A}_2$ by

$$(Q_{2,1} + Q_{2,1} + Q_{2,3} + 9) \cdot 3/2^\lambda$$

This yields the claim. □

9.6 EUF-CMA

Theorem 9.36 (FAEST is EUF-CMA secure). *Let* $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$ *and* H_4 *be modeled as random oracles and* H_3 *be a PRF. Then for any PPT adversary* $\mathcal{A}$ *which makes* Q_{sig} *queries to* $\mathcal{O}_{\mathsf{sig}}$ *and* $Q_0, Q_1, Q_{2,1}, Q_{2,2}$ *and* $Q_{2,3}$ *queries to* $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^1, \mathsf{H}_2^2$ *and* H_2^3 *respectively,*

$$\begin{aligned}\mathsf{AdvEUFCMA}_{\mathcal{A}}^{\mathsf{FAEST}}[Q_{\mathsf{sig}}] \leq\ & 2 \cdot \mathsf{AdvEUFKO}^{\mathsf{FAEST}} \\ & + 2 \cdot Q_{\mathsf{sig}} \cdot \left(\mathsf{AdvPRF}^{\mathsf{H}_3} + \lceil \log(L) \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1]\right) \\ & + 2 \cdot Q_{\mathsf{sig}} \cdot \left(\frac{2Q_1 + Q_{2,1}}{2^{2\lambda}} + \frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{5\lambda+64}} + \frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}}\right)\end{aligned}$$

For FAEST-EM*, the reduction and bound holds.*

Proof. Let $\mathcal{A}$ be an arbitrary adversary against the EUF-CMA security of FAEST. We define a sequence of games which begins with $\mathcal{A}$ playing the real EUF-CMA game, where the signing oracle $\mathcal{O}_{\mathsf{sig}}$ uses the real secret key sk to compute signatures, and ends with $\mathcal{O}_{\mathsf{sig}}$ simulating signatures without using sk.

We then define the reduction $\mathcal{B}$ playing in the EUF-KO game to play the role of $\mathcal{O}_{\mathsf{sig}}$ for $\mathcal{A}$ in the final game of the sequence and to output the forgery that $\mathcal{A}$ outputs. From this definition, it follows that $\mathcal{B}$ is successful in the EUF-KO game whenever $\mathcal{A}$ is in the final game.

The sequence of games is defined as follows:

G_1**:** $\mathcal{A}$ plays the EUF-CMA game with a real signature oracle. Clearly, we have:

$$\Pr[\mathsf{G}_1 = 1] = \mathsf{AdvEUFCMA}_{\mathcal{A}}^{\mathsf{FAEST}}[Q_{\mathsf{sig}}]. \tag{6}$$

G_2**:** $\mathcal{O}_{\mathsf{sig}}$ samples $r \in \{0,1\}^\lambda$ and $\mathsf{iv}^{\mathsf{pre}} \in \{0,1\}^{128}$ at random instead of computing $\mathsf{H}_3(\mathsf{sk}\|\mu\|\rho)$.
The difference between this game and the previous one reduces to the PRF security of H_3 with secret key ρ. Since we need to change (r, iv) in every query to $\mathcal{O}_{\mathsf{sig}}$, we run a hybrid argument and obtain:

$$\Pr[\mathsf{G}_1] \leq \Pr[\mathsf{G}_2] + Q_{\mathsf{sig}} \cdot \mathsf{AdvPRF}_{\mathcal{A}_{\mathsf{PRF}}}^{\mathsf{H}_3}. \tag{7}$$

G_3**:** $\mathcal{O}_{\mathsf{sig}}$ samples $\mathsf{chall}_1 \in \{0,1\}^{5\lambda+64}$ at random in every query. When $\mu\|\mathsf{com}\|\boldsymbol{c}_1\|\ldots\|\boldsymbol{c}_{\tau-1}\|\mathsf{iv}$ is queried to H_2^1, game G_3 abort if this query to H_2^1 has already been made during the game; otherwise program H_2^1 to output chall_1 on this input.
We argue indistinguishability by reduction to unpredictability as stated in Lemma 9.21. We use a hybrid argument, where $\mathsf{G}_{3,i}$ implements the abort only for the first i queries to $\mathcal{O}_{\mathsf{sig}}$. Clearly, $\mathsf{G}_{3,0} = \mathsf{G}_2$ and $\mathsf{G}_{3,Q_{\mathsf{sig}}} = \mathsf{G}_3$. Moreover, $\mathsf{G}_{3,i-1}$ and $\mathsf{G}_{3,i}$ only differ in the i-th $\mathcal{O}_{\mathsf{sig}}$ query, and only if $\mathcal{A}$ predicted the query $\mu\|\mathsf{com}\|\boldsymbol{c}_1\|\ldots\|\boldsymbol{c}_{\tau-1}\|\mathsf{iv}$. The i-th reduction $\mathcal{A}_{\mathsf{unpred},i}$ to unpredictability plays unpredictability game in Lemma 9.21 such that it emulates $\mathsf{G}_{3,i}$, except that for the i-th $\mathcal{O}_{\mathsf{sig}}$-query, it outputs its current state and the list $\mathcal{L}_{2,1}$ of queries made to $\mathsf{H}_{2,1}$, receives $(\mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1})$ from the unpredictability experiment (instead of running VOLECommit), and then continues from there. By construction, $\mathsf{G}_{3,i-1} = 1 \iff \mathsf{ExpUnpred}_{0,\mathcal{A}_{\mathsf{unpred}}}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}} = 1$ and $\mathsf{G}_{3,i} = 1 \iff \mathsf{ExpUnpred}_{1,\mathcal{A}_{\mathsf{unpred},i}}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}}$. Thus, from Lemma 9.21 and a hybrid argument we get

$$\begin{aligned}\Pr[\mathsf{G}_3 = 1] \leq & e^{Q_{\mathsf{sig}} \cdot (4\lceil \log(L) \rceil \beta^2 + 16\beta^2)/2^{128}} \cdot \Big(\Pr[\mathsf{G}_1 = 1] \\ & + Q_{\mathsf{sig}} \cdot \left(\lceil \log(L) \rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1] + \frac{(2Q_1 + Q_{2,1})}{2^{-2\lambda}}\right)\Big)\end{aligned} \tag{8}$$

G_4: $\mathcal{O}_{\mathsf{sig}}$ samples $\mathsf{chall}_2 \in \{0,1\}^{3\lambda+64}$ at random in every query. When $\mathsf{chall}_1 \| \tilde{\mathbf{u}} \| h_V \| \mathbf{d}$ is queried to H_2^2, abort if this query to H_2^2 has already been made during the game; otherwise program H_2^2 to output chall_2 on this input.
G_4 can only abort if a query starting with chall_1 has been made to H_2^2; since chall_1 is fresh because of G_3, by a hybrid argument, we get:

$$\Pr[\mathsf{G}_3 = 1] \leq \Pr[\mathsf{G}_4 = 1] + Q_{\mathsf{sig}} \cdot \frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{5\lambda+64}}.$$

G_5: $\mathcal{O}_{\mathsf{sig}}$ samples $\mathsf{chall}_3 \in \{0,1\}^{\lambda}$ at random in every query in every iteration of FAEST.Sign (in line (20)), until eventually the loop ends with $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}}) \neq 0^{w_{\mathsf{grind}}}$ and $I = \mathsf{DecodeChall}_3(\mathsf{chall}_3; \mathtt{param})$ such that $\mathsf{decom}_I \neq \bot$. When $\mathsf{chall}_2 \| \tilde{a} \| \tilde{b} \| \mathsf{ctr}$ is queried to H_2^3, abort if this query to H_2^3 has already been made during the game; otherwise program H_2^3 to output chall_3 on this input.
G_5 can only abort if a query starting with chall_2 has been previously made to H_2^3; since chall_2 is fresh because of G_4, by a hybrid argument, we get:

$$\Pr[\mathsf{G}_4 = 1] \leq \Pr[\mathsf{G}_5 = 1] + Q_{\mathsf{sig}} \cdot \frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}}.$$

Observe that the relevant elements of this game are now compatible with the hiding experiment for VOLECommit described in Lemma 9.20.

G_6: $\mathcal{O}_{\mathsf{sig}}$ replaces the call to VOLECommit by uniformly sampling $\mathbf{u} \in \mathbb{F}_2^{\hat{\ell}}$ and $(\boldsymbol{c}_i)_{i \in [1..\tau)}$ and computing $\mathbf{V} \in \mathbb{F}_2^{\hat{\ell} \times \lambda}$ such that the VOLE relation holds with the Δ and $\mathbf{q}_i$ values that result from chall_3; it does so in every signature query.
In the reduction, we sample chall_3 as in game G_5, except that we replace the successful challenge chall_3 (where $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}}) \neq 0^{w_{\mathsf{grind}}}$ and $\mathsf{decom}_I \neq \bot$) with the challenge of the hiding game. This game perfectly corresponds to game G_5 when $b = 0$, i.e. the real VOLECommit is used, or G_6 when $b = 1$. Too see this here, note that $\mathsf{decom}_I \neq \bot$ is derivable given only I (Remark 9.19), and thus embedding chall_3 obtained from the hiding game for the successful counter ctr poses no problem as decom_I is not needed to check if chall_3 satisfies $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}}) \neq 0^{w_{\mathsf{grind}}}$ and $\mathsf{decom}_I \neq \bot$. Moreover, since game G_5 establishes that programming works (or the game aborts), we can sample all chall_3 ahead of time. Hence, the non-adaptive nature of hiding as in Lemma 9.20 also poses no problem, and conceptual changes suffice to facilitate the reduction.
By Lemma 9.20 and a hybrid argument, the change in success is therefore bounded as

$$\Pr[\mathsf{G}_5 = 1] \leq e^{Q_{\mathsf{sig}}\tau\left(4\lceil\log(L)\rceil\beta^2 + 16\beta^2 N_0^2 + (\lceil\hat{\ell}/128\rceil)^2\right)/2^{128}} \cdot \Big(\Pr[\mathsf{G}_6 = 1] + Q_{\mathsf{sig}}\tau \cdot \Big(\lceil\log(L)\rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1] + \mathsf{AdvPRG}^{\mathsf{LeafCommit}}_{\mathcal{B}_2}[N_0] + \mathsf{AdvPRP}^{\mathsf{PRG}_{\hat{\ell}}}_{\mathcal{B}_3}[1]\Big)\Big)$$

After this change, the distribution of $(\mathbf{u}, \mathbf{V})$ is uniform (modulo the VOLE relation) and independent of the VOLE commitments $(\boldsymbol{c}_i)_{i \in [1..\tau)}$.

