FAEST v3: Algorithm Specifications

Version 3.0

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

[1] Technical University of Denmark
[2] IBM Research Europe
[3] Université Paris Cité, CNRS, IRIF
[4] 3MI Labs
[5] Karlsruhe Institute of Technology
[6] Graz University of Technology
[7] Know Center
[8] Bocconi University
[9] AIT Austrian Institute of Technology
[10] Aarhus University

September 1, 2026

Table of Contents

1 Introduction

This document describes and specifies version 3 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+23] 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-7 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 how these components are used to generate the VOLE correlations, using a CRT-based approach to create masks. Section 7 specifies the computation of the zero-knowledge proof for the AES and Rijndael circuits used in FAEST and FAEST-EM. Section 8 specifies the key generation, signing and verification algorithms.

The final part of this document analyses the FAEST signature algorithm. Section 9 provides the performance results of the implementations. Section 10 provides both a formal proof as well as an analysis of the concrete attacks that are relevant to the building blocks of FAEST. Section 11 discusses the advantages and limitations provided by the algorithm.

1.1 Changelog

FAEST version 3 incorporates several tweaks to improve performance and security:

- A new method for committing to λ-bit VOLE masks used to mask the QuickSilver check based on the Chinese remainder theorem. This reduces the signature size penalty from committing to the masks.
- To take advantage of the above, we now prove AES with a set of degree-7 constraints and a meet-in-the-middle strategy, compared with the previous (more complex) degree-3 constraints based on finite field norms. This leads to a smaller witness, reducing signatures sizes.
- The new VOLE mask commitment requires a different VOLEHash check, which now computes 4 linear combinations over $\mathbb{F}_{2^\lambda}$. The analysis of this check requires a new security proof for the KOS OT extension consistency check [KOS15].
- The IV is now derived as $\mathsf{iv} = \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu)$ instead of $\mathsf{iv} = \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}})$, where μ is the hash of the message and public key. This avoids a possible multi-target attack on the GGM tree leaf commitments.

– The first Fiat-Shamir hash function takes $\mathsf{iv}^{\mathsf{pre}}$ as input, not iv. The previous version of the spec was inconsistent on this aspect, but using $\mathsf{iv}^{\mathsf{pre}}$ prevents malleability attacks based on collisions in the 128-bit IV, giving conjectured strong unforgeability.
– The $(\tau, w_{\mathsf{grind}}, T_{\mathsf{open}})$ parameters defining each instance have been tuned to further optimize signature sizes and verification time.
– The architecture-specific implementation now also supports AArch64.

We have additionally updated both the classical and quantum security proofs to reflect the changes, and fixed a number of minor bugs in the previous proofs.

Acknowledgements. We used AI tools (Claude Opus 5 and Fable 5, and GPT-5.6) as an aid for writing parts of this document, and for helping to develop, check and debug proofs and implementations. All content was verified by the authors, who take full responsibility for its correctness.

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+23]. 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.

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.

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

Generating VOLE Commitments. Instead of directly generating VOLE correlations over $\mathbb{F}_{2^\lambda}$, VOLEitH first generates τ VOLE instances over smaller fields, where τ is a tunable parameter giving a tradeoff between signature size and speed. We use τ_1 small VOLE instances of bit length k and $\tau_0 = \tau - \tau_1$ instances of bit length $k-1$, where the parameters are chosen such that

$$\tau_1 k + \tau_0(k-1) = \lambda - w_{\mathsf{grind}},$$

for a small grinding parameter w_{grind} (see 'Challenge Grinding' below). 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 combined to build a VOLE correlation with a $(\lambda - w_{\mathsf{grind}})$-bit key, which is later lifted to $\mathbb{F}_{2^\lambda}$.

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 partitioning the leaves into τ consecutive blocks of size N, so that the j-th entry of the i-th committed vector is the leaf at index $i \cdot N + j$.

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 reduce the security of FAEST. Challenges are still sampled from the same domain as before, but some challenges are now always rejected, which makes forgery only more difficult. 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: Challenge Grinding. In addition to rejecting challenges with large openings, FAEST requires the last w_{grind} bits of the final Fiat–Shamir challenge to be zero; the signer repeatedly increments a counter and rehashes until this holds. Since any forger must perform the same $2^{w_{\mathsf{grind}}}$ expected hash evaluations per attempt, this proof-of-work allows reducing the number of challenge bits that determine the VOLE keys Δ_i from λ to $\lambda - w_{\mathsf{grind}}$, which lowers the cost of the GGM trees without changing security.

Optimization 3: 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 10.5.

[13] In our security proofs, we actually rely on a slight strengthening of collision resistance, where it must be hard to even find a PRG image for which a collision exists, without necessarily knowing the colliding preimages; see Definition 10.13 in Section 10.

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 $\mathbb{F}_{2^k}$,

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

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

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

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

Once this has been done, the parties concatenate the τ VOLE instances, forming the $\ell \times (\lambda - w_{\mathsf{grind}})$ matrices

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

while the bits of $\Delta_0, \ldots, \Delta_{\tau-1}$ are concatenated into $\Delta \in \mathbb{F}_2^{\lambda - w_{\mathsf{grind}}}$. Applying a fixed, public linear map (coming from the CRT-based construction described next) to Δ and to each row of $\mathbf{V}$ and $\mathbf{Q}$ lifts these to elements of $\mathbb{F}_{2^\lambda}$, after which we have the VOLE relation $\mathbf{Q}[i] = \mathbf{V}[i] + \mathbf{u}[i] \cdot \Delta$ over $\mathbb{F}_{2^\lambda}$.

More Efficient Generation of VOLE Masks with CRT. The VOLE correlations described so far commit the signer to random *bits* $u_i \in \mathbb{F}_2$. In addition to these, the ZK proof and the VOLE consistency check (described below) require a small number of VOLE *masks*: correlations $\bar{q}_m = \bar{v}_m + \bar{u}_m \cdot \Delta$, where $\bar{u}_m$ is uniform over the whole field $\mathbb{F}_{2^\lambda}$. Previous versions of FAEST assembled each mask bit-by-bit from λ rows of the subfield VOLE, so that each of the λ bits of $\bar{u}_m$ required its own $\tau - 1$ correction bits, adding $(\tau - 1)\lambda$ bits to the signature per mask.

In this version, we instead compute shares of the product $\bar{u}_m \cdot \Delta$ directly, using a multiplication algorithm based on the Chinese remainder theorem (CRT) for polynomials over $\mathbb{F}_2$. Viewing $\bar{u}_m$ and Δ as binary polynomials, it suffices to obtain shares of the products of their residues modulo sufficiently many pairwise coprime, low-degree moduli. The key observation is that the small-field VOLEs already provide many of these residues for free: by representing the field of the i-th small VOLE instance as $\mathbb{F}_2[x]/(M_i(x))$, for distinct irreducible polynomials M_i, the i-th instance directly yields shares of $(\bar{u}_m \bmod M_i) \cdot (\Delta \bmod M_i)$, with the small-VOLE challenge Δ_i now playing the role of $\Delta \bmod M_i$. These τ residues determine only around half of the coefficients of the product, so the remaining residues, taken with a set of additional low-degree moduli, are computed via a small bilinear circuit containing $n_{\mathsf{mult}} \approx 2.5\lambda$ AND gates. Each AND gate is evaluated with a hashing technique inspired by free-XOR garbling [KS08], which reuses the first λ witness rows of the VOLE and requires just one correction bit per mask. Overall, this reduces the cost of each mask from $(\tau-1)\lambda$ correction bits down to n_{mult}, a saving of 4–12× depending on the parameter set. The details of the construction are given in Section 6, with additional details describing precomputed tables to assist with CRT multiplication in Appendix B.

VOLE Consistency Check. Finally, we need a way of ensuring that the signer does not cheat when sending the correction values, either for the witness or for the mask commitments. To prevent this, the verifier challenges the signer to open a random, linear universal hash function applied to the committed values. The hash function, VOLEHash, is linear over $\mathbb{F}_{2^\lambda}$ and is applied to the ℓ committed witness bits (embedded into $\mathbb{F}_{2^\lambda}$), together with the $d_{\mathsf{zk}} - 1$ full-field masks used in the ZK proof. The signer computes the hash $\widetilde{\mathbf{u}}$ of its committed values and the hash $\widetilde{\mathbf{V}}$ of the corresponding VOLE tags, which are then bound into the next Fiat–Shamir challenge. After the VOLEs are opened, the verifier computes the hash $\widetilde{\mathbf{Q}}$ of its keys and checks that the VOLE relation still holds between $\widetilde{\mathbf{u}}, \widetilde{\mathbf{V}}$ and $\widetilde{\mathbf{Q}}$. To preserve zero-knowledge, we incorporate additional mask values into the hash.

Compared with previous versions of FAEST, which used a check over $\mathbb{F}_2$ based on SoftSpokenOT [Roy22], the new check is applied directly to the $\mathbb{F}_{2^\lambda}$ VOLE correlations. This check is now essentially the same as the consistency check in the KOS OT extension protocol [KOS15], however, the original KOS security proof was shown to be broken, and only a very weak security bound was previously known [Dia22]. We show that if the hash output is increased to 4λ bits, we obtain λ-bit security and can extract consistent VOLE commitments over $\mathbb{F}_{2^\lambda}$. One catch is that we cannot ensure the committed witness elements are all *bits*, and we need to add some additional constraints into the ZK proof stage to check that.

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 $\hat{\ell} = \ell + n_{\mathsf{mask}} \cdot d_0$, where the additional $n_{\mathsf{mask}} \cdot d_0$ rows are used to derive $n_{\mathsf{mask}} = d_{\mathsf{zk}} + 3$ masks in $\mathbb{F}_{2^\lambda}$. In the ZK proof, the first ℓ rows of the VOLE are used to commit to the extended witness, and the first $d_{\mathsf{zk}} - 1$ of the extra masks are used to mask the responses of the ZK check. The last four masks are used to mask the 4λ-bit output of the VOLE consistency check.

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 7.6.)

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 $d_{\mathsf{zk}} = 7$. 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_{zk} constraint $\hat{f}$, the prover could just send over the coefficients $(a_0, \ldots, a_{d_{\mathsf{zk}}})$ of its polynomial, at which point the verifier can check that $a_{d_{\mathsf{zk}}} = 0$ 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 $d_{\mathsf{zk}}/2^{\lambda}$, since there are at most d_{zk} possible values of Δ for which the check can pass, each corresponding to a root of the polynomial with coefficients $(a_0 + a'_0, \ldots, a_{d_{\mathsf{zk}}} + a'_{d_{\mathsf{zk}}})$, 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 $2^{-\lambda} + d_{\mathsf{zk}} \cdot 2^{-\lambda}$.

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. The QuickSilver protocol adds one more component to the constraint check in order to guarantee the zero-knowledge property of the protocol. Indeed, revealing $(a_0, \ldots, a_{d_{\mathsf{zk}}})$ to the verifier leaks information about the circuit values used to compute them. To prevent this, the prover uses $d_{\mathsf{zk}} - 1$ of the random field masks $(\bar{u}_m, \bar{v}_m)$ generated in the CRT-based mask generation phase. Each mask is a VOLE correlation whose committed value $\bar{u}_m$ is *uniform* over $\mathbb{F}_{2^\lambda}$, and is used to mask one of the a_i coefficients. Because there are effectively only $d_{\mathsf{zk}} - 1$ unknown degrees of freedom for the verifier (who knows its evaluation point as well as $a_{d_{\mathsf{zk}}} = 0$), $d_{\mathsf{zk}} - 1$ masks suffice 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:

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

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, together with a 128-bit salt and the 32-bit counter used for the rejection sampling of chall_3. 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.

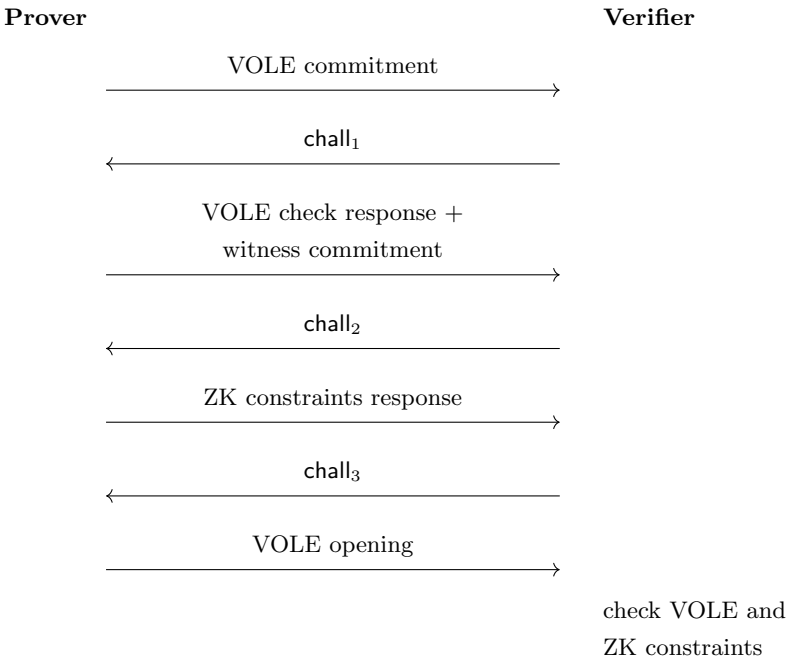

Fig. 2.1: Interactive argument used in FAEST

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:

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

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. All parameter sets use $d_{\mathsf{zk}} = 7$.

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

$$\begin{aligned}\text{size} = \underbrace{\tau \cdot \ell}_{\substack{\text{corrections } \mathbf{c}_i \\ + \text{ witness com. } \mathbf{d}}} + \underbrace{n_{\mathsf{mask}} \cdot \bar{n}_{\mathsf{mult}}}_{\text{gate corrections } \mathbf{c}_{\mathsf{mult}}} + \underbrace{4\lambda}_{\substack{\text{VOLE check} \\ \text{response } \tilde{\mathbf{u}}}} + \underbrace{(d_{\mathsf{zk}} - 1)\lambda}_{\substack{\text{ZK check} \\ \text{response } \tilde{a}_1, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}}} \\ + \underbrace{T_{\mathsf{open}} \cdot \lambda + n_{\mathsf{leafcom}}\lambda\tau}_{\text{GGM tree opening}} + \underbrace{\lambda}_{\mathsf{chall}_3} + \underbrace{128}_{\mathsf{iv}^{\mathsf{pre}}} + \underbrace{32}_{\mathsf{ctr}}\end{aligned}$$

Every term above is a multiple of 8, so the signature is exactly size/8 bytes long. Note that $\mathbf{c}_{\mathsf{mult}}$ explicitly already carries some padding, since $\bar{n}_{\mathsf{mult}}$ pads its row length n_{mult} up to an exact number of bytes.

3.1.1 Additional Parameters

Parameter	Description
λ	Security parameter in $\{128, 192, 256\}$
τ	Number of small-VOLE instances
w_{grind}	Grinding parameter: the w_{grind} upper bits of the challenge chall_3 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
$E_k(x)$	One-way function used to generate the public key
ℓ	Witness length for ZK proof (in bits)
d_{zk}	7; maximal degree of the QuickSilver constraints (Section 7)
Derived from the above:	
n_{mask}	Number of $\mathbb{F}_{2^\lambda}$ VOLE masks, $n_{\mathsf{mask}} := d_{\mathsf{zk}} + 3$
n_{mult}	Number of AND gate corrections in the VOLE mask generation part of $\mathtt{param}_{\mathsf{VOLE}}$, see Section 6.2
$\bar{n}_{\mathsf{mult}}$	Row length of $\mathbf{c}_{\mathsf{mult}}$ in bits, $\bar{n}_{\mathsf{mult}} := 8\lceil n_{\mathsf{mult}}/8 \rceil$
$\hat{\ell}$	Length of each small VOLE, $\hat{\ell} := \ell + n_{\mathsf{mask}} \cdot d_0$

Table 3.1: Main parameters for FAEST and FAEST-EM

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:

$$\mathtt{param}_{\mathsf{VOLE}} := \big(\tau_1, \tau_0, k, L, (N_i)_{i\in[0..\tau)}, n_{\mathsf{mult}}, (M_i)_{i\in[0..\tau)}, M_{\mathrm{tree}}, \mathbf{F}, \mathbf{G}, \mathbf{W}_{\mathrm{tree}}, \mathbf{W}_{\mathrm{gate}}, \mathbf{W}_{\mathrm{crt}}\big).$$

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:

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

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

Scheme	OWF $E_k(x)$	ℓ	τ	w_{grind}	T_{open}	n_{leafcom}	n_{mult}	sizes (bytes)	
								pk	sig.
FAEST-128s	$\mathsf{AES128}_k(x)$	960	11	7	102	3	312	32	4 066
FAEST-128f		960	17	8	108	3	312	32	5 170
FAEST-192s	$\mathsf{AES192}_k(x)\|\|\mathsf{AES192}_k(x \oplus 1)$	1728	16	12	162	3	504	48	9 410
FAEST-192f		1728	24	8	163	3	504	48	11 738
FAEST-256s	$\mathsf{AES256}_k(x)\|\|\mathsf{AES256}_k(x \oplus 1)$	2208	22	6	225	3	696	48	16 626
FAEST-256f		2208	33	8	229	3	704	48	20 856
FAEST-EM-128s	$\mathsf{AES128}_x(k) \oplus k$	640	11	7	103	2	312	32	3 466
FAEST-EM-128f		640	17	8	105	2	312	32	4 170
FAEST-EM-192s	$\mathsf{Rijndael192}_x(k) \oplus k$	1152	16	8	162	2	504	48	7 874
FAEST-EM-192f		1152	25	8	171	2	504	48	9 818
FAEST-EM-256s	$\mathsf{Rijndael256}_x(k) \oplus k$	1792	22	6	218	2	696	64	14 554
FAEST-EM-256f		1792	33	8	229	2	704	64	18 084

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; n_{leafcom} is the number of λ-bit blocks in each leaf commitment; and n_{mult} is the number of AND gates in the CRT multiplication circuit, which depends on $(\lambda, \tau, w_{\mathsf{grind}})$ and therefore may differ between the `s` and `f` variants at the same λ (as it does for $\lambda = 256$).

Param.	Formula	Description	FAEST			FAEST-EM		
λ			128	192	256	128	192	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}	$-\frac{\lambda}{8} + 56 + 28 \left\lfloor \frac{\lambda}{256} \right\rfloor$ 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}	$((R/2) - 1) \cdot N_{\mathsf{st\text{-}bits}}$	no. witness bits for enc. ($d_{\mathsf{zk}} = 7$)	512	640	768	512	960	1 536
ℓ	$\ell_{\mathsf{ke}} + \beta\ell_{\mathsf{enc}}$	witness length in bits ($d_{\mathsf{zk}} = 7$)	960	1 728	2 208	640	1 152	1 792
C	$\ell + 2S_{\mathsf{ke}} + \beta \cdot \frac{1}{2}S_{\mathsf{enc}} + 1$	no. constraints ($d_{\mathsf{zk}} = 7$)	1 121	1 985	2537	721	1 297	2 017

Table 3.3: Unified OWF parameters for all instances

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

Notation. For integers a, b, we write $[a, b) = \{a, \ldots, b-1\}$. For a vector $\boldsymbol{a}$, $\boldsymbol{a}[j]$ denotes the j-th entry and $\boldsymbol{a}[a..b)$ a sub-vector.

Finite field Arithmetic. FAEST uses finite field arithmetic over $\mathbb{F}_{2^8}$ (as part of AES)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 taken from a table of low Hamming weight irreducible polynomials [Ser98], which agrees with the AES specification [AES01] for P_8.

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

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, Polynomials and Integers. The following algorithms are used to convert and manipulate finite field elements.

- $\mathsf{ToField}(\mathbf{x}; k)$: 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.
- $\mathsf{ToBits}(\boldsymbol{x}; k, n)$: maps $\boldsymbol{x} \in \mathbb{F}_{2^k}^n$, for $n \geq 1$, into a bit string $\mathbf{x} \in \{0,1\}^{nk}$.
- $\mathsf{ToPoly}(\mathbf{a}; d)$: maps a bit-string $\mathbf{a} \in \{0,1\}^d$, for $d \geq 1$, into the polynomial $\sum_{i=0}^{d-1} \mathbf{a}[i] \cdot x^i \in \mathbb{F}_2[x]$ of degree less than d, using little-endian ordering.
- $\mathsf{ToBits}(f; d)$: maps a polynomial $f \in \mathbb{F}_2[x]$ of degree less than d into its coefficient vector in $\{0,1\}^d$. This is the inverse of ToPoly; we overload the name ToBits, since under the representation $\mathbb{F}_{2^k} = \mathbb{F}_2[x]/P_k(x)$ (with α_k corresponding to x), the two versions agree on polynomials of degree less than k.
- $\mathsf{ByteCombine}(\mathbf{x}; \lambda)$: 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].

Note that arithmetic on polynomials output by ToPoly is performed in $\mathbb{F}_2[x]$ without any modular reduction, unless explicitly specified.

In an implementation, depending on the chosen representation of finite field elements, ToField, ToPoly and ToBits may not require any operations (e.g. if field elements are stored as 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 $i \in [0..2^d)$.

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 : $\boldsymbol{x}[i] := \sum_{j=0}^{k-1} \mathbf{x}[ki+j] \cdot \alpha_k^j$
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** $\mathbf{x} \in \{0,1\}^{nk}$

ToPoly($\mathbf{a}; d$)

1 : // $\mathbf{a} \in \{0,1\}^d$, for $d \geq 1$
2 : **return** $\sum_{i=0}^{d-1} \mathbf{a}[i] \cdot x^i \in \mathbb{F}_2[x]$

ToBits($f; d$)

1 : // $f \in \mathbb{F}_2[x]$ with $\deg f < d$; inverse of ToPoly
2 : Parse $f_0 + f_1 x + \cdots + f_{d-1}x^{d-1} = f$ for $f_j \in \mathbb{F}_2$
3 : **return** $(f_0, \ldots, f_{d-1}) \in \{0,1\}^d$

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 $\mathbf{x}[i] \in \mathbb{F}_2 \subset \mathbb{F}_{2^k}$
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** $j \in [0..d)$
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

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 generating the BAVC commitment key
– $\mathsf{H}_1 : \{0,1\}^* \to \{0,1\}^{2\lambda}$, collision-resistant hash for vector commitments
– $\mathsf{H}_2^j : \{0,1\}^* \to \{0,1\}^*$, for $j \in \{0,1,2,3\}$, 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
– $\mathsf{H}_5 : \{0,1\}^{128} \times \{0,1\}^{32} \times \{0,1\}^\lambda \to \{0,1\}^n_{\mathsf{mask}}$, domain-separated hash used on the CRT gate labels

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 10.1.2). 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.

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** $(\mathsf{iv}_0, \mathsf{iv}_1, \mathsf{iv}_2, \mathsf{BitDec}(\mathsf{iv}'_3, 32))$

AddToLowerWord(iv, α)

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** $(\mathsf{BitDec}(\mathsf{iv}'_0, 32), \mathsf{iv}_1, \mathsf{iv}_2, \mathsf{iv}_3)$

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 : $h := \lceil m/128 \rceil$
2 : $r := m - (h-1) \cdot 128$
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 : $\quad \mathbf{s}_i \leftarrow \mathsf{AES}\text{-}\lambda.\mathsf{Encrypt}(\mathsf{AddToLowerWord}(\mathsf{iv}', i), \overline{\mathbf{k}})$
7 : **return** $\mathbf{s}_0 \,\|\, \cdots \,\|\, \mathbf{s}_{h-2} \,\|\, \mathbf{s}_{h-1}[0..r)$

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 $i \in \{0, 1, 3, 4\}$
- $\mathsf{H}_2^j(m) := \text{SHAKE}(m \| 8 + j, \ell)$
- $\mathsf{H}_5(\mathsf{iv}, e, L) := \text{SHAKE}\big(\mathsf{iv} \| \mathsf{BitDec}(e, 32) \| L \| 5, n_{\mathsf{mask}}\big)$

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

In our security proofs, we model $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^j, \mathsf{H}_4, \mathsf{H}_5$ as random oracles, while H_3 is modeled as a pseudo-random function where the last λ input bits are the key (for the case of deterministic signing, H_3 is instead modeled as a random oracle).

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:

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

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

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

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

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

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

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

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

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

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 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 4.1. *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 $\mathbf{v} \in \mathbb{F}_q^r$,*

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

$\mathsf{Encrypt}(\mathbf{in}, \bar{\mathbf{k}}; \mathtt{param}_{\mathsf{OWF}})$

1 : **new** $\mathtt{state} \in \{0,1\}^{32 \cdot N_{\mathsf{st}}}$
2 : $\mathtt{state} := \mathbf{in}$
3 : $\mathsf{AddRoundKey}(\mathtt{state}, \bar{\mathbf{k}}[0..N_{\mathsf{st}} - 1])$

4 : **for** $r \in [1..R)$ **do**
5 : $\quad \mathsf{SubBytes}(\mathtt{state})$
6 : $\quad \mathsf{ShiftRows}_{N_{\mathsf{st}}}(\mathtt{state})$
7 : $\quad \mathsf{MixColumns}(\mathtt{state})$
8 : $\quad \mathsf{AddRoundKey}(\mathtt{state}, \bar{\mathbf{k}}[N_{\mathsf{st}} r..N_{\mathsf{st}}(r+1) - 1])$

9 : $\mathsf{SubBytes}(\mathtt{state})$
10 : $\mathsf{ShiftRows}_{N_{\mathsf{st}}}(\mathtt{state})$
11 : $\mathsf{AddRoundKey}(\mathtt{state}, \bar{\mathbf{k}}[N_{\mathsf{st}} R..N_{\mathsf{st}}(R+1) - 1])$

12 : **return out** $:= \mathtt{state}$

Fig. 4.3: The encryption routine

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 4.2. *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.*

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

Proposition 4.3. *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 $\mathbf{H}\mathbf{x}_0 = -\mathbf{x}_1$. If $\mathbf{x}_0 = 0$, then $\mathbf{x}_1 \neq 0$ and hence $\mathbf{H}'\mathbf{x} = \mathbf{x}_1 \neq 0$, a contradiction. Therefore $\mathbf{x}_0 \neq 0$, 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. □

4.2.1 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.

VOLEHash$(\mathsf{sd}, (\boldsymbol{x}_0, \boldsymbol{x}_1) \in \mathbb{F}_{2^\lambda}^{\ell+d_{\mathsf{zk}}-1} \times \mathbb{F}_{2^\lambda}^4)$

$\ell' := \ell + d_{\mathsf{zk}} - 1$
// λ-bit $\mathbf{r}_i, \mathbf{s}_i$; 64-bit $\mathbf{t}$
Parse $\mathsf{sd} = (\mathbf{r}_0 \| \mathbf{r}_1 \| \mathbf{r}_2 \| \mathbf{r}_3 \| \mathbf{s}_0 \| \mathbf{s}_1 \| \mathbf{s}_2 \| \mathbf{s}_3 \| \mathbf{t}) \in \{0,1\}^{8\lambda+64}$
$r_i := \mathsf{ToField}(\mathbf{r}_i, \lambda)$, **for** $i \in [0..4)$
$s_i := \mathsf{ToField}(\mathbf{s}_i, \lambda)$,**for** $i \in [0..4)$
$s_4 := \mathsf{ToField}(\mathbf{t} \| 0^{\lambda-64}, \lambda)$
$y_j := \sum_{i=0}^{\ell'-1} s_j^{\ell'-1-i} \boldsymbol{x}_0[i] \in \mathbb{F}_{2^\lambda}$, **for** $j \in [0..5)$
$h_j := r_j y_0 + y_{j+1} + \boldsymbol{x}_1[j]$, **for** $j \in [0..4)$
$\mathbf{h} := (h_0, h_1, h_2, h_3) \in \mathbb{F}_{2^\lambda}^4$
return h

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

// 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** $i \in \{0,1\}$
$s := \mathsf{ToField}(\mathbf{s}, \lambda)$
$t := \mathsf{ToField}(\mathbf{t} \| 0^{\lambda-64}, \lambda)$
$h_0 := \sum_{i=0}^{C-1} s^{C-1-i} \cdot \boldsymbol{x}_0[i]$
$h_1 := \sum_{i=0}^{C-1} t^{C-1-i} \cdot \boldsymbol{x}_0[i]$
$\mathbf{h} := r_0 \cdot h_0 + r_1 \cdot h_1 + x_1 \in \mathbb{F}_{2^\lambda}$
return h

Fig. 4.4: Universal hashing algorithms

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

Proposition 4.4. *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 4.5. *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. □

4.2.3 VOLE Universal Hash. For the CRT mask generation, VOLEHash runs on $\ell + d_{\mathsf{zk}} - 1 + 4$ elements of $\mathbb{F}_{2^\lambda}$, and outputs 4λ bits to match what's needed for the analysis of the KOS check. The algorithm runs several polynomial-based hashes and then combines them with a random matrix hash.

To achieve 4λ bits of security, we use 4 polynomial hashes with full-length challenges in $\mathbb{F}_{2^\lambda}$, plus one with a shorter 64-bit challenge. This means that we consider the input vector $\mathbf{x}_0 \in \mathbb{F}_{2^\lambda}^{\ell'}$ as coefficients of a polynomial, which we evaluate in the randomly chosen points $s_0, \ldots, s_4$ by computing the inner product of $\mathbf{x}_0$ with $(s_j^{\ell'-1}, \ldots, s_j, 1)$ for $j \in [0,5)$. These give the five intermediate hash outputs $y_0, \ldots, y_4 \in \mathbb{F}_{2^\lambda}$, which we then map to four elements via

$$\begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix} = \begin{bmatrix} r_0 & 1 & 0 & 0 & 0 \\ r_1 & 0 & 1 & 0 & 0 \\ r_2 & 0 & 0 & 1 & 0 \\ r_3 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}.$$

where $r_0, r_1, r_2, r_3 \in \mathbb{F}_{2^\lambda}$ are additional random seed elements. The hiding property is achieved by adding the last four input elements $\boldsymbol{x}_1$ to (h_0, h_1, h_2, h_3).

Lemma 4.6. *Write $\ell' = \ell + d_{\mathsf{zk}} - 1$. VOLEHash is an ε_{v}-almost universal hash family in $\mathbb{F}_{2^\lambda}^{4\times(\ell'+4)}$, for $\varepsilon_{\mathsf{v}} = 2^{-4\lambda}(1 + 2^{-4})$, if $\ell' \leq 2^{12}$. Furthermore, VOLEHash is $\mathbb{F}_{2^\lambda}^{\ell'}$-hiding.*

By Proposition 4.3 it suffices to show ε_{v}-uniformity of (h_0, h_1, h_2, h_3) without adding $\boldsymbol{x}_1$. Note that this cannot be argued by composing Proposition 4.5 with a matrix hash, since the outer matrix hash is not uniform for all non-zero inputs (which is clearly seen if e.g. $y_0 = 0, y_1 \neq 0$).

Proof. Let $P(X) := \sum_{i=0}^{\ell'-1} \boldsymbol{x}_0[i] \cdot X^{\ell'-1-i}$ be the degree-$< \ell'$ polynomial defined by an input $\boldsymbol{x}_0$, so $y_j = P(s_j)$ for each $j \in [0..5)$. Since $\boldsymbol{x}_0 \neq 0$, $P(X)$ is not the zero polynomial. Fix an arbitrary target $(v_0, v_1, v_2, v_3) \in \mathbb{F}_{2^\lambda}^4$, and consider two cases depending on whether $y_0 = P(s_0)$ is zero. If $y_0 \neq 0$, each equation $r_i y_0 + y_{i+1} = v_i$ has exactly one solution in r_i, regardless of the value of y_{i+1}. Since r_0, r_1, r_2, r_3 are uniform and independent of $s_0, \ldots, s_4$, conditioned on $y_0 \neq 0$ the target is hit with probability exactly $2^{-4\lambda}$. Now suppose $y_0 = 0$. Since $P \neq 0$ has at most $\deg P \leq \ell' - 1$ roots, this event occurs with probability at most $(\ell'-1)/2^\lambda$ over s_0, and furthermore it forces $\deg P \geq 1$, since a nonzero constant polynomial has no roots. The output when $y_0 = 0$ is then exactly (y_1, y_2, y_3, y_4). For any fixed $v \in \mathbb{F}_{2^\lambda}$, $P(X) - v$ is a nonzero polynomial of degree $\leq \ell' - 1$, so $y_1 = P(s_1), y_2 = P(s_2), y_3 = P(s_3)$ each hit their target with probability at most $(\ell' - 1)/2^\lambda$, while $y_4 = P(s_4)$, with 64-bit s_4, does so with probability at most $(\ell' - 1)/2^{64}$. This gives $\Pr[(y_1, y_2, y_3, y_4) = (v_0, v_1, v_2, v_3) \mid y_0 = 0] \leq (\ell' - 1)^4/2^{3\lambda+64}$. Combining both cases, $\Pr[(h_0, h_1, h_2, h_3) = (v_0, v_1, v_2, v_3)] \leq 2^{-4\lambda} + (\ell' - 1)^5/2^{4\lambda+64} \leq 2^{-4\lambda}(1 + 2^{-4})$, for all $\ell' \leq 2^{12}$. □

4.2.4 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 $C' = C + 1$ field elements, where C is the number of constraints. 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

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

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

Lemma 4.7. ZKHash *is an* $\varepsilon_{\mathsf{zk}}$*-almost universal hash family in* $\mathbb{F}_{2^\lambda}^{1 \times C'}$*, for* $\varepsilon_{\mathsf{zk}} = 2^{-\lambda}(1 + 2^{-38})$*, if* $C' \leq 2^{13}$*. Furthermore,* ZKHash *is* $\mathbb{F}_{2^\lambda}^C$*-hiding.*

As with VOLEHash, by Proposition 4.3 it suffices to show $\varepsilon_{\mathsf{zk}}$-uniformity of the hash defined by $\mathbf{r}^\top \mathbf{x}_0$.