G_7: In every query, $\mathcal{O}_{\mathsf{sig}}$ samples $\tilde{\boldsymbol{u}}$ and $\tilde{\mathbf{V}}$ at random (modulo the VOLE relation with $\tilde{\mathbf{Q}}$ and Δ) instead of computing VOLEHash during the signing. It then adjusts the last $\lambda + B$ elements of $\mathbf{u}$ and rows of $\mathbf{V}$ to match $\tilde{\boldsymbol{u}}$ and $\tilde{\mathbf{V}}$ when VOLEHash is applied under challenge chall_1 to $\mathbf{u}, \mathbf{V}$.
Since the last $\lambda + B$ elements of $\mathbf{u}$ and rows of $\mathbf{V}$ are uniformly random (modulo the VOLE relation) because of G_6, and VOLEHash is $\mathbb{F}_2^{\ell+\lambda}$-hiding by Lemma 4.10, this game is perfectly indistinguishable from the previous one.

G_8: In every query, $\mathcal{O}_{\mathsf{sig}}$ samples $\tilde{a}_0, \tilde{a}_1, \tilde{a}_2$ at random (modulo the VOLE relation) instead of computing ZKHash. It then computes OWFProve, but adjusts the $\mathbf{u}, \mathbf{V}$ in rows $[\ell..\ell + 2\lambda)$ to match $\tilde{a}_0, \tilde{a}_1, \tilde{a}_2$ when ZKHash is applied under challenge chall_2 to $\mathbf{a}_0 \| u_0^*, \mathbf{a}_1 \| v_0^* + u_1^*, \mathbf{a}_2 \| v_1^*$.

Since the respective rows of $\mathbf{u}$ and $\mathbf{V}$ are uniformly random (modulo the VOLE relation) because of G_6, and ZKHash is $\mathbb{F}_{2^\lambda}^{\ell}$-hiding by Lemma 4.11, the distribution of $(\tilde{a}_0, \tilde{a}_1, \tilde{a}_2)$ is unchanged and this game is perfectly indistinguishable from the previous one.

G_9**:** Now that the computation of $\tilde{a}_0, \tilde{a}_1, \tilde{a}_2$ is independent of $\mathbf{w}$, $\mathcal{O}_{\mathsf{sig}}$ will sample $\mathbf{d}$ uniformly at random in every query instead of computing $\mathbf{d} := \mathbf{w} + \mathbf{u}_{[0..\ell)}$.
This does not change the distribution of $\mathbf{d}$ because $\mathbf{u}$ is uniform since G_6 and the first ℓ elements were not used in games G_7 and G_8 to produce their outputs, hence this game is perfectly indistinguishable from the previous one.

In G_9, we see that the distribution of σ produced by $\mathcal{O}_{\mathsf{sig}}$ no longer depends on the secret key sk and the success probability of $\mathcal{A}$. Taking together the intermediate advantages, we arrive at

$$\begin{aligned}
\Pr[\mathsf{G}_0 = 1] \leq \rho \cdot \Bigg(& \Pr[\mathsf{G}_9 = 1] \\
& + Q_{\mathsf{sig}} \cdot \mathsf{AdvPRF}^{\mathsf{H}_3} \\
& + Q_{\mathsf{sig}} \cdot \left(\lceil \log(L) \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1] + \frac{2Q_1 + Q_{2,1}}{2^{2\lambda}} \right) \\
& + Q_{\mathsf{sig}} \cdot \left(\frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{5\lambda+64}} + \frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}} \right) \Bigg) \\
= \rho \cdot & \Pr[\mathsf{G}_9 = 1] \\
& + \rho \cdot Q_{\mathsf{sig}} \cdot \left(\mathsf{AdvPRF}^{\mathsf{H}_3} + \lceil \log(L) \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1] \right) \\
& + \rho \cdot Q_{\mathsf{sig}} \cdot \left(\frac{2Q_1 + Q_{2,1}}{2^{2\lambda}} + \frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{5\lambda+64}} + \frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}} \right)
\end{aligned}$$

where

$$\rho = e^{Q_{\mathsf{sig}} \cdot (4\lceil \log(L) \rceil \beta^2 + 16\beta^2)/2^{128} + Q_{\mathsf{sig}} \tau \left(4\lceil \log(L) \rceil \beta^2 + 16\beta^2 N_0^2 + (\lceil \hat{\ell}/128 \rceil)^2 \right)/2^{128}}$$

is bounded by 2 for all proposed parameter sets. Observe also, that in the reduction up until now, there are no differences for FAEST or FAEST-EM.

At this point, we reduce to EUF-KO security. We note here, that there is a formal compositional problem: The EUF-CMA reduction programs random oracles, which interferes with the EUF-KO claims. However, these problems are of formal nature. If we would instead include the EUF-KO proof here, then the advantage terms due to collisions, pre-image attacks, or guesses of random oracle outputs would be the same. □

9.7 QROM proof

We present a proof of EUF-CMA security in the quantum-accessible random oracle model. The proof follows the following strategy.

The strategy for reducing breaking EUF-CMA to EUF-KO (Theorem 9.42) is identical to that in the ROM proof. Whenever a random oracle needs to be reprogrammed to a fresh, random output at a location that has min-entropy given the adversary's view, we use the adaptive reprogramming lemma (also called "resampling lemma") [**?**]. When replacing chall_1 by a random value, however, the corresponding input to H_1 is only computationally unpredictable for the adversary. To exploit this computational unpredictability in a clean but tight way, we prove the unpredictability of the BAVC commitments with "quantum guesses" (see Lemma 9.41).

For proving EUF-KO security, we employ a lossy key argument to reduce from soundness of FAEST as a non-interactive argument. This step is identical to the ROM reduction.

We would then like to follow the same strategy as in the ROM reduction: switch to an interactive version of FAEST and then exploit round-by-round (RBR) soundness. It is, however, unclear how to perform these two steps *separately* in the QROM: The equivalence of soundness of the Fiat-Shamir (FS) transformation of a given interactive proof system (IP) and its state restoration soundness seems to be unlikely in this setting, and the natural IP underlying FAEST does not have RBR soundness without idealizing the BAVC commitment. Instead we go one step further and formulate FAEST as a (slightly non-standard) FS transformation of an IP with more rounds, interpreting every random oracle call, including the ones inside BAVC.Commit in FAEST as a FS challenge generation. This way, we get to a RBR-sound IP in one step, as the commitments in the this IP are the statistically-binding leaf commitments (with the tree commitment that comes on top interpreted as rounds of interaction). This facilitates a QROM reduction via compressed-oracle-based search bound techniques [CFHL21, AHJ+23] (Lemma 9.40).

9.7.1 FAEST as non-standard Fiat Shamir transformation of 7-challenge IP (6-challenge for FAEST-EM). We begin by formulating FAEST as a slightly non-standard Fiat-Shamir transformation of a 7-challenge (6-challenge for FAEST-EM) interactive proof system, FAEST-IP$^+$. The 8 (7) prover messages of this protocol are

1. $\mathsf{pmsg}_{-3} = \mathsf{iv}^{\mathsf{pre}}$
2. $\mathsf{pmsg}_{-2} = \iota$, the empty string
3. $\mathsf{pmsg}_{-1} = \mathsf{com}_{0,0} \| \mathsf{com}_{0,1} \| \dots \mathsf{com}_{0,N_0} \| \mathsf{com}_{1,0} \| \dots \mathsf{com}_{\tau-1,N_{\tau-1}}$ as produced by FAEST.Sign in subroutine VOLECommit in subrouting BAVC.Commit
4. $\mathsf{pmsg}_0 = \iota$, the empty string
5. pmsg_1 is the argument of H_2^1 in FAEST.Sign with μ,com and iv removed
6. pmsg_2 is the argument of H_2^2 in FAEST.Sign with chall_1 removed
7. pmsg_3 is the argument of H_2^3 in FAEST.Sign with chall_2 removed
8. pmsg_4 is the signature, with chall_3 removed.

For FAEST-EM, pmsg_{-2} is omitted. For clarity, we keep the numbering of rounds in this case, meaning that the prover messages are $\mathsf{pmsg}_{-3}, \mathsf{pmsg}_{-1}, \dots$ and similar for the challenges. The challenges are

1. $\mathsf{chall}_{-3} = \mathsf{iv}$
2. $\mathsf{chall}_{-2} = (\mathsf{uhash}_0, \dots, \mathsf{uhash}_{\tau-1})$
3. $\mathsf{chall}_{-1} = h_0 \| \dots \| h_\tau$
4. $\mathsf{chall}_0 = \mathsf{com}$
5. chall_1
6. chall_2
7. chall_3

For FAEST-EM, chall_{-2} is omitted. The honest prover computes the prover messages as FAEST.Sign would compute the corresponding quantities, except: i) it samples $r, \mathsf{iv}^{\mathsf{pre}}$ at random, ii) there is only one attempt to obtain a chall_3 with a suffix of w_{grind} zeroes (denote the variant of FAEST where Line 4 of FAEST.Sign is replaced by random sampling, and where only one, say random, value of ctr is attempted, by FAEST_r ($\mathsf{FAEST\text{-}EM}_r$)), and iii) it uses the challenges in place of hash values where FAEST.Sign calls $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_4$ or any of the H_2^i, $i = 1, 2, 3$. Note that as the change from FAEST to FAEST_r leaves verification unchanged and thus does not affect soundness.

The FAEST-IP$^+$verifier runs FAEST.Verify, omitting Lines 2,3,10 and 15, and replacing Lines 18 and 19. Instead of Lines 18 and 19, the verifier compares the result $\tilde{a}_0'$ of Line 17 with the value $\tilde{a}_0$ from pmsg_3, and rejects if they differ, accepts otherwise. In case of FAEST-EM, the FAEST-IP$^+$ verifier additionally rejects if any of the $\mathsf{com}_{i,j}$ have more

than one pre-image. This verifier is not efficient, which is not a problem as we are only interested in soundness of the IP.

Note that FAEST-IP$^+$ is in the plain model: We replaced the call to H_3 by random sampling and all other hash calls by interaction.

We now describe the Fiat Shamir transformation we apply to FAEST-IP$^+$ to get back to a variant of FAEST (with random $\mathsf{iv}^{\mathsf{pre}}, r$ and with an equivalent verification). For readability, we omit the output length parameter of the hash functions. The challenges are derived as follows.

1. $\mathsf{chall}_{-3} = \mathsf{H}_4(\mathsf{pmsg}_{-3})$
2. $\mathsf{chall}_{-2} = \mathsf{H}_0(\mathsf{chall}_{-3})$
3. $\mathsf{chall}_{-1} = h_0 \| \dots \| h_\tau$ with $h_i = \mathsf{H}_1(\mathsf{com}_{i,0} \| \mathsf{com}_{i,1} \| \dots \mathsf{com}_{i,N_i})$
4. $\mathsf{chall}_0 := \mathsf{H}_1(\mathsf{chall}_{-1} \| \mathsf{pmsg}_0)$
5. $\mathsf{chall}_1 := \mathsf{H}_2^1(H_2^0(\mathsf{pk} \| \mathsf{msg}) \| \mathsf{com} \| \mathsf{pmsg}_1 \| \mathsf{iv}^{\mathsf{pre}})$
6. $\mathsf{chall}_2 := \mathsf{H}_2^2(\mathsf{chall}_1 \| \mathsf{pmsg}_2)$
7. $\mathsf{chall}_3 := \mathsf{H}_2^3(\mathsf{chall}_2 \| \mathsf{pmsg}_3)$

There are three main ways in which this transformation deviates from all standard variants of Fiat Shamir: i) chall_{-1} is derived without hashing the previous challenge. The commitment $\mathsf{com}_{i,j}$ is, however, an injective function of uhash_i. ii) The function for deriving chall_{-1} is not a random oracle. The two-step computation of com is, however, only for efficiency (to allow parallel hashing), so it should be plausible that this is not a problem, and will be taken into account in the proof. iii) The instance (the public key) is pre-hashed and only included in the hash argument for deriving chall_1, the 5th challenge.

The standard Fiat Shamir prover (outputting all pmsg_i) and verifier (recomputing all chall_i) with challenges derived as described above yield a variant $\mathsf{FAEST}_r^{\mathrm{elc}}$ ("explicit leaf commitment") of FAEST_r ($\mathsf{FAEST\text{-}EM}_r^{\mathrm{elc}}$ of $\mathsf{FAEST\text{-}EM}_r$) that is equivalent from a security point of view, and differs only in efficiency (see Lemma 9.40)

9.7.2 RBR soundness of FAEST-IP$^+$.

We prove RBR soundness for the expanded IP.

Lemma 9.37. FAEST-*IP*$^+$ *has RBR soundness with passive prefix up to* chall_{-3}*, where challenge* chall_{-2} *is isolated, with RBR soundness errors*

$$\varepsilon_{-3} = \varepsilon_{-1} = \varepsilon_0 = 0 \tag{9}$$

$$\varepsilon_{-2} = \tau \cdot 2^{-\lambda} \tag{10}$$

$$\varepsilon_1 = \varepsilon_2 = \varepsilon_3 = \varepsilon := 3 \cdot 2^{-\lambda} \tag{11}$$

Proof. Define the state function RBRState to output 1 if one of the values uhash_i in chall_{-2} defines a non-injective LeafHash. If that is not the case, let $g_1 = (\mathbf{u}_i, \mathbf{V}_i)_i$ be the VOLEs defined by the unique preimages of the $\mathsf{com}_{i,j}$. If none of the values uhash_i in chall_{-2} defines a non-injective LeafHash, RBRState computes its output using the RBR state function from Lemma 9.34 on input $(\mathbf{c}_1 \| \dots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}^{\mathsf{pre}}) \| g_1$. Now the expression for ε_{-2} follows by the analysis in the proof of Lemma 9.15, and the expression for $\varepsilon_1, \varepsilon_2, \varepsilon_3$ is due to Lemma 9.34. The passive prefix and isolation statements follow by construction of RBRState.

For FAEST-EM, it suffices to additionally observe that we defined the verifier to reject whenever the extraction of the $\mathsf{com}_{i,j}$ yields multiple pre-images. □

9.7.3 FAEST$_r^{\mathrm{elc}}$ soundness to FAEST-IP$^+$ RBR soundness.

We recall a lemma for QROM query bounds from [AHJ+23]. In the following, we identify $D \in (\mathcal{Y} \cup \{\perp\})^{\mathcal{X}}$ with a partial function from $\mathcal{X}$ to $\mathcal{Y}$, where $D(x) = \perp$ indicates that $D(x)$ is undefined. We

write $D[x \mapsto u]$ for the partial function that maps x to u and x' to $D(x')$ for all $x' \neq x$. We say that a predicate P on $(\mathcal{Y} \cup \{\bot\})^{\mathcal{X}}$ has q_{P}-local witnesses if for all $D \in (\mathcal{Y} \cup \{\bot\})^{\mathcal{X}}$ with $\mathsf{P}(D) = 1$ there exists a tuple $z = (x_1, \ldots, x_i) \in \mathcal{X}^i$ for $i \leq q_{\mathsf{P}}$ such that for all $D' \in (\mathcal{Y} \cup \{\bot\})^{\mathcal{X}}$ that agree with D on z, i.e. $D'(x_j) = D(x_j)$ for $j = 1, \ldots, i$, it holds that $\mathsf{P}(D') = 1$. We denote the validity of a witness z for a predicate P and a (partial) function D as $\langle z, \mathsf{P}, D\rangle$. Furthermore we define $|D| = |\{x \in \mathcal{X} | D(x) \neq \bot\}$.

Lemma 9.38 (Special case of Lemma 1 in [AHJ+23]). *Let $\mathsf{H} : \mathcal{X} \to \mathcal{Y}$ be a random oracle and let $\mathsf{P} : (\mathcal{Y} \cup \{\bot\})^{\mathcal{X}} \to \{0,1\}$ be a predicate with q_{P}-local witnesses. Let further $\mathcal{A}^{\mathsf{H}}$ be a QROM algorithm making at most q quantum queries to H and outputs a candidate witness z for P. Then*

$$\sqrt{\Pr_{z \leftarrow \mathcal{A}^{\mathsf{H}}}[\langle z, \mathsf{P}, \mathsf{H}\rangle]} \leq \sum_{k=1}^{q+q_{\mathsf{P}}} \sqrt{10 \max_{\substack{x, D: \\ |D| \leq k \\ \neg \mathsf{P}(D)}} \Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}(D[x \mapsto u])]}. \tag{12}$$

Here the maximum is taken over $x \in \mathcal{X}$ and $D \in (\mathcal{Y} \cup \{\bot\})^{\mathcal{X}}$.

We are now ready to prove the soundness of FAEST based on the round-by-round soundness of FAEST-IP$^+$. For signature schemes, we consider a notion of soundness where an adversary presented with an invalid public key cannot produce a valid message-signature pair, except with the soundness advantage AdvSnd.

Lemma 9.39. *For any static QROM adversary $\mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_2^0,\mathsf{H}_2^1,\mathsf{H}_2^2,\mathsf{H}_2^3,\mathsf{H}_4}$ against $\mathsf{FAEST}_r^{\mathrm{elc}} = \mathsf{FS}[\mathsf{FAEST}\text{-}IP^+]$ that makes a total of $Q_0, Q_1, Q_{2,0}, Q_{2,1}, Q_{2,2}, Q_{2,3}, Q_4$ quantum queries to its oracles $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_4$, respectively,*

$$\mathsf{AdvSnd}_{\mathcal{A}}^{\mathsf{FAEST}_r^{\mathrm{elc}}} \leq 10 \cdot Q^3 \cdot 2^{-2\lambda} + 10 \cdot Q^2 \cdot \tau \cdot 2^{-\lambda}.$$

For FAEST-EM, the advantage is instead bounded as

$$\mathsf{AdvSnd}_{\mathcal{A}}^{\mathsf{FAEST\text{-}EM}_r^{\mathrm{elc}}} \leq 10(\tau+1) \cdot Q^3 \cdot 2^{-2\lambda} + 30 \cdot Q^2 \cdot 2^{-\lambda}.$$