Proof. 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 $\varepsilon_0 = (C' - 1)/2^\lambda$, and the second for $\varepsilon_1 = (C' - 1)/2^{64}$; together, this gives an $\varepsilon_0\varepsilon_1$-almost universal hash (via Proposition 4.4), where $\varepsilon_0\varepsilon_1 \leq (C')^2/2^{\lambda+64} \leq 2^{-\lambda-38}$, for all $C' \leq 2^{13}$. Combining the inner and outer parts, from Proposition 4.5 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 define the batch all-but-one vector commitment scheme used by FAEST and the conversion of its leaf seeds into the τ small VOLE instances. The BAVC construction is specified in Section 5.2, and ConvertToVOLE is specified in Section 5.4. The challenge-decomposition procedures $\mathsf{DecodeChall}_3$ and $\mathsf{DecodeAllChall}_3$ are specified in Section 5.5, and the algorithm ConvertToVOLE for creating the small VOLE instances is in Section 5.4.

We defer the CRT-based VOLECommit and VOLEReconstruct algorithms until Section 6. They use the BAVC and small-VOLE components defined here to construct VOLE correlations for witness bits in $\mathbb{F}_2$ and masks in $\mathbb{F}_{2^\lambda}$.

The parameters used by the BAVC and small-VOLE components are listed in Table 5.1. The BAVC uses one global GGM tree with $L = \sum_{i=0}^{\tau-1} N_i$ leaves. Its leaves are partitioned into τ BAVC vectors. The i-th vector contains $N_i = 2^{d_i}$ leaves, where d_i is the hidden-index bit length and the number of binary VOLE columns returned by ConvertToVOLE for small-VOLE instance i.

Parameter	Formula	Description
n_{leafcom}	3 for FAEST, 2 for FAEST-EM	Number of λ-bit blocks in each leaf commitment
n_Δ	$\lambda - w_{\mathsf{grind}}$	Number of non-grinding challenge bits
τ_1	$n_\Delta \bmod \tau$	Number of larger small-VOLE instances
τ_0	$\tau - \tau_1$	Number of smaller small-VOLE instances
k	$\lfloor n_\Delta/\tau \rfloor + 1$	Bit length of larger small-VOLE instances
d_i	$\begin{cases} k & \text{if } i < \tau_1, \\ k-1 & \text{otherwise} \end{cases}$	Hidden-index bit length and number of binary VOLE columns for instance i
d_0	$\max_{i \in [0..\tau)} d_i$	Maximum hidden-index bit length; each CRT-mask block is allocated d_0 rows
N_i	2^{d_i}	Number of leaves in small-VOLE instance i
L	$\sum_{i=0}^{\tau-1} N_i = \tau_1 2^k + \tau_0 2^{k-1}$	Number of leaves in the GGM tree
n_{mask}	$(d_{\mathsf{zk}} - 1) + 4$	Number of CRT-generated field-mask correlations: $d_{\mathsf{zk}} - 1$ for QuickSilver and 4 for VOLE Hash
$\hat{\ell}$	$\ell + n_{\mathsf{mask}} d_0$	Binary length of each vector produced by ConvertToVOLE: ℓ witness rows + n_{mask} CRT-mask blocks of d_0 rows

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 $\alpha \in [L - 1..2L - 1)$.
- FAEST.LeafHash. It describes the hash used in FAEST.LeafCommit. To build the hash, it maps the public commitment key $\mathsf{uhash} \in \{0,1\}^{3\lambda}$ 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. Note that the first input x_0 is the derived leaf seed sd, while x_1 is the remaining 3λ-bit PRG output.

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, $\mathbf{x}_1$) $\leftarrow$ PRG(r, iv, twk; 4λ)
2 : com $\leftarrow$ LeafHash(uhash, (sd, $\mathbf{x}_1$))
3 : **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

– 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 $\mathsf{sd}_{i,j}$, $i \in [0..\tau)$ and $j \in [0..N_i)$.

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 $I = (\Delta_0, \ldots, \Delta_{\tau-1})$, thenReconstruct indeed recovers the original commitment com and the correct seeds of all non-hidden leaves.

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*

$\forall(\mathsf{com}, \mathsf{decom}, ((\mathsf{sd}_{i,j})_{j\in[0..N_i)})_{i\in[0..\tau)}) \leftarrow \mathsf{BAVC.Commit}(r, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$, *where* $r \leftarrow \{0,1\}^\lambda$ *is the root seed,* $\mathsf{iv} \leftarrow \{0,1\}^{128}$ *is an AES-CTR IV,* $\mathtt{param}, \mathtt{param}_{\mathsf{VOLE}}$ *are public parameters, and* $\forall I = (\Delta_0, \ldots, \Delta_{\tau-1}) \in [0..N_0) \times \cdots \times [0..N_{\tau-1})$, *so that* $\mathsf{decom}_I \leftarrow \mathsf{BAVC.Open}(\mathsf{decom}, I; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$,

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

$$\mathsf{BAVC.Reconstruct}(\mathsf{decom}_I, I, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}}) = \left(\mathsf{com}, \left((\mathsf{sd}_{i,j})_{j\in[0..N_i)\setminus\{\Delta_i\}}\right)_{i\in[0..\tau)}\right).$$

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: $\mathsf{com} \in \{0,1\}^{2\lambda}$, $\mathsf{decom} \in \{0,1\}^{(2L-1)\lambda + n_{\mathsf{leafcom}}\lambda L}$, $\left((\mathsf{sd}_{i,j})_{j\in[0..N_i)}\right)_{i\in[0..\tau)} \in (\{0,1\}^\lambda)^L$

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

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

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

OUTPUT: $\mathsf{decom}_I \in \{0,1\}^{n_{\mathsf{leafcom}}\lambda\tau + T_{\mathsf{open}}\lambda} \cup \{\perp\}$

1: Parse $\mathsf{decom} := \left((k_\alpha)_{\alpha\in[0..2L-1)}, \left((\mathsf{com}_{i,j})_{j\in[0..N_i)}\right)_{i\in[0..\tau)}\right)$
2: // Initialize the opening with the τ hidden leaf commitments
3: $\mathsf{decom}_I \leftarrow (\mathsf{com}_{0,\Delta_0}, \ldots, \mathsf{com}_{\tau-1,\Delta_{\tau-1}})$
4: // Mark the tree nodes that must remain hidden
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: $\quad$ // Mark the hidden leaf and all its unmarked ancestors
9: $\quad \alpha \leftarrow \mathsf{BAVC.PosInTree}(i, \Delta_i)$
10: $\quad S[\alpha] \leftarrow 1$
11: $\quad n_h \leftarrow n_h + 1$
12: $\quad$ **while** $\alpha > 0$ and $S[\lfloor(\alpha-1)/2\rfloor] = 0$ **do**
13: $\quad\quad \alpha \leftarrow \lfloor(\alpha-1)/2\rfloor$
14: $\quad\quad S[\alpha] \leftarrow 1$
15: $\quad\quad n_h \leftarrow n_h + 1$
16: **if** $n_h - 2\tau + 1 > T_{\mathsf{open}}$ **then**
17: $\quad$ **return** $\perp$
18: // Add the boundary sibling nodes to the opening
19: **for** $i = L-2$ to 0 **do**
20: $\quad$ **if** $S[2i+1] \oplus S[2i+2] = 1$ **then**
21: $\quad\quad \alpha \leftarrow 2i + 1 + S[2i+1]$
22: $\quad\quad$ Append k_α to decom_I
23: Zero-append decom_I to length $n_{\mathsf{leafcom}}\lambda\tau + T_{\mathsf{open}}\lambda$ bits
24: **return** decom_I

Fig. 5.4: Commit and open algorithms for the 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**
  $\mathsf{uhash} \leftarrow \mathsf{H}_0(\mathsf{iv})$
// 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 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
      if variant is FAEST-λ **then**
        $(\mathsf{sd}_{i,j}, h_{i,j}) \leftarrow$ FAEST.LeafCommit$(k_\alpha, \mathsf{iv}, i + L - 1, \mathsf{uhash})$
      else
        $(\mathsf{sd}_{i,j}, h_{i,j}) \leftarrow$ FAEST-EM.LeafCommit$(k_\alpha, \mathsf{iv}, i + L - 1)$
  $h_i \leftarrow \mathsf{H}_1(h_{i,0} \| \ldots \| h_{i,N_i-1})$
$\mathsf{com} \leftarrow \mathsf{H}_1(h_0 \| \ldots \| h_{\tau-1})$
return $\left(\mathsf{com}, \left((\mathsf{sd}_{i,j})_{j \in [0..N_i) \setminus \{\Delta_i\}}\right)_{i \in [0..\tau)}\right)$

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 complete ordered family $\big((\mathsf{com}_{i,j})_{j\in[0..N_i)}\big)_{i\in[0..\tau)}$ matches the family from BAVC.Commit. For every i, the hidden commitment $\mathsf{com}_{i,\Delta_i}$ comes from the opening, while all the other commitments are recomputed. Hence $\mathsf{com}' = \mathsf{com}$.

Other than the commitment com, the BAVC.Reconstruct algorithm also returns $\big((\mathsf{sd}_{i,j})_{j\in[0..N_i)\setminus\{\Delta_i\}}\big)_{i\in[0..\tau)}$ 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. The hidden leaf seeds are not returned by BAVC.Reconstruct. This proves correctness whenever BAVC.Open does not abort.

□

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 by PRG to a vector in $\mathbb{F}_2^{\hat{\ell}}$. 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

$$\mathbf{q} = \sum_{i\in[0..N)\setminus\{\Delta\}} (\Delta - i) \cdot \mathsf{PRG}(\mathsf{sd}_i)$$

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

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

INPUT: $\mathsf{sd}_0 \in \{0,1\}^\lambda \cup \{\bot\}$, $\mathsf{sd}_i \in \{0,1\}^\lambda$ for $i \in [1..N)$, $\mathsf{iv} \in \{0,1\}^{128}$, $\mathsf{twk} \in \{0,1\}^{32}$, $\hat{\ell} \in \mathbb{N}$, $N = 2^d$

OUTPUT: $(\mathbf{u}, \mathbf{v}_0, \ldots, \mathbf{v}_{d-1}) \in (\mathbb{F}_2^{\hat{\ell}})^{d+1}$

$d \leftarrow \log_2 N$

$\mathbf{r}_{0,0} := 0^{\hat{\ell}}$ **if** $\mathsf{sd}_0 = \bot$ **else** PRG$(\mathsf{sd}_0, \mathsf{iv}, \mathsf{twk}; \hat{\ell})$

for $i \in [1..N)$ **do**

  $\mathbf{r}_{0,i} :=$ PRG$(\mathsf{sd}_i, \mathsf{iv}, \mathsf{twk}; \hat{\ell})$

$\mathbf{v}_0 := \cdots := \mathbf{v}_{d-1} := 0^{\hat{\ell}}$

for $j \in [0..d)$ **do**

  for $i \in [0..N/2^{j+1})$ **do**

    $\mathbf{v}_j := \mathbf{v}_j \oplus \mathbf{r}_{j,2i+1}$

    $\mathbf{r}_{j+1,i} := \mathbf{r}_{j,2i} \oplus \mathbf{r}_{j,2i+1}$

$\mathbf{u} := \mathbf{r}_{d,0}$

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

to $\bot$ and ignores the $\mathbf{u}$ part of the output. For correctness, the verifier must additionally permute its seeds according to the permutation $i \mapsto i \oplus \Delta$.

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

Proposition 5.2. *Let $N = 2^d$, let $\Delta \in [0..N)$, and let $(\mathsf{sd}_0, \ldots, \mathsf{sd}_{N-1}) \in (\{0,1\}^\lambda)^N$. Define $\mathsf{sd}'_0 := \bot$ and $\mathsf{sd}'_i := \mathsf{sd}_{i \oplus \Delta}$ for every $i \in [1..N)$. Here $i \oplus \Delta$ denotes bitwise XOR of the d-bit binary representations of the two indices. Let $\mathsf{iv} \in \{0,1\}^{128}$ and $\mathsf{twk} \in \{0,1\}^{32}$, and use the same values in both executions. Then, if*

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

and

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

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 $\mathbf{u} = \mathbf{r}_{d,0} = \mathbf{r}_{0,0} \oplus \cdots \oplus \mathbf{r}_{0,N-1}$.

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 $\mathbf{v}_0 \oplus \delta_0 \cdot \mathbf{u}$ 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 $\mathbf{r}'_{0,2i+1} = \mathbf{r}_{0,(2i+1) \oplus \Delta}$, and $(2i+1) \oplus \Delta$ is even. It follows that $\mathbf{q}_0 \oplus \mathbf{v}_0 = \bigoplus_{i=0}^{N-1} \mathbf{r}_{0,i} = \delta_0 \cdot \mathbf{u}$.

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 $i \mapsto i \oplus \Delta'$. The outputs $\mathbf{q}_j$ for $j \in [1..d)$ do not depend on $\mathbf{r}'_{1,0}$, because index 0 has every remaining index bit equal to zero and is never included in the sums defining these output columns. Hence, for the purpose of computing $\mathbf{q}_1, \ldots, \mathbf{q}_{d-1}$, we may replace $\mathbf{r}'_{1,0}$ by $0^{\hat{\ell}}$. 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 Challenge Decomposition.

Let $\mathsf{chall} \in \{0,1\}^\lambda$, the procedures $\mathsf{DecodeChall}_3$ and $\mathsf{DecodeAllChall}_3$, shown in Figure 5.7, verify the grinding suffix and divide the first $n_\Delta = \lambda - w_{\mathsf{grind}}$ challenge bits among the τ small-VOLE instances. For $i \in [0..\tau)$, $\mathsf{DecodeChall}_3(\mathsf{chall}, i; \mathtt{param})$ returns the d_i-bit block assigned to instance i, and $\mathsf{DecodeAllChall}_3$ converts all these blocks into $(\Delta_0, \ldots, \Delta_{\tau-1})$. 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 $i \geq \tau_1$. 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 procedure returns $\perp$. Otherwise, it checks that the final w_{grind} bits are zero and sets $\mathsf{chall}' := \mathsf{chall}[0..\lambda - w_{\mathsf{grind}})$. 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.

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

1 : **if** $i \notin [0..\tau)$ **then**
2 : **return** $\perp$
3 : **if** $\mathsf{chall}[\lambda - w_{\mathsf{grind}}..\lambda) \neq 0^{w_{\mathsf{grind}}}$ **then**
4 : **return** $\perp$
5 : $\mathsf{chall}' \leftarrow \mathsf{chall}[0..\lambda - w_{\mathsf{grind}})$
6 : **if** $i < \tau_1$ **then**
7 : $\mathtt{lo} := i \cdot k$
8 : **else**
9 : $t := i - \tau_1$
10 : $\mathtt{lo} := \tau_1 k + t(k-1)$
11 : **return** $\mathsf{chall}'[\mathtt{lo}..\mathtt{lo} + d_i)$

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

1 : **for** $i \in [0..\tau)$ **do**
2 : $b_i \leftarrow \mathsf{DecodeChall}_3(\mathsf{chall}, i; \mathtt{param})$
3 : **if** $b_i = \perp$ **then**
4 : **return** $\perp$
5 : $\Delta_i \leftarrow \mathsf{NumRec}(d_i, b_i)$
6 : **return** $(\Delta_0, \ldots, \Delta_{\tau-1})$

Fig. 5.7: Challenge decoding algorithm

The CRT-based VOLECommit and VOLEReconstruct algorithms use these small-VOLE challenges and are specified in Section 6.

6 Creating VOLE Commitments and Masks with CRT

This section details the VOLECommit and VOLEReconstruct algorithms for creating and verifying VOLE correlations using the components from Section 5.

6.1 Overview.

The algorithms serve two purposes. Firstly, for the ℓ witness bits, they create VOLE correlations for random bits over $\mathbb{F}_2$, that is, random values $u_j \in \mathbb{F}_2, v_j, q_j \in \mathbb{F}_{2^\lambda}$ such that $q_j = v_j + u_j \cdot \Delta$, where u_j, v_j are initially computed by the signer, then $\Delta \in \mathbb{F}_{2^\lambda}$ is derived from challenge chall_3, forcing the signer to open q_j to the verifier. These are correlations are set up by correcting each small-VOLE instance from ConvertToVOLE to use the same u_j bits, requiring $\tau - 1$ correction bits per u_j.

Secondly, we need to create VOLE *masks*, which are relations $\bar{q}_j = \bar{v}_j + \bar{u}_j \cdot \Delta$, where now $\bar{u}_j$ is a uniform element of $\mathbb{F}_{2^\lambda}$; these are used to mask the VOLE consistency check and the ZK constraint check. Previous versions of FAEST built up these masks bit-by-bit, by sending $\tau - 1$ corrections *per bit* of $\bar{u}_j$, for a total of $(\tau - 1)\lambda$ bits. This is, however, very wasteful. In particular, if we did a naive version of small-field VOLE-in-the-head using parallel repetition across the τ instances — skipping the VOLE check — then these ZK masks would be much cheaper to generate, as they could be independent in each repetition. Unfortunately, that version would not be round-by-round sound, and we would lose much more from that than we would save from the cheaper masks. Instead, we devise a new way to generate the masks, which are VOLEs over $\mathbb{F}_{2^\lambda}$, without having to build them up bit-by-bit from $\mathbb{F}_2$ (subfield) VOLE.

Generating VOLE Masks via CRT: the High-Level Idea. To generate secret shares of $\bar{\mathbf{u}}_m \cdot \Delta$, where $\bar{\mathbf{u}}_m \in \mathbb{F}_{2^\lambda}$ is the m-th field mask, the idea is to use CRT-based multiplication. Let $P_\lambda(x)$ be the degree-λ irreducible polynomial used to represent $\mathbb{F}_{2^\lambda}$ as $\mathbb{F}_2[x]/P_\lambda(x)$, and let $\hat{u}_m \in \mathbb{F}_2[x]$ and $\hat{\Delta} \in \mathbb{F}_2[x]$ be the polynomial representatives of $\bar{\mathbf{u}}_m \in \mathbb{F}_{2^\lambda}$ and $\Delta \in \mathbb{F}_{2^\lambda}$. In this construction, $\deg \hat{u}_m \le \lambda - 1$ and $\deg \hat{\Delta} < n_\Delta = \lambda - w_{\mathsf{grind}}$.

Choose an integer $s \ge \tau$, pairwise-coprime monic polynomials $M_0(x), \ldots, M_{s-1}(x) \in \mathbb{F}_2[x]$ with $\deg M_i = d_i$ for every $i \in [0..\tau)$, and a multiplicity $m_\infty \ge 0$ for the *place at infinity*, such that

$$\sum_{j=0}^{s-1} \deg M_j + m_\infty = \lambda + n_\Delta - 1 = 2\lambda - w_{\mathsf{grind}} - 1.$$

Define $M(x) := \prod_{j=0}^{s-1} M_j(x)$, of degree $D := 2\lambda - w_{\mathsf{grind}} - 1 - m_\infty$. The product $\hat{u}_m\hat{\Delta}$ can be obtained as follows:

1. Compute each $\hat{u}_m\hat{\Delta} \bmod M_j$, for $j \in [0..s)$.
2. Lift the individual products to the unique $\hat{u}_m\hat{\Delta} \bmod M$ using that $\{M_i\}_i$ are pair-wise coprime.
3. If $m_\infty > 0$, compute the residue of $\hat{u}_m\hat{\Delta}$ at the place at infinity with multiplicity m_∞, that is, its top m_∞ coefficients, and use these to recover $\hat{u}_m\hat{\Delta}$ from $\hat{u}_m\hat{\Delta} \bmod M$.

When $m_\infty = 0$ we have $\deg(\hat{u}_m\hat{\Delta}) < \deg M$, so reduction modulo M does not wrap around and $\hat{u}_m\hat{\Delta} \bmod M$ already equals $\hat{u}_m\hat{\Delta}$ over $\mathbb{F}_2[x]$. For $m_\infty > 0$ the two kinds of residue together still determine the product, since the $\mathbb{F}_2$-linear map

$$f \longmapsto \big(f \bmod M,\ (f_D, \ldots, f_{D+m_\infty-1})\big)$$

on polynomials $f = \sum_k f_k x^k$ of degree less than $D + m_\infty$ is a bijection: a nonzero element of its kernel would be a multiple of M, hence of degree at least D, yet have vanishing top m_∞ coefficients, hence degree less than D.

Small VOLEs from Vector Commitments. For every $i \in [0..\tau)$, the modulus $M_i(x)$ is associated with small-VOLE instance i. These first τ moduli are fixed to be the lexicographically smallest distinct irreducibles with $\deg M_i = d_i$. Using the pseudorandom outputs of ConvertToVOLE for BAVC vector i, we obtain shares of

$$(\hat{u}_m \bmod M_i)(\hat{\Delta} \bmod M_i) \bmod M_i.$$

The d_i-bit mask and challenge strings used in small-VOLE instance i are interpreted as coefficient vectors in the quotient algebra $\mathbb{F}_2[x]/(M_i(x))$.

This construction first leads to a problem when defining $u \in \mathbb{F}_{2^\lambda}$, since the tree residues only determine u modulo $M_{\text{tree}}(x) := \prod_{i=0}^{\tau-1} M_i(x)$, which has degree $\lambda - w_{\mathsf{grind}}$: they fix a unique representative $\mathbf{u}_{\mathsf{low}} \in \mathbb{F}_2[x]$ of degree less than $\lambda - w_{\mathsf{grind}}$. We cannot simply define $u := \mathbf{u}_{\mathsf{low}}$, since the mask would then not have enough entropy to mask a field element in $\mathbb{F}_{2^\lambda}$. Instead, to define the mask, the prover samples w_{grind} additional secret bits $\mathbf{u}_{\mathsf{hi}} \in \mathbb{F}_2^{w_{\mathsf{grind}}}$ and defines

$$\hat{u}(x) := \mathbf{u}_{\mathsf{low}} + \mathsf{ToPoly}(\mathbf{u}_{\mathsf{hi}}; w_{\mathsf{grind}}) \cdot M_{\text{tree}}(x), \text{ and } u := \hat{u}(x) \bmod P_\lambda(x) \in \mathbb{F}_{2^\lambda}.$$

where $\mathsf{ToPoly}(\mathbf{u}_{\mathsf{hi}}; w_{\mathsf{grind}})$ denotes the polynomial with coefficient vector $\mathbf{u}_{\mathsf{hi}}$ (see Figure 3.4). Since division with remainder by M_{tree} is a bijection

$$\{f \in \mathbb{F}_2[x] \mid \deg(f) < \lambda\} \cong \{(f, g) \in (\mathbb{F}_2[x])^2 \mid \deg(f) < \lambda - w_{\mathsf{grind}} \wedge \deg(g) < w_{\mathsf{grind}}\}$$

then if the residue bits and quotient bits are uniform and independent, then $\bar{\mathbf{u}}_m$ is uniform in $\mathbb{F}_{2^\lambda}$.

In the real protocol both $\mathbf{u}_{\mathsf{low}}$ and $\mathbf{u}_{\mathsf{hi}}$ are produced by domain-separated PRG calls, and their pseudorandomness is established by the security proof. The security proof does not require them to be separately committed — instead, the VOLE consistency check enforces consistency of the resulting correlations $\bar{\mathbf{u}}_m \Delta$ in $\mathbb{F}_{2^\lambda}$.

Note that all the VOLEs modulo $M_0, \ldots, M_{\tau-1}$ only commit to $\mathbf{u}_{\mathsf{low}}$, so we have to take care of committing to $\mathbf{u}_{\mathsf{hi}}$ in a different way. More importantly, we so far only generated shares that reconstruct $\hat{u} \cdot \hat{\Delta}$ of degree $\lambda - w_{\mathsf{grind}} - 1$, so we are still missing the remaining degree $\lambda - 1$ needed to represent the full multiplication modulo M.

Filling the degree gap. Assume the remaining $M_\tau, \ldots, M_{s-1}$ are of small degree and coprime, then we can write each multiplication $\bmod M_j$ in terms of bit multiplications between $\mathbb{F}_2$-linear functions of tbe secret u and Δ (using fast bilinear multiplication algorithms that minimize the number of AND gates). We denote the total number of such multiplications as n_{mult} and the linear functions as f_e, g_e for u, Δ respectively. Note that n_{mult} depends on the whole modulus set $M_0, \ldots, M_{s-1}$ together with m_∞, and hence on $(\lambda, \tau, w_{\mathsf{grind}})$ rather than on λ alone.

With this viewpoint, all we need is a way to get shares of $f_e(u)g_e(\Delta)$ for all $e \in [0..n_{\mathsf{mult}})$ and then take a linear combination of them (along with the previously obtained shares of $\hat{u}\hat{\Delta} \bmod M_i$, for $i \in [0..\tau)$) to get the VOLE output — corresponding to the CRT lifting (and, when $m_\infty > 0$, the interpolation at infinity). In the algorithms below, these linear maps f_e, g_e are the rows of the precomputed matrices $\mathbf{F}$ and $\mathbf{G}$.)

To compute shares of the bit $f_e(u)g_e(\Delta)$, we use an idea inspired by free-XOR garbling [KS08]/IKNP OT extension [IKNP03]. First, we want to obtain an $\mathbb{F}_2$ subfield-VOLE with the roles reversed, where $g_e(\Delta)$ is the secret bit held by the verifier, authenticated under a secret key of the signer. It turns out that this can be obtained without any additional GGM trees by taking the first λ witness VOLE rows $(\mathbf{V}_0, \ldots, \mathbf{V}_{\tau-1})$. For ordinary FAEST, these rows correspond to additive shares of products of the secret cipher key with Δ, while for FAEST-EM, they correspond to the secret input block and Δ.

Concretely, these witness VOLEs give additive shares of $u'_i \cdot \Delta$ where u'_i is used to mask the i-th bit of the AES key. By "transposing" these VOLEs, we can consider the additive shares of the matrix $u'_1 \cdot \Delta, \ldots, u'_\lambda \cdot \Delta$, whose ith row we write as $u' \cdot \Delta$. As the sharing operation is XOR, the parties can thus obtain shares of $\Delta_i \cdot u'$, where $u' \in \mathbb{F}_2^\lambda$ and $\Delta_i \in \{0,1\}$; these can in turn be used to obtain shares of $g_e(\Delta) \cdot u'$ for *any* linear function g_e.

To see this, note that the prover can compute two strings $A, A + u'$ while the verifier learns $A + g_e(\Delta) \cdot u'$. Consider, for the sake of simplicity, a hash function H with output in $\{0,1\}$. Then hashing the two strings $A, A + u'$ and setting the random bit $r = H(A) \oplus H(A + u')$, we see that if the verifier computes $H(A + g_e(\Delta) \cdot u')$ then

$$H(A) \oplus H(A + g_e(\Delta) \cdot u') = r \cdot g_e(\Delta)$$

The signer then turns these into shares of $g_e(\Delta) \cdot f_e(u)$ by sending a 1-bit correction $r \oplus f_e(u)$. We denote the matrix of correction bits as $\mathbf{c}_{\mathsf{mult}}$. To simplify parsing, each row of $\mathbf{c}_{\mathsf{mult}}$ is zero-padded to an exact multiple of bytes, and the padded version used in the signature output.

Consistency Check. The verifier computes each value $\gamma_e = g_e(\Delta) = \mathbf{G}[e] \cdot \Delta'$ directly from the challenge, so the signer cannot choose the $g_e(\Delta)$ values. The possible malicious behavior is that the signer sends correction bits $\mathbf{c}_{\mathsf{mult}}[m, e]$ that are inconsistent with the claimed value $f_e(\bar{\mathbf{u}}_m)$. The VOLE consistency check must therefore establish that all correlations share a consistent field-VOLE relation. We use a four-output $\mathbb{F}_{2^\lambda}$-linear consistency hash, analyzed in Section 10.6. The first $d_{\mathsf{zk}} - 1$ CRT-generated field-mask correlations are used to establish correctness of the QuickSilver proof of the witness relations, while the final 4 correlations hide the four outputs of the VOLE consistency hash.

6.2 Precomputed CRT Tables

To avoid having to implement the necessary arithmetic, we provide precomputed tables to store the necessary matrices and polynomials for computing (shares of) the bilinear product $u \cdot \Delta \in \mathbb{F}_{2^\lambda}$ for each parameter set. The algorithms in this section are specified assuming access to these tables, which are given in the `tables` folder of each reference implementation. In Appendix B, we explain how the tables can be reproduced and verified, and how an implementation may avoid them and implement the underlying arithmetic directly, if preferred.

- $\mathbf{F} \in \mathbb{F}_2^{n_{\mathsf{mult}} \times \lambda}$: linear map applied to the coefficient vector of a field mask. For $z \in \mathbb{F}_{2^\lambda}$, row e defines $f_e(z) := \langle \mathbf{F}[e], \mathsf{ToBits}(z; \lambda) \rangle \in \mathbb{F}_2$.
- $\mathbf{G} \in \mathbb{F}_2^{n_{\mathsf{mult}} \times (\lambda - w_{\mathsf{grind}})}$: linear map applied to the bits of the residues of Δ; row e is the map g_e, so that $g_e(\Delta) = \mathbf{G}[e] \cdot \Delta'$
- $\mathbf{W}_{\mathrm{tree}} \in \mathbb{F}_2^{\lambda \times (\lambda - w_{\mathsf{grind}})}$: linear map applied to the $\hat{u}_i \cdot \hat{\Delta}_i$ products from the GGM trees
- $\mathbf{W}_{\mathrm{gate}} \in \mathbb{F}_2^{\lambda \times n_{\mathsf{mult}}}$: linear map applied to the output of the n_{mult} AND gates
- $\mathbf{W}_{\mathrm{crt}} \in \mathbb{F}_2^{\lambda \times (\lambda - w_{\mathsf{grind}})}$: CRT map lifting $\lambda - w_{\mathsf{grind}}$ tree residue bits into the corresponding element of $\mathbb{F}_{2^\lambda}$ of degree less than $\lambda - w_{\mathsf{grind}}$. Used to lift Δ' to Δ, to assemble the mask values from their residues, and to lift the rows of $\mathbf{V}, \mathbf{Q}$ to field elements.
- Pairwise-coprime polynomial moduli $M_0, \ldots, M_{\tau-1}$ used for the small-VOLE instances, such that each $M_i \in \mathbb{F}_2[X]$ is irreducible of degree d_i and is represented as a coefficient vector in $\mathbb{F}_2^{d_i+1}$.
- Modulus $M_{\mathrm{tree}}(x) = \prod_{i=0}^{\tau-1} M_i(x)$, of degree $\sum d_i = \lambda - w_{\mathsf{grind}}$. Represented as a coefficient vector in $\mathbb{F}_2^{\lambda - w_{\mathsf{grind}} + 1}$.

These parameters satisfy the following properties:

1. *(CRT map)* For every column vector $\Delta' \in \mathbb{F}_2^{\lambda - w_{\mathsf{grind}}}$, let $\Delta_{\mathsf{bin}} := \mathbf{W}_{\mathrm{crt}} \cdot \Delta' \in \mathbb{F}_2^{\lambda}$. Then

$$\mathsf{ToBits}\big(\mathsf{ToPoly}(\Delta_{\mathsf{bin}}; \lambda) \bmod M_i(x); d_i\big) = \Delta'[\mathsf{start}_i..\mathsf{start}_i + d_i) \tag{2}$$

   for every $i \in [0..\tau)$, where $\mathsf{start}_i := \sum_{j=0}^{i-1} d_j$.
2. *($\mathbb{F}_{2^\lambda}$ multiplication)* For every $\bar{\mathbf{u}}_m \in \mathbb{F}_{2^\lambda}$ and $\Delta' \in \mathbb{F}_2^{\lambda - w_{\mathsf{grind}}}$, let $\Delta := \mathsf{ToField}(\mathbf{W}_{\mathrm{crt}} \cdot \Delta', \lambda) \in \mathbb{F}_{2^\lambda}$. Then

$$\mathsf{ToBits}(\bar{\mathbf{u}}_m \cdot \Delta; \lambda) = \mathbf{W}_{\mathrm{tree}} \cdot \mathsf{res}_\tau(\bar{\mathbf{u}}_m, \Delta) \oplus \mathbf{W}_{\mathrm{gate}} \cdot \big((\mathbf{F} \cdot \mathsf{ToBits}(\bar{\mathbf{u}}_m; \lambda)) * (\mathbf{G} \cdot \Delta')\big). \tag{3}$$

   Here,

$$\mathsf{res}_\tau(\bar{\mathbf{u}}_m, \Delta) := \Big(\mathsf{ToBits}(\hat{u}_m \hat{\Delta} \bmod M_0(x); d_0) \,\|\, \cdots \,\|\, \mathsf{ToBits}(\hat{u}_m \hat{\Delta} \bmod M_{\tau-1}(x); d_{\tau-1})\Big)^\top,$$

   where $\hat{u}_m$ and $\hat{\Delta}$ are the polynomial representatives of $\bar{\mathbf{u}}_m$ and Δ, and $*$ denotes component-wise AND.

To confirm correctness of the tables, these properties can be easily verified by testing on basis vectors, as detailed in Appendix B.6.