Here,

$$Q = Q_0 + Q_1 + Q_{2,0} + Q_{2,1} + Q_{2,2} + Q_{2,3} + Q_4 + 2\tau + 12.$$

Proof. We instantiate the random oracles via domain separation of a random oracle H with sufficiently large output length for this proof, by setting $\mathsf{H}_i^j = \mathsf{H}(i\|j\|\cdot)$ truncated to the correct output length, with appropriate encoding of the number prefixes, where $\mathsf{H}_i^0 := \mathsf{H}_i$ for $i \in \{0,1,3,4\}$. For ease of notation, we define $\hat{\mathsf{H}}_i$ such that $\hat{\mathsf{H}}_i$ is used to derive chall_i. In particular, $\hat{\mathsf{H}}_{-1}$ is the function used to map $\mathsf{com}_{0,0}\|\mathsf{com}_{0,1}\| \ldots \mathsf{com}_{0,N_0}\|\mathsf{com}_{1,0}\| \ldots \mathsf{com}_{\tau-1,N_{\tau-1}}$ to $h_0\| \ldots \|h_{\tau-1}$. Let T^{Π} be the set of transcripts for Π and $\mathcal{R}_i^j$ the range of H_i^j, $\hat{\mathcal{R}}_i$ the range of $\hat{\mathsf{H}}_i$, $M_i^j = \left|\mathcal{R}_i^j\right|$ and $\hat{M}_i = |\mathcal{R}_i|$. To use Lemma 9.38, we define the following predicate on pairs of partial transcripts $\mathsf{trc} = (\mathsf{msg}, \mu, \mathsf{pmsg}_{-3}, \mathsf{chall}_{-3}, \ldots, \mathsf{chall}_i)$,

$$\begin{aligned}
\mathrm{pred}_{\mathsf{pk},\mathsf{H}}(\mathsf{trc}, \mathsf{msg}) = & \mathsf{RBRState}(\mathsf{pk}, \mathsf{trc}) \\
& \wedge \neg \mathsf{RBRState}(\mathsf{pk}, \mathsf{trc}_{i-1}) \\
& \wedge \left(\forall j \in [-3 : i] \setminus \{-1, 1\} : \mathsf{chall}_j = \hat{\mathsf{H}}_j(\mathsf{chall}_{j-1}\|\mathsf{pmsg}_j)\right) \\
& \wedge \left(i < -1 \vee \mathsf{chall}_{-1} = \hat{\mathsf{H}}_{-1}(\mathsf{pmsg}_{-1})\right) \\
& \wedge \left(i < 1 \vee \mathsf{chall}_1 = \hat{\mathsf{H}}_1(\mu\|\mathsf{chall}_0\|\mathsf{pmsg}_1)\right) \\
& \wedge \mathsf{H}_2^0(\mathsf{pk}, \mathsf{msg}) = \mu
\end{aligned}$$

Here we have defined the truncation $\mathsf{trc}_{i-1} = (\mathsf{msg}, \mu, \mathsf{pmsg}_1, \mathsf{chall}_1, \ldots, \mathsf{chall}_{i-1})$. If $\mathrm{pred}_{\mathsf{pk},\mathsf{H}}(\mathsf{trc}, \mu)$, we say (trc, μ) is *bad* for H. We further define a predicate $\mathsf{P} : (\mathcal{Y} \cup \{\bot\})^{\mathcal{X}} \to \{0,1\}$ by $\mathsf{P}(D) :\Leftrightarrow$ there exists bad (trc, μ) for D. P has $(\tau+4)$-local witnesses. I particular if (trc, μ) is bad for D and trc ends in chall_{-3} or chall_{-2}, then a 1-local witness exists as chall_{-2} is isolated and chall_{-3} has RBR soundness error 0 so trc cannot actually end in chall_{-2} and fulfil pred. If trc ends in chall_i for $i \in [-1:3]$, the list of random oracle inputs in the domains of H_1, H_2^0 and $\hat{\mathsf{H}}_i$, $i = 1,2,3$, queried when evaluating $\mathrm{pred}_{\mathsf{pk},\mathsf{H}}(\mathsf{trc}, \mathsf{msg})$ is a witness if $\mathrm{pred}_{\mathsf{pk},\mathsf{H}}(\mathsf{trc}, \mu)$. The at most $\tau + 5$ inputs include the τ inputs to H_1 for deriving chall_{-1}. We call i the length of the witness. In the following we denote by D_i^j and $\hat{D}_i$ the partial functions obtained in the same way as H_i^j and $\hat{\mathsf{H}}_i$ are obtained from H. We also define the collision predicate P_C by setting $\mathsf{P}_C(D)$ iff there exist inputs x, x' in the domain of the same hash function H_1 or H_2^i, $i = 0,1,2,3$ such that $D(x) = D(x') \neq \bot$. Note that this prevents collisions for chall_{-1} as well as $\hat{\mathsf{H}}_0 = \mathsf{H}_1$ is used for deriving chall_{-1}. The collision predicate clearly has $(\tau + 5)$-local witnesses (it has 2-local witnesses). We construct algorithm $\mathcal{B}^{\mathsf{H}}$ as follows. $\mathcal{B}^{\mathsf{H}}$ runs $(\mathsf{msg}, \pi) \leftarrow \mathcal{A}^{\mathsf{H}}$ and recomputes the challenges. Denote the completed transcript by trc. If there exists i such that $\mathsf{RBRState}(\mathsf{pk}, \mathsf{trc}_i)$ and $\neg\mathsf{RBRState}(\mathsf{pk}, \mathsf{trc}_{i-1})$, output the predicate witness z, else, output $\bot$. By RBR soundness (Lemma 9.37) we have

$$\begin{aligned}\Pr_{\pi \leftarrow \mathcal{A}^{\mathsf{H}}(\mathsf{pk})}[\text{verifier accepts } \pi] &\leq \Pr_{z \leftarrow \mathcal{B}^{\mathsf{H}}}[\langle z, \mathsf{P}, \mathsf{H}\rangle] \\ &\leq \Pr_{z \leftarrow \mathcal{B}^{\mathsf{H}}}[\langle z, \mathsf{P} \vee \mathsf{P}_C, \mathsf{H}\rangle] \qquad (13)\end{aligned}$$

We now aim to bound the quantity

$$\max_{\substack{x, D: \\ |D| \leq k \\ \neg\mathsf{P}'(D)}} \Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}'(D[x \mapsto u])]$$

for $\mathsf{P}' = \mathsf{P} \vee \mathsf{P}_C$. Here, D is a partial function with the same domain and range as H. We denote by $|D_i|$ the number of different inputs x in the domain of $\hat{\mathsf{H}}_i$ such that $D(x) \neq \bot$. We distinguish different cases depending on which domain x is from.

1. x is in the domain of $\hat{\mathsf{H}}_{-3}$. As the RBR soundness error for this round is 0 and RBR soundness holds with passive prefix up to chall_{-2}, we have
$$\Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}'(D[x \mapsto u])] \leq 0$$
2. x is in the domain of $\hat{\mathsf{H}}_{-2}$. There are two ways to fulfill $\mathsf{P}'(D[x \mapsto u])$, i) if a collision occurs, i.e., $\exists x' : \hat{D}_{-2}(x) = \hat{D}_{-2}(x') \neq \bot$, or ii) if a u is chosen such that $\mathsf{RBRState}(\mathsf{pk}, \tau|x|u) = 1$. By Lemma 9.37, chall_{-2} is isolated, thus we have
$$\Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}'(D[x \mapsto u])] \leq \frac{|\hat{D}_{-2}|}{\hat{M}_{-2}} + \tau \cdot 2^{-\lambda} =: \hat{\delta}_{-2}$$
using the RBR soundness error given by Lemma 9.37.
3. x is in the domain of H_1. For D with $\neg\mathsf{P}'(D)$, there are 2 ways to fulfil $\mathsf{P}'(D[x \mapsto u])$. The first one is if a new collision is found, i.e. $\exists x' : D_1^0(x) = D_1^0(x') \neq \bot$. This happens with probability at most $\frac{k}{M_1^0}$. The second one is that there exists $x' = h_0 \| \ldots \| h_{\tau-1}$ in the domain of H_1 such that $D_1^0(x') \neq \bot$ and there exists $i \in [0, \tau)$ such that $u = h_i$, or $x' = \mu \| \mathsf{com} \| \mathsf{pmsg}_0$ in the domain of $\hat{\mathsf{H}}_1$ such that $u = \mathsf{com}$. This happens with probability at most $\frac{\tau k}{M_1} + \frac{k}{M_2^1}$. In summary, we get
$$\Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}'(D[x \mapsto u])] \leq \frac{(\tau+1)|D_1|}{M_1} + \frac{\hat{D}_1}{\hat{M}_1} =: \delta_1.$$

4. x is in the domain of H_2^0. The only way to fulfil $\mathsf{P}'(D[x \mapsto u])$ in this case is by forming a collision, so

$$\Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}'(D[x \mapsto u])] \leq \frac{|D_2^0|}{\hat{M}_i} =: \delta_{2,0}$$

5. x is in the domain of $\hat{\mathsf{H}}_i$ for $i = 1, 2$. For D with $\neg\mathsf{P}'(D)$, there are 3 ways to fulfil $\mathsf{P}'(D[x \mapsto u])$. The first option is that there exists x' in the domain of $\hat{\mathsf{H}}_i$, $x' \neq x$ such that $D(x') = u$ (i.e. a new collision has formed). The second option is that a new witness of length $i' > i$ for P is completed. In this case, there exists pmsg_{i+1} such that $\hat{D}_{i+1}(u \| \mathsf{pmsg}_{i+1}) \neq \perp$. The third option is that a new witness of length i for P is completed. In this case, it is necessary that there exists a partial witness (which is unique by $\neg\mathsf{P}_C(D)$)

$$z = \Big(0\|0\|\mathsf{iv}, 1\|1\|\mathsf{com}_{0,0}\| \ldots \|\mathsf{com}_{0,N_0}, \ldots, 1\|1\|\mathsf{com}_{\tau-1,0}\| \ldots \|\mathsf{com}_{\tau-1,N_{\tau-1}},$$
$$1\|1\|\mathsf{chall}_{-1}, 2\|1\|\mu\|\mathsf{chall}_0|\mathsf{pmsg}_1\Big)$$

if $i = 2$, and a partial witness of the same form except without the last entry if $i = 1$, with the following properties. Let $\mathsf{chall}_{i-1} = \hat{D}_{i-1}(z_{\text{last}})$, where z_{last} is the last entry of z, let trc be the transcript obtained from z, and let trc_{i-2} be its truncation to the second-to-last challenge. Then

(a) z is consistent, i.e., recomputing the chall_i that appear in the entries of z using D succeeds
(b) $\mathsf{RBRState}(\mathsf{pk}, \mathsf{trc}_{i-2}) = 0$
(c) there exists pmsg_i such that $x = 2\|i\|\mathsf{chall}_{i-1}\|\mathsf{pmsg}_i$ and $\mathsf{RBRState}(\mathsf{pk}, (\mathsf{pmsg}_1, \mathsf{chall}_1, \ldots, \mathsf{pmsg}_i, u)) = 1$.