6.3 VOLECommit and VOLEReconstruct Algorithms

We present these algorithms — for committing to and reconstructing both the witness bits and the mask values — in Figure 6.2 and Figure 6.3.

Matrix Layout. The VOLE output by ConvertToVOLE, stored in the signer vectors $\mathbf{u}_i$, signer matrices $\mathbf{V}_i$, and verifier matrices $\mathbf{Q}_i$, has length $\hat{\ell} = \ell + n_{\mathsf{mask}} d_0$, organised as shown in Figure 6.1. Rows $[0..\ell)$ are the *witness rows*, used to commit to the extended witness; as before, the τ small-VOLE instances are corrected to agree with the same secret $\mathbf{u}$ on these rows, via the correction vectors $\mathbf{c}_i$. The remaining *mask rows* form n_{mask} blocks of d_0 rows each, the m-th block defining the mask $\bar{\mathbf{u}}_m$. Within a block, small-VOLE instance i uses its first $d_i \le d_0$ rows, whose output defines the bits of $\bar{\mathbf{u}}_m$ mod M_i; the remaining $d_0 - d_i$ rows of the shorter small-VOLE instances are unused. When $\tau_1 = 0$, every instance has $d_i = d_0$ and the block is exactly full. This allows the parties to obtain shares of $\bar{\mathbf{u}}_m \cdot \Delta \bmod M_i$. The full field element $\bar{\mathbf{u}}_m \in \mathbb{F}_{2^\lambda}$ is recovered by CRT interpolation, with the addition of the quotient term whose coefficient vector is $\mathbf{u}_{\mathsf{hi}}^{(m)}$.

In the figures, $\mathbf{V}[j]$ is the j-th row of a matrix, $\mathbf{V}|_{[a..b)}$ the restriction of a matrix to rows $[a..b)$, and $[\boldsymbol{a}_0 \cdots \boldsymbol{a}_{n-1}]$ the matrix with the given columns. We identify a bit-string in $\{0,1\}^n$ with the corresponding vector in $\mathbb{F}_2^n$.

VOLECommit. The algorithm proceeds in four steps. (1) As before, it runs BAVC.Commit and converts each tree into a small VOLE of length $\hat{\ell}$ via ConvertToVOLE; the witness rows are corrected to the secret $\mathbf{u} := \boldsymbol{u}_0[0..\ell)$ via $\mathbf{c}_i$. (2) It then samples the upper mask bits $\mathbf{u}_{\mathsf{hi}}^{(0)}, \ldots, \mathbf{u}_{\mathsf{hi}}^{(n_{\mathsf{mask}}-1)}$ from $n_{\mathsf{mask}} \cdot w_{\mathsf{grind}}$ congiguous bits output by PRG. It then reads the residue bits $\mathbf{u}_{\mathsf{low}}^{(m)}$ from the relevant part of each tree's slice of the mask window and assembles these into the mask value $\bar{\mathbf{u}}_m$; the signer's v-shares $\tilde{\mathbf{r}}_{m,i}$ of the products modulo M_i are extracted similarly from the $\mathbf{V}$ matrix (with the CRT step postponed until later). (3) Using the free-XOR wire labels $\mathbf{a}_j$, taken from the first λ rows of the $\mathbf{V}$ matrix, we then define for each gate e the two candidate labels L_e^0, L_e^1 that authenticate the bit

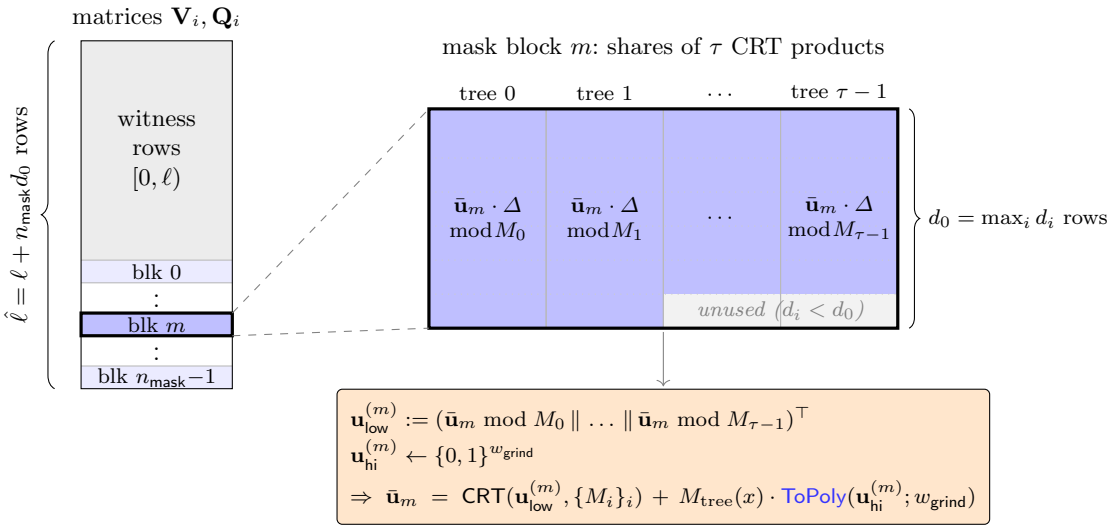

Fig. 6.1: Structure of the CRT-encoded mask shares stored in the VOLE matrices $\mathbf{V}_i$ and $\mathbf{Q}_i$

$\gamma_e = \mathbf{G}[e] \cdot \Delta'$, and hash these with H_5 to derive the gate correction bit $\mathbf{c}_{\mathsf{mult}}[m, e]$ that derandomizes the resulting $\mathbb{F}_2$-VOLE to the gate value $\mathbf{F}[e] \cdot \bar{\mathbf{u}}_m$. (4) Finally, it recombines the tree and gate v-shares into $\bar{\mathbf{v}}_m$ via $\mathbf{W}_{\mathrm{tree}}, \mathbf{W}_{\mathrm{gate}}$, and applies the CRT lift $\mathbf{W}_{\mathrm{crt}}$ row-wise to the common part of $\mathbf{V}$, so that the output VOLE relations hold directly over $\mathbb{F}_{2^\lambda}$.

VOLEReconstruct. The verifier mirrors these steps with its q-shares: after $\mathsf{DecodeAllChall}_3$ and BAVC.Reconstruct, it runs ConvertToVOLE and corrects the witness rows as before. It then assembles its shares $\tilde{\mathbf{r}}'_{m,i}$ of the small-VOLE products from the mask rows, recomputes each gate's wire label from the (corrected) columns $\mathbf{a}'_j$ of $\mathbf{Q}$, obtaining the label $L_e^{\gamma_e}$ for $\gamma_e = \mathbf{G}[e] \cdot \Delta'$, and obtains its gate shares by hashing and applying the corrections $\mathbf{c}_{\mathsf{mult}}$. Finally, the verifier applies the maps $\mathbf{W}_{\mathrm{tree}}, \mathbf{W}_{\mathrm{gate}}$ to get the mask shares and then the CRT map to the witness rows of $\mathbf{Q}$ to get the witness product shares.

Correctness of the CRT-based VOLE commitment.

Proposition 6.1. *Let*

$$\begin{pmatrix} \mathsf{com}, \mathsf{decom}, (\mathbf{c}_i)_{i\in[1..\tau)}, \mathbf{c}_{\mathsf{mult}}, \\ \mathbf{u}, \mathbf{V}, (\bar{\mathbf{u}}_m, \bar{\mathbf{v}}_m)_{m\in[0..n_{\mathsf{mask}})} \end{pmatrix} \leftarrow \mathsf{FAEST.VOLECommit}(r, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}}),$$

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

for $\mathsf{chall}_3 \in \{0,1\}^\lambda$ *and* $\hat{\ell} = \ell + n_{\mathsf{mask}} \cdot d_0$ *and suppose that*

$$\mathsf{decom}_I \leftarrow \mathsf{BAVC.Open}(\mathsf{decom}, I; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}}) \text{ with } \mathsf{decom}_I \neq \bot.$$

If

$$(\mathsf{com}', \mathbf{Q}, (\bar{\mathbf{q}}_m)_{m\in[0..n_{\mathsf{mask}})}, \Delta) \leftarrow$$
$$\mathsf{FAEST.VOLEReconstruct}(\mathsf{chall}_3, \mathsf{decom}_I, (\mathbf{c}_i)_{i\in[1..\tau)}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$$

then $\mathsf{com}' = \mathsf{com}$ *and*

$$\mathbf{Q}[j] = \mathbf{V}[j] + \mathbf{u}[j] \cdot \Delta \qquad \text{for every } j \in [0..\ell),$$

and

$$\bar{\mathbf{q}}_m = \bar{\mathbf{v}}_m + \bar{\mathbf{u}}_m \cdot \Delta \qquad \text{for every } m \in [0..n_{\mathsf{mask}}).$$

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: $\mathsf{com} \in \{0,1\}^{2\lambda}$, $\mathsf{decom} \in \{0,1\}^{(2L-1)\lambda + n_{\mathsf{leafcom}}\lambda L}$, $(\mathbf{c}_i)_{i\in[1..\tau)} \in (\{0,1\}^{\ell})^{\tau-1}$, $\mathbf{c}_{\mathsf{mult}} \in \{0,1\}^{n_{\mathsf{mask}}\times \bar{n}_{\mathsf{mult}}}$,
$\mathbf{u} \in \{0,1\}^{\ell}$, $\mathbf{V} \in \mathbb{F}_{2^\lambda}^{\ell}$, $(\bar{\mathbf{u}}_m, \bar{\mathbf{v}}_m)_{m\in[0..n_{\mathsf{mask}})} \in (\mathbb{F}_{2^\lambda} \times \mathbb{F}_{2^\lambda})^{n_{\mathsf{mask}}}$

// Small VOLEs from the GGM trees, as before (recall $\hat{\ell} = \ell + n_{\mathsf{mask}} d_0$)

$(\mathsf{com}, \mathsf{decom}, (\mathsf{sd}_{i,0}, \dots, \mathsf{sd}_{i,N_i-1})_{i\in[0..\tau)}) \leftarrow \mathsf{BAVC.Commit}(r, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$

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

  $(\boldsymbol{u}_i, \mathbf{v}_{i,0}, \dots, \mathbf{v}_{i,d_i-1}) := \mathsf{ConvertToVOLE}(\mathsf{sd}_{i,0}, \dots, \mathsf{sd}_{i,N_i-1}, \mathsf{iv}, i + 2^{31}; \hat{\ell}) \in (\mathbb{F}_2^{\hat{\ell}})^{1+d_i}$

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

// Correct the witness rows of all trees to the secret $\mathbf{u}$

$\mathbf{u} := \boldsymbol{u}_0[0..\ell) \in \mathbb{F}_2^{\ell}$

for $i \in [1..\tau)$ **do**

  $\mathbf{c}_i := \mathbf{u} \oplus \boldsymbol{u}_i[0..\ell) \in \mathbb{F}_2^{\ell}$

// Derive mask values and first τ product shares; each $\mathbf{u}_{\mathsf{hi}}^{(m)} \in \mathbb{F}_2^{w_{\mathsf{grind}}}$

$(\mathbf{u}_{\mathsf{hi}}^{(0)}, \dots, \mathbf{u}_{\mathsf{hi}}^{(n_{\mathsf{mask}}-1)}) := \mathsf{PRG}(r, \mathsf{iv}, 2^{31}-1; n_{\mathsf{mask}} \cdot w_{\mathsf{grind}}) \in \mathbb{F}_2^{n_{\mathsf{mask}} \cdot w_{\mathsf{grind}}}$ // mask top bits

for $m \in [0..n_{\mathsf{mask}})$ **do**

  $\ell_m := \ell + m \cdot d_0$

  $\mathbf{u}_{\mathsf{low}}^{(m)} := \big(\boldsymbol{u}_0[\ell_m..\ell_m + d_0) \,\|\, \boldsymbol{u}_1[\ell_m..\ell_m + d_1) \,\|\, \cdots \,\|\, \boldsymbol{u}_{\tau-1}[\ell_m..\ell_m + d_{\tau-1})\big)^{\top} \in \mathbb{F}_2^{\lambda - w_{\mathsf{grind}}}$

  $\bar{\mathbf{u}}_m := \mathsf{ToField}\Big(\mathbf{W}_{\mathsf{crt}} \cdot \mathbf{u}_{\mathsf{low}}^{(m)} \oplus \mathsf{ToBits}\big(M_{\mathsf{tree}}(x) \cdot \mathsf{ToPoly}(\mathbf{u}_{\mathsf{hi}}^{(m)}; w_{\mathsf{grind}}); \lambda\big), \lambda\Big) \in \mathbb{F}_{2^\lambda}$

  for $i \in [0..\tau)$ **do** // v-shares of the products $\bar{u}_m \cdot \hat{\Delta}_i \bmod M_i$

    // Each row $\mathbf{V}_i[\ell_m + t]$ authenticates one coeff. of $\bar{u}_m \bmod M_i$

    $\tilde{\mathbf{r}}_{m,i} := \mathsf{ToBits}\Big(\sum_{t=0}^{d_i-1} x^t \cdot \mathsf{ToPoly}\big(\mathbf{V}_i[\ell_m + t]; d_i\big) \bmod M_i(x); d_i\Big) \in \mathbb{F}_2^{d_i}$

// Free-XOR wire labels: columns of the first λ rows of $\mathbf{V}$

$(\mathbf{a}_0, \cdots, \mathbf{a}_{\lambda - w_{\mathsf{grind}}-1}) := [\mathbf{V}_0 \cdots \mathbf{V}_{\tau-1}]|_{[0..\lambda)} \in (\mathbb{F}_2^{\lambda})^{\lambda - w_{\mathsf{grind}}}$ // $\mathbf{a}_j$: label for bit j of Δ'

// Define the gate labels and hash them to mask the AND gates

for $e \in [0..n_{\mathsf{mult}})$ **do**

  $L_e^0 := \bigoplus_{j\,:\,\mathbf{G}[e][j]=1} \mathbf{a}_j \in \mathbb{F}_2^{\lambda}, \quad L_e^1 := L_e^0 \oplus \mathbf{u}[0..\lambda) \in \mathbb{F}_2^{\lambda}$

  $\boldsymbol{h}_e^0 := \mathsf{H}_5(\mathsf{iv}, e, L_e^0) \in \mathbb{F}_2^{n_{\mathsf{mask}}}, \quad \boldsymbol{h}_e^1 := \mathsf{H}_5(\mathsf{iv}, e, L_e^1) \in \mathbb{F}_2^{n_{\mathsf{mask}}}$

  for $m \in [0..n_{\mathsf{mask}})$ **do** // gate v-shares, and correction bits for the values $f_e(\bar{\mathbf{u}}_m)$

    $\tilde{v}_{m,e} := \boldsymbol{h}_e^0[m] \in \mathbb{F}_2$

    $\mathbf{c}_{\mathsf{mult}}[m, e] := \tilde{v}_{m,e} \oplus \boldsymbol{h}_e^1[m] \oplus f_e(\bar{\mathbf{u}}_m) \in \mathbb{F}_2$

$\mathbf{c}_{\mathsf{mult}}[m, e] := 0$, **for** $m \in [0..n_{\mathsf{mask}})$, $e \in [n_{\mathsf{mult}}..\bar{n}_{\mathsf{mult}})$ // Zero-pad to $\lceil n_{\mathsf{mult}}/8 \rceil$ bytes

// Recombine the v-shares of $\bar{u}_m \cdot \Delta$

for $m \in [0..n_{\mathsf{mask}})$ **do**

  $\bar{\mathbf{v}}_m := \mathsf{ToField}\Big(\mathbf{W}_{\mathsf{tree}} \cdot (\tilde{\mathbf{r}}_{m,0} \| \cdots \| \tilde{\mathbf{r}}_{m,\tau-1})^{\top} \oplus \mathbf{W}_{\mathsf{gate}} \cdot (\tilde{v}_{m,0}, \dots, \tilde{v}_{m,n_{\mathsf{mult}}-1})^{\top}, \lambda\Big) \in \mathbb{F}_{2^\lambda}$

$\mathbf{V}_{\mathsf{bin}} := \big([\mathbf{V}_0 \cdots \mathbf{V}_{\tau-1}]|_{[0..\ell)}\big) \cdot \mathbf{W}_{\mathsf{crt}}^{\top} \in \mathbb{F}_2^{\ell\times\lambda}$ // row-wise CRT lift

$\mathbf{V} := \big(\mathsf{ToField}(\mathbf{V}_{\mathsf{bin}}[j], \lambda)\big)_{j\in[0..\ell)} \in \mathbb{F}_{2^\lambda}^{\ell}$

return $(\mathsf{com}, \mathsf{decom}, \mathbf{c}_1, \dots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathbf{u}, \mathbf{V}, (\bar{\mathbf{u}}_m, \bar{\mathbf{v}}_m)_{m\in[0..n_{\mathsf{mask}})})$

Fig. 6.2: FAEST VOLE commitments with CRT-based masks

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

INPUT: $\mathsf{chall}_3 \in \{0,1\}^{\lambda}, \mathsf{decom}_I \in \{0,1\}^{n_{\mathsf{leafcom}}\lambda\tau + T_{\mathsf{open}}\lambda}, \mathbf{c}_i \in \{0,1\}^{\ell}, \mathbf{c}_{\mathsf{mult}} \in \{0,1\}^{n_{\mathsf{mask}} \times \bar{n}_{\mathsf{mult}}}, \mathsf{iv} \in \{0,1\}^{128}$
OUTPUT: $\mathsf{com} \in \{0,1\}^{2\lambda}, \mathbf{Q} \in \mathbb{F}_{2^\lambda}^{\ell}, (\bar{\mathbf{q}}_m)_{m \in [0..n_{\mathsf{mask}})} \in \mathbb{F}_{2^\lambda}^{n_{\mathsf{mask}}}, \Delta \in \mathbb{F}_{2^\lambda}$ or $\perp$

$I = (\Delta_0, \dots, \Delta_{\tau-1}) \leftarrow \mathsf{DecodeAllChall}_3(\mathsf{chall}_3; \mathtt{param}) \in \big([0..N_0) \times \cdots \times [0..N_{\tau-1})\big) \cup \{\perp\}$
if $I = \perp$ **then return** $\perp$
$\mathsf{rec} \leftarrow \mathsf{BAVC.Reconstruct}(\mathsf{decom}_I, I, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$
if $\mathsf{rec} = \perp$ **then return** $\perp$
$\Big(\mathsf{com}, \big((\mathsf{sd}_{i,j})_{j \in [0..N_i) \setminus \{\Delta_i\}}\big)_{i \in [0..\tau)}\Big) \leftarrow \mathsf{rec}$
for $i \in [0..\tau)$ **do**
  for $j \in [1..N_i)$ **do** $\mathsf{sd}'_{i,j} \leftarrow \mathsf{sd}_{i,j \oplus \Delta_i}$ // XOR j and Δ_i as bit-strings
  $(\cdot, \mathbf{q}_{i,0}, \dots, \mathbf{q}_{i,d_i-1}) := \mathsf{ConvertToVOLE}(\perp, \mathsf{sd}'_{i,1}, \dots, \mathsf{sd}'_{i,N_i-1}, \mathsf{iv}, i + 2^{31}; \hat{\ell}) \in (\mathbb{F}_2^{\hat{\ell}})^{d_i+1}$
  $(\delta_{i,0}, \dots, \delta_{i,d_i-1}) \leftarrow \mathsf{BitDec}(\Delta_i, d_i) \in \mathbb{F}_2^{d_i}$
  $\mathbf{Q}_i := [\mathbf{q}_{i,0} \cdots \mathbf{q}_{i,d_i-1}] \in \mathbb{F}_2^{\hat{\ell} \times d_i}$ // stored in column major representation
// Correct the witness rows; mask rows receive no correction
  if $i \geq 1$ **then** $\mathbf{Q}_i|_{[0..\ell)} \leftarrow \mathbf{Q}_i|_{[0..\ell)} \oplus [\delta_{i,0} \cdot \mathbf{c}_i \cdots \delta_{i,d_i-1} \cdot \mathbf{c}_i]$
$\Delta' := (\delta_{0,0}, \dots, \delta_{0,d_0-1}, \ \dots, \ \delta_{\tau-1,0}, \dots, \delta_{\tau-1,d_{\tau-1}-1})^{\top} \in \mathbb{F}_2^{\lambda - w_{\mathsf{grind}}}$
$\Delta := \mathsf{ToField}(\mathbf{W}_{\mathrm{crt}} \cdot \Delta', \lambda) \in \mathbb{F}_{2^\lambda}$
// q-shares of the first τ small products, from the mask rows
for $m \in [0..n_{\mathsf{mask}})$ **do**
  $\ell_m := \ell + m \cdot d_0$
  for $i \in [0..\tau)$ **do** // q-shares of the products $\bar{u}_m \cdot \hat{\Delta}_i \bmod M_i$
    $\tilde{\mathbf{r}}'_{m,i} := \mathsf{ToBits}\Big(\sum_{t=0}^{d_i-1} x^t \cdot \mathsf{ToPoly}\big(\mathbf{Q}_i[\ell_m + t]; d_i\big) \bmod M_i(x); d_i\Big) \in \mathbb{F}_2^{d_i}$
// Free-XOR wire labels: columns of the first λ rows of $\mathbf{Q}$
$(\mathbf{a}'_0, \cdots, \mathbf{a}'_{\lambda - w_{\mathsf{grind}}-1}) := [\mathbf{Q}_0 \cdots \mathbf{Q}_{\tau-1}]|_{[0..\lambda)} \in (\mathbb{F}_2^{\lambda})^{\lambda - w_{\mathsf{grind}}}$ // $\mathbf{a}'_j = \mathbf{a}_j \oplus \Delta'[j] \cdot \mathbf{u}[0..\lambda)$
// Recompute the gate labels and hash them to get the gate q-shares
for $e \in [0..n_{\mathsf{mult}})$ **do**
  $\gamma_e := \mathbf{G}[e] \cdot \Delta' \in \mathbb{F}_2, \quad L'_e := \bigoplus_{j \,:\, \mathbf{G}[e][j]=1} \mathbf{a}'_j \in \mathbb{F}_2^{\lambda}$ // $L'_e = L_e^{\gamma_e}$
  $\boldsymbol{h}_e := \mathsf{H}_5(\mathsf{iv}, e, L'_e) \in \mathbb{F}_2^{n_{\mathsf{mask}}}$
  for $m \in [0..n_{\mathsf{mask}})$ **do**
    $\tilde{q}_{m,e} := \boldsymbol{h}_e[m] \oplus \gamma_e \cdot \mathbf{c}_{\mathsf{mult}}[m, e] \in \mathbb{F}_2$
// Recombine the q-shares of $\bar{u}_m \cdot \Delta$
for $m \in [0..n_{\mathsf{mask}})$ **do**
  $\bar{\mathbf{q}}_m := \mathsf{ToField}\Big(\mathbf{W}_{\mathrm{tree}} \cdot (\tilde{\mathbf{r}}'_{m,0} \| \cdots \| \tilde{\mathbf{r}}'_{m,\tau-1})^{\top} \oplus \mathbf{W}_{\mathrm{gate}} \cdot (\tilde{q}_{m,0}, \dots, \tilde{q}_{m,n_{\mathsf{mult}}-1})^{\top}, \lambda\Big) \in \mathbb{F}_{2^\lambda}$
$\mathbf{Q}_{\mathsf{bin}} := \big([\mathbf{Q}_0 \cdots \mathbf{Q}_{\tau-1}]|_{[0..\ell)}\big) \cdot \mathbf{W}_{\mathrm{crt}}^{\top} \in \mathbb{F}_2^{\ell \times \lambda}$ // row-wise CRT lift
$\mathbf{Q} := \big(\mathsf{ToField}(\mathbf{Q}_{\mathsf{bin}}[j], \lambda)\big)_{j \in [0..\ell)} \in \mathbb{F}_{2^\lambda}^{\ell}$
return $(\mathsf{com}, \mathbf{Q}, (\bar{\mathbf{q}}_m)_{m \in [0..n_{\mathsf{mask}})}, \Delta)$

Fig. 6.3: FAEST VOLE reconstruction with CRT-based masks

Proof. By Proposition 5.1, the verifier reconstructs the same commitment $\mathsf{com}' = \mathsf{com}$ and all the non-hidden leaf seeds, i.e. all seeds $\mathsf{sd}_{i,j}$ where $j \neq \Delta_i$. In particular, BAVC.Reconstruct does not return $\perp$ on this opening.

VOLEReconstruct applies ConvertToVOLE to every independent small-VOLE instance. By assumption, VOLECommit and VOLEReconstruct use the same iv and the same tweak $i + 2^{31}$ in the two executions of ConvertToVOLE. Let $(\delta_{i,0}, \ldots, \delta_{i,d_i-1}) := \mathsf{BitDec}(\Delta_i, d_i)$ for every $i \in [0..\tau)$. Then by Proposition 5.2, for every $t \in [0..d_i)$ we have that

$$\mathbf{q}_{i,t} = \mathbf{v}_{i,t} \oplus \delta_{i,t} \cdot \boldsymbol{u}_i.$$

Note that these are $\mathbf{v}_{i,t}, \mathbf{u}_i, \mathbf{q}_{i,t}$ as directly output by ConvertToVOLE in VOLECommit, VOLEReconstruct, i.e. before correcting for the same witness.

Set $\mathbf{c}_0 := 0^\ell$, by the definition of the correction vectors, we have $\boldsymbol{u}_i[0..\ell) \oplus \mathbf{c}_i = \mathbf{u}$ for every $i \in [0..\tau)$ so, after applying the correction in line 12, the t-th witness column of instance i satisfies

$$\mathbf{q}^{\mathsf{corr}}_{i,t}[0..\ell) = \mathbf{v}_{i,t}[0..\ell) \oplus \delta_{i,t} \cdot \mathbf{u}.$$

Let $\mathbf{V}' := [\mathbf{V}_0 \cdots \mathbf{V}_{\tau-1}]|_{[0..\ell)}$ and $\mathbf{Q}' := [\mathbf{Q}_0 \cdots \mathbf{Q}_{\tau-1}]|_{[0..\ell)}$, for $\Delta' := (\delta_{0,0}, \ldots, \delta_{0,d_0-1}, \ldots, \delta_{\tau-1,0}, \ldots, \delta_{\tau-1,d_{\tau-1}-1})^\top$, it follows for ever row $j \in [0..\ell)$ that

$$\mathbf{Q}'[j] = \mathbf{V}'[j] \oplus \mathbf{u}[j] \cdot (\Delta')^\top$$

where $\mathbf{u}[j] \cdot (\Delta')^\top$. By lines 31 & 32 in VOLECommit and 31 & 32 in VOLEReconstruct, we have that

$$\mathbf{V}[j] = \mathsf{ToField}\big(\mathbf{V}'[j]\mathbf{W}_{\mathrm{crt}}^\top, \lambda\big), \qquad \mathbf{Q}[j] = \mathsf{ToField}\big(\mathbf{Q}'[j]\mathbf{W}_{\mathrm{crt}}^\top, \lambda\big).$$

Since ToField is $\mathbb{F}_2$-linear, $\mathbf{u}[j] \in \mathbb{F}_2$ and $(\Delta')^\top \mathbf{W}_{\mathrm{crt}}^\top = (\mathbf{W}_{\mathrm{crt}}\Delta')^\top$,

$$\mathbf{Q}[j] = \mathsf{ToField}\big(\mathbf{V}'[j]\mathbf{W}_{\mathrm{crt}}^\top, \lambda\big) + \mathbf{u}[j] \cdot \mathsf{ToField}\big((\Delta')^\top \mathbf{W}_{\mathrm{crt}}^\top, \lambda\big) = \mathbf{V}[j] + \mathbf{u}[j] \cdot \Delta,$$

therefore,

$$\mathbf{Q}[j] = \mathbf{V}[j] + \mathbf{u}[j] \cdot \Delta \qquad \text{for every } j \in [0..\ell),$$

where $\Delta = \mathsf{ToField}(\mathbf{W}_{\mathrm{crt}} \cdot \Delta', \lambda)$.

It remains to prove the mask relation $\bar{\mathbf{q}}_m = \bar{\mathbf{v}}_m + \bar{\mathbf{u}}_m \cdot \Delta$. Let $\hat{u}_m$ and $\hat{\Delta}$ be the polynomial representatives of $\bar{\mathbf{u}}_m$ and Δ. Given $m \in [0..n_{\mathsf{mask}})$ and $i \in [0..\tau)$, and setting $\ell_m := \ell + md_0$, define

$$A_{m,i}(x) := \sum_{t=0}^{d_i-1} \boldsymbol{u}_i[\ell_m + t]x^t, \qquad D_i(x) := \sum_{t=0}^{d_i-1} \delta_{i,t}x^t.$$

By the definition of $\mathbf{u}^{(m)}_{\mathsf{low}}$, the CRT-map property, and the fact that $M_i(x)$ divides $M_{\mathrm{tree}}(x)$, the quotient term $\mathbf{u}^{(m)}_{\mathsf{hi}}$ used to complete $\bar{\mathbf{u}}_m$ vanishes modulo $M_i(x)$. Therefore,

$$A_{m,i}(x) = \hat{u}_m(x) \bmod M_i(x)$$

and, similarly, by the definition of Δ' and the CRT-map property,

$$D_i(x) = \hat{\Delta}(x) \bmod M_i(x).$$

Since the mask rows receive no witness correction, for every $t \in [0..d_i)$ we obtain from Proposition 5.2 that

$$\mathsf{ToPoly}\big(\mathbf{Q}_i[\ell_m + t]; d_i\big) = \mathsf{ToPoly}\big(\mathbf{V}_i[\ell_m + t]; d_i\big) + \boldsymbol{u}_i[\ell_m + t]D_i(x).$$

Then multiplying the equality for row t by x^t, summing over $t \in [0..d_i)$, and reducing modulo $M_i(x)$ gives

$$\sum_{t=0}^{d_i-1} x^t \,\mathsf{ToPoly}\big(\mathbf{Q}_i[\ell_m+t]; d_i\big) \equiv \sum_{t=0}^{d_i-1} x^t \,\mathsf{ToPoly}\big(\mathbf{V}_i[\ell_m+t]; d_i\big) + A_{m,i}(x)D_i(x) \pmod{M_i(x)}.$$

By the definitions of $\tilde{\mathbf{r}}_{m,i}$ and $\tilde{\mathbf{r}}'_{m,i}$ and because ToBits is additively homomorphic, it follows that

$$\tilde{\mathbf{r}}'_{m,i} = \tilde{\mathbf{r}}_{m,i} \oplus \mathsf{ToBits}\big((\hat{u}_m \bmod M_i(x))(\hat{\Delta} \bmod M_i(x)) \bmod M_i(x); d_i\big).$$

For every challenge-bit position j, the VOLE column relation yields

$$\mathbf{a}'_j = \mathbf{a}_j \oplus \Delta'[j] \cdot \mathbf{u}[0..\lambda)$$

by Proposition 5.2. For every gate $e \in [0..n_{\mathsf{mult}})$, we define $\gamma_e := \mathbf{G}[e] \cdot \Delta'$. Then taking the linear combination selected by row $\mathbf{G}[e]$ therefore gives

$$L'_e = L_e^0 \oplus \gamma_e \cdot \mathbf{u}[0..\lambda) = L_e^{\gamma_e}.$$

In this way the verifier obtains $\boldsymbol{h}_e = \boldsymbol{h}_e^{\gamma_e}$ and applies the correction value as

$$\tilde{q}_{m,e} = \boldsymbol{h}_e^{\gamma_e}[m] \oplus \gamma_e \cdot \mathbf{c}_{\mathsf{mult}}[m, e].$$

If $\gamma_e = 0$, then $\tilde{q}_{m,e} = \boldsymbol{h}_e^0[m] = \tilde{v}_{m,e}$, otherwise if $\gamma_e = 1$, then the definition of $\mathbf{c}_{\mathsf{mult}}[m, e]$ gives

$$\tilde{q}_{m,e} = \boldsymbol{h}_e^1[m] \oplus \tilde{v}_{m,e} \oplus \boldsymbol{h}_e^1[m] \oplus f_e(\bar{\mathbf{u}}_m) = \tilde{v}_{m,e} \oplus f_e(\bar{\mathbf{u}}_m)$$

and therefore

$$\tilde{q}_{m,e} = \tilde{v}_{m,e} \oplus \gamma_e \cdot f_e(\bar{\mathbf{u}}_m).$$

Define

$$\mathsf{res}_\tau(\bar{\mathbf{u}}_m, \Delta) := \big(\mathsf{ToBits}\big((\hat{u}_m \bmod M_i(x))(\hat{\Delta} \bmod M_i(x)) \bmod M_i(x); d_i\big)\big)_{i \in \{0,\dots,\tau-1\}}.$$

By the definitions of $\bar{\mathbf{v}}_m$ and $\bar{\mathbf{q}}_m$, by $\mathbf{x} = \mathsf{ToBits}(\mathsf{ToField}(\mathbf{x}, \lambda), \lambda)$ for $\mathbf{x} \in \{0,1\}^\lambda$ and by the $\mathbb{F}_2$-linearity of $\mathsf{ToField}$, we obtain

$$\begin{aligned}\mathsf{ToBits}(\bar{\mathbf{q}}_m + \bar{\mathbf{v}}_m; \lambda) &= \mathbf{W}_{\text{tree}} \cdot \mathsf{res}_\tau(\bar{\mathbf{u}}_m, \Delta) \oplus \mathbf{W}_{\text{gate}} \cdot (\gamma_e \cdot f_e(\bar{\mathbf{u}}_m))_{e \in \{0,\dots,n_{\mathsf{mult}}-1\}} \\ &= \mathbf{W}_{\text{tree}} \cdot \mathsf{res}_\tau(\bar{\mathbf{u}}_m, \Delta) \oplus \mathbf{W}_{\text{gate}} \cdot \big(\big(\mathbf{F} \cdot \mathsf{ToBits}(\bar{\mathbf{u}}_m; \lambda)\big) * \big(\mathbf{G} \cdot \Delta'\big)\big)\end{aligned}$$

Finally, by the bilinear identity (3), the right-hand side is $\mathsf{ToBits}(\bar{\mathbf{u}}_m \cdot \Delta; \lambda)$ and hence $\bar{\mathbf{q}}_m = \bar{\mathbf{v}}_m + \bar{\mathbf{u}}_m \cdot \Delta$, for every $m \in [0..n_{\mathsf{mask}})$. □

7 Zero-Knowledge Constraints for AES and Rijndael

This section specifies how to compute the QuickSilver proof of knowledge of the OWF for 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.

7.1 VOLE-ZK Operations

We use the notation $\langle x \rangle_e$ to denote a commitment to a value $x \in \mathbb{F}_{2^e}$, where $e \in \{1, 8, \lambda\}$, through VOLE. Note that this x exists in a subfield, but will for technical reasons in an algorithm be represented as an element in $\mathbb{F}_{2^\lambda}$ once $e > 1$.

In the full correlation, 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 7.1 (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 7.1.

Remark 7.2.

- When computing constraints, we usually end up with some commitment $\langle z \rangle_e^d$ 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 $\langle z \rangle_e^d$, the signer can skip computation of the degree-d coefficient.
- By convention, if superscript d is omitted, we assume $d = 1$. Similarly, if subscript e is omitted, we assume $e = 1$.

Lemma 7.3. *Let $\langle x \rangle_e^d$ be a VOLE commitment with commitment polynomial $\rho_x(X)$, and let $S \subseteq \mathbb{F}_{2^\lambda}$ be nonempty. Assume that Δ is uniform in S and independent of $\rho'(X)$. Assume that $\mathcal{P}$ sends $\rho'(X) \in \mathbb{F}_{2^\lambda}[X]$ to $\mathcal{V}$ such that $\deg(\rho') \leq d$ but the coefficient of X^d in $\rho'(X)$ is not x. Let $\mathcal{V}$ accept if $\rho'(\Delta) = \gamma_x$, and otherwise reject. Then $\mathcal{V}$ accepts with probability at most $d/|S|$.*

Proof. By definition, $\gamma_x = \rho_x(\Delta)$. Since the coefficient of X^d in $\rho'(X)$ is not x, the polynomial $\rho'(X) - \rho_x(X)$ is nonzero and has degree at most d. It therefore has at most d roots in $\mathbb{F}_{2^\lambda}$, and hence at most d roots in S. Since Δ is uniform in S and independent of $\rho'(X)$, the verifier accepts with probability at most $d/|S|$. □

7.2 An Overview of the OWF evaluation using VOLE Commitments

An overview of the OWF evaluation is also given in Figure 7.2. At the outset, we assume that the prover has committed to the individual bits of k as $\langle k_1 \rangle_1^1, \ldots, \langle k_{128} \rangle_1^1$.

Signer: VOLE-ZK operations	**Verifier:** VOLE-ZK operations
// Represent $\langle x\rangle_e^d$ as $(x_0,\dots,x_{d-1},x) \in \mathbb{F}_{2^\lambda}^d \times \mathbb{F}_{2^e}$,	// Represent $\langle x\rangle_e^d$ as $\gamma_x \in \mathbb{F}_{2^\lambda}$
// coefficients of $\rho_x(X) = x_0 + x_1X + \cdots + xX^d$	// 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** $(z_0,\dots,z_d)$ (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** $(z_0,\dots,z_{d_1+d_2-1},xy)$ (coeffs of ρ_z)	**return** $\gamma_x\cdot\gamma_y$

Fig. 7.1: VOLE-ZK operations. The extension degrees e, e_1, e_2 may be any combination in $\{1,\lambda\}$. The output degree d is at most d_{zk}.

Computing the key expansion. Towards computing KeyExpansion on the aforementioned commitments to obtain commitments $\langle\bar{k}_1\rangle_1^1,\dots,\langle\bar{k}_{\lambda(R+1)}\rangle_1^1$ to $\bar{\mathbf{k}}$, all operations besides SubWord can be performed directly on either $\langle k_1\rangle_1^1,\dots,\langle k_{128}\rangle_1^1$ 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 $\langle b_1\rangle_1^1,\dots,\langle b_{32}\rangle_1^1$. 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 $\langle b_1\rangle_1^1,\dots,\langle b_{32}\rangle_1^1$ to the outputs are actually consistent with the inputs a for SubWord (or rather, with the input commitments $\langle a_1\rangle_1^1,\dots,\langle a_{32}\rangle_1^1$), we perform the following operations:

1. Apply the inverse of step 2 of SubBytes on each byte of b, leading to commitments $\langle\bar{b}_1\rangle_1^1,\dots,\langle\bar{b}_{32}\rangle_1^1$. 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 $\langle\hat{b}\rangle_8^1 = \sum_{i=1}^8\langle\bar{b}_i\rangle_1^1\cdot\alpha^{i-1}$. 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 7.4. *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

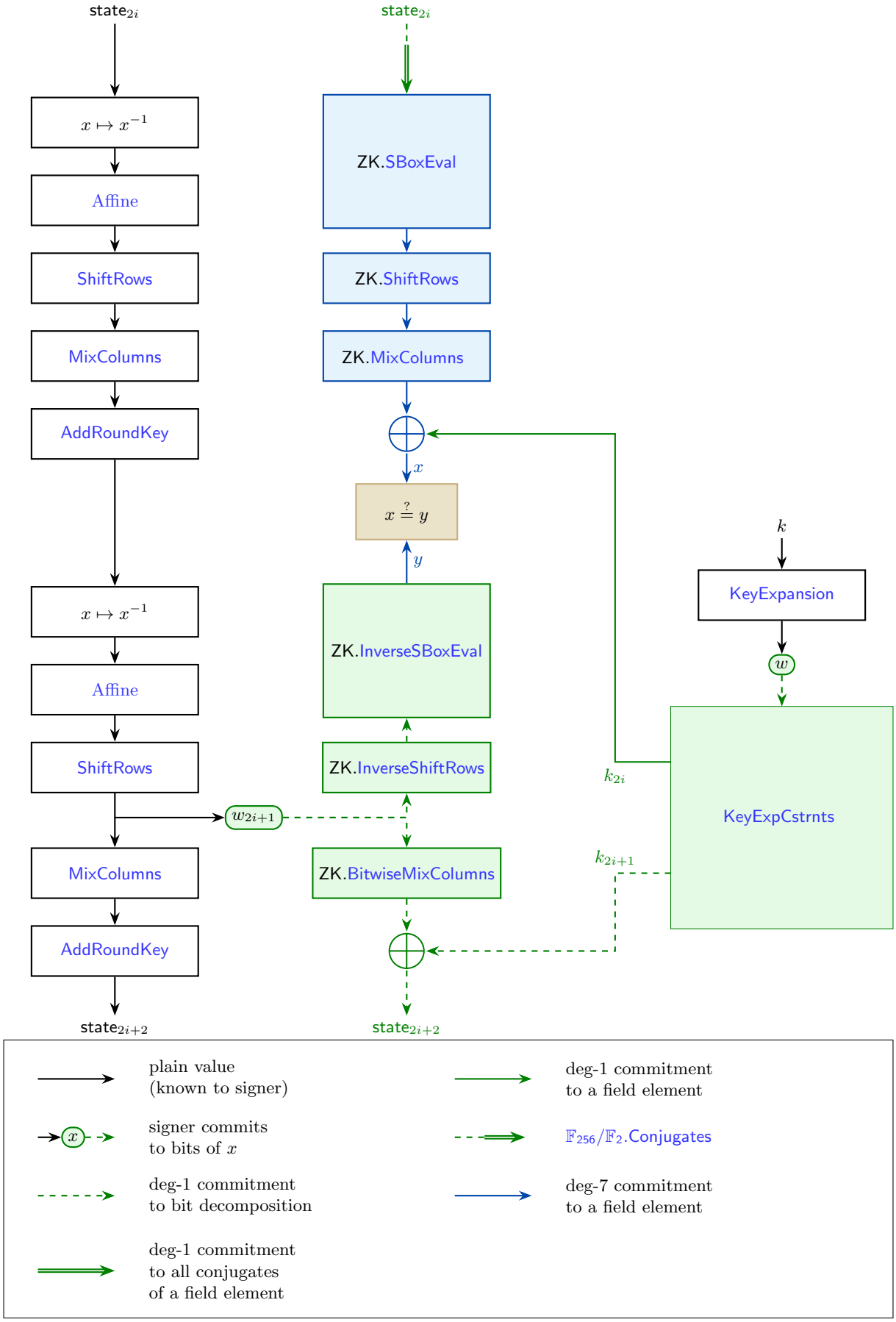

Fig. 7.2: A diagrammatic overview of the QuickSilver proof of knowledge of two consecutive AES rounds in FAEST. The FAEST-EM version is similar, except that the AES key is public, so the key schedule is evaluated in the clear, rather than on degree-1 VOLE commitments.

bits of the round key for each round are available as described above. We could 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 use a commitment to **out** (as being identical to y) as part of the consistency check.

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 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[14].

Matching round evaluations. We treat odd and even rounds differently, where we will commit to the full state in even rounds, which results in only having to commit to approximately $(R-1) \cdot 64$ bits. We then evaluate, from the committed state, one round of AES "forward" on the bit commitments of the state and the S-Box. Since each S-Box output can be written as a degree-7 polynomial in its inputs, this creates VOLE commitments of degree at most 7. We evaluate the round function until the input layer of the S-Boxes in the odd round, which does not increase the degree of the VOLE commitments further. In addition, we evaluate "backwards" from the committed state of the next even round (or the public output) using the bit commitment to the round keys. We again evaluate the non-linear part of the S-Box, now backwards, obtaining degree-7 VOLE commitments. Note that if commitments to the even round states are correct, then these degree-7 forward and backward commitments must now be commitments to the same secret.

Optimizing the S-Box evaluation. Evaluating the S-Boxes directly on the bit commitments is computationally suboptimal, as many VOLE commitments must be multiplied in the process. Towards implementing backward evaluation (which is simpler), observe that squaring $x \in \mathbb{F}_{2^8}$ (and, more generally, computing all Galois conjugates of x) is a linear operation on the bit decomposition. We thus compute $x^{-1} = x^{254}$ by precomputing the bit decompositions of the Galois conjugates, lifting those into $\mathbb{F}_{2^8}$ and computing the inverse using 6 multiplications. We thus obtain $\langle \cdot \rangle_8^7$ commitments to the S-Box inputs, which is sufficient to the check outlined above.

We then aim to use the same computational optimization to evaluate the S-Box forward, starting from a bit decomposition of its input. However, the SubBytes procedure consists of the inversion step, followed by an affine transformation on the bit decomposition of the inverse. However, as we now have $\langle \cdot \rangle_8^7$ commitments (and not to the bit decomposition), the affine step is not directly applicable. However, we can also compute the Galois conjugates of the S-Box outputs and use that any $\mathbb{F}_2$-linear function can be expressed as a $\mathbb{F}_{2^8}$-linear combination of Galois conjugates (Remark **??**). We then further lower the number of VOLE multiplications by analyzing which terms are re-used when computing the conjugates, and applying the affine step directly without explicitly computing all conjugates in full. Note that the AES round function after the S-Box is F_{2^8}-linear in these outputs up until the inputs of the next S-Boxes, and we can apply AddRoundKey, MixColumns and ShiftRows with minor modifications on the $\langle \cdot \rangle_8^7$ commitments.

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 7.6: FAEST.ExtendWitness in Figure 7.13

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

Signer.ConstantToVOLE($\mathbf{x} \in \mathbb{F}_2^n$)

1 : **return** $\langle \mathbf{x} \rangle := (\mathbf{0}, \mathbf{x}) \in \mathbb{F}_{2^\lambda}^n \times \mathbb{F}_2^n$

Verifier.ConstantToVOLE($\mathbf{x} \in \mathbb{F}_2^n$)

1 : // verifier holds $\Delta \in \mathbb{F}_{2^\lambda}$

2 : **return** $\langle \mathbf{x} \rangle := \mathbf{x} \cdot \Delta \in \mathbb{F}_{2^\lambda}^n$

Signer.DegLift($\langle \boldsymbol{z} \rangle_e^d; f$)

NEW: this function now lifts to higher degrees.

INPUT: Commitment of degree d, $f \in \mathbb{N}$

OUTPUT: Commitment of degree $d+f$

1 : // parse $\langle \boldsymbol{z} \rangle_e^d$ as poly $\boldsymbol{z}x^d + \sum_{i=0}^{d-1} \boldsymbol{z}_i x^i$,

2 : // where $\boldsymbol{z} = \mathbf{0} \in \mathbb{F}_{2^e}^n$ and $\boldsymbol{z}_i \in \mathbb{F}_{2^\lambda}^n$

3 : **return** $\langle \boldsymbol{z} \rangle_e^{d+f} := (\underbrace{\mathbf{0}, \ldots, \mathbf{0}}_{f \text{ times}}, \boldsymbol{z}_0, \ldots, \boldsymbol{z}_{d-1}, \mathbf{0}) \in (\mathbb{F}_{2^\lambda}^n)^{d+f} \times \mathbb{F}_{2^e}^n$

Verifier.DegLift($\langle \boldsymbol{z} \rangle_e^d; f$)

NEW: this function now lifts to higher degrees.

INPUT: Commitment of degree d, $f \in \mathbb{N}$

OUTPUT: Commitment of degree $d+f$

1 : // Parse $\langle \boldsymbol{z} \rangle_e^d$ as $\boldsymbol{\gamma}_x \in \mathbb{F}_{2^\lambda}^n$

2 : **return** $\langle \boldsymbol{z} \rangle_e^{d+f} := \boldsymbol{\gamma}_x \cdot \Delta^f \in \mathbb{F}_{2^\lambda}^n$

Fig. 7.3: Auxiliary operations used for constraints

– Section 7.7: FAEST.KeyExpFwd in Figure 7.14.
– Section 7.7: FAEST.KeyExpBkwd in Figure 7.15.
– Section 7.7: FAEST.KeyExpCstrnts in Figure 7.16.
– Section 7.8: FAEST.EncCstrnts in Figure 7.17.

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

7.3 Building Blocks for VOLE-ZK

In Figure 7.3, we have procedures ConstantToVOLE and DegLift. 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 $\langle \mathbf{x} \rangle^1$ 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.
- DegLift extends a degree-d commitment (resp. from the signer or verifier) to a degree-$d+f$ commitment, placing zeros in the unneeded coefficients. This is used when a procedure generates a deg $< d_{\mathsf{zk}}$ polynomial in $\mathbb{F}_{2^\lambda}[x]$ and we wish to embed it consistently in a deg $= d_{\mathsf{zk}}$ polynomial for the QuickSilver proof. The signer and verifier each have a slightly different way of "shifting" such polynomials.

Throughout the rest of this section, we will simply use ConstantToVOLE or DegLift directly in algorithms, and the implementation has to choose based on the context to use the Signer or Verifier version.

7.4 AES-related field arithmetic in VOLE-ZK

This section introduces some basic functions used to emulate the field arithmetic of the $\mathbb{F}_{2^8}$-field used in AES to be emulated in VOLE-ZK.

– ZK.ByteCombine (Figure 7.4): 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.
– SquareBits, SquareAndCombine (Figure 7.5): 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 outputs of SquareBits to the corresponding element in $\mathbb{F}_{2^8}$.

ZK.ByteCombine$(\langle x_0\rangle_1^d, \ldots, \langle x_7\rangle_1^d; \lambda)$

INPUT: $\langle x_i\rangle_1^d$: bit commitments representing an element in $\mathbb{F}_{2^8}$
OUTPUT: $\langle \boldsymbol{y}\rangle_8^d$: $\mathbb{F}_{2^8}$ commitment to encoded value
1 : **let** $\alpha_8 \in \mathbb{F}_{2^\lambda}$ // Basis element of $\mathbb{F}_{2^8}$ within $\mathbb{F}_{2^\lambda}$
2 : **return** $\langle \boldsymbol{y}\rangle_8^d := \sum_{i=0}^{7} \langle x_i\rangle_1^d \cdot \alpha_8^i$

Fig. 7.4: Lifting 8 committed bits to a committed byte in $\mathbb{F}_{2^\lambda}$.

– $\mathbb{F}_{256}/\mathbb{F}_2$.Conjugates (Figure 7.5): the algorithm computes the Galois conjugates (the powers x^{2^i}) 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.

7.5 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.

Affine Map on Bytes. The inputs to the affine map are VOLE commitments to bytes in $\mathbb{F}_{2^8}$, each packing exactly an inverted field element. As described in SubBytes, we must now apply an $\mathbb{F}_2$-linear operation on the individual bits. Using **??**, we can instead precompute public coefficients in $\mathbb{F}_{2^8}$ and express the linear function in terms of the coefficients and the Galois conjugates (i.e. Frobenius automorphisms) of the field element. These elements are exactly

$$(\chi_0, \ldots, \chi_8) = (\texttt{0x05}, \texttt{0x09}, \texttt{0xf9}, \texttt{0x25}, \texttt{0xf4}, \texttt{0x01}, \texttt{0xb5}, \texttt{0x8f}, \texttt{0x63})$$

and we can apply linear layer on $x \in \mathbb{F}_{2^8}$ as

$$\zeta_8 + \zeta_7 \cdot \sigma^7(x) + \zeta_6 \cdot \sigma^6(x) + \cdots + \zeta_0 \cdot x,$$

where $\zeta_i = \text{ByteCombine}(\chi_i; \lambda)$ lifts the constants to $\mathbb{F}_{2^\lambda}$.

Inverse Affine Map. The inverse affine map, Figure 7.6, 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.

7.5.1 S-Box Evaluation.

Inverse S-Box. The function ZK.InverseSBoxEval in Figure 7.7 obtains as inputs all the $N_{\text{st-bits}}$ bits of the state as VOLE commitments. It then first applies ZK.InverseAffine to apply the inverse affine layer of the S-Box, and then evaluates the map $x \mapsto x^{254}$ in $\mathbb{F}_{2^8}$. For each byte x of the state — given in bit-decomposed form — the ZK.$\mathbb{F}_{256}/\mathbb{F}_2$.Conjugates algorithm repeatedly applies the Frobenius map to compute all the 2^i-th powers of x, embedded into $\mathbb{F}_{2^\lambda}$. Using these, we multiply $x^2, \ldots, x^{2^7}$, yielding the result x^{254}.

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

Input: $\langle x_i\rangle$: bit commitments representing an element in $\mathbb{F}_{2^8}$

Output: $\langle y_i\rangle$: bit commitments representing square in $\mathbb{F}_{2^8}$

$\langle y_0\rangle \leftarrow \langle x_0\rangle + \langle x_4\rangle + \langle x_6\rangle$

$\langle y_1\rangle \leftarrow \langle x_4\rangle + \langle x_6\rangle + \langle x_7\rangle$

$\langle y_2\rangle \leftarrow \langle x_1\rangle + \langle x_5\rangle$

$\langle y_3\rangle \leftarrow \langle x_4\rangle + \langle x_5\rangle + \langle x_6\rangle + \langle x_7\rangle$

$\langle y_4\rangle \leftarrow \langle x_2\rangle + \langle x_4\rangle + \langle x_7\rangle$

$\langle y_5\rangle \leftarrow \langle x_5\rangle + \langle x_6\rangle$

$\langle y_6\rangle \leftarrow \langle x_3\rangle + \langle x_5\rangle$

$\langle y_7\rangle \leftarrow \langle x_6\rangle + \langle x_7\rangle$

return $(\langle y_0\rangle, \ldots, \langle y_7\rangle)$

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

Input: $\langle x_i\rangle$: bit commitments representing an element in $\mathbb{F}_{2^8}$

Output: $\langle \boldsymbol{y}\rangle_8$: commitment to $\mathbb{F}_{2^8}$ value of square

$\langle y_0\rangle, \ldots, \langle y_7\rangle \leftarrow$ SquareBits$(\langle x_0\rangle, \ldots, \langle x_7\rangle)$

return ZK.ByteCombine$(\langle y_0\rangle, \ldots, \langle y_7\rangle; \lambda)$

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

Input: $\langle s_i\rangle$: 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. 7.5: Bitwise squaring and conjugates of VOLE commitments in $\mathbb{F}_{2^8}$

ZK.InverseAffine$(\langle \texttt{state-bits}\rangle_1; n)$

Input: For each byte $i \in [0..n)$, commitments $\langle x_{i,j}\rangle_1$ to $x_{i,j} \in \mathbb{F}_2$ for $j \in \{0, \ldots, 7\}$.

Output: For each byte i, commitments $\langle y_{i,j}\rangle_1$ to $y_{i,j} \in \mathbb{F}_2$ for $j \in \{0, \ldots, 7\}$

for $i \in [0..n)$ **do**

  for $j \in [0..8)$ **do**

    $c_j \leftarrow 1$ **if** $j \in \{0, 2\}$ **else** 0

    $\langle y_{i,j}\rangle \leftarrow \langle x_{i,(j-1) \bmod 8}\rangle + \langle x_{i,(j-3) \bmod 8}\rangle + \langle x_{i,(j-6) \bmod 8}\rangle +$ ConstantToVOLE(c_j)

return $((\langle y_{i,0}\rangle, \ldots, \langle y_{i,7}\rangle))_{i \in [0..n)}$

Fig. 7.6: Inverse affine layer on the entire state. Each byte is processed as an 8-bit Boolean circuit in $\mathbb{F}_2$.

Forwards S-Box. The function ZK.SBoxEval in Figure 7.7 obtains as inputs all the $N_{\text{st-bits}}$ bits of the state as VOLE commitments. Towards understanding the forward S-Box evaluation algorithm, let us consider the individual steps of the S-Box:

1. To compute the Galois conjugates of x^{254} based on powers $x, x^2, \ldots, x^{128}$, note that the exponents $a_i = 254 \cdot 2^i \bmod 255$ are actually the powers

$$254, 253, 251, 247, 239, 223, 191, 127$$

ZK.InverseSBoxEval($\langle$`state-bits`$\rangle$)

NEW: this whole procedure

INPUT: Commitments to state bits after inverse shift rows

OUTPUT: Commitments to $N_{\text{st-bytes}}$ S-Box inputs, packed as bytes

1 : $\langle \mathbf{s} \rangle := \text{InverseAffine}(\langle \texttt{state-bits} \rangle; N_{\text{st-bytes}})$ // bit-wise

2 : // $s_{\text{conj},i,j} = s_i^{2^j} \in \mathbb{F}_{2^8}$, where s_i is i-th byte of $\mathbf{s}$, for $j \in [0..7]$

3 : $(\langle \boldsymbol{s}_{\text{conj},0,0} \rangle_8, \ldots, \langle \boldsymbol{s}_{\text{conj},N_{\text{st-bytes}}-1,7} \rangle_8) := \mathbb{F}_{256}/\mathbb{F}_2.\text{Conjugates}(\langle \mathbf{s} \rangle)$ // prepare for inverses

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

5 : // Inverse $x \mapsto x^{254}$

6 : $\langle \boldsymbol{b}_j \rangle_8^7 := \prod_{i=1}^{7} \langle \boldsymbol{s}_{\text{conj},j,i} \rangle_8$

7 : **return** $(\langle \boldsymbol{b}_0 \rangle_8^7, \ldots, \langle \boldsymbol{b}_{N_{\text{st-bytes}}-1} \rangle_8^7)$

ZK.SBoxEval($\langle$`state-bits`$\rangle$)

NEW: this whole procedure

INPUT: Commitments to state bits

OUTPUT: Commitments to $N_{\text{st-bytes}}$ S-Box outputs, including affine layer

1 : $(\langle \boldsymbol{s}_{\text{conj},0,0} \rangle_8, \ldots, \langle \boldsymbol{s}_{\text{conj},N_{\text{st-bytes}}-1,7} \rangle_8) := \mathbb{F}_{256}/\mathbb{F}_2.\text{Conjugates}(\langle \texttt{state-bits} \rangle)$ // Output in bytes

2 : // Coefficients for the affine layer over $\mathbb{F}_{2^8}$; ζ_i lifts these to $\mathbb{F}_{2^\lambda}$ (can be precomputed)

3 : $(\chi_0, \ldots, \chi_8) = (\texttt{0x05}, \texttt{0x09}, \texttt{0xf9}, \texttt{0x25}, \texttt{0xf4}, \texttt{0x01}, \texttt{0xb5}, \texttt{0x8f}, \texttt{0x63})$

4 : **for** $i \in [0..8]$ **do**

5 : $\zeta_i = \text{ByteCombine}(\chi_i; \lambda)$

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

7 : // Compute exponentiation and sum more efficiently using prefixes and including the ζ_i's

8 : **for** $i \in [0..4)$ **do**

9 : $\langle p_{1,i} \rangle_8^2 := \langle \boldsymbol{s}_{\text{conj},j,2i} \rangle_8 \cdot \langle \boldsymbol{s}_{\text{conj},j,2i+1} \rangle_8$

10 : $\langle f_{1,i} \rangle_8 := \zeta_{2i} \cdot \langle \boldsymbol{s}_{\text{conj},j,2i+1} \rangle_8 + \zeta_{2i+1} \cdot \langle \boldsymbol{s}_{\text{conj},j,2i} \rangle_8$

11 : **for** $i \in [0..2)$ **do**

12 : $\langle p_{2,i} \rangle_8^4 := \langle p_{1,2i} \rangle_8^2 \cdot \langle p_{1,2i+1} \rangle_8^2$

13 : $\langle f_{2,i} \rangle_8^3 := \langle f_{1,2i} \rangle_8 \cdot \langle p_{1,2i+1} \rangle_8^2 + \langle f_{1,2i+1} \rangle_8 \cdot \langle p_{1,2i} \rangle_8^2$

14 : $\langle \text{state}_j \rangle_8^7 := \langle f_{2,0} \rangle_8^3 \cdot \langle p_{2,1} \rangle_8^4 + \langle f_{2,1} \rangle_8^3 \cdot \langle p_{2,0} \rangle_8^4 + \zeta_8$

15 : **return** $\langle \text{state}_0 \rangle_8^7, \ldots, \langle \text{state}_{N_{\text{st-bytes}}-1} \rangle_8^7$

Fig. 7.7: Forward and backward S-box evaluation on committed state bits.

of x in $\mathbb{F}_{2^8}$. These powers can be computed as $\sigma^i(x^{254}) = \prod_{k=0,k\neq i}^{7} x^{2^k}$.

2. We then need to apply the affine layer based on the Galois conjugates of the inverse x^{254}, using the formula

$$S(x) = \zeta_8 + \sum_{i=0}^{7} \zeta_i \, \sigma^i(x^{254})$$

where each $\zeta_i = \text{ByteCombine}(\chi_i; \lambda)$ is derived using the constant $\chi_i \in \mathbb{F}_{2^8}$ that can be obtained from Lagrange interpolation of the affine layer.

However, we observe that the sum can be computed more efficiently: we can rewrite the sum

$$\sum_{i=0}^{7} \zeta_i \sigma^i(x^{254}) = \sum_{i=0}^{7} \zeta_i \prod_{k=0,k\neq i}^{7} x^{2^k}$$

Notice that the terms of $\zeta_0, \ldots, \zeta_3$ share the factor $p_{2,1} = x^{2^4} \cdots x^{2^7}$, while $\zeta_4, \ldots, \zeta_7$ share the factor $p_{2,0} = x^{2^0} \cdots x^{2^3}$. Thus we can rewrite the sum as

$$\sum_{i=0}^{7} \zeta_i \sigma^i(x^{254}) = p_{2,1} \cdot \sum_{i=0}^{3} \zeta_i \prod_{k=0,k\neq i}^{3} x^{2^k} + p_{2,0} \cdot \sum_{i=4}^{7} \zeta_i \prod_{k=4,k\neq i}^{7} x^{2^k}.$$

Thus, instead of computing 8 products of 7 VOLE-commitments, we can instead precompute commitments to $p_{2,0}, p_{2,1}$ and then compute a sum, reducing the number of non-linear multiplications.

We can now recursively apply the same technique, observing that e.g. the first summand $\sum_{i=0}^{3} \zeta_i \prod_{k=0,k\neq i}^{3} x^{2^k}$ can again be written as a sum $\sum_{i=0}^{1} \zeta_i \prod_{k=0,k\neq i}^{3} x^{2^k} + \sum_{i=2}^{3} \zeta_i \prod_{k=0,k\neq i}^{3} x^{2^k}$, whose first summation (including ζ_0, ζ_1) contains a $x^{2^2} \cdot x^{2^3}$ factor, while the second summation (including ζ_2, ζ_3) has a factor $x \cdot x^2$ that can be precomputed.

Overall, this technique reduces the cost of the S-box evaluation from 18 non-linear multiplications down to 12.

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

- ZK.MixColumns (Figure 7.8): This procedure implements the MixColumns transformation on committed state bytes. It takes as input a sequence of degree-7 commitments to bytes (elements of $\mathbb{F}_{2^8}$). The output is a new sequence of degree-7 commitments obtained by mixing each column according to the MixColumns linear transformation.

- ZK.BitwiseMixColumns (Figure 7.9): This procedure performs the MixColumns operation at the bit level on the committed state bits. It processes the state byte-by-byte 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.MixColumns($\langle \boldsymbol{x} \rangle_8^7$)

INPUT: $\langle \boldsymbol{x} \rangle_8^7 = (\langle x_0 \rangle_8^7, \ldots, \langle x_{(N_{\text{st-bytes}}-1)} \rangle_8^7)$, each $\langle x_i \rangle_8^7$ is degree-7 commitment to $x_i \in \mathbb{F}_{2^8} \subset \mathbb{F}_{2^\lambda}$

OUTPUT: $\langle \boldsymbol{y} \rangle_8^7 = (\langle y_0 \rangle_8^7, \ldots, \langle y_{(N_{\text{st-bytes}}-1)} \rangle_8^7)$, each $\langle y_i \rangle_8^7$ is degree-7 commitment to $y_i \in \mathbb{F}_{2^8} \subset \mathbb{F}_{2^\lambda}$

1 : $\nu_1 \leftarrow$ ByteCombine(0x01; λ)
2 : $\nu_2 \leftarrow$ ByteCombine(0x02; λ)
3 : $\nu_3 \leftarrow$ ByteCombine(0x03; λ)
4 : **for** $c \in [0..N_{\text{st}})$ **do**
5 : $i_0 \leftarrow 4c,\ i_1 \leftarrow 4c+1,\ i_2 \leftarrow 4c+2,\ i_3 \leftarrow 4c+3$
6 : // Compute $\langle y[i_0..i_3] \rangle$ via $(2,3,1,1)$ or $(4,5,1,1)$ in $\mathbb{F}_{2^8}$.
7 : $\langle y_{i_0} \rangle_8^7 \leftarrow \langle x_{i_0} \rangle_8^7 \cdot \nu_2 + \langle x_{i_1} \rangle_8^7 \cdot \nu_3 + \langle x_{i_2} \rangle_8^7 \cdot \nu_1 + \langle x_{i_3} \rangle_8^7 \cdot \nu_1$
8 : $\langle y_{i_1} \rangle_8^7 \leftarrow \langle x_{i_0} \rangle_8^7 \cdot \nu_1 + \langle x_{i_1} \rangle_8^7 \cdot \nu_2 + \langle x_{i_2} \rangle_8^7 \cdot \nu_3 + \langle x_{i_3} \rangle_8^7 \cdot \nu_1$
9 : $\langle y_{i_2} \rangle_8^7 \leftarrow \langle x_{i_0} \rangle_8^7 \cdot \nu_1 + \langle x_{i_1} \rangle_8^7 \cdot \nu_1 + \langle x_{i_2} \rangle_8^7 \cdot \nu_2 + \langle x_{i_3} \rangle_8^7 \cdot \nu_3$
10 : $\langle y_{i_3} \rangle_8^7 \leftarrow \langle x_{i_0} \rangle_8^7 \cdot \nu_3 + \langle x_{i_1} \rangle_8^7 \cdot \nu_1 + \langle x_{i_2} \rangle_8^7 \cdot \nu_1 + \langle x_{i_3} \rangle_8^7 \cdot \nu_2$
11 : **return** $\langle \boldsymbol{y} \rangle_8^7$

Fig. 7.8: MixColumns applied to commitments to bytes

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

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

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

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

Fig. 7.9: MixColumns operation performed on commitments to bits.

ZK.ShiftRows($\langle \text{state} \rangle$)

INPUT: $\text{state} = (\langle s_0 \rangle_8^7, \ldots, \langle s_{(N_{\text{st-bytes}}-1)} \rangle_8^7)$ in byte order (each $\langle s_i \rangle_8^7$ is a commitment to s_i in $\mathbb{F}_{2^8}$).

OUTPUT: $\text{state}' = (\langle s'_0 \rangle_8^7, \ldots, \langle s'_{(N_{\text{st-bytes}}-1)} \rangle_8^7)$, with each row r shifted left by r positions.