In summary, we have

$$\Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}'(D[x \mapsto u])] \leq \frac{|\hat{D}_i| + |\hat{D}_{i+1}|}{\hat{M}_i} + \varepsilon =: \hat{\delta}_i,$$

where the term $\frac{|\hat{D}_i|+|\hat{D}_{i+1}|}{\hat{M}_i}$ is from the first and second option, we have used RBR soundness from Lemma 9.37 to bound the probability

$$\Pr_{u \leftarrow \mathcal{Y}}[\mathsf{RBRState}(\mathsf{pk}, (\mathsf{pmsg}_1, \mathsf{chall}_1, \ldots, \mathsf{pmsg}_i, u))] \leq \varepsilon$$

in the third option, and ε is defined in Lemma 9.37.

6. x is in the domain of $\hat{\mathsf{H}}_3$. The analysis here is identical to item 4., except that $\hat{\mathsf{H}}_3$ is not included in the collision predicate, and chall_3 is the last challenge, so only option 3 exists for $\hat{\mathsf{H}}_3$. We thus get

$$\Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}'(D[x \mapsto u])] \leq \varepsilon =: \hat{\delta}_3$$

in this case

In conclusion, we have

$$\max_{\substack{x,D: \\ |D| \leq k \\ \neg\mathsf{P}'(D)}} \Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}'(D[x \mapsto u])] \leq \max(\hat{\delta}_{-2}, \delta_1, \delta_{2,0}, \hat{\delta}_1, \hat{\delta}_2, \hat{\delta}_3) := \delta(k).$$

Using Equation (13), and applying Lemma 9.38 to the $(\tilde{Q}+\tau+7)$-query algorithm $\mathcal{B}$, where $\tilde{Q} = Q_0 + Q_1 + Q_{2,0} + Q_{2,1} + Q_{2,2} + Q_{2,3} + Q_4$, we obtain

$$\begin{aligned}
&\Pr_{(\mathsf{pk},\pi)\leftarrow\mathcal{A}^{\mathsf{H}_1,\mathsf{H}_2^1,\ldots,\mathsf{H}_2^\ell}}[\mathsf{pk}\notin\mathcal{L}\wedge\text{verifier accepts}] \\
&\leq \left(\sum_{k=1}^{\tilde{Q}+2\tau+12}\sqrt{10\max_{\substack{x,D:\\ |D|\leq k\\ \neg\mathsf{P}(D)}}\Pr_{u\leftarrow\mathcal{Y}}[\mathsf{P}(D[x\mapsto u])]}\right)^2 \\
&\leq \left(\sum_{k=1}^{\tilde{Q}+2\tau+12}\sqrt{10\delta(k)}\right)^2 \\
&\leq 10(\tilde{Q}+2\tau+12)\sum_{k=1}^{\tilde{Q}+2(\ell+1)}\delta(k)
\end{aligned}$$

Here, the third inequality is the Cauchy-Schwarz inequality. Evaluating the maximum for the FAEST parameters and upper-bounding by a sum when it depends on Q which term is larger we get $\delta(k) \leq (\tau+1)(\tilde{Q}+2\tau+12)2^{-2\lambda} + \tau\cdot 2^{-\lambda}$

$$\begin{aligned}
&\Pr_{(\mathsf{pk},\pi)\leftarrow\mathcal{A}^{\mathsf{H}_1,\mathsf{H}_2^1,\ldots,\mathsf{H}_2^\ell}}[\mathsf{pk}\notin\mathcal{L}\wedge\text{verifier accepts}] \\
&\leq 10(\tau+1)(\tilde{Q}+2\tau+12)^3 2^{-2\lambda} + 10(\tilde{Q}+2\tau+12)^2\tau\cdot 2^{-\lambda}.
\end{aligned}$$

For FAEST-EM, the term $10(\tilde{Q}+2\tau+12)^2\tau\cdot 2^{-\lambda}$ is replaced by the next largest term that scales with Q^2, $30(\tilde{Q}+2\tau+12)^2\cdot 2^{-\lambda}$. □

9.7.4 FAEST EUF-KO ⇐ FAEST$_r^{\text{elc}}$ soundness

Lemma 9.40. *Let $\mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_2^0,\mathsf{H}_2^1,\mathsf{H}_2^2,\mathsf{H}_2^3,\mathsf{H}_3,\mathsf{H}_4}$ be a QROM* EUF-KO *adversary against* FAEST *that makes a total of $Q_0, Q_1, Q_{2,0}, Q_{2,1}, Q_{2,2}, Q_{2,3}, Q_3, Q_4$ queries to its oracles. Then there exist a quantum PRG adversary $\mathcal{B}_1$ against* OWF *and a QROM adversary $\mathcal{B}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_2^0,\mathsf{H}_2^1,\mathsf{H}_2^2,\mathsf{H}_2^3,\mathsf{H}_4}$ against* FAEST$_r^{\text{elc}}$ *that makes at most a total of $Q_0+1, Q_1+\tau+1, Q_{2,0}+1, Q_{2,1}+1, Q_{2,2}+1, Q_{2,3}+1, Q_4+1$ queries to its oracles such that*

$$\mathsf{AdvEUFKO}_{\mathcal{A}}^{\mathsf{FAEST}} \leq \frac{1}{1-\frac{\mathsf{dom(OWF)}}{\mathsf{cod(OWF)}}}\cdot\left(\mathsf{AdvSnd}_{\mathcal{B}}^{\mathsf{FAEST}_r^{\text{elc}}} + \mathsf{AdvPRG}_{\mathcal{B}_1}^{\mathsf{OWF}}\right)$$

For FAEST-EM*, there exists an adversary $\mathcal{C}$ against the almost injectivity (Definition 9.13) of* PRG *such that the following similar bound holds.*

$$\mathsf{AdvEUFKO}_{\mathcal{A}}^{\mathsf{FAEST\text{-}EM}} \leq \frac{1}{1-\frac{\mathsf{dom(OWF)}}{\mathsf{cod(OWF)}}}\cdot\left(\mathsf{AdvSnd}_{\mathcal{B}}^{\mathsf{FAEST\text{-}EM}_r^{\text{elc}}} + \mathsf{AdvPRG}_{\mathcal{B}_1}^{\mathsf{OWF}} + \mathsf{AdvOivInj}_{\mathcal{C}}^{\mathsf{PRG}_{2\lambda}}[L]\right).$$

Proof. First, define $\mathcal{A}'^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_2^0,\mathsf{H}_2^1,\mathsf{H}_2^2,\mathsf{H}_2^3,\mathsf{H}_4}$ as identical to $\mathcal{A}$ except that it simulates H_3 locally. As FAEST.Verify does not call H_3, this change is perfectly indistinguishable, so

$$\mathsf{AdvSnd}_{\mathcal{A}}^{\mathsf{FAEST}} = \mathsf{AdvSnd}_{\mathcal{A}}^{\mathsf{FAEST}_r} = \mathsf{AdvSnd}_{\mathcal{A}'}^{\mathsf{FAEST}_r}.$$

Now define a prover $\mathcal{B}$ for FAEST$_r^{\text{elc}}$ as follows. $\mathcal{B}(\mathsf{pk})$ runs $(\mathsf{msg},\sigma)\leftarrow\mathcal{A}(\mathsf{pk})$ and then FAEST.Verify$(\mathsf{msg},\mathsf{pk},\sigma)$, storing intermediate results of the verification computation as

needed for the following. For each prover message of the 7-challenge IP, we will now describe how the data contained in it (and thus in a $\mathsf{FAEST}_r^{\mathrm{elc}}$ proof) that are not part of a FAEST_r signature are generated. For pmsg_{-1}, use the values $h_{i,j}$ hashed in Line 36 of BAVC.Reconstruct (as called by VOLEReconstruct, as called by FAEST.Verify) in place of $\mathsf{com}_{i,j}$. For pmsg_i, $i = 1, 2, 3$, use the hash input of lines 10, 15, and 18 of FAEST.Verify, respectively. By contsruction, if FAEST.Verify accepts, then the resulting proof is accepted by $\mathsf{FAEST}_r^{\mathrm{elc}}$ verification. We have thus shown that

$$\mathsf{AdvSnd}_{\mathcal{A}}^{\mathsf{FAEST}} = \mathsf{AdvSnd}_{\mathcal{B}}^{\mathsf{FAEST}_r^{\mathrm{elc}}}.$$

For FAEST-EM, the same argument shows that

$$\mathsf{AdvSnd}_{\mathcal{A}}^{\mathsf{FAEST\text{-}EM}} = \mathsf{AdvSnd}_{\mathcal{B}}^{\widetilde{\mathsf{FAEST\text{-}EM}}_r^{\mathrm{elc}}},$$

where $\widetilde{\mathsf{FAEST\text{-}EM}}_r^{\mathrm{elc}}$ is identical to $\mathsf{FAEST\text{-}EM}_r^{\mathrm{elc}}$ except that the verifier does not reject when one of the $\mathsf{com}_{i,j}$ has multiple pre-images. We now construct adversary $\mathcal{C}$ against the almost injectivity game from Definition 9.13 as follows. $\mathcal{C}$ runs $\mathcal{A}$ while simulating oracles locally. It then outputs the list of $\mathsf{com}_{i,j}$ submitted by $\mathcal{A}$, together with the challenge iv output by H_4 when queried on $\mathsf{iv}^{\mathsf{pre}}$. We thus have

$$\mathsf{AdvSnd}_{\mathcal{B}}^{\widetilde{\mathsf{FAEST\text{-}EM}}_r^{\mathrm{elc}}} - \mathsf{AdvSnd}_{\mathcal{B}}^{\mathsf{FAEST\text{-}EM}_r^{\mathrm{elc}}} \leq \mathsf{AdvOivInj}_{\mathcal{C}}^{\mathsf{PRG}_{2\lambda}}[L].$$

The claim then follows by applying Equation (5) proven in the proof of Theorem 9.24, which holds for any computational model. □

9.7.5 Quantum-guess unpredictability of VOLECommit. We start by proving a slight quantum generalization of Lemma 9.21.

Lemma 9.41 (VOLECommit is quantum-guess unpredictable). *Fix some $\hat{\ell} \in \mathbb{N}$ and admissible $\mathtt{param}$, $\mathtt{param}_{\mathsf{VOLE}}$. Let $\mathcal{A}$ be an adversary in the following game $\mathsf{ExpUnpred}_{b,\mathcal{A}}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}}$*