1 : // Internally, AES stores the $N_{\text{st-bytes}}$ bytes in a $4 \times N_{\text{st}}$ matrix by column.
2 : // Let $\langle s_{4c+r} \rangle_8^7$ be the commitment corresponding to the (r, c) entry.
3 : // We rotate row r left by r positions, unless $N_{\text{st}} = 8$, when the last two rows are rotated by $r + 1$.
4 : **for** $r \in [0..4)$ **do**
5 : **for** $c \in [0..N_{\text{st}})$ **do**
6 : **if** $N_{\text{st}} \neq 8$ or $r \in \{0, 1\}$ **do**
7 : $\langle s'_{4c+r} \rangle_8^7 \leftarrow \langle s_{4 \cdot ((c+r) \bmod N_{\text{st}}) + r} \rangle_8^7$
8 : **else**
9 : $\langle s'_{4c+r} \rangle_8^7 \leftarrow \langle s_{4 \cdot ((c+r+1) \bmod N_{\text{st}}) + r} \rangle_8^7$
10 : **return** $(\langle s'_0 \rangle_8^7, \ldots, \langle s'_{(N_{\text{st-bytes}}-1)} \rangle_8^7)$

Fig. 7.10: ShiftRows on a committed state .

- ZK.ShiftRows (Figure 7.10): 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 7.11): 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 7.12): 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.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$' = (\langle s'_0\rangle, \ldots, \langle s'_{N_{\text{st-bits}}-1}\rangle)$

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

Fig. 7.11: Inverse of the ShiftRows operation on a bit-level $N_{\text{st-bits}}$-bit array. The procedure reverses the row rotations applied by ShiftRows.

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 7.12): This procedure performs the AddRoundKey operation at the byte level. It takes as input an array of degree-7 VOLE commitments representing the state bytes and an array of VOLE commitments representing the corresponding round key bits, where the key commitments have degree 1. For each byte index i in the range $[0, N_{\text{st-bytes}})$, it first lifts all bits of the corresponding key indices into a byte (i.e. an element in $\mathbb{F}_{2^8}$). It then lifts this degree-1 VOLE commitment to a byte to a degree-7 commitment and adds it to the state byte. This yields new degree-7 commitments for the state after the key addition.

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** $i \in [0..N_{\text{st-bits}})$ **do**
2 : $\langle o_i\rangle \leftarrow \langle s_i\rangle + \langle k_i\rangle$
3 : **return** $(\langle o_0\rangle, \ldots, \langle o_{N_{\text{st-bits}}-1}\rangle)$

ZK.AddRoundKeyBytes$(\langle s_0\rangle_8^7, \ldots, \langle s_{N_{\text{st-bytes}}-1}\rangle_8^7, \langle k_0\rangle, \ldots, \langle k_{N_{\text{st-bits}}-1}\rangle)$

INPUT: $\langle s_i\rangle_8^7, \langle k_i\rangle$: commitments to state bytes, $\langle k_i\rangle$ is an array of $N_{\text{st-bits}}$ commitments to round key bits
OUTPUT: $\langle o_i\rangle_8^7$: commitments to state bytes after AddRoundKey

1 : **for** $i \in [0..N_{\text{st-bytes}})$ **do**
2 : $\langle b_i\rangle_8 :=$ ZK.ByteCombine$(\langle k_{8\cdot i}\rangle, \ldots, \langle k_{8\cdot i+7}\rangle)$
3 : $\langle o_i\rangle_8^7 \leftarrow \langle s_i\rangle_8^7 + \langle b_i\rangle_8$
4 : **return** $(\langle o_0\rangle_8^7, \ldots, \langle o_{N_{\text{st-bytes}}-1}\rangle_8^7)$

Fig. 7.12: The AddRoundKey procedure XORs consecutive bits of the expanded key with the current AES/Rijndael state. AddRoundKeyBytes does the same, but where the state is packed into bytes.

7.6 Witness Extension

The FAEST.ExtendWitness algorithm (Figure 7.13) 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 $b \in [0..\beta)$, the algorithm initializes the state as $\mathbf{in}_b$ and performs rounds $1, \ldots, R-2$. It stores the state bits after ShiftRows in the even-numbered rounds $2, 4, \ldots, R-2$. Specifically, for each of these even rounds j, it appends $N_{\mathsf{st\text{-}bits}}$ bits of column-major state to $\mathbf{w}$, denoted by $\mathtt{shift\_row\_bits}_j$.

The extended witness $\mathbf{w} \in \{0,1\}^{\ell}$ can be represented as:

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

If $\beta = 2$, the algorithm stores $\mathtt{shift\_row\_bits}$ concatenated for the two encryption blocks.

7.7 Deriving Constraints for the Key Expansion Routine

The KeyExpCstrnts algorithm, described in Figure 7.16, 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 commitments. In particular, the algorithm combines the outputs of two forward/backward key schedule routines, namely KeyExpFwd (see Figure 7.14) and KeyExpBkwd (see Figure 7.15), 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 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 $\mathtt{rconEvry}$ to track how often round constants must be removed.

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

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

```
1 :  // Set EM = false for FAEST and EM = true for FAEST-EM
2 :  // Initialize empty extended witness array, eventually of length ℓ
3 :  new w ∈ {0,1}^{≤ℓ}
4 :  // Store round keys as an array of 32-bit words
5 :  new k̄ ∈ [{0,1}^32; N_st(R+1)]
6 :  if EM = false then
7 :    k̄ ← KeyExpansion(k; param_OWF)
8 :    w ← k̄[0..N_k − 1]    // Saving the first N_k words, which contain k
9 :    i_k ← N_k
10 :   for j ∈ [0..S_ke/4) do
11 :     w ← w‖k̄[i_k]    // Saving the word that depends on SubWord
12 :     if λ = 192 then i_k ← i_k + 6
13 :     else i_k ← i_k + 4
14 :   in_0 ← in
15 :   if β = 2 then in_1 ← in ⊕ 1
16 : else
17 :   k̄ ← KeyExpansion(in; param_OWF)    // The public input is the Rijndael key
18 :   w ← k    // The secret key is the Rijndael message block
19 :   in_0 ← k
20 : // Encryption routine and saving witness bits
21 : for b ∈ [0..β) do
22 :   new state ∈ {0,1}^{N_st-bits}
23 :   state ← in_b
24 :   AddRoundKey(state, k̄[0..N_st − 1])
25 :   for j ∈ [1..R − 1) do    // Rounds are numbered 1, ..., R
26 :     Modification: no more storing of InvNorm
27 :     SubBytes(state)
28 :     ShiftRows(state)
29 :     if j mod 2 = 0 then
30 :       w := w‖state    // Save S-box outputs, in column-major order
31 :     MixColumns(state)
32 :     AddRoundKey(state, k̄[N_st · j..N_st(j+1) − 1])    // Round key k̄_j
       // The final two rounds R − 1, R are evaluated by EncCstrnts and not computed here
33 : return w ∈ {0,1}^ℓ
```

Fig. 7.13: Extending the witness from the AES/Rijndael key and the input plaintext block(s).

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($\langle \mathbf{w} \rangle^{\lambda \cdot \cdot}, \langle \mathbf{k} \rangle$) 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 $\mathbb{F}_{2^\lambda}$ 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

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. 7.14: Transforming witness or VOLE values into AES round keys.

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 inverse of $\hat{w}$. Hence, if all such constraints are satisfied, the final commitments are effectively commitments to zero.

7.8 Deriving Constraints for the Encryption Routine

ZK.EncCstrnts *(Figure 7.17)*. 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 $(2r+1)$ and $(2r+2)$, as follows:

1. *Forward computation of the S-Box in round* $2r+1$. Using the bitwise commitments to the state, the algorithm evaluates the AES S-Box using ZK.SBoxEval, combining the non-linear and linear steps. This outputs commitments to bytes of degree 7.
2. *Forward computation of round* $2r+1$. The algorithm applies ShiftRows, MixColumns, and AddRoundKey with $\langle \mathbf{k}_{2r+1}\rangle$. The resulting byte commitments are commitments to the S-Box inputs of round $2r+2$.
3. *Backward reconstruction of round* $2r+2$. If $2r+2$ is not the final iteration, the state after ShiftRows in round $2r+2$ is obtained from the extended witness; otherwise it is reconstructed from **out**. The algorithm applies InverseShiftRows, followed by ZK.InverseSBoxEval. The latter first applies ZK.InverseAffine and then evaluates inversion by multiplying the conjugates.

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

INPUT: VOLE-committed extended witness bits $\{\langle \mathbf{x}[i] \rangle\}_{i=0}^{8S_{\mathsf{ke}}-1}$ 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$(\langle \tilde{\mathbf{x}} \rangle; 1)$ // 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. 7.15: Transforming VOLE-committed extended witness for the key schedule into $\mathbb{F}_{2^8}$-inverse outputs.

4. *Checking equality of the forward and backward evaluations.* For every byte i, the algorithm defines $\langle z_{r \cdot N_{\text{st-bytes}}+i} \rangle_8^7$ as the difference between the forward S-Box input for round $2r+2$ and the corresponding backward reconstruction.
5. *Next-round state.* Except in the final iteration, the procedure applies BitwiseMixColumns and AddRoundKey with $\langle \mathbf{k}_{2r+2} \rangle$ to the saved round-$2r+2$ state. This produces the committed S-Box inputs for round $2r+3$.

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: $\langle \boldsymbol{z} \rangle_8^2$: 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 $\langle \hat{\boldsymbol{k}} \rangle_8, \langle \hat{\boldsymbol{k}}_{\mathsf{sq}} \rangle_8, \langle \hat{\boldsymbol{w}} \rangle_8, \langle \hat{\boldsymbol{w}}_{\mathsf{sq}} \rangle_8$, each a commitment to vector in $\mathbb{F}_{2^\lambda}^4$

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

    $r' \leftarrow r$

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

    $\langle \hat{\boldsymbol{k}}[r'] \rangle_8 \leftarrow \mathsf{ZK.ByteCombine}\big(\langle \mathbf{k}[\, i_{\mathsf{wd}} + 8r \,..\, i_{\mathsf{wd}} + 8r + 7] \rangle,\ \lambda\big)$

    $\langle \hat{\boldsymbol{k}}_{\mathsf{sq}}[r'] \rangle_8 \leftarrow \mathsf{SquareAndCombine}\big(\langle \mathbf{k}[\, i_{\mathsf{wd}} + 8r \,..\, i_{\mathsf{wd}} + 8r + 7] \rangle,\ \lambda\big)$

    $\langle \hat{\boldsymbol{w}}[r] \rangle_8 \leftarrow \mathsf{ZK.ByteCombine}\big(\langle \bar{\mathbf{w}}[\, 32j + 8r \,..\, 32j + 8r + 7] \rangle,\ \lambda\big)$

    $\langle \hat{\boldsymbol{w}}_{\mathsf{sq}}[r] \rangle_8 \leftarrow \mathsf{SquareAndCombine}\big(\langle \bar{\mathbf{w}}[\, 32j + 8r \,..\, 32j + 8r + 7] \rangle,\ \lambda\big)$

  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

    $\langle z_{8j+2r} \rangle_8^2 \leftarrow \langle \hat{\boldsymbol{k}}_{\mathsf{sq}}[r] \rangle_8 \cdot \langle \hat{\boldsymbol{w}}[r] \rangle_8 - \langle \hat{\boldsymbol{k}}[r] \rangle_8$

    $\langle z_{8j+2r+1} \rangle_8^2 \leftarrow \langle \hat{\boldsymbol{k}}[r] \rangle_8 \cdot \langle \hat{\boldsymbol{w}}_{\mathsf{sq}}[r] \rangle_8 - \langle \hat{\boldsymbol{w}}[r] \rangle_8$

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

return $(\langle z_0 \rangle_8^2, \ldots, \langle z_{2S_{\mathsf{ke}}-1} \rangle_8^2, \langle \mathbf{k} \rangle)$

Fig. 7.16: Deriving the $2S_{\mathsf{ke}}$ constraints for the AESλ key expansion routine.

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

INPUT: Commitments to $\mathbf{in} \in \mathbb{F}_2^{N_{\text{st-bits}}}, \mathbf{out} \in \mathbb{F}_2^{N_{\text{st-bits}}}$, $\mathbf{w} \in \mathbb{F}_2^{\ell_{\text{enc}}}$ and round keys $\mathbf{k}_i \in \mathbb{F}_2^{N_{\text{st-bits}}}$
OUTPUT: $\langle z_0 \rangle_8^7, \ldots, \langle z_{S_{\text{enc}}/2-1} \rangle_8^7$

1 : $\langle \mathtt{state\text{-}bits} \rangle \leftarrow$ ZK.AddRoundKey($\langle \mathbf{in} \rangle, \langle \mathbf{k}_0 \rangle$) // Input and output in bits
2 : **for** $r \in [0..R/2)$ **do**
3 : NEW: no more evaluation on squared S-Box output
4 : $\langle st \rangle_8^7 :=$ ZK.SBoxEval($\langle \mathtt{state\text{-}bits} \rangle$)
5 : $\langle st \rangle_8^7 :=$ ZK.ShiftRows($\langle st \rangle_8^7$)
6 : $\langle st \rangle_8^7 :=$ ZK.MixColumns($\langle st \rangle_8^7$)
7 : $\langle st \rangle_8^7 :=$ ZK.AddRoundKeyBytes($\langle st \rangle_8^7, \langle \mathbf{k}_{2r+1} \rangle$) // round $2r+2$ S-box inputs
8 : // Now go backwards to get S-box outputs for round $2r+2$
9 : // Now go backwards to get S-box outputs for round $2r+1$
10 : **if** $r = \frac{R}{2} - 1$ **then**
11 : $\langle \tilde{\mathbf{s}} \rangle :=$ ZK.AddRoundKey($\langle \mathbf{out} \rangle, \langle \mathbf{k}_R \rangle$)
12 : **else**
13 : // Extract the bits corresponding to ShiftRows output from the extended witness
14 : $\langle \tilde{\mathbf{s}} \rangle := \big(\langle \mathbf{w}[N_{\text{st-bits}} \cdot r] \rangle, \ldots, \langle \mathbf{w}[N_{\text{st-bits}} \cdot (r+1) - 1] \rangle\big)$
15 : $\langle \mathbf{s}'' \rangle :=$ ZK.InverseShiftRows($\langle \tilde{\mathbf{s}} \rangle$) // Computed bit-wise
16 : $\langle \mathbf{b} \rangle_8^7 :=$ ZK.InverseSBoxEval($\langle \mathbf{s}'' \rangle$)
17 : **for** $i \in [0..N_{\text{st-bytes}})$ **do**
18 : // Check computed S-Box input against output from round eval
19 : $\langle z_{r \cdot N_{\text{st-bytes}}+i} \rangle_8^7 := \langle \boldsymbol{b}_i \rangle_8^7$ - $\langle st_i \rangle_8^7$
20 : // Go forwards to compute the next round's state
21 : **if** $r \neq \frac{R}{2} - 1$ **then**
22 : $\langle \mathtt{state\text{-}bits} \rangle :=$ ZK.BitwiseMixColumns($\langle \tilde{\mathbf{s}} \rangle$)
23 : $\langle \mathtt{state\text{-}bits} \rangle :=$ ZK.AddRoundKey($\langle \mathtt{state\text{-}bits} \rangle, \langle \mathbf{k}_{2r+2} \rangle$)
24 : **return** $(\langle z_0 \rangle_8^7, \ldots, \langle z_{S_{\text{enc}}/2-1} \rangle_8^7)$

Fig. 7.17: Deriving the constraints for the encryption routine

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

NEW: now raises to higher degree than 3, checks that witness consists of bits.

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

OUTPUT: Committed constraints $\langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}}$, for $\boldsymbol{z} \in \mathbb{F}_{2^\lambda}^{C-\ell}$ (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 $\langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}}$, $\boldsymbol{z} \in \mathbb{F}_{2^\lambda}^{\le C-\ell}$
// constraint to reduce keyspace by 1 bit
$\langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}} := \langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}} \parallel \mathsf{DegLift}(\langle \mathbf{w}[0] \rangle \cdot \langle \mathbf{w}[1] \rangle; d_{\mathsf{zk}} - 2)$
// Define committed $\mathbf{in}, \mathbf{out}$ for EM compatibility
if $\mathsf{EM} = \mathtt{true}$ **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
  $\langle \mathbf{out}_0 \rangle \leftarrow \langle \mathbf{w}[0..\lambda) \rangle + \mathsf{ConstantToVOLE}(\mathbf{y})$ // 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** $\langle \mathbf{out}_1 \rangle := \mathsf{ConstantToVOLE}(\mathbf{y}[128..255])$
  $(\langle \tilde{\boldsymbol{z}} \rangle_8^2, \langle \mathbf{k} \rangle) := \mathsf{FAEST.KeyExpCstrnts}(\langle \mathbf{w}[0..\ell_{\mathsf{ke}}) \rangle; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
  $\langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}} := \langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}} \parallel \mathsf{DegLift}(\langle \tilde{\boldsymbol{z}} \rangle_8^2; d_{\mathsf{zk}} - 2)$ // 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** $\langle \mathbf{in}[0] \rangle \leftarrow \langle \mathbf{in}[0] \rangle + \mathsf{ConstantToVOLE}(1)$
  $\langle \tilde{\boldsymbol{z}} \rangle_8^{d_{\mathsf{zk}}} := \mathsf{ZK.EncCstrnts}(\langle \mathbf{in} \rangle, \langle \mathbf{out}_b \rangle, \langle \tilde{\mathbf{w}} \rangle, \langle \mathbf{k}_0 \rangle, \ldots, \langle \mathbf{k}_R \rangle; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
  $\langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}} := \langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}} \parallel \langle \tilde{\boldsymbol{z}} \rangle_8^{d_{\mathsf{zk}}}$ // Appending values
return $\langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}}$

Fig. 7.18: Proof of one-way function constraints

7.9 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 three algorithms:

1. The main OWFConstraints algorithm that combines both key-schedule and encryption constraints.
2. The signer-side procedure OWFProve.
3. The verifier-side procedure OWFVerify.

ZK.OWFConstraints($\langle \mathbf{w} \rangle, \mathsf{pk}$) *(Figure 7.18)*. 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 $\langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}}$ for the accumulated constraints.
2. It first appends a small keyspace-reduction constraint $\langle \mathbf{w}[0] \rangle \times \langle \mathbf{w}[1] \rangle$ and uses DegLift to ensure that the constraint has the correct degree d_{zk}.
3. If $\mathsf{EM} = \mathtt{true}$, 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 $\langle \tilde{\boldsymbol{z}} \rangle_{\lambda}^{d_{\mathsf{zk}}}$. This is appended to the main buffer by calling DegLift.
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 $\langle \boldsymbol{z} \rangle_{\lambda}^{d_{\mathsf{zk}}}$.
5. Finally, the algorithm returns $\langle \tilde{\boldsymbol{z}} \rangle_{8}^{2}$.

ZK.OWFProve, ZK.OWFVerify *(Figure 7.19)*. 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. The VOLE commitments $(\bar{\mathbf{u}}_j, \bar{\mathbf{v}}_j)_{j \in [0..d_{\mathsf{zk}}-1)}$ are treated as masks and will be used during the hashing step later.
2. For each of the ℓ witness bits $\langle \mathbf{w}[i] \rangle$, it adds a constraint $\langle \mathbf{w}[i] \rangle \cdot (1 - \langle \mathbf{w}[i] \rangle) = 0$ using a multiplication, and lifts the resulting constraint to degree d_{zk} using DegLift.
   This check ensures that, in the CRT-based commitment of Section 6, a malicious signer only commits to witness elements that are *bits*.
3. Then, it invokes OWFConstraints$(\langle \mathbf{w} \rangle, \mathsf{pk})$, which returns a vector of polynomials of degree $\leq d_{\mathsf{zk}}$. These polynomials are appended to those of the bit check.
4. The polynomials returned by the bit-check and by OWFConstraints are then parsed as d_{zk} $\mathbb{F}_{2^\lambda}^{C}$ vectors $\boldsymbol{a}_0, \dots, \boldsymbol{a}_{d_{\mathsf{zk}}-1}$, where C is from Table 3.3.
5. Finally, it generates the QuickSilver response $\tilde{a}_0, \dots, \tilde{a}_{d_{\mathsf{zk}}-1}$ by hashing $\boldsymbol{a}_j \| (\text{masks})$ with chall_2. The values $\tilde{a}_0, \dots, \tilde{a}_{d_{\mathsf{zk}}-1}$ are then returned as the final proof message.

The algorithm OWFVerify is the corresponding verifier procedure. It reconstructs the commitments values $\langle \mathbf{w}[i] \rangle$ from $\mathbf{d}[i]$ and $\mathbf{Q}$ and the hash mask q^* from $(\bar{\mathbf{q}}_j)_{j \in [0..d_{\mathsf{zk}}-1)}$. It first computes the constraints $\langle \boldsymbol{z} \rangle_{\lambda}^{d_{\mathsf{zk}}}$ based on the bit-check and using the function OWFConstraints$(\langle \mathbf{w} \rangle, \mathsf{pk})$, and then checks consistency of the QuickSilver hash with the prover's $\tilde{a}_1, \dots, \tilde{a}_{d_{\mathsf{zk}}-1}$. In particular, it computes $\tilde{q} = \mathsf{ZKHash}(\mathsf{chall}_2, \boldsymbol{b} \| q^*)$ using the mask q^* and constraint vectors $\boldsymbol{b}$ derived from the commitment polynomials $\boldsymbol{z}$ derived as in OWFProve. Then it returns $\tilde{q} - \sum_{j=1}^{d_{\mathsf{zk}}-1} \tilde{a}_j \Delta^j$. If the proof is correct, this final expression should be the same as $\tilde{a}_0$.

$\mathsf{ZK.OWFProve}(\mathbf{w}, \mathbf{V}, (\bar{\mathbf{u}}_j, \bar{\mathbf{v}}_j)_{j\in[0..d_{\mathsf{zk}}-1)}, \mathsf{pk}, \mathsf{chall}_2; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$

INPUT: $\mathbf{w} \in \{0,1\}^\ell$,$\mathbf{V} \in \mathbb{F}_{2^\lambda}^\ell$, $(\bar{\mathbf{u}}_j, \bar{\mathbf{v}}_j) \in \mathbb{F}_{2^\lambda} \times \mathbb{F}_{2^\lambda}$ for $j \in [0..d_{\mathsf{zk}}-1)$

OUTPUT: $(\tilde{a}_0, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}) \in \mathbb{F}_{2^\lambda}^{d_{\mathsf{zk}}}$

// Construct the degree-1 commitments to the witness bits
for $i \in [0..\ell)$ **do**
  Let $\langle \mathbf{w}[i] \rangle := (\mathbf{V}[i], \mathbf{w}[i])$ // $\rho_{\mathbf{w}[i]}(X) = \mathbf{V}[i] + \mathbf{w}[i]X$
// Compute the degree-d_{zk} zero constraints
// Polynomial $\boldsymbol{a}_0 + \boldsymbol{a}_1 X + \cdots + \boldsymbol{a}_{d_{\mathsf{zk}}-1} X^{d_{\mathsf{zk}}-1} + \boldsymbol{z} X^{d_{\mathsf{zk}}}$, where $\boldsymbol{z} = 0$
// Allocate space for the constraint commitments
new $\langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}}$, $\boldsymbol{z} \in \mathbb{F}_{2^\lambda}^C$ // $C = \ell + C_{\mathsf{owf}}$, see Table 3.3
// Check that each $\langle \mathbf{w}[i] \rangle$ is a bit
$\langle \boldsymbol{z}[0..\ell) \rangle_\lambda^{d_{\mathsf{zk}}} := \big(\mathsf{DegLift}(\langle \mathbf{w}[i] \rangle \cdot (\mathsf{ConstantToVOLE}(1) - \langle \mathbf{w}[i] \rangle); d_{\mathsf{zk}} - 2)\big)_{i\in[0..\ell)}$
// Now check OWF constraints
$\langle \boldsymbol{z}[\ell..C) \rangle_\lambda^{d_{\mathsf{zk}}} := \mathsf{OWFConstraints}(\langle \mathbf{w} \rangle, \mathsf{pk})$
Parse $\langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}}$ as $(\boldsymbol{a}_0, \ldots, \boldsymbol{a}_{d_{\mathsf{zk}}-1}) \in (\mathbb{F}_{2^\lambda}^C)^{d_{\mathsf{zk}}}$
// Use the first $d_{\mathsf{zk}} - 1$ CRT-generated field VOLE masks
// The final four masks are reserved for the VOLE consistency check
$\tilde{a}_0 := \mathsf{ZKHash}(\mathsf{chall}_2, (\boldsymbol{a}_0 \| \bar{\mathbf{v}}_0))$
for $j \in [1..d_{\mathsf{zk}} - 2]$ **do**
  $\tilde{a}_j := \mathsf{ZKHash}(\mathsf{chall}_2, (\boldsymbol{a}_j \| \bar{\mathbf{u}}_{j-1} + \bar{\mathbf{v}}_j))$
$\tilde{a}_{d_{\mathsf{zk}}-1} := \mathsf{ZKHash}(\mathsf{chall}_2, (\boldsymbol{a}_{d_{\mathsf{zk}}-1} \| \bar{\mathbf{u}}_{d_{\mathsf{zk}}-2}))$
return $(\tilde{a}_0, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1})$

$\mathsf{ZK.OWFVerify}(\mathbf{d}, \mathbf{Q}, (\bar{\mathbf{q}}_j)_{j\in[0..d_{\mathsf{zk}}-1)}, \mathsf{pk}, \mathsf{chall}_2, \Delta, \tilde{a}_1, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$

INPUT: $\mathbf{d} \in \{0,1\}^\ell$, $\mathbf{Q} \in \mathbb{F}_{2^\lambda}^\ell$, $(\bar{\mathbf{q}}_j)_{j\in[0..d_{\mathsf{zk}}-1)} \in \mathbb{F}_{2^\lambda}^{d_{\mathsf{zk}}-1}$, $\Delta \in \mathbb{F}_{2^\lambda}$, and $\tilde{a}_1, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1} \in \mathbb{F}_{2^\lambda}$

OUTPUT: $\tilde{a}_0 \in \mathbb{F}_{2^\lambda}$

// Construct the verifier-side degree-1 commitments to the witness bits
for $i \in [0..\ell)$ **do**
  Let $\langle \mathbf{w}[i] \rangle := \mathbf{Q}[i] + \mathbf{d}[i] \cdot \Delta$ // $\mathbf{Q}[i] = \mathbf{V}[i] + \mathbf{u}[i]\Delta$ and $\mathbf{d}[i] = \mathbf{w}[i] + \mathbf{u}[i]$
// Combine the first $d_{\mathsf{zk}} - 1$ CRT-generated field VOLE mask correlations
$q^* := \sum_{j=0}^{d_{\mathsf{zk}}-2} \bar{\mathbf{q}}_j \cdot \Delta^j$ // $\bar{\mathbf{q}}_j = \bar{\mathbf{v}}_j + \bar{\mathbf{u}}_j \Delta$
// Allocate space for the constraint commitments
new $\langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}}$, $\boldsymbol{z} \in \mathbb{F}_{2^\lambda}^C$ // $C = \ell + C_{\mathsf{owf}}$, see Table 3.3
// Verifier-side constraints that each of the ℓ committed values is a bit
$\langle \boldsymbol{z}[0..\ell) \rangle_\lambda^{d_{\mathsf{zk}}} := \big(\mathsf{DegLift}(\langle \mathbf{w}[i] \rangle \cdot (\mathsf{ConstantToVOLE}(1) - \langle \mathbf{w}[i] \rangle); d_{\mathsf{zk}} - 2)\big)_{i\in[0..\ell)}$
// Compute the verifier-side evaluation of the degree-d_{zk} OWF constraints
$\langle \boldsymbol{z}[\ell..C) \rangle_\lambda^{d_{\mathsf{zk}}} := \mathsf{OWFConstraints}(\langle \mathbf{w} \rangle, \mathsf{pk})$
Parse $\langle \boldsymbol{z} \rangle_\lambda^{d_{\mathsf{zk}}}$ as $\boldsymbol{b} \in \mathbb{F}_{2^\lambda}^C$
// Hash the constraint evaluation together with the mask-polynomial evaluation
$\tilde{q} := \mathsf{ZKHash}(\mathsf{chall}_2, (\boldsymbol{b} \| q^*))$
// Reconstruct the value $\tilde{a}_0$
return $\tilde{q} - \sum_{j=1}^{d_{\mathsf{zk}}-1} \tilde{a}_j \cdot \Delta^j$

Fig. 7.19: Signer and verifier algorithms for the ZK proof of the one-way function.

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

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 : $\mathbf{y} := E_{\mathbf{k}}(\mathbf{x})$
6 : $\mathsf{sk} := \mathbf{k}$
7 : $\mathsf{pk} := (\mathbf{x}, \mathbf{y})$
8 : **return** $(\mathsf{sk}, \mathsf{pk})$

Fig. 8.1: FAEST key generation algorithm

8 The FAEST Signature Scheme

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

- Section 8.1: Key Generation algorithm FAEST.KeyGen in Figure 8.1
- Section 8.2: Signing algorithm FAEST.Sign in Figure 8.2
- Section 8.3: Verification algorithm FAEST.Verify in Figure 8.3.

8.1 Key Generation

The key generation algorithm takes as input one of the parameter sets described in Table 3.2 via $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$. The selected parameter set fixes the security parameter λ, the OWF input length $N_{\text{st-bits}}$, the OWF output length $\beta N_{\text{st-bits}}$, and the one-way function E (which is also specified in Table 3.2). Thus, $\mathbf{x} \in \{0,1\}^{N_{\text{st-bits}}}$ and $E_{\mathbf{k}}(\mathbf{x}) \in \{0,1\}^{\beta N_{\text{st-bits}}}$. The algorithm, given in Figure 8.1, is 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 not 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 evaluates the selected one-way function to obtain $\mathbf{y} = E_{\mathbf{k}}(\mathbf{x}) \in \{0,1\}^{\beta N_{\text{st-bits}}}$. The secret key is $\mathsf{sk} = \mathbf{k}$, and the public key is $\mathsf{pk} = (\mathbf{x}, \mathbf{y})$.

Representation of keys. Although Figure 8.1 defines the secret key abstractly as $\mathsf{sk} = \mathbf{k}$, in an implementation, keys are stored as byte strings where the secret key additionally contains the public one-way function input $\mathbf{x}$. Concretely, we encode:

$$\mathsf{sk} = (\mathbf{x} \| \mathbf{k}), \qquad \mathsf{pk} = (\mathbf{x} \| \mathbf{y}),$$

where $\mathbf{x}$ is the public OWF input and $\mathbf{k}$ the OWF key. The secret key is thus $N_{\text{st-bits}}/8 + \lambda/8$ bytes long and the public key $N_{\text{st-bits}}/8 + \beta N_{\text{st-bits}}/8$ bytes (recall that $N_{\text{st-bits}} = \lambda$ for FAEST-EM and 128 for all FAEST variants). Including $\mathbf{x}$ in the secret key allows a signer to recompute pk by evaluating the one-way function.

8.2 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.

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

$\rho \leftarrow \{0,1\}^{\lambda}$ // For deterministic signing, set $\rho := 0^{\lambda}$

1 : $\mu := \mathsf{H}_2^0(\mathsf{pk} \| \mathsf{msg}; 2\lambda)$

2 : $(r, \mathsf{iv}^{\mathsf{pre}}) := \mathsf{H}_3(\mathsf{sk} \| \mu \| \rho)$ // $r \in \{0,1\}^{\lambda}$ and $\mathsf{iv}^{\mathsf{pre}} \in \{0,1\}^{128}$

3 : $\mathsf{iv} := \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu)$

4 : // Commit to the VOLEs and generate the CRT field masks

5 : $(\mathsf{com}, \mathsf{decom}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathbf{u}, \mathbf{V}, (\bar{\mathbf{u}}_m, \bar{\mathbf{v}}_m)_{m \in [0..n_{\mathsf{mask}})}) := \mathsf{FAEST.VOLECommit}(r, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$

6 : $\mathsf{chall}_1 := \mathsf{H}_2^1(\mu \| \mathsf{com} \| \mathbf{c}_1 \| \cdots \| \mathbf{c}_{\tau-1} \| \mathbf{c}_{\mathsf{mult}} \| \mathsf{iv}^{\mathsf{pre}}; 8\lambda + 64)$ // $\mathbf{c}_{\mathsf{mult}} \in \{0,1\}^{n_{\mathsf{mask}} \times \bar{n}_{\mathsf{mult}}}$ read row-major

7 : // Prepare the two inputs to the four-output VOLE consistency hash

8 : $\mathbf{u}_0^{\mathsf{h}} := (\mathbf{u}[0], \ldots, \mathbf{u}[\ell-1], \bar{\mathbf{u}}_0, \ldots, \bar{\mathbf{u}}_{d_{\mathsf{zk}}-2}) \in \mathbb{F}_{2^{\lambda}}^{\ell + d_{\mathsf{zk}} - 1}$

9 : $\mathbf{u}_1^{\mathsf{h}} := (\bar{\mathbf{u}}_{d_{\mathsf{zk}}-1}, \bar{\mathbf{u}}_{d_{\mathsf{zk}}}, \bar{\mathbf{u}}_{d_{\mathsf{zk}}+1}, \bar{\mathbf{u}}_{d_{\mathsf{zk}}+2}) \in \mathbb{F}_{2^{\lambda}}^{4}$

10 : $\mathbf{V}_0^{\mathsf{h}} := (\mathbf{V}[0], \ldots, \mathbf{V}[\ell-1], \bar{\mathbf{v}}_0, \ldots, \bar{\mathbf{v}}_{d_{\mathsf{zk}}-2}) \in \mathbb{F}_{2^{\lambda}}^{\ell + d_{\mathsf{zk}} - 1}$

11 : $\mathbf{V}_1^{\mathsf{h}} := (\bar{\mathbf{v}}_{d_{\mathsf{zk}}-1}, \bar{\mathbf{v}}_{d_{\mathsf{zk}}}, \bar{\mathbf{v}}_{d_{\mathsf{zk}}+1}, \bar{\mathbf{v}}_{d_{\mathsf{zk}}+2}) \in \mathbb{F}_{2^{\lambda}}^{4}$

12 : $\widetilde{\mathbf{u}} := \mathsf{VOLEHash}(\mathsf{chall}_1, (\mathbf{u}_0^{\mathsf{h}}, \mathbf{u}_1^{\mathsf{h}})) \in \mathbb{F}_{2^{\lambda}}^{4}$

13 : $\widetilde{\mathbf{V}} := \mathsf{VOLEHash}(\mathsf{chall}_1, (\mathbf{V}_0^{\mathsf{h}}, \mathbf{V}_1^{\mathsf{h}})) \in \mathbb{F}_{2^{\lambda}}^{4}$

14 : // Commit to the extended witness and derive the QuickSilver challenge

15 : $\mathbf{w} := \mathsf{FAEST.ExtendWitness}(\mathsf{sk}, \mathsf{pk}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}) \in \{0,1\}^{\ell}$

16 : $\mathbf{d} := \mathbf{w} \oplus \mathbf{u}$

17 : $\mathsf{chall}_2 := \mathsf{H}_2^2\big(\mathsf{chall}_1 \| \mathsf{ToBits}(\widetilde{\mathbf{u}}; \lambda, 4) \| \mathsf{ToBits}(\widetilde{\mathbf{V}}; \lambda, 4) \| \mathbf{d}; 3\lambda + 64\big)$

18 : // Compute the degree-d_{zk} QuickSilver proof using the first $d_{\mathsf{zk}} - 1$ CRT masks

19 : $(\tilde{a}_0, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}) := \mathsf{ZK.OWFProve}(\mathbf{w}, \mathbf{V}, (\bar{\mathbf{u}}_j, \bar{\mathbf{v}}_j)_{j \in [0..d_{\mathsf{zk}}-1)}, \mathsf{pk}, \mathsf{chall}_2; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$

20 : $\mathsf{ctr} \leftarrow -1, \quad \mathsf{decom}_I \leftarrow \bot$

21 : **while** $\mathsf{decom}_I = \bot$ **do**

22 : **if** $\mathsf{ctr} = 2^{32} - 1$ **then**

23 : **return** $\bot$

24 : $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$

25 : $\mathsf{chall}_3 \leftarrow \mathsf{H}_2^3\big(\mathsf{chall}_2 \| \mathsf{ToBits}((\tilde{a}_j)_{j \in [0..d_{\mathsf{zk}})}; \lambda, d_{\mathsf{zk}}) \| \mathsf{BitDec}(\mathsf{ctr}, 32); \lambda\big)$

26 : **if** $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}}..\lambda) \neq 0^{w_{\mathsf{grind}}}$ **then**

27 : **continue**

28 : $I \leftarrow \mathsf{DecodeAllChall}_3(\mathsf{chall}_3; \mathtt{param})$

29 : $\mathsf{decom}_I \leftarrow \mathsf{BAVC.Open}(\mathsf{decom}, I; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$

30 : **return** $\sigma := ((\mathbf{c}_i)_{i \in [1..\tau)}, \mathbf{c}_{\mathsf{mult}}, \widetilde{\mathbf{u}}, \mathbf{d}, \tilde{a}_1, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}, \mathsf{decom}_I, \mathsf{chall}_3, \mathsf{iv}^{\mathsf{pre}}, \mathsf{ctr})$

Fig. 8.2: FAEST signing algorithm.

In phase 1, the signing procedure (Figure 8.2) starts by sampling $\rho \in \{0,1\}^{\lambda}$ 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, $\mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu)$, produces the 128-bit IV iv; including the digest μ binds the IV to the signed message. Using (r, iv), the signer calls $\mathsf{FAEST.VOLECommit}(r, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$, which returns the commitment com, the bulk decommitment decom, the correction vectors $\mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}$, the gate-correction matrix $\mathbf{c}_{\mathsf{mult}}$, the witness correlation $(\mathbf{u}, \mathbf{V})$, and the field-mask correlations $(\bar{\mathbf{u}}_m, \bar{\mathbf{v}}_m)_{m \in [0..n_{\mathsf{mask}})}$. Finally, the signer derives chall_1 from $\mu, \mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}^{\mathsf{pre}}$ by calling H_2^1 and requesting $8\lambda + 64$ output bits.

In phase 2, the signer prepares $\widetilde{\mathbf{V}}$ and $\widetilde{\mathbf{u}}$ for the VOLE consistency check. It sets $\mathbf{u}_0^{\mathsf{h}} := (\mathbf{u}[0], \ldots, \mathbf{u}[\ell-1], \bar{\mathbf{u}}_0, \ldots, \bar{\mathbf{u}}_{d_{\mathsf{zk}}-2})$ and $\mathbf{u}_1^{\mathsf{h}} := (\bar{\mathbf{u}}_{d_{\mathsf{zk}}-1}, \bar{\mathbf{u}}_{d_{\mathsf{zk}}}, \bar{\mathbf{u}}_{d_{\mathsf{zk}}+1}, \bar{\mathbf{u}}_{d_{\mathsf{zk}}+2})$, embedding

each bit $\mathbf{u}[i]$ canonically in $\mathbb{F}_{2^\lambda}$. Similarly, it sets $\mathbf{V}_0^{\mathsf{h}} := (\mathbf{V}[0], \ldots, \mathbf{V}[\ell-1], \bar{\mathbf{v}}_0, \ldots, \bar{\mathbf{v}}_{d_{\mathsf{zk}}-2})$ and $\mathbf{V}_1^{\mathsf{h}} := (\bar{\mathbf{v}}_{d_{\mathsf{zk}}-1}, \bar{\mathbf{v}}_{d_{\mathsf{zk}}}, \bar{\mathbf{v}}_{d_{\mathsf{zk}}+1}, \bar{\mathbf{v}}_{d_{\mathsf{zk}}+2})$. It evaluates VOLEHash on both inputs, obtaining $\widetilde{\mathbf{u}}, \widetilde{\mathbf{V}} \in \mathbb{F}_{2^\lambda}^4$ directly. Next, it generates the extended witness by calling FAEST.ExtendWitness, producing $\mathbf{w} \in \{0,1\}^\ell$, and computes $\mathbf{d} := \mathbf{w} \oplus \mathbf{u}$ and derives 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. It uses $\mathbf{V}$ to construct the degree-1 commitments to the bits of $\mathbf{w}$, and uses $(\bar{\mathbf{u}}_j, \bar{\mathbf{v}}_j)_{j\in[0..d_{\mathsf{zk}}-1)}$ as the field-mask correlations for the QuickSilver check. It calls

$$(\tilde{a}_0, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}) \leftarrow \mathsf{ZK.OWFProve}(\mathbf{w}, \mathbf{V}, (\bar{\mathbf{u}}_j, \bar{\mathbf{v}}_j)_{j\in[0..d_{\mathsf{zk}}-1)}, \mathsf{pk}, \mathsf{chall}_2; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}),$$

which generates the d_{zk} QuickSilver proof values $\tilde{a}_0, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}$.
Using these proof components and after initializing a counter ctr for the grinding step, the signer creates the final Fiat–Shamir challenge chall_3. It repeatedly calls Using these proof components, the signer repeatedly computes

$$\mathsf{chall}_3 := \mathsf{H}_2^3\big(\mathsf{chall}_2 \| \mathsf{ToBits}((\tilde{a}_j)_{j\in[0..d_{\mathsf{zk}})}; \lambda, d_{\mathsf{zk}}) \| \mathsf{BitDec}(\mathsf{ctr}, 32); \lambda\big).$$

It checks whether $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}}..\lambda) = 0^{w_{\mathsf{grind}}}$. Once it has found such a challenge, it calls $I \leftarrow \mathsf{DecodeAllChall}_3(\mathsf{chall}_3; \mathtt{param})$. The challenge decoder checks the grinding suffix and decodes the first $\lambda - w_{\mathsf{grind}}$ bits into $I = (\Delta_0, \ldots, \Delta_{\tau-1})$. The signer then calls $\mathsf{decom}_I \leftarrow \mathsf{BAVC.Open}(\mathsf{decom}, I; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$. If the opening procedure returns $\perp$, the signer increments ctr and repeats the grinding step.

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 gate corrections $\mathbf{c}_{\mathsf{mult}}$ for the VOLE generation;
- The hashed VOLE secret $\widetilde{\mathbf{u}}$;
- The commitment to the extended witness $\mathbf{d}$;
- The QuickSilver proof parts $\tilde{a}_1, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}$;
- The partial decommitment decom_I;
- The last challenge chall_3;
- The 128-bit string $\mathsf{iv}^{\mathsf{pre}}$;
- The final counter ctr.

The signature is composed by the elements given in the ordered list above. Each $\mathbf{c}_i$ and $\mathbf{d}$ consists of an ℓ-bit vector; every element of $\mathbb{F}_{2^\lambda}$ is described using $\mathsf{ToBits}(\cdot; \lambda)$; decom_I has length $n_{\mathsf{leafcom}}\lambda\tau + T_{\mathsf{open}}\lambda$; chall_3 has length λ; $\mathsf{iv}^{\mathsf{pre}}$ has length 128; and ctr is given as $\mathsf{BitDec}(\mathsf{ctr}, 32)$.

The gate corrections $\mathbf{c}_{\mathsf{mult}} \in \{0,1\}^{n_{\mathsf{mask}} \times \bar{n}_{\mathsf{mult}}}$ are represented in row-major order in both the signature and the input to H_2^1. Since $\bar{n}_{\mathsf{mult}} = 8\lceil n_{\mathsf{mult}}/8 \rceil$, each row occupies exactly $\lceil n_{\mathsf{mult}}/8 \rceil$ bytes, with $\bar{n}_{\mathsf{mult}} - n_{\mathsf{mult}}$ bits of zero padding.

8.3 Verification

The verification algorithm FAEST.Verify is shown in Figure 8.3. The verifier begins by parsing the signature σ into its components: the correction vectors $\{\mathbf{c}_i\}$, the gate corrections $\mathbf{c}_{\mathsf{mult}}$, the hashed VOLE secret $\widetilde{\mathbf{u}}$, the witness commitment $\mathbf{d}$, the QuickSilver proof pieces $\tilde{a}_1, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}$, 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 $\mu := \mathsf{H}_2^0(\mathsf{pk} \| \mathsf{msg}; 2\lambda)$ and obtains $\mathsf{iv} = \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu)$. 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

$$(\mathsf{com}, \mathbf{Q}, (\bar{\mathbf{q}}_m)_{m\in[0..n_{\mathsf{mask}})}, \Delta) \leftarrow \mathsf{FAEST.VOLEReconstruct}(\mathsf{chall}_3, \mathsf{decom}_I, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}}),$$

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

INPUT: $\mathsf{msg} \in \{0,1\}^*$, $\mathsf{pk} \in \{0,1\}^{N_{\text{st-bits}}} \times \{0,1\}^{\beta N_{\text{st-bits}}}$, and $\sigma \in \{0,1\}^*$

OUTPUT: $b \in \{\mathtt{false}, \mathtt{true}\}$

Parse $\sigma = ((\mathbf{c}_i)_{i\in[1..\tau)}, \mathbf{c}_{\mathsf{mult}}, \widetilde{\mathbf{u}}, \mathbf{d}, \tilde{a}_1, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}, \mathsf{decom}_I, \mathsf{chall}_3, \mathsf{iv}^{\mathsf{pre}}, \mathsf{ctr})$
if parsing fails **then**
  return false
$\mu \leftarrow \mathsf{H}_2^0(\mathsf{pk}\|\mathsf{msg}; 2\lambda)$
$\mathsf{iv} \leftarrow \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}}\|\mu)$
if $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}}..\lambda) \neq 0^{w_{\mathsf{grind}}}$ **then**
  return false
// Reconstruct the witness VOLE, the CRT field masks, and the field challenge
$\mathsf{rec} \leftarrow \mathsf{FAEST.VOLEReconstruct}(\mathsf{chall}_3, \mathsf{decom}_I, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$
if $\mathsf{rec} = \perp$ **then**
  return false
$(\mathsf{com}, \mathbf{Q}, (\bar{\mathbf{q}}_m)_{m\in[0..n_{\mathsf{mask}})}, \Delta) \leftarrow \mathsf{rec}$
$\mathsf{chall}_1 \leftarrow \mathsf{H}_2^1(\mu\|\mathsf{com}\|\mathbf{c}_1\|\cdots\|\mathbf{c}_{\tau-1}\|\mathbf{c}_{\mathsf{mult}}\|\mathsf{iv}^{\mathsf{pre}}; 8\lambda+64)$
$\mathbf{Q}_0^{\mathsf{h}} := (\mathbf{Q}[0], \ldots, \mathbf{Q}[\ell-1], \bar{\mathbf{q}}_0, \ldots, \bar{\mathbf{q}}_{d_{\mathsf{zk}}-2}) \in \mathbb{F}_{2^\lambda}^{\ell+d_{\mathsf{zk}}-1}$
$\mathbf{Q}_1^{\mathsf{h}} := (\bar{\mathbf{q}}_{d_{\mathsf{zk}}-1}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}+1}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}+2}) \in \mathbb{F}_{2^\lambda}^4$
$\widetilde{\mathbf{Q}} := \mathsf{VOLEHash}(\mathsf{chall}_1, (\mathbf{Q}_0^{\mathsf{h}}, \mathbf{Q}_1^{\mathsf{h}})) \in \mathbb{F}_{2^\lambda}^4$
$\mathbf{D} := \widetilde{\mathbf{Q}} + \Delta \cdot \widetilde{\mathbf{u}} \in \mathbb{F}_{2^\lambda}^4$
$\mathsf{chall}_2 \leftarrow \mathsf{H}_2^2\big(\mathsf{chall}_1\|\mathsf{ToBits}(\widetilde{\mathbf{u}}; \lambda, 4)\|\mathsf{ToBits}(\mathbf{D}; \lambda, 4)\|\mathbf{d}; 3\lambda+64\big)$
// Verify the degree-d_{zk} QuickSilver proof using the first $d_{\mathsf{zk}}-1$ field-mask correlations
$\tilde{a}_0 \leftarrow \mathsf{ZK.OWFVerify}(\mathbf{d}, \mathbf{Q}, (\bar{\mathbf{q}}_j)_{j\in[0..d_{\mathsf{zk}}-1)}, \mathsf{pk}, \mathsf{chall}_2, \Delta, \tilde{a}_1, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
$\mathsf{chall}_3' \leftarrow \mathsf{H}_2^3\big(\mathsf{chall}_2\|\mathsf{ToBits}((\tilde{a}_j)_{j\in[0..d_{\mathsf{zk}})}; \lambda, d_{\mathsf{zk}})\|\mathsf{BitDec}(\mathsf{ctr}, 32); \lambda\big)$
return true if $\mathsf{chall}_3 = \mathsf{chall}_3'$, and false otherwise

Fig. 8.3: FAEST verification algorithm.

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, add each VOLE correction vector $\mathbf{c}_i$, and finally produce the VOLE commitment com along with a re-constructed matrix $\mathbf{Q}$ of VOLE tags for the proofs and $(\bar{\mathbf{q}}_m)_{m\in[0..n_{\mathsf{mask}})}$ for the checks. The function also returns Δ as used by the QuickSilver proof. If VOLEReconstruct fails or returns $\perp$, the verifier rejects. Otherwise, it re-computes the first challenge $\mathsf{chall}_1 := \mathsf{H}_2^1(\mu\|\mathsf{com}\|\mathbf{c}_1\|\ldots\|\mathbf{c}_{\tau-1}\|\mathbf{c}_{\mathsf{mult}}\|\mathsf{iv}^{\mathsf{pre}}; 8\lambda+64)$.

Next, the verifier splits up the $\mathbf{Q}$ VOLE tags in those that are hashed using VOLEHash ($(\mathbf{Q}_0^{\mathsf{h}})$ and those that serve as blinding ($\mathbf{Q}_1^{\mathsf{h}}$) and evaluates the hash with input chall_1, computing $\widetilde{\mathbf{Q}}$. It then calculates the value $\mathbf{D} := \widetilde{\mathbf{Q}} + \Delta \cdot \widetilde{\mathbf{u}}$ that should correspond to a correct evaluation of the VOLE hash on the prover's secret $\mathbf{V}$ and its blinding values. This permits to reconstruct

$$\mathsf{chall}_2 := \mathsf{H}_2^2\big(\mathsf{chall}_1\|\mathsf{ToBits}(\widetilde{\mathbf{u}}; \lambda, 4)\|\mathsf{ToBits}(\mathbf{D}; \lambda, 4)\|\mathbf{d}; 3\lambda+64\big).$$

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

$$\tilde{a}_0 \leftarrow \mathsf{ZK.OWFVerify}\big(\mathbf{d}, \mathbf{Q}, (\bar{\mathbf{q}}_j)_{j\in[0..d_{\mathsf{zk}}-1)}, \mathsf{pk}, \mathsf{chall}_2, \Delta, \tilde{a}_1, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}\big),$$

recovering the missing piece $\tilde{a}_0$. With all d_{zk} proof values, the verifier is able to compute $\mathsf{chall}_3' \leftarrow \mathsf{H}_2^3\big(\mathsf{chall}_2\|\mathsf{ToBits}((\tilde{a}_j)_{j\in[0..d_{\mathsf{zk}})}; \lambda, d_{\mathsf{zk}})\|\mathsf{BitDec}(\mathsf{ctr}, 32); \lambda\big)$. If chall_3' does not match the provided chall_3, the verifier rejects; otherwise, it accepts.

9 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 modern 64-bit CPUs It supports x86-64 with the AVX2 and AES-NI instruction set extensions, as commonly seen in Intel and AMD CPUs, as well as AArch64 (ARM) which are common in mobile devices and Apple M-series 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].[15]

Benchmarking Setup. We measured the performance of both implementations using a single core of the benchmarking system while having the systems otherwise idle.

We consider two different benchmarking platforms:

(1) A notebook with an Intel Core Ultra 9 285H processor of the Arrow Lake family running Linux 7.1.9. We compile with GCC 16.2.1 and pin the benchmark to a performance core with up to 5.4 GHz.
(2) A 13-inch MacBook Pro (2020) with an Apple M1 processor running macOS 26.5, with performance cores of up to 3.2 GHz. We compile with Apple Clang 21.0.0 for the `armv8-a+crypto` target.
(3) A Google Pixel 6a phone from 2022 running GrapheneOS (Android 17) after a reboot. We compile with Clang 21.0.0 Android NDK r29 and pin the benchmark to a Cortex-X1 core with up to 2.8 GHz).

Our benchmarks are implemented using the Catch2 framework[16]. Additionally, we count CPU cycles on the x86-64 platform using code from PQShield's Signature Zoo[17]. Runtimes of the respective algorithms are given in milliseconds and CPU cycles. Times and cycles for the architecture-specific implementation were averaged over 1000 runs on platforms (1) and (2), while the times for the phone (3) and for the reference implementation were averaged over 100 runs.

Performance. Table 9.1 shows the performance of the architecture-specific implementation on the benchmarking platforms. For completeness, we also provide runtimes of our reference implementation in Table 9.2, *which is not optimized for performance.* Finally, Table 9.3 summarizes the changes compared to FAEST v2 of the signing and verification performance (measured on platform (1)), as well as signature size.

[15] 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.

[16] https://github.com/catchorg/Catch2

[17] https://github.com/PQShield/nist-sigs-zoo/blob/1dddd7e76a7565d350a90cd9da333fecf8fc946f/bench-common/harness_common.h

Table 9.1: Benchmark results for the architecture-specific implementation.

Scheme	Runtimes (in ns, cyc.)		Runtimes (in ms, Mcyc.)			
	KeyGen		Sign		Verify	
Intel Arrow Lake Notebook (System (1))						
FAEST-128f	77	389	0.352	1.7	0.239	1.2
FAEST-128s	77	388	2.172	10.6	1.679	8.1
FAEST-192f	138	696	1.817	9.1	1.422	7.3
FAEST-192s	137	695	10.824	53.9	8.428	42.6
FAEST-256f	165	839	1.981	9.7	1.603	7.9
FAEST-256s	165	837	12.860	62.6	12.417	60.6
FAEST-EM-128f	83	420	0.307	1.5	0.190	0.9
FAEST-EM-128s	83	421	1.532	7.5	1.176	5.7
FAEST-EM-192f	143	726	1.136	5.7	0.845	4.3
FAEST-EM-192s	143	723	6.374	31.9	5.793	29.1
FAEST-EM-256f	190	964	1.702	8.5	1.416	6.9
FAEST-EM-256s	190	965	10.761	51.4	9.980	49.4
Apple M1 MacBook (System (2))						
FAEST-128f	601		0.727		0.480	
FAEST-128s	589		4.758		3.354	
FAEST-192f	624		2.191		1.701	
FAEST-192s	615		17.090		10.625	
FAEST-256f	622		3.229		2.658	
FAEST-256s	620		20.616		20.050	
FAEST-EM-128f	582		0.685		0.386	
FAEST-EM-128s	590		3.595		2.574	
FAEST-EM-192f	594		1.520		1.174	
FAEST-EM-192s	591		10.987		9.589	
FAEST-EM-256f	593		2.701		2.147	
FAEST-EM-256s	594		16.393		15.170	
Google Pixel 6a Phone (System (3))						
FAEST-128f	4 699		1.177		0.810	
FAEST-128s	4 760		8.690		6.630	
FAEST-192f	4 745		4.343		3.325	
FAEST-192s	4 801		32.540		22.559	
FAEST-256f	4 929		6.792		5.531	
FAEST-256s	4 805		46.909		46.469	
FAEST-EM-128f	5 823		1.336		0.810	
FAEST-EM-128s	5 984		7.533		5.675	
FAEST-EM-192f	6 067		3.244		2.532	
FAEST-EM-192s	5 758		23.694		22.409	
FAEST-EM-256f	5 952		6.203		4.896	
FAEST-EM-256s	6 181		40.598		38.304	

Table 9.2: Benchmark results for the reference implementation measured on benchmarking system (1).

Scheme	Runtimes (in µs)	Runtimes (in ms)	
	KeyGen	Sign	Verify
FAEST-128f	0.080	20.335	11.148
FAEST-128s	0.079	38.930	29.011
FAEST-192f	0.092	92.968	49.427
FAEST-192s	0.092	152.313	103.969
FAEST-256f	0.114	132.376	72.850
FAEST-256s	0.113	246.704	187.455
FAEST-EM-128f	0.082	15.636	7.785
FAEST-EM-128s	0.083	18.533	10.235
FAEST-EM-192f	15.912	62.417	30.798
FAEST-EM-192s	15.866	71.113	38.321
FAEST-EM-256f	26.476	110.206	54.899
FAEST-EM-256s	26.413	121.449	65.475

Table 9.3: Changes in runtime and signature size compared to FAEST v2. The runtime are from the architecture-specific implementation for x86-64 with AVX2 measured on benchmarking system (1).

Scheme	Sign (ms)			Verify (ms)			Signature (B)		
	v3	v2	Change	**v3**	v2	Change	**v3**	v2	Change
FAEST-128f	0.352	0.302	16.6 %	0.239	0.244	**−1.9 %**	5 170	5 924	**−12.7 %**
FAEST-128s	2.172	2.192	**−0.9 %**	1.679	1.726	**−2.7 %**	4 066	4 506	**−9.8 %**
FAEST-192f	1.817	1.757	3.4 %	1.422	1.537	**−7.5 %**	11 738	14 948	**−21.5 %**
FAEST-192s	10.824	11.195	**−3.3 %**	8.428	8.928	**−5.6 %**	9 410	11 260	**−16.4 %**
FAEST-256f	1.981	1.862	6.4 %	1.603	1.791	**−10.4 %**	20 856	26 548	**−21.4 %**
FAEST-256s	12.860	13.479	**−4.6 %**	12.417	13.428	**−7.5 %**	16 626	20 696	**−19.7 %**
FAEST-EM-128f	0.307	0.229	34.2 %	0.190	0.187	1.7 %	4 170	5 060	**−17.6 %**
FAEST-EM-128s	1.532	1.596	**−4.1 %**	1.176	1.286	**−8.6 %**	3 466	3 906	**−11.3 %**
FAEST-EM-192f	1.136	1.142	**−0.5 %**	0.845	0.996	**−15.2 %**	9 818	12 380	**−20.7 %**
FAEST-EM-192s	6.374	6.887	**−7.5 %**	5.793	6.436	**−10.0 %**	7 874	9 340	**−15.7 %**
FAEST-EM-256f	1.702	1.585	7.4 %	1.416	1.573	**−10.0 %**	18 084	23 476	**−23.0 %**
FAEST-EM-256s	10.761	11.201	**−3.9 %**	9.980	10.975	**−9.1 %**	14 554	17 984	**−19.1 %**

10 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.

10.1 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 10.2, 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.

10.1.1 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 10.2.4, 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.

10.1.2 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[18] 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.

10.1.3 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 10.30. 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 10.30, 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 10.13), which cannot be efficiently verified. We discuss its plausibility in Remark 10.14; 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

[18] 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.

for an attacker 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 10.30 is due to our proof technique and related to tightly secure signatures following the approach of Katz and Wang [KW03].)

10.1.4 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.

10.2 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.

10.2.1 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 10.1. *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.[19] 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 $P(k', u_i) = P(k, u_i)$.

[19] 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.

10.2.2 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.,

$$F_k^\pi(z) = k + \pi(k + z)\,. \tag{4}$$

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:

$$\Pr[z \leftarrow \{0,1\}^\lambda, k \leftarrow \{0,1\}^\lambda, k' \leftarrow A^\pi(z, F_k^\pi(z)) : F_{k'}^\pi(0) = F_k^\pi(0)]. \tag{5}$$

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 10.2. *The single-key Even–Mansour construction (Equation (4)) 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 (5) 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.

10.2.3 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 10.1 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 10.3 (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 10.1 with $m = \lambda$ and $\beta = 1$.

Corollary 10.4. *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.

10.2.4 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.

10.2.5 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].

10.3 Preliminaries for Security Reductions

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

10.3.1 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.

10.3.2 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 the hardness assumptions on the hashes $\mathsf{H}_0, \ldots, \mathsf{H}_4$ explicit; these are idealized as random oracles in the security proofs. We use the standard notions of computational indistinguishability, pseudorandom functions and pseudorandom permutations.

Definition 10.5 (Indistinguishability). *Let $\mathcal{D}_0, \mathcal{D}_1$ be two distributions. Let $\mathcal{A}$ be an adversary in the following experiment $\mathsf{ExpDist}_{\mathcal{A}}$:*

1. $x_0 \leftarrow \mathcal{D}_0$, $x_1 \leftarrow \mathcal{D}_1$, $b \leftarrow \{0,1\}$,
2. $b' = \mathcal{A}(1^\lambda, x_b)$
3. *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 $\mathbf{k} \leftarrow \mathcal{K}_\lambda$, oracle-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 $\mathbf{k} \in \mathcal{K}_\lambda$.

10.3.3 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 10.6 (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 the PRG construction from Figure 3.6, we define two types of security, namely PRP-type and PRF-type security in Definition 10.7, which capture a multi-instance setting where an adversary receives multiple PRG outputs for the same key, under different tweaks (the tweak is viewed as the input to the PRP or PRF). The PRP-type security notion is exactly what is achieved by modeling AES as a pseudorandom permutation. While the PRP/PRF switching lemma could be used to show PRF-type security, this would lead to a poor concrete bound, due to birthday attacks and the fixed block size of 128 in AES. However, by using a divergence-based switching lemma (Corollary 10.10), we can still show PRF-type security in our setting, where signature queries will be limited to $Q_{\mathsf{sig}} \leq 2^{64}$.

Definition 10.7 (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 $b \leftarrow \{0,1\}$, 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 $i = [1, \alpha]$ sample $y_i \leftarrow \{0,1\}^{128} \setminus B$, and 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.

Definition 10.7 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 10.8 (PRP-type security). It is easy to see that our PRG constructed in Figure 3.6 satisfies PRP-type security of Definition 10.7 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 in the following.

10.3.4 Divergence Analysis for PRF-type Security.

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 10.9 (∞-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 10.10. *Let $\mathcal{X}$ be a set of size N and let $n \in \mathbb{N}$ with $n \leq N$. 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 10.9, it suffices to show $\frac{\Pr[P=x]}{\Pr[R=x]} \leq e^{\frac{n(n-1)}{N}}$. Since $\Pr[P=x] \leq \frac{1}{N\cdot(N-1)\cdot\ldots\cdot(N-n+1)}$ and $\Pr[R=x] = \frac{1}{N^n}$, we have to show

$$\frac{N^n}{(N\cdot(N-1)\cdot(N-n+1))} = \left(\prod_{i=1}^{n-1}\left(1-\frac{i}{N}\right)\right)^{-1} \overset{(?)}{\leq} e^{\frac{n(n-1)}{N}}.$$

When $n-1 \leq 0.79N$, this follows directly from the bound $e^{-x} \leq 1 - x/2$ for $\frac{x}{2} \in [0, 0.79]$. To handle general $n \leq N$, we need a more technical argument. First, we take logarithms to express the claim to prove as

$$\sum_{i=1}^{n-1} \ln\left(1-\frac{i}{N}\right) \overset{(?)}{\geq} -\frac{n(n-1)}{N}. \tag{6}$$

Observe that $f(x) = \ln(1-x)$ is concave and monotone decreasing over $[0,1)$ (since $f'(x) = -\frac{1}{1-x} < 1$, and $f''(x) = -(1-x)^{-2} < 0$). Dividing the sum by N, we can interpret it as a right-sided Riemann sum which approximates an integral over f with $n-1$ rectangles of width $1/N$. Let $g(x) := -f(x)$ so that $g(x)$ is a monotone increasing function over $[0,1)$. Let $S_L := \frac{1}{N}\cdot\sum_{i=0}^{n-2} g(\frac{i}{N})$ and $S_R := \frac{1}{N}\cdot\sum_{i=1}^{n-1} g(\frac{i}{N})$ denote the left-sided and right-sided Riemann sums bounding the integral: $S_L \leq \int_0^{\frac{n-1}{N}} g(x)\,\mathrm{d}x \leq S_R$. Note that the difference between the Riemann sums is exactly $S_R - S_L = \frac{1}{N}\cdot g(\frac{n-1}{N}) - \frac{1}{N}\cdot g(\frac{0}{N}) = \frac{1}{N}\cdot g(\frac{n-1}{N})$ by construction as $g(0) = 0$. Thus we arrive at the inequality

$$\frac{1}{N}\cdot\sum_{i=1}^{n-1} g\left(\frac{i}{N}\right) = S_R = S_L + \frac{1}{N}\cdot g\left(\frac{n-1}{N}\right) \leq \int_0^{\frac{n-1}{N}} g(x)\,\mathrm{d}x + \frac{1}{N}\cdot g\left(\frac{n-1}{N}\right).$$

Plugging in $g(x) = -\ln(1-x)$ and simplifying the resulting lower bound, we find

$$\begin{aligned}\frac{1}{N}\cdot\sum_{i=1}^{n-1}\ln\left(1-\frac{i}{N}\right) &\geq \int_0^{\frac{n-1}{N}} \ln(1-x)\,\mathrm{d}x + \frac{1}{N}\cdot\ln\left(1-\frac{n-1}{N}\right)\\ &= \Big[-(1-x)\cdot\ln(1-x) - x\Big]_0^{\frac{n-1}{N}} + \frac{1}{N}\cdot\ln\left(1-\frac{n-1}{N}\right)\\ &= -\left(1-\frac{n-1}{N}\right)\cdot\ln\left(1-\frac{n-1}{N}\right) - \frac{n-1}{N} + \frac{1}{N}\cdot\ln\left(1-\frac{n-1}{N}\right)\\ &= -\left(1-\frac{n}{N}\right)\cdot\ln\left(1-\frac{n-1}{N}\right) - \frac{n-1}{N}.\end{aligned}$$

Multiplying Equation (6) by $\frac{1}{N}$ and using the above bound, it suffices to show that

$$-\left(1-\frac{n}{N}\right)\cdot\ln\left(1-\frac{n-1}{N}\right)-\frac{n-1}{N}\overset{(?)}{\geq}-\frac{n(n-1)}{N^2}. \tag{7}$$

First, suppose that $n=N$. Then the claim holds because $(1-\frac{n}{N})=0$ and $\frac{n-1}{N}=\frac{n(n-1)}{N^2}$. Now assume $n<N$. Then we can rewrite Equation (7) as

$$-\ln\left(1-\frac{n-1}{N}\right)\overset{(?)}{\geq}\frac{1}{\left(1-\frac{n}{N}\right)}\left(-\frac{n(n-1)}{N^2}+\frac{n-1}{N}\right)=\frac{n-1}{N}.$$

From the inequality $-\ln(1-x)\geq x$ for $x<1$, the above claim follows for all $n<N$. This proves Equation (6) and completes the proof. □

In particular, we can use Corollary 10.10 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 10.11 (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.

We are now ready to prove PRF-type security of the PRG.