1. $(L, \mathsf{state}) \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1}(1^\lambda)$, *where G is a quantum register with the right number of qubits to hold a (superposition of) list(s) $\mathcal{L}_{\mathsf{com}}$ of commitment guesses.*
2. $r \leftarrow \{0,1\}^\lambda$, $\mathsf{iv} \leftarrow \{0,1\}^{128}$
3. $C = (\mathsf{com}, \mathsf{decom}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{u}, \mathbf{V}) \leftarrow \mathsf{VOLECommit}(r, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$
4. *Set $b' = 1$. If $b = 1$, apply the binary measurement to L that tests membership of* com *in the list $\mathcal{L}_{\mathsf{com}}$ the register L contains (in superposition). If yes (output 1), set $b' = 0$.*
5. *If $b' = 1$, set $b' \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1}(\mathsf{state}, L, C)$.*
6. *Output b'.*

For brevity, let G_b denote $\mathsf{ExpQUnpred}_{b,\mathcal{A}}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}}$ for $b \in \{0,1\}$. Also let $\beta = \frac{\lambda}{128} \in \mathbb{N}$. For any adversary $\mathcal{A}$ which makes at most Q_1 queries to H_1, and outputs lists $\mathcal{L}$ of size at most Q_{com} we have

$$\begin{aligned}\Pr[\mathsf{G}_0 = 1] \leq & e^{(4\lceil\log(L)\rceil\beta^2+16\beta^2)/2^{128}} \cdot \Big(\Pr[\mathsf{G}_1 = 1] \\ & + \lceil\log(L)\rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1] \\ & + \left(3\sqrt{Q_1} + 2\sqrt{Q_{\mathsf{com}}}\right) \cdot 2^{-\lambda} + Q_{\mathsf{com}} \cdot 2^{-2\lambda}\Big)\end{aligned}$$

where by abuse of notation, we write $\mathsf{AdvPRP}^{\mathsf{LeafCommit}}$ for $\mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}$ in FAEST*, and $\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}$ in* FAEST-EM*.*

Proof. The proof proceeds identically to the proof of Lemma 9.21, until Equation (4). Now, define G_2'' where h_0 is sampled independently uniformly random. By Proposition 1 in [GHHM21] we have

$$|\Pr[\mathsf{G}_2' = 1] - \Pr[\mathsf{G}_2'' = 1]| \leq \frac{3}{2}\sqrt{Q_1}2^{-\lambda}$$

Define G_2''' where additionally, com is sampled uniformly at random. Similarly we get

$$|\Pr[\mathsf{G}_2'' = 1] - \Pr[\mathsf{G}_2''' = 1]| \leq \frac{3}{2}\sqrt{Q_1}2^{-\lambda}$$

Finally, we need to add the the membership test measurement and analyze the probability that it returns 1. Define G_3' like G_2''' except that the membership test measurement from step 4. is applied (but the output is not used). Let ε be the probability that the measurement outputs 1. Then by a gentle measurement lemma we get

$$|\Pr[\mathsf{G}_3' = 1] - \Pr[\mathsf{G}_2''' = 1]| \leq 2\sqrt{\varepsilon}$$

Finally we observe that the outputs of G_3' and G_0 are identical except if the measurement in Line 4 returns 1. We thus get

$$|\Pr[\mathsf{G}_0 = 1] - \Pr[\mathsf{G}_3' = 1]| \leq \varepsilon$$

To bound ε, let $L = L_0 L_1 \dots L_{Q_{\mathsf{com}}-1}$, where each subregister L_i contains a commitment guess. We observe that for fixed com, the projector corresponding to the 1 outcome of the measurement in step 4 is

$$P_{1,\mathsf{com}} = \sum_{i=0}^{Q_{\mathsf{com}}-1} |\mathsf{com}\rangle\langle\mathsf{com}|_{L_i} \otimes \mathbb{1}_{L_{i^c}},$$

where $L_{i^c} = L_0 L_1 \dots L_{i-1} L_{i+1} \dots L_{Q_{\mathsf{com}}-1}$. Taking the expectation over a uniformly random com we get

$$\mathbb{E}_{\mathsf{com}\leftarrow\{0,1\}^{2\lambda}} P_{1,\mathsf{com}} = \frac{Q_{\mathsf{com}}}{2^{2\lambda}}\mathbb{1}.$$

It follows that $\varepsilon = \frac{Q_{\mathsf{com}}}{2^{2\lambda}}$. Collecting all the terms yields the claimed bound. □

9.7.6 FAEST EUF-CMA ⇐ FAEST EUF-KO

Theorem 9.42 (FAEST is EUF-CMA secure). *Let $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$ and H_4 be modeled as quantum-accessible random oracles and H_3 be a PRF. Then for any QPT adversary $\mathcal{A}$ which makes Q_{sig} queries to $\mathcal{O}_{\mathsf{sig}}$ and $Q_0, Q_1, Q_{2,1}, Q_{2,2}$ and $Q_{2,3}$ queries to $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^1, \mathsf{H}_2^2$ and H_2^3 respectively,*

$$\begin{aligned}\mathsf{AdvEUFCMA}^{\mathsf{FAEST}}_{\mathcal{A}}[Q_{\mathsf{sig}}] \leq\ & 2 \cdot \mathsf{AdvEUFKO}^{\mathsf{FAEST}} \\ & + 2 \cdot Q_{\mathsf{sig}} \cdot \left(\mathsf{AdvPRF}^{\mathsf{H}_3} + \lceil \log(L) \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1]\right) \\ & + 2 \cdot Q_{\mathsf{sig}} \cdot \left(\left(3\sqrt{Q_1} + 2\sqrt{Q_{2,1}}\right) \cdot 2^{-\lambda} + Q_{2,1} \cdot 2^{-2\lambda}\right. \\ & \left. + \frac{3}{2} \cdot \sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{5\lambda+64}}} + \frac{3}{2}\sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}}}\right),\end{aligned}$$

where again $\mathsf{AdvPRP}^{\mathsf{LeafCommit}}$ for $\mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}$ in FAEST, and $\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}$ in FAEST-EM.

Proof. We follow the same sequence of game hops as in the proof of Theorem 9.36, and only prove bounds for game hops where they differ from the classical ones beyond advantage terms now being defined with respect to quantum algorithms.

G_3: In G_3, the quantum-accessible random oracle is simulated using the compressed oracle technique [Zha19]. Upon a query to $\mathcal{O}_{\mathsf{sig}}$, $\mathsf{chall}_1 \in \{0,1\}^{5\lambda+64}$ is sampled at random. When $\mathcal{O}_{\mathsf{sig}}$ is about to query H_2^1 on $\mu\|\mathsf{com}\|\boldsymbol{c}_1\|\ldots\|\boldsymbol{c}_{\tau-1}\|\mathsf{iv}$, a two-outcome measurement is applied to the compressed oracle database to test whether it contains an entry with this input. If yes, abort, else program H_2^1 to output chall_1 on this input.
We argue indistinguishability by reduction to unpredictability as stated in Lemma 9.41. We use a hybrid argument, where $\mathsf{G}_{3,i}$ implements measurement and abort only for the first i queries to $\mathcal{O}_{\mathsf{sig}}$. Clearly, $\mathsf{G}_{3,0} = \mathsf{G}_2$ and $\mathsf{G}_{3,Q_{\mathsf{sig}}} = \mathsf{G}_3$. Moreover, $\mathsf{G}_{3,i-1}$ and $\mathsf{G}_{3,i}$ only differ in the i-th $\mathcal{O}_{\mathsf{sig}}$ query. The i-th reduction $\mathcal{A}_{\mathsf{unpred},i}$ to quantum-list unpredictability plays unpredictability game in Lemma 9.41 such that it emulates $\mathsf{G}_{3,i}$, except that for the i-th $\mathcal{O}_{\mathsf{sig}}$-query, it outputs its current state and the subregisters of all compressed oracle database input registers corresponding to the commitments. It then receives $(\mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1})$ from the unpredictability experiment (instead of running VOLECommit), and then continues from there. By construction, $\mathsf{G}_{3,i-1} = 1 \iff \mathsf{ExpQUnpred}_{0,\mathcal{A}_{\mathsf{unpred}}}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}} = 1$ and $\mathsf{G}_{3,i} = 1 \iff \mathsf{ExpQUnpred}_{1,\mathcal{A}_{\mathsf{unpred},i}}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}}$. Thus, from Lemma 9.41 and a hybrid argument we get

$$\begin{aligned}\Pr[\mathsf{G}_3 = 1] \leq & e^{Q_{\mathsf{sig}}\cdot(4\lceil\log(L)\rceil\beta^2+16\beta^2)/2^{128}} \cdot \Big(\Pr[\mathsf{G}_1 = 1] \\ & + Q_{\mathsf{sig}} \cdot \Big(\lceil\log(L)\rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1] \\ & + \left(3\sqrt{Q_1} + 2\sqrt{Q_{2,1}}\right)\cdot 2^{-\lambda} + Q_{2,1}\cdot 2^{-2\lambda}\Big)\Big)\end{aligned} \tag{14}$$

G_4&G_5: For G_4 and G_5, instead of adding an abort condition, we just reprogram H_2^2 repsectively H_2^3 to the randomly sampled values for chall_2 and chall_3. By the adaptive reprogramming lemma (Proposition 1 in [GHHM21]) we get

$$\Pr[\mathsf{G}_3 = 1] \leq \Pr[\mathsf{G}_4 = 1] + \frac{3}{2}\cdot Q_{\mathsf{sig}} \cdot \sqrt{\frac{Q_{\mathsf{sig}}+Q_{2,2}}{2^{5\lambda+64}}}$$

and

$$\Pr[\mathsf{G}_4 = 1] \leq \Pr[\mathsf{G}_5 = 1] + \frac{3}{2} Q_{\mathsf{sig}} \cdot \sqrt{\frac{Q_{\mathsf{sig}}+Q_{2,3}}{2^{3\lambda+64}}}.$$

Collecting all the terms, including from the game hops that are unchanged compared to Theorem 9.36, finishes the proof. □

We can now combine the above to get the EUF-CMA security of FAEST in the QROM.