Lemma 10.12. *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 for PRF-type security of PRG (Definition 10.7). Let $\mathsf{G}_{\mathsf{PRP}}$ denote the output of $\mathcal{A}$ in the 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*

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

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 10.10, 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

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

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 10.13 (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 10.13. 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 10.13 for the PRG constructed in Figure 3.6 under heuristic simplifications.

Remark 10.14 (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 $\frac{Q^2_{\mathsf{AES}}}{n} + \frac{Q_{\mathsf{out}}(m - Q_{\mathsf{AES}})}{n} = \frac{Q_{\mathsf{AES}}(Q_{\mathsf{AES}} - Q_{\mathsf{out}})}{n} + \frac{Q_{\mathsf{out}} \cdot m}{n}$.

10.4 Bounding the Preimages of H_4

In our EU-CMA proof, we will need to use a preimage bound on the hash function H_4, which is used to derive $\mathsf{iv} = \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu)$, where μ is the hash of the message and public key.

Definition 10.15. *Let* $h : \{0,1\}^{128+2\lambda} \to \{0,1\}^{128}$. *We say* h *is* K*-preimage bounded if for every* $\mu \in \{0,1\}^{2\lambda}$ *and* $v \in \{0,1\}^{128}$,

$$\left|\{x \in \{0,1\}^{128} : \mathsf{H}_4(x \| \mu) = v\}\right| \leq K.$$

Lemma 10.16 (Preimage-bounds for $H\colon [M]\times[N]\to[M]$). *Let $M, N \in \mathbb{N}$ and let $H\colon [M]\times[N]\to[M]$ be a random oracle. Let $K \in \mathbb{N}$ and let Fail be the event over the random choice of H, that*

$$\exists n \in [N] \;:\; |\{x \in [M] : \mathsf{H}_4(x\|\mu) = v\}| > K$$

which means that for some $n \in [N]$, the function $x \mapsto H(x,n)$ fails to be at most K-to-one. Then

$$\Pr[\mathsf{Fail}] \;\le\; \frac{M\cdot N}{(K+1)!} \;\le\; M\cdot N\cdot \Big(\frac{e}{K+1}\Big)^{K+1},$$

Proof. Observe that for each $n \in [N]$, the functions $H_n \coloneqq H(\cdot, n)$ are i.i.d. as a uniform $G\colon [M]\to[M]$. Thus, we first bound

$$\Pr[\exists y \in [M]\colon \big|G^{-1}(x)\big| > K].$$

Towards this, fix some $y \in [M]$ and observe that $\big|G^{-1}(y)\big| > K$ means that there is *some* subset of $K+1$ elements of $[M]$ that all map to y. Therefore,

$$\Pr[\big|G^{-1}(y)\big| > K] \le \binom{M}{K+1} M^{-(K+1)} \le \frac{M^{K+1}}{(K+1)!} M^{-(K+1)} = \frac{1}{(K+1)!}$$

where we used $\binom{n}{k} \le n^k/k!$ in the second inequality. Taking a union bound over $y \in [M]$, we find

$$\Pr[\exists y \in [M]\colon \big|G^{-1}(x)\big| > K] \le \frac{M}{(K+1)!}. \tag{8}$$

The probability p that some H_n is not K-to-one is the same for each $n \in [N]$, since all H_n are i.i.d. as G, by a union bound

$$\Pr[\mathsf{Fail}] \le Np \le \frac{MN}{(K+1)!} \le MN\Big(\frac{e}{K+1}\Big)^{K+1}$$

where the penultimate inequality follows by Equation (8), and the final inequality by $(n+1)! \ge (n/e)^n$. □

Lemma 10.17 (Uniform distribution under at most K-to-one function). *Let $\mathcal{X}, \mathcal{Y}$ be finite and let $F\colon \mathcal{X}\to\mathcal{Y}$ be at most K-to-one, i.e., for any $y \in \mathcal{Y}$, the number of preimages of y satisfies $|\{x \in [L]\colon F(x) = y\}| \le K$. Then for any $y \in \mathcal{X}$, we have for uniformly random $x \leftarrow \mathcal{X}$ that*

$$\Pr[f(x) = y] \le \frac{K}{|\mathcal{X}|}$$

Proof. Since $\big|f^{-1}(y)\big| \le K$ and x is uniform, we get $\Pr[f(x) = y] = \Pr[x \in f^{-1}(y)] \le \frac{K}{|\mathcal{X}|}$. □

Corollary 10.18 (H_4 is preimage bounded). *Let $\mathsf{H}_4\colon \{0,1\}^{128+2\lambda}\to\{0,1\}^{128}$ be modelled as a random oracle and let $K \in \mathbb{N}$. Then, H_4 is K-preimage bounded, except with probability at most $2^{2\lambda+128}/(K+1)!$, which is $\le 2^{-\lambda}$ for $K = K_{\max} \coloneqq \lambda$.*

$$\Pr[\mathsf{H}_4 \textit{ is not } K\textit{-preimage bounded}] \;\le\; \frac{2^{2\lambda+128}}{(K+1)!} \;\le\; 2^{2\lambda+128}\Big(\frac{e}{K+1}\Big)^{K+1}.$$

Moreover, this is $\le 2^{-\lambda}$ for any $K \ge \frac{3\lambda+128}{\log_2(3\lambda+128)-\log_2 e-\log_2\log_2(3\lambda+128)} - 1$. In particular, $K \ge \lambda \ge 128$ satisfies this.

Proof. The first claim follows immediately from Lemma 10.16 after setting $N = 2^{2\lambda}$ and $M = 2^{128}$. For the claim of $2^{-\lambda}$, when $K+1 = \frac{3\lambda+128}{\log_2(3\lambda+128)-\log_2 e-\log_2\log_2(3\lambda+128)}$, then we have

$$\begin{aligned}
\log_2 \frac{K+1}{e} &= \log_2(3\lambda+128) - \log_2(e) - \log_2\Big(\log_2(3\lambda+128) - \log_2 e - \log_2\log_2(3\lambda+128)\Big) \\
&\geq \log_2(3\lambda+128) - \log_2(e) - \log_2\log_2(3\lambda+128) \\
K+1 &= \frac{3\lambda+128}{\log_2(3\lambda+128) - \log_2 e - \log_2\log_2(3\lambda+128)} \\
&\geq \frac{3\lambda+128}{\log_2 \frac{K+1}{e}}.
\end{aligned}$$

It can be checked that $K+1 \geq e$, so we have that $\left(\frac{K+1}{e}\right)^{K+1}$ is increasing. Therefore,

$$\left(\frac{K+1}{e}\right)^{K+1} \geq \left(\frac{K+1}{e}\right)^{\frac{3\lambda+128}{\log_2 \frac{K+1}{e}}} = 2^{3\lambda+128},$$

and so $2^{2\lambda+128}\left(\frac{e}{K+1}\right)^{K+1} \leq 2^{-\lambda}$. □

10.5 Properties of VOLECommit

10.5.1 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 all complete responses to which the adversary can later open with high probability. The analysis is similar to [BBD+23, 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 complete reconstructed VOLE responses, rather than the original vectors committed using VC (even though these are also still binding). By arguing security for our specific construction, we also avoid the security loss incurred by generic reductions in [BBD+23]. (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.

Throughout this subsection, H_5 is a fixed public function independent of the later protocol challenges. We also set $\mathbb{K} := \mathbb{F}_{2^\lambda}$ and define the set of field challenges $\mathcal{D}_\Delta := \{\mathsf{ToField}(\mathbf{W}_{\mathrm{crt}}\Delta'; \lambda) : \Delta' \in \mathbb{F}_2^{\lambda - w_{\mathsf{grind}}}\}$.

Lemma 10.19 (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 and set* $t := 0$.
   (a) *Set* $t \leftarrow t+1$ *and let* $\mathbf{C} = (\mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}^{\mathsf{pre}}, \mu) \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_4}(\mathtt{commit})$.
   (b) $\mathbf{S} = \mathsf{Ext}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_4}(\mathcal{L}_0, \mathcal{L}_1, \mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}^{\mathsf{pre}}, \mu)$*, 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* $\mathcal{C}[t] = (\mathbf{C}, \mathbf{S})$. *If* $\mathbf{S} = \perp$*, then set* $\mathbf{S} := \emptyset$*, equivalently,* $\emptyset$ *is the graph of the everywhere-*$\perp$ *partial response function.*
   (d) $\mathcal{A}$ *can choose to go to Step 1a or it outputs* $(t^*, \mathsf{chall}_3, \mathsf{decom}_I)$ *where* $t^* \in [1..t]$ *and the game continues below.*
2. Decommitment-Phase: *Set*

$$\begin{aligned}
(\mathbf{C}, \mathbf{S}) &:= \mathcal{C}[t^*] \\
\mathbf{C} &:= (\mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}^{\mathsf{pre}}, \mu) \\
\mathsf{iv} &:= \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu) \\
\mathsf{rec} &:= \mathsf{VOLEReconstruct}^{\mathsf{H}_0,\mathsf{H}_1}(\mathsf{chall}_3, \mathsf{decom}_I, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})
\end{aligned}$$

3. *If* $\mathsf{rec} = \bot$ *then output* 0 *(failure).*
4. *Otherwise, parse* $\mathsf{rec} = (\overline{\mathsf{com}}, \mathbf{Q}, (\overline{\mathbf{q}}_m)_{m\in[0..n_{\mathsf{mask}})}, \Delta)$.
5. *Output* 0 *(failure) if*
   (a) $\mathsf{com} \neq \overline{\mathsf{com}}$ *(i.e., verifier rejects), or*
   (b) $(\Delta, (\mathbf{Q}, \overline{\mathbf{q}}_0, \ldots, \overline{\mathbf{q}}_{n_{\mathsf{mask}}-1})) \in \mathbf{S}$ *(extraction correct)*
6. *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, and at most* Q_C *commitment attempts. Assume also that* $\tau > 1$ *and* $n_{\mathsf{leafcom}} N_i > 2$ *for every* $i \in [0..\tau)$*. Moreover, suppose that, immediately before outputting the tuple* $(t^*, \mathsf{chall}_3, \mathsf{decom}_I)$*,* $\mathcal{A}$ *executes the complete specified* VOLEReconstruct *algorithm on this opening and the selected first-message components, including all calls to* H_0 *and* H_1*. These calls are included in* Q_0 *and* Q_1*, respectively, and* $\mathcal{A}$ *outputs* $\bot$ *if reconstruction returns* $\bot$*. There is a deterministic (unbounded) straightline extractor* Ext *such that the advantage of* $\mathcal{A}$ *in the above game is at most*

$$\frac{(Q_1 + Q_C)^2}{2^{2\lambda}} + \frac{\tau(Q_0 + Q_C)}{2^{\lambda}}.$$

For FAEST-EM*, there is a deterministic (unbounded) straightline extractor* Ext *such that the advantage is at most*

$$\frac{(Q_1 + Q_C)^2}{2^{2\lambda}} + \sum_{t=1}^{Q_C} \mathsf{AdvInj}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_t}[L] \leq \frac{(Q_1 + Q_C)^2}{2^{2\lambda}} + Q_C \max_t \mathsf{AdvInj}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_t}[L].$$

The running time of every $\mathcal{B}_t$ *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* $[L-1, L+\tau-1]$.

In both variants, for every indexed attempt t*, writing* $\mathbf{S}_t$ *for the set stored in* $\mathcal{C}[t]$*, there is a partial response function*

$$\mathsf{Resp}_t : \mathcal{D}_\Delta \longrightarrow \mathbb{K}^{\ell+n_{\mathsf{mask}}} \cup \{\bot\},$$

fixed when Ext *is run in the Commit-Phase, such that* $\mathbf{S}_t = \{(\Delta, \mathsf{Resp}_t(\Delta)) : \mathsf{Resp}_t(\Delta) \neq \bot\}$ *and* $|\mathbf{S}_t| \leq 2^{\lambda - w_{\mathsf{grind}}}$.

Intuitively, Lemma 10.19 asserts that extraction succeeds unless $\mathcal{A}$ cheats by breaking collision or preimage resistance of the random oracle, or some parsed leaf commitment is consistent with two distinct induced seeds; the probability of the latter case is bounded separately in the lemma and it gives the advantage bounds in the lemma. The extractor will brute-force some preimage and thus be inefficient. This will be unproblematic for FAEST since the binding property will be statistical and the proof system will be statistically sound.
Note that $\mathcal{A}$ chooses the challenge chall_3, hence it can cheat in the sense that only an opening for some Δ_i is "known" to $\mathcal{A}$, i.e. the (extracted) commitment cannot be opened for any other challenge than certain Δ_i. However, as long as the extractor's response set contains the complete reconstructed response, this does not constitute a success for the adversary as the adversary only wins if the opened response is not in the extracted set.

Remark 10.20. 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 10.21. Lemma 10.19 is a multi-challenge setting, i.e., the adversary may submit several first messages during the Commit-Phase, and the extractor stores a separate response graph $\mathbf{S}_t$ for every indexed attempt t. This allows a reduction to extract every attempt without guessing in advance which one the adversary will later open. If an accepted opening is inconsistent with its stored response graph, the reduction selects that attempt for the Decommitment-Phase and wins the binding game. An extractor output $\mathbf{S}_t = \emptyset$ does not by itself constitute a binding violation, since the adversary may never open that attempt or its eventual opening may be rejected. A further timing issue is that a structural H_1-preimage which was absent when $\mathbf{S}_t$ was fixed might be queried later, before the adversary supplies its opening. The marker query made by the extractor ensures that any such late preimage, or any alternative preimage, triggers the random-oracle graph bad event. Consequently, except with the probability already bounded in the lemma, every subsequently accepted opening belongs to the response graph stored for its indexed attempt.

We use following lemma to simplify the proof of Lemma 10.19.

Lemma 10.22 (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

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

Proof of Lemma 10.22. 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 at most $\frac{|V'|}{|\mathcal{Y}|}$ 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 10.19. 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 induced seeds $\mathsf{sd}_{i,j}$. Transforming the seeds, given $(\mathsf{chall}_3, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv})$ to the VOLE output $\mathbf{Q}$ and the blindings $\overline{\mathbf{q}}_m$ is simply post-processing of the opening.

We define the straightline extractor Ext: for the indexed attempt under consideration, Ext first sets $\mathsf{iv} := \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu)$ and queries $\mathsf{uhash} = \mathsf{H}_0(\mathsf{iv})$. It then looks through the list of queries $\mathcal{L}_1$ as follows.

1. Find a preimage $(h_0\| \dots \|h_{\tau-1})$ 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 $(\mathsf{com}_{i,0}, \dots, \mathsf{com}_{i,N_i-1})$ 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 $j \in [0..N_i)$, compute by exhaustive search the set

$$\begin{aligned}\mathsf{SeedPre}_{\mathsf{iv},i,\mathsf{uhash}}(\mathsf{com}_{i,j}) := \{s \in \{0,1\}^\lambda : \\ \exists r \in \{0,1\}^\lambda \text{ such that} \\ \mathsf{LeafCommit}(r, \mathsf{iv}, i+L-1, \mathsf{uhash}) = (s, \mathsf{com}_{i,j})\}.\end{aligned}$$

   If this set is empty, set $\mathsf{sd}_{i,j} = \perp$. If it contains one element, store that induced seed as $\mathsf{sd}_{i,j}$. If it contains two distinct induced seeds, immediately return $\emptyset$.

Then, for each of the $i \in [0,\tau)$ VOLEs generated in Step 3 of VOLECommit, Ext distinguishes following cases:

- Case 1 (All seeds found): If all $(\mathsf{sd}_{i,j})_{j\in[0..N_i)}$ are not $\perp$ for i, then Ext computes

$$(\mathbf{u}_i^*, \mathbf{v}_{i,0}^*, \dots, \mathbf{v}_{i,d_i-1}^*) := \mathsf{ConvertToVOLE}(\mathsf{sd}_{i,0}, \dots, \mathsf{sd}_{i,N_i-1}, \mathsf{iv}, i+2^{31}; \hat{\ell})$$

  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 VOLEReconstruct, computes $(\delta_{i,0}^*, \dots, \delta_{i,d_i-1}^*) := \mathsf{BitDec}(\Delta_i^*, d_i)$ and

$$(\mathbf{u}_i^*, \mathbf{q}'_{i,0}, \dots, \mathbf{q}'_{i,d_i-1}) := \mathsf{ConvertToVOLE}(\perp, \mathsf{sd}_{i,1\oplus\Delta_i^*}, \dots, \mathsf{sd}_{i,(N_i-1)\oplus\Delta_i^*}, \mathsf{iv}, i+2^{31}; \hat{\ell}).$$

  It then sets $\mathbf{v}_{i,a}^* := \mathbf{q}'_{i,a} + \delta_{i,a}^* \mathbf{u}_i^*$, for every $a \in [0..d_i)$.
- Case 3 (Missing multiple seeds): If two or more seeds are $\perp$, say $\mathsf{sd}_{i,j_1^*} = \perp = \mathsf{sd}_{i,j_2^*}$ for $j_1^* \neq j_2^*$, then Ext outputs $\perp$.

Let $\mathbf{S}$ be an initially empty set, we have that the extracted $\overline{\mathbf{q}}_m$ may depend on the choice of Δ. Thus, for every choice of $\Delta' \in \mathbb{F}_2^{\lambda - w_{\mathsf{grind}}}$ we now run the following steps:

1. Compute $\Delta := \mathsf{ToField}(\mathbf{W}_{\mathrm{crt}} \cdot \Delta', \lambda)$ and set $\Delta' = (\delta_{0,0}, \dots, \delta_{0,d_0-1}, \dots, \delta_{\tau-1,0}, \dots, \delta_{\tau-1,d_{\tau-1}-1})$.
2. For each $i \in [0..\tau)$ set $\Delta_i := \mathsf{NumRec}(d_i, (\delta_{i,0}, \dots, \delta_{i,d_i-1}))$. If $\perp \neq \Delta_i^*$ and $\Delta_i^* \neq \Delta_i$ then skip to the next Δ'.
3. For each $i \in [0..\tau)$, set

$$\mathbf{Q}_i(\Delta_i) := [\mathbf{v}_{i,0}^* \ \cdots \ \mathbf{v}_{i,d_i-1}^*] + [\delta_{i,0}\mathbf{u}_i^* \ \cdots \ \delta_{i,d_i-1}\mathbf{u}_i^*].$$

   If $i \geq 1$, apply the public correction only to the witness rows:

$$\mathbf{Q}_i(\Delta_i)|_{[0..\ell)} \leftarrow \mathbf{Q}_i(\Delta_i)|_{[0..\ell)} + [\delta_{i,0}\mathbf{c}_i \ \cdots \ \delta_{i,d_i-1}\mathbf{c}_i].$$

4. Generate $\mathbf{Q}$ by setting

$$\mathbf{Q}_{\mathsf{bin}} = \left([\mathbf{Q}_0(\Delta_0) \cdots \mathbf{Q}_{\tau-1}(\Delta_{\tau-1})]|_{[0..\ell)}\right) \cdot \mathbf{W}_{\mathrm{crt}}^\top$$

   and

$$\mathbf{Q} := \left(\mathsf{ToField}(\mathbf{Q}_{\mathsf{bin}}[j], \lambda)\right)_{j\in[0..\ell)}.$$

5. In addition, for each $m \in [0..n_{\mathsf{mask}}), i \in [0..\tau)$, set

$$\tilde{\mathbf{r}}'_{m,i} := \mathsf{ToBits}\Big(\sum_{t=0}^{d_i-1} x^t \cdot \mathsf{ToPoly}\big(\mathbf{Q}_i(\Delta_i)[\ell + m \cdot d_0 + t]; d_i\big) \bmod M_i(x); d_i \Big)$$

and compute the Free-XOR wire labels as

$$(\mathbf{a}'_0, \cdots, \mathbf{a}'_{\lambda - w_{\mathsf{grind}} - 1}) := [\mathbf{Q}_0(\Delta_0) \cdots \mathbf{Q}_{\tau-1}(\Delta_{\tau-1})]|_{[0..\lambda)}.$$

6. For all $e \in [0..n_{\mathsf{mult}})$, set $\gamma_e = \mathbf{G}[e] \cdot \Delta'$ and $L'_e = \bigoplus_{j\,:\,\mathbf{G}[e][j]=1} \mathbf{a}'_j$. This allows the extractor to compute $\boldsymbol{h}_e = \mathsf{H}_5(\mathsf{iv}, e, L'_e)$.
7. For all $e \in [0..n_{\mathsf{mult}})$ and $m \in [0..n_{\mathsf{mask}})$, compute

$$\tilde{q}_{m,e} := \boldsymbol{h}_e[m] \oplus \gamma_e \cdot \mathbf{c}_{\mathsf{mult}}[m, e].$$

8. Finally, for each $m \in [0..n_{\mathsf{mask}})$, compute

$$\bar{\mathbf{q}}_m := \mathsf{ToField}\Big(\mathbf{W}_{\mathrm{tree}} \cdot (\tilde{\mathbf{r}}'_{m,0} \| \cdots \| \tilde{\mathbf{r}}'_{m,\tau-1})^\top \oplus \mathbf{W}_{\mathrm{gate}} \cdot (\tilde{q}_{m,0}, \ldots, \tilde{q}_{m,n_{\mathsf{mult}}-1})^\top, \lambda \Big).$$

9. Add $(\Delta, (\mathbf{Q}, \bar{\mathbf{q}}_0, \ldots, \bar{\mathbf{q}}_{n_{\mathsf{mask}}-1}))$ to $\mathbf{S}$.

The CRT property of $\mathbf{W}_{\mathrm{crt}}$ implies that $\Delta' \mapsto \mathsf{ToField}(\mathbf{W}_{\mathrm{crt}} \Delta'; \lambda)$ is injective, so $\mathbf{S}$ is the graph of a partial function from $\mathcal{D}_\Delta$ to $\mathbb{K}^{\ell + n_{\mathsf{mask}}}$, and $|\mathbf{S}| \le 2^{\lambda - w_{\mathsf{grind}}}$. This describes the output of Ext.

Finally, to invoke Lemma 10.22 later, we add the following actions to Ext if extraction failed:

– If extraction failed because the preimage for com in Step 1, or the preimage for some h_i in Step 2, was missing, let x be the missing value, namely $x = \mathsf{com}$ in the former case and $x = h_i$ in the latter, query $\mathsf{H}_1(x)$, and only then return $\emptyset$.

Note that Ext will make at most one query to H_1 per indexed attempt, and hence at most Q_C such queries in total.

To analyse Ext, we first consider the graph from the game in Lemma 10.22 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 to the random oracle graph game in Lemma 10.22. If Fail_1 occurs, this adversary wins the random oracle graph game: either a collision is found, or an accepted opening supplies a preimage for a value x which was missing at extraction time. In the latter case, the additional query $\mathsf{H}_1(x)$ made by Ext inserted x as a node, so the later preimage triggers the preimage or collision condition of Lemma 10.22.

As a consequence, failing to look up com or h_i can trigger an extra query to H_1 by Ext. By assumption, $\mathcal{A}$ has already run VOLEReconstruct on its purported opening, so the final reconstruction in the game only repeats stored H_1 queries and need not be forwarded as fresh queries to the random oracle graph game. Including the adversary's Q_1 queries and the at most Q_C queries, Lemma 10.22 therefore gives

$$\Pr[\mathsf{Fail}_1] \le (Q_1 + Q_C)^2 / 2^{2\lambda}.$$

As a second step, let $\mathsf{Fail}_{\mathsf{uhash}}$ be the event that, for one of the indexed attempts, some fixed $i \in [0..\tau)$, and a leaf commitment c parsed by the extractor, the set $\mathsf{SeedPre}_{\mathsf{iv},i,\mathsf{uhash}}(c)$ contains two distinct induced seeds. Observe that hash LeafHash (used in LeafCommit to hash $(\mathsf{sd}, \mathbf{x}_1)$) is $\varepsilon_{\mathsf{uhash}}$-almost universal for $\varepsilon_{\mathsf{uhash}} = 2^{-3\lambda}$. For fixed (iv, i), the 2^λ possible

roots produce at most 2^λ inputs $(\mathsf{sd}, \mathbf{x}_1)$ to LeafHash. If two roots induce distinct seeds but the same leaf commitment, their LeafHash inputs are distinct and collide; roots that induce the same seed do not cause extraction ambiguity. Since uhash is a random oracle output and thus is independent of the hashed inputs, a union bound over pairs of roots shows that, for fixed (iv, i), an induced-seed ambiguity occurs with probability less than $2^{-\lambda}$. The extractor queries $\mathsf{H}_0(\mathsf{iv})$ once for every indexed attempt, so at most $Q_0 + Q_C$ IVs are used. A union bound over those IVs and all τ choices of i gives

$$\Pr[\mathsf{Fail}_{\mathsf{uhash}}] < \tau(Q_0 + Q_C) \cdot 2^{-\lambda}.$$

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 required H_1-preimage the game will not later encounter one, and no leaf commitment encountered in the game has two distinct induced seeds. We observe the following:

- com has at most one H_1-preimage $(h_0 \| \dots \| h_{\tau-1})$.
- h_i has at most one H_1-preimage $(\mathsf{com}_{i,0}, \dots, \mathsf{com}_{i,N_i-1})$ encountered in the game, for $i \in [0, \tau)$.
- $\mathsf{com}_{i,j}$ uniquely determines its induced seed if such a seed exists.[20]

Indeed, an accepted BAVC opening reveals an induced seed for every $j \neq \Delta_i$. If exactly one induced seed was missing during extraction, acceptance therefore forces that leaf to be the hidden leaf, so $\Delta_i = \Delta_i^*$; if two or more induced seeds were missing, every choice of Δ_i would reveal at least one missing seed and no opening could be accepted. Thus, it is only possible for $\mathcal{A}$ to open com successfully (i.e., $\mathsf{com} = \overline{\mathsf{com}} \neq \bot$) if the resulting Δ_i is arbitrary when $\Delta_i^* = \bot$ and satisfies $\Delta_i = \Delta_i^*$ otherwise.

Hence, if Fail does not occur, every accepted opening respects the extracted missing-seed restrictions. If two or more induced seeds were missing for some instance i, no opening for that instance is accepted. Therefore, for every accepted opening, it remains to distinguish the following two cases for each $i \in [0..\tau)$:

- If $\Delta_i^* = \bot$, then all induced seeds of instance i were extracted (Case 1). For the challenge Δ_i appearing in the opening, Proposition 5.2 shows that the evaluator's columns
$$\mathbf{v}_{i,a}^* + \delta_{i,a}\mathbf{u}_i^*, \quad a \in [0..d_i),$$
are exactly the columns obtained by the missing-seed execution of ConvertToVOLE in VOLEReconstruct.
- If $\Delta_i^* \neq \bot$, then exactly one induced seed was missing (Case 2). Acceptance forces the corresponding leaf to be hidden, and therefore $\Delta_i = \Delta_i^*$ and $\delta_{i,a} = \delta_{i,a}^*$ for every $a \in [0..d_i)$. By $\mathbf{v}_{i,a}^* := \mathbf{q}'_{i,a} + \delta_{i,a}^*\mathbf{u}_i^*$, it follows that
$$\mathbf{v}_{i,a}^* + \delta_{i,a}\mathbf{u}_i^* = \mathbf{q}'_{i,a} + \delta_{i,a}^*\mathbf{u}_i^* + \delta_{i,a}\mathbf{u}_i^* = \mathbf{q}'_{i,a},$$
where the last equality uses $\delta_{i,a} = \delta_{i,a}^*$ and the fact that the computation is over characteristic two. These are exactly the columns obtained by the same missing-seed execution of ConvertToVOLE in VOLEReconstruct.

Thus, in both cases, the extractor and VOLEReconstruct obtain the same small-VOLE matrices. Then they apply the same witness-row corrections, CRT lift, mask-row computations, and $\mathsf{H}_5/\mathbf{c}_{\mathsf{mult}}$ gate computations to the same public values. The Ext complete response is $(\mathbf{Q}, \overline{\mathbf{q}}_0, \dots, \overline{\mathbf{q}}_{n_{\mathsf{mask}}-1})$. So we have that

$$\left(\Delta, (\mathbf{Q}, \overline{\mathbf{q}}_0, \dots, \overline{\mathbf{q}}_{n_{\mathsf{mask}}-1})\right) \in \mathbf{S},$$

[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 $(\mathsf{sd}_{i,j}, \mathsf{com}_{i,j}) =$ LeafCommit$(\dots)$, in the notation of BAVC.Reconstruct.

and extraction succeeds. Combining the two bad-event bounds gives the claimed bound 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 10.13) to ensure that extraction of the challenge commitments succeeds. The FAEST-EM extractor omits the query to H_0 and, in Step 2b, instead computes

$$\mathsf{SeedPre}^{\mathsf{FAEST\text{-}EM}}_{\mathsf{iv},i}(c) := \left\{ r \in \{0,1\}^\lambda : \mathsf{PRG}(r, \mathsf{iv}, i+L-1; 2\lambda) = c \right\}.$$

It stores $\perp$ if this set is empty, stores its unique root if it is a unique element, and returns $\emptyset$ if it contains two distinct roots. It then uses the same three cases and the same response computation as above, with these roots as the induced seeds. Thus the $\mathsf{Fail}_{\mathsf{uhash}}$ analysis and all H_0 calls made by the extractor are omitted in the FAEST-EM branch.

Here the induced seed output by LeafCommit is the root r itself, so one leaf commitment has two distinct induced seeds precisely when it has two distinct PRG preimages. For each indexed attempt t, construct an almost-injectivity adversary $\mathcal{B}_t$ by simulating the binding game through attempt t and parsing the correctly typed top-level and lower-level H_1 preimages available at that point. The adversary $\mathcal{B}_t$ outputs the corresponding IV iv_t and retains every parsed tuple $(i+L-1, \mathsf{com}_{i,j})$, padding with allowed tweaks and arbitrary outputs if necessary to obtain exactly L tuples. It never performs the extractor's exhaustive PRG inversion. Every induced-seed ambiguity encountered by the extractor in attempt t is therefore among the outputs of $\mathcal{B}_t$ and makes it win the game of Definition 10.13. A union bound over the at most Q_C indexed attempts replaces the $\mathsf{Fail}_{\mathsf{uhash}}$ term by $\sum_{t=1}^{Q_C} \mathsf{AdvInj}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_t}[L]$, as claimed. Each $\mathcal{B}_t$ has running time roughly that of $\mathcal{A}$. □

10.5.2 Hiding

Next, we show that VOLECommit is hiding.

*Remark 10.23 (*BAVC *opening aborts).* Observe that BAVC.Open aborts for certain challenges $I = (\Delta_0, \ldots, \Delta_{\tau-1})$. 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 10.24 (VOLECommit is multi-hiding). *Let $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_4, \mathsf{H}_5$ be random oracles. Consider the following multi-hiding experiment for $\mathsf{VOLECommit}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_5}$ and adversary $\mathcal{A}$, that makes at most Q_4 queries to H_4 and at most Q_5 queries to H_5 before its final invocation of $\mathcal{O}_{b^*}$, defined for some arbitrary $\hat{\ell} = \ell + n_{\mathsf{mask}} \cdot d_0$ and $\mathtt{param}$, such that $(\tau+1)2^k < 2^{31}$ (cf. Table 5.1).*

1. *Sample $b^* \leftarrow \{0,1\}$ and set $j = 1$, $\mathsf{fail} = 0$.*
2. *$b \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_4,\mathsf{H}_5,\mathcal{O}_{b^*}}(1^\lambda)$, where $\mathcal{O}_{b^*}$ is defined below.*
3. *If $\mathsf{fail} = 1$, output 0 (failure). Else output b.*

We jointly define oracles $\mathcal{O}_0$ and $\mathcal{O}_1$ as follows: On its j-th invocation with $\mu^j \in \{0,1\}^{2\lambda}$, oracle $\mathcal{O}_{b^*}(\mu^j)$ does the following:

1. *$r^j \leftarrow \{0,1\}^\lambda$, $\mathsf{iv}^{\mathsf{pre},j} \leftarrow \{0,1\}^{128}$*
2. *$\mathsf{iv}^j \leftarrow \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre},j} \| \mu^j)$*
3. *$(\mathsf{com}^j, \mathsf{decom}^j, \mathbf{c}^j_1, \ldots, \mathbf{c}^j_{\tau-1}, \mathbf{c}^j_{\mathsf{mult}}, \mathbf{u}^j, \mathbf{V}^j, (\bar{\mathbf{u}}^j_m, \bar{\mathbf{v}}^j_m)_{m\in[0..n_{\mathsf{mask}})}) :=$ $\mathsf{VOLECommit}(r^j, \mathsf{iv}^j, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$, where, if $b^* = 1$, the H_5 calls used to compute $\mathbf{c}^j_{\mathsf{mult}}$ and $\bar{\mathbf{v}}^j_m$ are deferred until the programming step below.*
4. *Sample $\mathsf{chall}^j_3 \leftarrow \{0,1\}^{\lambda - w_{\mathsf{grind}}} \times 0^{w_{\mathsf{grind}}}$ uniformly, conditioned on BAVC.Open not aborting for $I^j = (\Delta^j_0, \ldots, \Delta^j_{\tau-1}) := \mathsf{DecodeAllChall}_3(\mathsf{chall}^j_3; \mathtt{param})$ (cf. Remark 10.23).*
5. *$\mathsf{decom}^j_{I^j} :=$ BAVC.Open(decom^j, I^j)*

6. *Define*

$$(\widehat{\mathbf{c}}_1^j, \dots, \widehat{\mathbf{c}}_{\tau-1}^j, \widehat{\mathbf{u}}^j, (\widehat{\mathbf{u}}_m^j)_{m\in[0..n_{\mathsf{mask}})}) = \begin{cases} (\mathbf{c}_1^j, \dots, \mathbf{c}_{\tau-1}^j, \mathbf{u}^j, (\bar{\mathbf{u}}_m^j)_{m\in[0..n_{\mathsf{mask}})}) & \text{if } b^*=0 \\ \textit{random in } (\mathbb{F}_2^\ell)^\tau \times \mathbb{F}_{2^\lambda}^{n_{\mathsf{mask}}} & \text{if } b^*=1 \end{cases}$$

Set $\widehat{\mathbf{c}_{\mathsf{mult}}}^j := \mathbf{c}_{\mathsf{mult}}^j$ *if* $b^*=0$. *If* $b^*=1$, *sample* $\widehat{\mathbf{c}_{\mathsf{mult}}}^j[\cdot,e] \leftarrow \mathbb{F}_2^{n_{\mathsf{mask}}}$ *for* $e \in [0..n_{\mathsf{mult}})$, *set* $\widehat{\mathbf{c}_{\mathsf{mult}}}^j[\cdot,e] := 0^{n_{\mathsf{mask}}}$ *for* $e \in [n_{\mathsf{mult}}..\bar{n}_{\mathsf{mult}})$, *and program* $\mathsf{H}_5(\mathsf{iv}^j, e, L_e^{\gamma_e^j \oplus 1}) := \widehat{\mathbf{c}_{\mathsf{mult}}}^j[\cdot,e] \oplus \mathsf{H}_5(\mathsf{iv}^j, e, L_e^{\gamma_e^j}) \oplus (f_e(\widehat{\mathbf{u}}_m^j))_{m\in[0..n_{\mathsf{mask}})}$, *where* γ_e^j *is as in* VOLEReconstruct *(line 24),*[21] *except if* $L_e^{\gamma_e^j} = L_e^{\gamma_e^j\oplus 1}$, $\mathcal{A}$ *previously queried* $(\mathsf{iv}^j, e, L_e^{\gamma_e^j\oplus 1})$ *or a previous loop (with* $j' < j$*) queried or programmed it. If programming fails, set* $\mathsf{fail} := 1$ *(so the experiment eventually outputs* 0 *(failure)).*

7. *Let* $\mathsf{out} := ((\widehat{\mathbf{c}}_i^j)_{i\in[1..\tau)}, \widehat{\mathbf{c}_{\mathsf{mult}}}^j, \widehat{\mathbf{u}}^j, (\widehat{\mathbf{u}}_m^j)_m, \mathsf{decom}_{I^j}^j, \mathsf{chall}_3^j, \mathsf{com}^j, \mathsf{iv}^{\mathsf{pre},j})$.
8. *Update counter* $j := j+1$.
9. *Return* out.