Corollary 9.43 (FAEST is EUF-CMA secure in the QROM). *Let* $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$ *and* H_4 *be modeled as quantum-accessible random oracles and* H_3 *be a PRF. Then for any QPT adversary* $\mathcal{A}$ *which makes* Q_{sig} *queries to* $\mathcal{O}_{\mathsf{sig}}$ *and* $Q_0, Q_1, Q_{2,1}, Q_{2,2}$ *and* $Q_{2,3}$ *queries to* $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^1, \mathsf{H}_2^2$ *and* H_2^3 *respectively,*

$$\begin{aligned}\mathsf{AdvEUFCMA}_{\mathcal{A}}^{\mathsf{FAEST}}[Q_{\mathsf{sig}}] \leq & \ 2\cdot\mathsf{AdvEUFKO}_{\mathcal{A}}^{\mathsf{FAEST}} \\ & + 2\cdot Q_{\mathsf{sig}}\cdot\left(\mathsf{AdvPRF}^{\mathsf{H}_3} + \lceil\log(L)\rceil\cdot\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1]\right) \\ & + 2\cdot Q_{\mathsf{sig}}\cdot\left(\left(3\sqrt{Q_1}+2\sqrt{Q_{2,1}}\right)\cdot 2^{-\lambda} + Q_{2,1}\cdot 2^{-2\lambda}\right) \\ & + 3\cdot Q_{\mathsf{sig}}\cdot\left(\sqrt{\frac{Q_{\mathsf{sig}}+Q_{2,2}}{2^{5\lambda+64}}} + \sqrt{\frac{Q_{\mathsf{sig}}+Q_{2,3}}{2^{3\lambda+64}}}.\right)\end{aligned}$$

with

$$\mathsf{AdvEUFKO}^{\mathsf{FAEST}}_{\mathcal{A}} \leq \frac{1}{1-\frac{\mathsf{dom(OWF)}}{\mathsf{cod(OWF)}}} \cdot \left(10 \cdot Q^3 \cdot 2^{-2\lambda} + 10 \cdot Q^2 \cdot \tau \cdot 2^{-\lambda} + \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1}\right),$$

for FAEST.

For FAEST-EM*, there exists a QROM adversary against the almost injectivity of* PRG *such that*

$$\begin{aligned}\mathsf{AdvEUFKO}^{\mathsf{FAEST\text{-}EM}}_{\mathcal{A}} \leq \frac{1}{1-\frac{\mathsf{dom(OWF)}}{\mathsf{cod(OWF)}}} \cdot \big(&10 \cdot Q^3 \cdot 2^{-2\lambda} + 10 \cdot Q^2 \cdot 2^{-\lambda} \\ &+ \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1} + \mathsf{AdvOivInj}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{C}}[L]\big).\end{aligned}$$

Here,

$$Q = Q_0 + Q_1 + Q_{2,0} + Q_{2,1} + Q_{2,2} + Q_{2,3} + Q_4 + 3\tau + 19.$$

Remark 9.44. We have chosen to instantiate all random oracles via domain separation to facilitate the proof of Lemma 9.39 in its relatively simple form. A tighter bound differentiating between the query number of the different random oracles instead of upper-bounding by the sum can be obtained by proving a straightforward, albeit tedious, generalization of Lemma 9.38 to multiple distinct quantum-accessible random oracles and applying it to improve Lemma 9.39.

10 Advantages and Limitations

10.1 Advantages

Minimal security assumptions. FAEST uses only symmetric primitives, with security relying on the already standard assumptions about the one-wayness and pseudorandomness of AES, and collision-resistance and random oracle-like properties of the SHA3 hash function family. In particular, FAEST does not need any structured or novel assumptions, and is fairly straightforward to analyze against concrete attacks.

Good, general-purpose performance. Overall, FAEST has good performance across public key and signature sizes, signing speed and verification speed. This makes it a strong candidate for general-purpose use, for instance, in real-time protocols like TLS as well as more static use-cases like code signing. Compared with hash-based signatures like SPHINCS+, based on similarly conservative assumptions, FAEST enjoys much faster signing and smaller signatures.

Small (key+signature) sizes. FAEST has very small keys, with secret keys of size 16–32 bytes and public keys 32–64 bytes. This makes the *combined size* of a public key and signature fairly small, for example, 3938 bytes for FAEST-EM-128s, very close to the 3732 bytes of the lattice-based scheme ML-DSA-44 (Dilithium2). This metric is particularly important to optimize in applications like certificates.

Modularity. FAEST uses a modular design with several independent building blocks. The PRGs or hash functions can easily be swapped out with alternatives that may improve performance, or security in case of an unexpected weakness in one of the primitives. Moreover, using one-way functions other than AES in the zero-knowledge proof system can lead to different tradeoffs in terms of performance and assumptions.

Performance trade-offs. FAEST provides a large amount of flexibility in terms of different parameters settings. While this document only specifies two parameter settings for each security level, further choices are possible that give a larger range of performance tradeoffs between signature size and signing/verification speed.

Security proof. FAEST has a security proof in the ROM and the QROM, via a reduction to the security of the underlying primitives. Since both proofs are almost tight, except for a multiplicative loss in the number of signing queries, this provides strong evidence of security for FAEST.

10.2 Limitations

Verification speed. Despite the overall good performance, FAEST has slightly slower verification compared with SPHINCS+, which may make it less desirable in applications where verification is performed much more often than signing. However, this is less significant compared with the improvements in both signing time and signature size.

Signature sizes. While FAEST signatures are very small amongst signature schemes based on symmetric primitives, they are still somewhat larger than lattice-based signature schemes like Dilithium and FALCON. This may make it less suitable in a setting where transmitting signatures is the bottleneck in a network.

References

ABKM22. Gorjan Alagic, Chen Bai, Jonathan Katz, and Christian Majenz. Post-quantum security of the Even-Mansour cipher. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 458–487. Springer, Cham, May / June 2022.

AES01. Advanced Encryption Standard (AES). National Institute of Standards and Technology, NIST FIPS PUB 197, U.S. Department of Commerce, November 2001.

AHJ+23. Carlos Aguilar Melchor, Andreas Hülsing, David Joseph, Christian Majenz, Eyal Ronen, and Dongze Yue. SDitH in the QROM. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VII*, volume 14444 of *LNCS*, pages 317–350. Springer, Singapore, December 2023.

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

BBD+23b. Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from vole-in-the-head. In *CRYPTO*. Springer, 2023.

BBM+24. Carsten Baum, Ward Beullens, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. One tree to rule them all: Optimizing GGM trees and OWFs for post-quantum signatures. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part I*, volume 15484 of *LNCS*, pages 463–493. Springer, Singapore, December 2024.

BCG+19. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 291–308. ACM Press, November 2019.

BCGI18. Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912. ACM Press, October 2018.

BCS16. Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 31–60. Springer, Berlin, Heidelberg, October / November 2016.

BDF11. Charles Bouillaguet, Patrick Derbez, and Pierre-Alain Fouque. Automatic search of attacks on round-reduced AES and applications. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 169–187. Springer, Berlin, Heidelberg, August 2011.

BDK+21. Carsten Baum, Cyprien Delpech de Saint Guilhem, Daniel Kales, Emmanuela Orsini, Peter Scholl, and Greg Zaverucha. Banquet: Short and fast signatures from AES. In Juan Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 266–297. Springer, Cham, May 2021.

BGI14. Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Berlin, Heidelberg, March 2014.

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

BKR11. Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full AES. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 344–371. Springer, Berlin, Heidelberg, December 2011.

BMRS21. Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 2021. Springer, Cham.

BR19a. Navid Ghaedi Bardeh and Sondre Rønjom. The exchange attack: How to distinguish six rounds of AES with $2^{88.2}$ chosen plaintexts. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part III*, volume 11923 of *LNCS*, pages 347–370. Springer, Cham, December 2019.

BR19b. Navid Ghaedi Bardeh and Sondre Rønjom. Practical attacks on reduced-round AES. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi, editors, *AFRICACRYPT 19*, volume 11627 of *LNCS*, pages 297–310. Springer, Cham, July 2019.

BR22. Navid Ghaedi Bardeh and Vincent Rijmen. New key-recovery attack on reduced-round AES. *IACR Trans. Symm. Cryptol.*, 2022(2):43–62, 2022.

BW13. Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Berlin, Heidelberg, December 2013.

CCH+18. Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, and Ron D. Rothblum. Fiat-Shamir from simpler assumptions. Cryptology ePrint Archive, Report 2018/1004, 2018.

CDG+17. Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, October / November 2017.

CFHL21. Kai-Min Chung, Serge Fehr, Yu-Hsuan Huang, and Tai-Ning Liao. On the compressed-oracle technique, and post-quantum security of proofs of sequential work. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 598–629. Springer, Cham, October 2021.

CW79. Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

CY24. Alessandro Chiesa and Eylon Yogev. *Building Cryptographic Proofs from Hash Functions*. self-published, 2024.

DFJ13. Patrick Derbez, Pierre-Alain Fouque, and Jérémy Jean. Improved key recovery attacks on reduced-round AES in the single-key setting. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 371–387. Springer, Berlin, Heidelberg, May 2013.

DIO21. Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

DKR+22. Christoph Dobraunig, Daniel Kales, Christian Rechberger, Markus Schofnegger, and Greg Zaverucha. Shorter signatures based on tailor-made minimalist symmetric-key crypto. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 843–857. ACM Press, November 2022.

DKS12. Orr Dunkelman, Nathan Keller, and Adi Shamir. Minimalism in cryptography: The Even-Mansour scheme revisited. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 336–354. Springer, Berlin, Heidelberg, April 2012.

DN19. Itai Dinur and Niv Nadler. Multi-target attacks on the Picnic signature scheme and related protocols. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 699–727. Springer, Cham, May 2019.

DR02. Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

EM97. Shimon Even and Yishay Mansour. A construction of a cipher from a single pseudorandom permutation. *Journal of Cryptology*, 10(3):151–162, June 1997.

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

GGM84. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th FOCS*, pages 464–479. IEEE Computer Society Press, October 1984.

GHHM21. Alex B. Grilo, Kathrin Hövelmanns, Andreas Hülsing, and Christian Majenz. Tight adaptive reprogramming in the QROM. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part I*, volume 13090 of *LNCS*, pages 637–667. Springer, Cham, December 2021.

GLNP15. Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 567–578. ACM Press, October 2015.

GRR17. Lorenzo Grassi, Christian Rechberger, and Sondre Rønjom. A new structural-differential property of 5-round AES. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 289–317. Springer, Cham, April / May 2017.

IKNP03. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Berlin, Heidelberg, August 2003.

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

KPTZ13. Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013.

KW03. Jonathan Katz and Nan Wang. Efficiency improvements for signature schemes with tight security reductions. In Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger, editors, *ACM CCS 2003*, pages 155–164. ACM Press, October 2003.

LLH20. Zhengrui Li, Sian-Jheng Lin, and Yunghsiang S. Han. On the exact lower bounds of encoding circuit sizes of hamming codes and hadamard codes. In *2020 IEEE International Symposium on Information Theory (ISIT)*, pages 2831–2836, 2020.

MRZ15. Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 591–602. ACM Press, October 2015.

MS04. Alfred Menezes and Nigel Smart. Security of signature schemes in a multi-user setting. *Designs, Codes and Cryptography*, 33(3):261–274, 2004.

Nao91. Moni Naor. Bit commitment using pseudorandomness. *Journal of Cryptology*, 4(2):151–158, January 1991.

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

RRR16. Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. In Daniel Wichs and Yishay Mansour, editors, *48th ACM STOC*, pages 49–62. ACM Press, June 2016.

Ser98. Gadiel Seroussi. Table of low-weight binary irreducible polynomials. Technical Report HPL-98-135, Hewlett Packard Laboratories, 1998. https://www.hpl.hp.com/techreports/98/HPL-98-135.pdf.

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

TE76. R. E. Twogood and M. P. Ekstrom. An extension of Eklundh's matrix transposition algorithm and its application in digital image processing. *IEEE Transactions on Computers*, C-25(9):950–952, 1976.

WYKW21. Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091. IEEE Computer Society Press, May 2021.

YSWW21. Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021.

ZCD+20. Greg Zaverucha, Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Jonathan Katz, Xiao Wang, Vladmir Kolesnikov, and Daniel Kales. Picnic. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

Zha19. Mark Zhandry. How to record quantum queries, and applications to quantum indifferentiability. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part II*, volume 11693 of *LNCS*, pages 239–268. Springer, Cham, August 2019.

A Details on Finite Fields

A.1 Finite Field Generator Elements

Below, we specify the generators of $\mathbb{F}_{2^8}$ we use when lifting to each of the fields $\mathbb{F}_{2^{128}}, \mathbb{F}_{2^{192}}$ and $\mathbb{F}_{2^{256}}$ using ByteCombine. We first give the hexadecimal representation in little-endian order, followed by the human-readable polynomial. The generators were obtained using SageMath.

$\mathbb{F}_{2^{128}}$:

```
{0x0d, 0xce, 0x60, 0x55, 0xac, 0xe8, 0x3f, 0xa1,
0x1c, 0x9a, 0x97, 0xa9, 0x55, 0x85, 0x3d, 0x05}
```

$$z^{122} + z^{120} + z^{117} + z^{116} + z^{115} + z^{114} + z^{112} + z^{111} + z^{106} + z^{104} + z^{102} + z^{100} + z^{98} + z^{96} + z^{95} + z^{93} + z^{91} + z^{88} + z^{87} + z^{84} + z^{82} + z^{81} + z^{80} + z^{79} + z^{76} + z^{75} + z^{73} + z^{68} + z^{67} + z^{66} + z^{63} + z^{61} + z^{56} + z^{53} + z^{52} + z^{51} + z^{50} + z^{49} + z^{48} + z^{47} + z^{46} + z^{45} + z^{43} + z^{39} + z^{37} + z^{35} + z^{34} + z^{30} + z^{28} + z^{26} + z^{24} + z^{22} + z^{21} + z^{15} + z^{14} + z^{11} + z^{10} + z^{9} + z^{3} + z^{2} + 1$$

$\mathbb{F}_{2^{192}}$:

```
{0x63, 0x97, 0x38, 0x6f, 0xd5, 0xa3, 0xc8, 0xcc,
0xea, 0xbd, 0x6e, 0x96, 0x6c, 0xd7, 0x65, 0xe6,
0x62, 0x36, 0x6b, 0x0e, 0x14, 0xc8, 0x0b, 0x31}
```

$$z^{189} + z^{188} + z^{184} + z^{179} + z^{177} + z^{176} + z^{175} + z^{174} + z^{171} + z^{164} + z^{162} + z^{155} + z^{154} + z^{153} + z^{150} + z^{149} + z^{147} + z^{145} + z^{144} + z^{141} + z^{140} + z^{138} + z^{137} + z^{134} + z^{133} + z^{129} + z^{127} + z^{126} + z^{125} + z^{122} + z^{121} + z^{118} + z^{117} + z^{114} + z^{112} + z^{111} + z^{110} + z^{108} + z^{106} + z^{105} + z^{104} + z^{102} + z^{101} + z^{99} + z^{98} + z^{95} + z^{92} + z^{90} + z^{89} + z^{86} + z^{85} + z^{83} + z^{82} + z^{81} + z^{79} + z^{77} + z^{76} + z^{75} + z^{74} + z^{72} + z^{71} + z^{70} + z^{69} + z^{67} + z^{65} + z^{63} + z^{62} + z^{59} + z^{58} + z^{55} + z^{54} + z^{51} + z^{47} + z^{45} + z^{41} + z^{40} + z^{39} + z^{38} + z^{36} + z^{34} + z^{32} + z^{30} + z^{29} + z^{27} + z^{26} + z^{25} + z^{24} + z^{21} + z^{20} + z^{19} + z^{15} + z^{12} + z^{10} + z^{9} + z^{8} + z^{6} + z^{5} + z + 1$$

$\mathbb{F}_{2^{256}}$:

```
{0xe7, 0xfe, 0xde, 0x0b, 0x42, 0x88, 0x97, 0x96,
0x67, 0x4e, 0x47, 0xa0, 0x38, 0x8d, 0xd6, 0xbe,
0x6a, 0xe1, 0xf1, 0xf8, 0x45, 0x98, 0x22, 0xdf,
0x33, 0x58, 0xc9, 0x20, 0xcf, 0xa8, 0xc9, 0x04}
```

$$z^{250} + z^{247} + z^{246} + z^{243} + z^{240} + z^{239} + z^{237} + z^{235} + z^{231} + z^{230} + z^{227} + z^{226} + z^{225} + z^{224} + z^{221} + z^{215} + z^{214} + z^{211} + z^{208} + z^{206} + z^{204} + z^{203} + z^{197} + z^{196} + z^{193} + z^{192} + z^{191} + z^{190} + z^{188} + z^{187} + z^{186} + z^{185} + z^{184} + z^{181} + z^{177} + z^{175} + z^{172} + z^{171} + z^{166} + z^{162} + z^{160} + z^{159} + z^{158} + z^{157} + z^{156} + z^{155} + z^{151} + z^{150} + z^{149} + z^{148} + z^{144} + z^{143} + z^{142} + z^{141} + z^{136} + z^{134} + z^{133} + z^{131} + z^{129} + z^{127} + z^{125} + z^{124} + z^{123} + z^{122} + z^{121} + z^{119} + z^{118} + z^{116} + z^{114} + z^{113} + z^{111} + z^{107} + z^{106} + z^{104} + z^{101} + z^{100} + z^{99} + z^{95} + z^{93} + z^{86} + z^{82} + z^{81} + z^{80} + z^{78} + z^{75} + z^{74} + z^{73} + z^{70} + z^{69} + z^{66} + z^{65} + z^{64} + z^{63} + z^{60} + z^{58} + z^{57} + z^{55} + z^{52} + z^{50} + z^{49} + z^{48} + z^{47} + z^{43} + z^{38} + z^{33} + z^{27} + z^{25} + z^{24} + z^{23} + z^{22} + z^{20} + z^{19} + z^{18} + z^{17} + z^{15} + z^{14} + z^{13} + z^{12} + z^{11} + z^{10} + z^{9} + z^{7} + z^{6} + z^{5} + z^{2} + z + 1$$

A.2 Affine Layer of S-box as a Function of the Conjugates

For each choice of $\lambda \in \{128, 192, 256\}$, we define the coefficients $\zeta_i \in \mathbb{F}_{2^\lambda}$ using the $\mathbb{F}_{2^8}$ embedding element $\alpha_8 \in \mathbb{F}_{2^\lambda}$, as follows:

$$\begin{aligned}
\zeta_0 &= \alpha_8^2 + 1 \\
\zeta_1 &= \alpha_8^3 + 1 \\
\zeta_2 &= \alpha_8^7 + \alpha_8^6 + \alpha_8^5 + \alpha_8^4 + \alpha_8^3 + 1 \\
\zeta_3 &= \alpha_8^5 + \alpha_8^2 + 1 \\
\zeta_4 &= \alpha_8^7 + \alpha_8^6 + \alpha_8^5 + \alpha_8^4 + \alpha_8^2 \\
\zeta_5 &= 1 \\
\zeta_6 &= \alpha_8^7 + \alpha_8^5 + \alpha_8^4 + \alpha_8^2 + 1 \\
\zeta_7 &= \alpha_8^7 + \alpha_8^3 + \alpha_8^2 + \alpha_8 + 1 \\
\zeta_8 &= \alpha_8^6 + \alpha_8^5 + \alpha_8 + 1
\end{aligned}$$

The affine component of the S-box on an input $x \in \mathbb{F}_{2^8}$, embedded in $\mathbb{F}_{2^\lambda}$, can then be computed as $\sum_{i=0}^{7} \zeta_i x^{2^i} + \zeta_8$.