Write $\mathsf{PRG}_{\mathsf{len}(s)} = \mathsf{PRG}(s;\mathsf{len})$ *and* $\beta = \frac{\lambda}{128}$. *Set* $b_{\mathsf{hi}} := \lceil n_{\mathsf{mask}} w_{\mathsf{grind}}/128 \rceil$ *and* $\beta_{\mathsf{root}} := 2\beta + b_{\mathsf{hi}}$, *and define the joint generator* $\mathsf{PRG}_{\mathsf{root}}(r,\mathsf{iv}) := \mathsf{PRG}(r,\mathsf{iv},0;2\lambda) \parallel \mathsf{PRG}(r,\mathsf{iv},2^{31}-1;n_{\mathsf{mask}}w_{\mathsf{grind}})$. *The notation* $\mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}[1]$ *denotes the one-query advantage of this generator. Let* G_i^* *be the hiding game with* $b^* = i$ *and suppose* $\mathcal{A}$ *is a* Q*-query hiding adversary, i.e., it makes at most* Q *queries to* $\mathcal{O}_{b^*}$. *Then we have*

$$\begin{aligned}\Pr[\mathsf{G}_0^*=1] \le e^{Q\cdot(\beta_{\mathsf{root}}^2+4\tau\lceil\log(L)-1\rceil\beta^2+\tau a_{\mathsf{leaf}}^2)/2^{128}} \cdot \Big(&\Pr[\mathsf{G}_1^*=1] \\ &+ Q\cdot\mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}_{\mathcal{B}_{1.1}}[1] + Q\cdot\tau\lceil\log(L)-1\rceil\cdot\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1] \\ &+ Q\cdot\tau\cdot\mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}} \\ &+ 2^{-\lambda} + Q2^{-\lambda} + K_{\max}(QQ_5 + 2n_{\mathsf{mult}}Q^2)2^{-(128+\lambda)}\Big)\end{aligned}$$

where $K_{\max} := \lambda$ *is the preimage bound on* H_4 *(see Corollary 10.18), and the leaf terms* a_{leaf} *and* $\mathsf{AdvPRP}^{\mathsf{leaf}}$ *are defined per variant, with* $h_{\hat{\ell}} := \lceil \hat{\ell}/128 \rceil$*:*

- *for* FAEST*:* $a_{\mathsf{leaf}} = a_{\mathsf{CTR}} = 4\beta N_0 + h_{\hat{\ell}}$ *and* $\mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}} = \mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}_{\mathcal{B}_2}[N_0] + \mathsf{AdvPRP}^{\mathsf{PRG}_{\hat{\ell}}}_{\mathcal{B}_3}[1]$;
- *for* FAEST-EM*:* $a_{\mathsf{leaf}} = a_{\mathsf{EM}} = 2\beta + h_{\hat{\ell}}$ *and* $\mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}} := \mathsf{AdvPRP}^{\mathsf{PRG}^{\mathsf{pair}}_{\mathsf{EM}}}_{\mathcal{B}_{2,3}}[1]$, *where* $\mathsf{PRG}^{\mathsf{pair}}_{\mathsf{EM},i}(s,\mathsf{iv}) := \mathsf{PRG}(s,\mathsf{iv},i+L-1;2\lambda) \parallel \mathsf{PRG}(s,\mathsf{iv},i+2^{31};\hat{\ell})$ *and the advantage is the maximum over* $i \in [0..\tau)$.

Note that we can sample the challenges $(\mathsf{chall}_3^j)_{j\in[1..Q]}$ before committing without changing the oracle's output, 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, while μ^j is adaptively chosen, we only assert selective security for the actual VOLE commitment.

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

[21] Given chall_3 and $(\widehat{\mathbf{c}}_1,\dots,\widehat{\mathbf{c}}_{\tau-1},\widehat{\mathbf{u}},(\widehat{\mathbf{u}}_m)_{m\in[0..n_{\mathsf{mask}})})$ the parameters L_e^0, L_e^1 are defined via VOLEReconstruct (as are $\mathbf{Q}, \bar{\mathbf{q}}_{m\in[0..n_{\mathsf{mask}})}$). Thus, for $b^*=1$, it makes sense to (re-)program H_5 as defined, which is consistent with VOLEReconstruct by construction. Note also, that instead of reprogramming H_5, the distinction between computing $\mathbf{c}_{\mathsf{mult}}$ in $b^*=0$ and $b^*=1$ could be implemented already during VOLECommit. However, to avoid repeating the code of VOLECommit here, we opt for the current presentation.

openings. Moreover, as noted in Footnote 21, we can now compute the opening information $\mathbf{Q}^j, \bar{\mathbf{q}}_{m\in[0..n_{\mathsf{mask}})}, \mathbf{c}^j_{\mathsf{mult}}$ given only iv^j, chall_3^j and

$$(\widehat{\mathbf{c}}^j_1, \ldots, \widehat{\mathbf{c}}^j_{\tau-1}, \widehat{\mathbf{c}_{\mathsf{mult}}}^j, \widehat{\mathbf{u}}^j, (\widehat{\bar{\mathbf{u}}}^j_m)_{m\in[0..n_{\mathsf{mask}})})$$

Hence, we concentrate on the distribution of the above values.

We argue in two main steps: First, we replace the hidden outputs $(\mathbf{u}^j, \bar{\mathbf{u}}^j_m)$ of the BAVC commitments with true randomness. Then, we replace the output of ConvertToVOLE and $\mathbf{c}^j_{\mathsf{mult}}$. By analyzing the resulting distribution, we find that each $\mathbf{u}^j_i, \bar{\mathbf{u}}^j_m$ and $\mathbf{c}^j_i, \mathbf{c}^j_{\mathsf{mult}}$ will be uniformly random, as required.

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 $\mathcal{O}_0$ is used and all oracle outputs

$$(\widehat{\mathbf{c}}^j_1, \ldots, \widehat{\mathbf{c}}^j_{\tau-1}, \mathbf{c}^j_{\mathsf{mult}}, \widehat{\mathbf{u}}^j, (\widehat{\bar{\mathbf{u}}}^j_m)_{m\in[0..n_{\mathsf{mask}})})$$

correspond to the generated commitment.

We describe the changes to one oracle response using games $\mathsf{G}_0, \ldots, \mathsf{G}_5$. In the complete experiment, we first apply the changes from G_0 to G_3, one response at a time, to all Q responses, while H_5 is still evaluated honestly. After these computational changes have been completed, we take a statistical hop to G_4, which aborts if H_4 fails to satisfy the bounded preimage property, and then apply the final change from G_4 to G_5, again one response at a time. Each game still answers every oracle invocation immediately, so this order preserves the adaptive interface. For readability, we omit the superscript j in the reductions and variables.

Game G_1 (Randomize hidden GGM tree seeds and $(\mathbf{u}^{(0)}_{\mathsf{hi}}, \ldots, \mathbf{u}^{(n_{\mathsf{mask}}-1)}_{\mathsf{hi}})$). In game G_1, we replace all hidden seeds k_{α_i} in BAVC.Commit (line 5) and and $(\mathbf{u}^{(0)}_{\mathsf{hi}}, \ldots, \mathbf{u}^{(n_{\mathsf{mask}}-1)}_{\mathsf{hi}})$ in FAEST.VOLECommit (line 10) in the commitment computation with truly random seeds, and consistently use these during decommitment.

The top-level replacement jointly replaces $\mathsf{PRG}(r, \mathsf{iv}, 0; 2\lambda)$ in the first GGM layer and $\mathsf{PRG}(r, \mathsf{iv}, 2^{31} - 1; n_{\mathsf{mask}} \cdot w_{\mathsf{grind}})$ in FAEST.VOLECommit (line 10) using $\mathsf{PRG}_{\mathsf{root}}$. The two tweak ranges are disjoint by $(\tau + 1)2^k < 2^{31}$. For the GGM tree in layers in BAVC.Commit (line 5), we use an additional hybrid argument. Namely, 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 $\tau\lceil\log(L)-1\rceil$ hybrid steps (omitting the top layer), the leaf seeds k_{α_i} are replaced with random values. By applying Corollary 10.10 and Remark 10.11 we arrive at

$$\begin{aligned}
\Pr[\mathsf{G}_0 = 1] &\le e^{\beta^2_{\mathsf{root}}/2^{128}} \cdot (e^{(2\beta)^2/2^{128}})^{\tau\lceil\log(L)-1\rceil} \\
&\quad \cdot (\Pr[\mathsf{G}_1 = 1] + \mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}_{\mathcal{B}_{1.1}}[1] + \sum_{i=2}^{1+\tau\lceil\log(L)-1\rceil} \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_{1.i}}[1]) \\
&= e^{(\beta^2_{\mathsf{root}}+4\tau\lceil\log(L)-1\rceil\beta^2)/2^{128}} \\
&\quad \cdot (\Pr[\mathsf{G}_1 = 1] + \mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}_{\mathcal{B}_{1.1}}[1] + \tau\lceil\log(L) - 1\rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1])
\end{aligned}$$

since each call to $\mathsf{PRG}(k_\alpha, \mathsf{iv}, \alpha; 2\lambda)$ in line 5 of BAVC.Commit outputs $2\beta = \frac{2\lambda}{128}$ blocks for $T = 1$ queries and we have at most $\tau\lceil\log(L) - 1\rceil$ hybrid steps, and the first special step outputs a total of $\beta_{\mathsf{root}} = 2\beta + \lceil n_{\mathsf{mask}} w_{\mathsf{grind}}/128\rceil$ PRG blocks due to the joint root and mask-high-bits generation.

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 $\alpha_0, \ldots, \alpha_{\tau-1}$ in line 8 of

$\mathsf{BAVC.Commit}$ by $(\mathsf{sd}_{i,\Delta_i}, \mathsf{LeafHash}(\mathsf{uhash}, \mathsf{sd}_{i,\Delta_i} \| s_0 \| s_1 \| s_2))$ for random $(\mathsf{sd}_{i,\Delta_i}, s_0, s_1, s_2) \leftarrow \{0,1\}^{4\lambda}$. Note that $(\mathsf{sd}, \mathbf{x}_1) \mapsto (\mathsf{sd}, \mathsf{LeafHash}(\mathsf{uhash}, (\mathsf{sd}, \mathbf{x}_1)))$ is a permutation on $\{0,1\}^{4\lambda}$ for fixed uhash. Hence, the output $(\mathsf{sd}_{i,\Delta_i}, \mathsf{com}_{i,\Delta_i})$ will be uniform.

For $\mathsf{FAEST\text{-}EM}$, the changes of games G_2 and G_3 are performed in one joint hybrid, since $\mathsf{com}_{i,\Delta_i} = \mathsf{PRG}(k_{\alpha_i}, \mathsf{iv}, i + L - 1; 2\lambda)$ and $\mathbf{r}_{0,\Delta_i} = \mathsf{PRG}(k_{\alpha_i}, \mathsf{iv}, i + 2^{31}; \hat{\ell})$ use the same seed k_{α_i}. The derived generator $\mathsf{PRG}^{\mathsf{pair}}_{\mathsf{EM},i}$ defined in the lemma statement replaces both values by independent uniform strings in this joint hybrid.

For FAEST, since in game G_1, all seeds k_{α_i} are truly random, we can apply pseudorandomness of $\mathsf{LeafCommit}$ to replace the outputs $(\mathsf{sd}_{i,\Delta_i}, \mathsf{com}_{i,\Delta_i})$ corresponding to leaves $\alpha_0, \dots, \alpha_{\tau-1}$ in line 8 of $\mathsf{BAVC.Commit}$ by randomness. Again, we apply Corollary 10.10 and Remark 10.11 to arrive at

$$\begin{aligned}\Pr[\mathsf{G}_1 = 1] &\le (e^{(4\beta)^2 N_0^2/2^{128}})^\tau (\Pr[\mathsf{G}_2 = 1] + \sum_{i=0}^{\tau-1} \mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}_{\mathcal{B}_{2.i}}[N_i]) \\ &\le e^{16\tau\beta^2 N_0^2/2^{128}} (\Pr[\mathsf{G}_2 = 1] + \tau \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}_{\mathcal{B}_2}[N_0])\end{aligned}$$

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 $N_i \le N_0$ by definition of N_i (see Table 5.1).

For $\mathsf{FAEST\text{-}EM}$, the joint change gives

$$\Pr[\mathsf{G}_1 = 1] \le e^{\tau a^2_{\mathsf{EM}}/2^{128}} \left(\Pr[\mathsf{G}_3 = 1] + \tau \mathsf{AdvPRP}^{\mathsf{PRG}^{\mathsf{pair}}_{\mathsf{EM}}}_{\mathcal{B}_{2,3}}[1] \right),$$

where the advantage is the maximum over $i \in [0..\tau)$.

Game G_3 (Switch $\mathsf{ConvertToVOLE}$ to random). In game G_3, for FAEST, we replace the output of $\mathbf{r}_{0,\Delta_i} = \mathsf{PRG}(\mathsf{sd}_{i,\Delta_i}, \mathsf{iv}, i + 2^{31}; \hat{\ell})$ in line 2 of that $\mathsf{ConvertToVOLE}$ computation called from $\mathsf{VOLECommit}$ line 3 by true randomness.

In the output of $\mathsf{ConvertToVOLE}$, $\mathbf{u}_i$ equals $\mathbf{u}_i = \sum_{j \in [0,N_i)} \mathsf{PRG}(\mathsf{sd}_{i,j}, \mathsf{iv}, i + 2^{31}; \hat{\ell})$ and thus if a single summand $\mathbf{r}_{0,\Delta_i} = \mathsf{PRG}(\mathsf{sd}_{i,\Delta_i}, \dots)$ is truly random, so is $\mathbf{u}_i$. Hence, all $\mathbf{u}_i$ in $\mathsf{VOLECommit}$ line 3 are truly random. Consequently, the outputs $\mathbf{u}$ and $\mathbf{c}_i = \mathbf{u}_i \oplus \mathbf{u}$ (from lines 6 and 8) are truly random in G_3. Notice also that all $\bar{\mathbf{u}}_m$ are uniformly random: the disjoint coordinates forming $\mathbf{u}^{(m)}_{\mathsf{low}}$ are uniform after the hidden $\mathsf{ConvertToVOLE}$ summands have been replaced, the values $\mathbf{u}^{(m)}_{\mathsf{hi}}$ are independent and uniform by game G_1, and the CRT decomposition used in their definition is bijective.

By the change in game G_2, we know that the seed sd_{i,Δ_i} which is input into $\mathsf{ConvertToVOLE}$ in line 3 of $\mathsf{VOLECommit}$ is truly random, so we can reduce to PRG security as before. We get

$$\begin{aligned}\Pr[\mathsf{G}_2 = 1] &\le (e^{(\lceil \hat{\ell}/128 \rceil)^2/2^{128}})^\tau (\Pr[\mathsf{G}_3 = 1] + \sum_{i=0}^{\tau-1} \mathsf{AdvPRP}^{\mathsf{PRG}_{\hat{\ell}}}_{\mathcal{B}_{3.i}}[1]) \\ &\le e^{\tau(\lceil \hat{\ell}/128 \rceil)^2/2^{128}} (\Pr[\mathsf{G}_3 = 1] + \tau \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{\hat{\ell}}}_{\mathcal{B}_3}[1])\end{aligned}$$

since we have $\lceil \hat{\ell}/128 \rceil$ blocks of output in $\mathsf{PRG}(\mathsf{sd}_i, \mathsf{iv}, i; \hat{\ell})$ in for the i-th run of $\mathsf{ConvertToVOLE}$ (for $i \in [0, \tau - 1]$), where each only has $T = 1$ queries.

Game G_4 (Abort if H_4 not preimage bounded). If H_4 is not $K_{\max}$-preimage bounded (Definition 10.15), the experiment outputs 0. By Corollary 10.18,

$$\Pr[\mathsf{G}_3 = 1] \le \Pr[\mathsf{G}_4 = 1] + 2^{-\lambda}.$$

Since this and the following hops are statistical, it's not a problem that this event is inefficiently decidable as no reduction needs to test it.

Game G_5 (Switch $\mathbf{c}_{\mathsf{mult}}$ to random). In game G_5, we sample $\mathbf{c}_{\mathsf{mult}} \leftarrow \mathbb{F}_2^{n_{\mathsf{mask}} \times n_{\mathsf{mult}}}$ uniformly random in $\mathsf{VOLECommit}$ (line 26), set $\mathbf{c}_{\mathsf{mult}}[\cdot, e] := 0^{n_{\mathsf{mask}}}$ for $e \in [n_{\mathsf{mult}}..\bar{n}_{\mathsf{mult}})$, and

program H_5 consistently. If H_5 is already defined on a query the game wants to program, the experiment outputs 0 (i.e., the adversary loses). Note that, since chall_3 is known from the start, we can derive L'_e (VOLEReconstruct, line 24) and program H_5 as claimed.[22]

Next, we bound the change in advantage by these modifications. In VOLECommit, we have

$$\begin{aligned}(\mathbf{a}_0, \cdots, \mathbf{a}_{\lambda - w_{\mathsf{grind}} - 1}) &= [\mathbf{V}_0 \cdots \mathbf{V}_{\tau-1}]|_{[0..\lambda)} \in (\mathbb{F}_2^\lambda)^{\lambda - w_{\mathsf{grind}}} \\ \mathbf{c}_{\mathsf{mult}}[m, e] &= \mathsf{H}_5(\mathsf{iv}, e, L_e^0)[m] \oplus \mathsf{H}_5(\mathsf{iv}, e, L_e^1)[m] \oplus f_e(\bar{\mathbf{u}}_m) \in \mathbb{F}_2 \\ L_e^0 &\coloneqq \bigoplus_{j \,:\, \mathbf{G}[e][j]=1} \mathbf{a}_j \in \mathbb{F}_2^\lambda \\ L_e^1 &\coloneqq L_e^0 \oplus \mathbf{u}[0..\lambda)\end{aligned}$$

and in VOLEReconstruct, we have

$$\begin{aligned}(\mathbf{a}'_0, \cdots, \mathbf{a}'_{\lambda - w_{\mathsf{grind}} - 1}) &= [\mathbf{Q}_0 \cdots \mathbf{Q}_{\tau-1}]|_{[0..\lambda)} \in (\mathbb{F}_2^\lambda)^{\lambda - w_{\mathsf{grind}}} \\ \gamma_e &= \mathbf{G}[e] \cdot \Delta' \in \mathbb{F}_2 \\ L'_e &= \bigoplus_{j \,:\, \mathbf{G}[e][j]=1} \mathbf{a}'_j \in \mathbb{F}_2^\lambda \\ \tilde{q}_{m,e} &= \mathbf{h}_e[m] \oplus \gamma_e \cdot \mathbf{c}_{\mathsf{mult}}[m, e] \in \mathbb{F}_2\end{aligned}$$

where $L'_e = L_e^{\gamma_e}$ holds by construction. Note that, by definition, $L_e^{\gamma_e \oplus 1} = L'_e \oplus \mathbf{u}[0..\lambda)$. In game G_4, the vector $(\mathbf{u}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1})$ is jointly uniform, while the reconstruction from the opened seeds is independent of the hidden random summands. Since L'_e depends only on that reconstruction, the challenge, and the correction vectors, it is independent of $\mathbf{u}$ even conditioned on the preceding transcript, so $L_e^{\gamma_e \oplus 1}$ is uniformly distributed conditioned on the preceding transcript. Hence, we can program

$$\mathsf{H}_5(\mathsf{iv}, e, L_e^{\gamma_e \oplus 1}) \coloneqq \mathbf{c}_{\mathsf{mult}}[\cdot, e] \oplus \mathsf{H}_5(\mathsf{iv}, e, L_e^{\gamma_e}) \oplus (f_e(\bar{\mathbf{u}}_m))_{m \in [0..n_{\mathsf{mask}})} \in \mathbb{F}_2^{n_{\mathsf{mask}}} \tag{9}$$

dependent on $\mathbf{c}_{\mathsf{mult}}[m, e]$, unless the event that $(\mathsf{iv}, e, L_e^{\gamma_e \oplus 1})$ was already programmed occurs. We bound this probability below. Observe that, conditioned on H_4 being $K_{\max}$-preimage bounded and on the complete transcript and all game randomness fixed before the fresh randomness for the j-th oracle response is sampled, for every $v \in \{0,1\}^{128}$,

$$\Pr[\mathsf{iv}^j = v] \leq K_{\max} 2^{-128}.$$

Moreover, even after fixing $\mathsf{iv}^j = v$, the value $L_e^{\gamma_e^j \oplus 1}$ is uniformly distributed in $\mathbb{F}_2^\lambda$, so, for every $e \in [0..n_{\mathsf{mult}})$, $v \in \{0,1\}^{128}$, and $z \in \mathbb{F}_2^\lambda$,

$$\Pr\left[(\mathsf{iv}^j, e, L_e^{\gamma_e^j \oplus 1}) = (v, e, z)\right] \leq K_{\max} 2^{-(128+\lambda)},$$

where both probabilities are conditioned on the transcript and previously fixed randomness described above. A programming failure for $(\mathsf{iv}, e, L_e^{\gamma_e \oplus 1})$ happens only if:

- $L_e^0 = L_e^1$, i.e., if $\mathbf{u}[0..\lambda) = 0^\lambda$; or
- $\mathcal{A}$ queried some $(\mathsf{iv}, e, L_e^{\gamma_e \oplus 1})$ previously; or
- A previous internal query or programming in oracle $\mathcal{O}_{b^*}$ defined it.

[22] As mentioned in Footnote 21, we can (and do) modify the computation of $\mathbf{c}_{\mathsf{mult}}$ in VOLECommit directly and program H_5 directly, instead of computing VOLECommit honestly and then resampling $\mathbf{c}_{\mathsf{mult}}$ and reprogramming H_5 (as described in the oracle).

Notice that $e \in [0..n_{\mathsf{mult}})$ effectively act as domain separators for H_5 queries. In particular, the Q_5 queries of $\mathcal{A}$ are separated by e. Let $Q_{5,e}^{<j}$ be the number of prior adversarial queries with gate index e, so $\sum_e Q_{5,e}^{<j} \le Q_5$. For the j-th oracle query and fixed e, there are at most $Q_{5,e}^{<j} + 2(j-1)$ previously queried or programmed points in H_5. Using the min-entropy, a union bound, and accounting for domain-separating by e in $\mathcal{A}$'s queries, the programming failure is at most

$$2^{-\lambda} + K_{\max}(Q_5 + 2n_{\mathsf{mult}}(j-1)) \cdot 2^{-(128+\lambda)}.$$

In that case, G_5 also sets $\mathsf{fail} = 1$ in the oracle call. Since games G_4 and G_5 are identically distributed unless a programming failure happens, we find

$$\Pr[\mathsf{G}_4 = 1] \le \Pr[\mathsf{G}_5 = 1] + 2^{-\lambda} + K_{\max}(Q_5 + 2n_{\mathsf{mult}}(j-1))2^{-(128+\lambda)} \tag{10}$$

Completing the proof. Finally, note that since the outputs $\mathbf{c}_i$, $\mathbf{u}$, $\bar{\mathbf{u}}_m$ and $\mathbf{c}_{\mathsf{mult}}$ are now uniformly random, the only difference between the ideal G_1^* and the last game is the preimage-boundedness abort condition. It therefore holds that $\Pr[\mathsf{G}_5 = 1] \le \Pr[\mathsf{G}_1^* = 1]$.

Putting everything together, we get

$$\begin{aligned}\Pr[\mathsf{G}_0^* = 1] \le (e^{(\beta_{\mathsf{root}}^2 + 4\tau\lceil\log(L)-1\rceil\beta^2 + \tau a_{\mathsf{leaf}}^2)/2^{128}})^Q \cdot \Big(&\Pr[\mathsf{G}_1^* = 1] \\ &+ Q \cdot \mathsf{AdvPRP}_{\mathcal{B}_{1.1}}^{\mathsf{PRG}_{\mathsf{root}}}[1] + Q \cdot \tau\lceil\log(L)-1\rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}_1}^{\mathsf{PRG}_{2\lambda}}[1] \\ &+ Q \cdot \tau \cdot \mathsf{AdvPRP}_{\mathcal{B}_{2,3}}^{\mathsf{leaf}} \\ &+ 2^{-\lambda} + Q \cdot \Big(2^{-\lambda} + K_{\max}(Q_5 + 2n_{\mathsf{mult}}Q)2^{-(128+\lambda)}\Big)\Big)\end{aligned}$$

where the first $2^{-\lambda}$ term is the cost of the G_4 abort, while the remaining programming terms follow from summing over the Q applications of G_5 over all queries.

For FAEST, we collected the leaf terms of games G_2 and G_3 using $16\beta^2 N_0^2 + h_{\hat{\ell}}^2 \le (4\beta N_0 + h_{\hat{\ell}})^2 = a_{\mathsf{leaf}}^2$ and $\mathsf{AdvPRP}_{\mathcal{B}_2}^{\mathsf{PRG}_{4\lambda}}[N_0] + \mathsf{AdvPRP}_{\mathcal{B}_3}^{\mathsf{PRG}_{\hat{\ell}}}[1] = \mathsf{AdvPRP}_{\mathcal{B}_{2,3}}^{\mathsf{leaf}}$. For FAEST-EM, the joint hybrid replacing games G_2 and G_3 contributes exactly $\tau a_{\mathsf{EM}}^2 = \tau a_{\mathsf{leaf}}^2$ to the exponent and $\tau\mathsf{AdvPRP}_{\mathcal{B}_{2,3}}^{\mathsf{PRG}_{\mathsf{EM}}^{\mathsf{pair}}}[1] = \tau\mathsf{AdvPRP}_{\mathcal{B}_{2,3}}^{\mathsf{leaf}}$ per oracle query, so the displayed bound holds verbatim. This concludes the proof.

□

10.5.3 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 10.25 (VOLECommit is unpredictable). *Fix* $\mathtt{param}$, $\mathtt{param}_{\mathsf{VOLE}}$ *and let* $\hat{\ell} := \ell + n_{\mathsf{mask}}d_0$. *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. $(\mathcal{L}_{\mathsf{com}}, \mu, \mathsf{state}) \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_4,\mathsf{H}_5}(1^\lambda)$, *where* $\mu \in \{0,1\}^{2\lambda}$ *is an IV label.*
2. $r \leftarrow \{0,1\}^\lambda$, $\mathsf{iv}^{\mathsf{pre}} \leftarrow \{0,1\}^{128}$
3. $\mathsf{iv} \leftarrow \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu)$
4. $C = (\mathsf{com}, \mathsf{decom}, \mathbf{c}_1, \dots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathbf{u}, \mathbf{V}, (\bar{\mathbf{u}}_m, \bar{\mathbf{v}}_m)_{m\in[0..n_{\mathsf{mask}})}) := \mathsf{VOLECommit}(r, \mathsf{iv}, \hat{\ell}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$
5. *Let* $C_{\mathsf{pub}} := (\mathsf{com}, \mathbf{c}_1, \dots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}^{\mathsf{pre}})$.
6. *Let* $b' \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_4,\mathsf{H}_5}(\mathsf{state}, C, \mathsf{iv}^{\mathsf{pre}})$.

7. *If* $C_{\mathsf{pub}} \in \mathcal{L}_{\mathsf{com}}$ *and* $b = 1$ *output* 0. *Else output* b'.

For brevity, let $\mathsf{G}_b^{\mathsf{unpred}}$ *denote the output of* $\mathsf{ExpUnpred}_{b,\mathcal{A}}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}}$ *for* $b \in \{0,1\}$. *Also let* $\beta = \frac{\lambda}{128}$. *For any adversary* $\mathcal{A}$ *which makes at most* Q_1 *queries to* H_1, *and outputs lists* $\mathcal{L}_{\mathsf{com}}$ *of size at most* Q_{com}, *whose entries have the same shape as* C_{pub}, *we have*

$$\begin{aligned}\Pr[\mathsf{G}_0^{\mathsf{unpred}} = 1] \le \Pr[\mathsf{G}_1^{\mathsf{unpred}} = 1] + e^{(\beta_{\mathsf{root}}^2 + 4\lceil \log(L)-1\rceil \beta^2 + a_{\mathsf{leaf}}^2)/2^{128}} \cdot \Big(\\ \mathsf{AdvPRP}_{\mathcal{B}_{1.1}}^{\mathsf{PRG}_{\mathsf{root}}}[1] + \lceil \log(L) - 1\rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}_1}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}_{\mathcal{B}_{2,3}}^{\mathsf{leaf}} \\ + (2Q_1 + Q_{\mathsf{com}}) \cdot 2^{-2\lambda}\Big)\end{aligned}$$

where $\mathsf{PRG}_{\mathsf{root}}$, β_{root}, $\mathsf{AdvPRP}^{\mathsf{leaf}}$ *and* a_{leaf} *are the variant-dependent quantities defined in Lemma 10.24.*

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 single-query ($Q = 1$) version of the first two hops of Lemma 10.24, applied to a truncated game which only computes the BAVC commitment.

Step 1 (Reduce to guessing com*).* The games $\mathsf{G}_0^{\mathsf{unpred}}$ and $\mathsf{G}_1^{\mathsf{unpred}}$ are identically distributed unless $C_{\mathsf{pub}} \in \mathcal{L}_{\mathsf{com}}$, and this event is determined by Lines 1–5, which are identical in both games. Moreover, it implies the event $\mathsf{Fail}_{\mathsf{com}}$ that some entry of $\mathcal{L}_{\mathsf{com}}$ has com as its first component. Hence

$$\Pr[\mathsf{G}_0^{\mathsf{unpred}} = 1] \le \Pr[\mathsf{G}_1^{\mathsf{unpred}} = 1] + \Pr[\mathsf{Fail}_{\mathsf{com}}],$$

and it remains to show that com is hard to guess ahead of time.

Step 2 (A truncated guessing game). Let G_0' denote the game that runs the first stage of $\mathcal{A}$, samples r and $\mathsf{iv}^{\mathsf{pre}}$ as in Line 2, sets $\mathsf{iv} := \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu)$, computes com by running $\mathsf{BAVC.Commit}(r, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$, and outputs 1 if and only if $\mathsf{Fail}_{\mathsf{com}}$ occurs; clearly $\Pr[\mathsf{Fail}_{\mathsf{com}}] = \Pr[\mathsf{G}_0' = 1]$. Two properties of G_0' are used below. First, G_0' never runs the second stage of $\mathcal{A}$, so the decommitment decom, which contains all GGM seeds including the root r, is never revealed. Second, the output of G_0' depends on com only, so ConvertToVOLE, the values $(\mathbf{u}_{\mathsf{hi}}^{(0)}, \ldots, \mathbf{u}_{\mathsf{hi}}^{(n_{\mathsf{mask}}-1)})$, the correction values $\mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}$ and $\mathbf{c}_{\mathsf{mult}}$ (and hence all evaluations of H_5) are not needed to compute it.

Step 3 (Randomize a single leaf commitment). Let $\alpha := \mathsf{BAVC.PosInTree}(0,0)$ and let G_1' be obtained from G_0' by successively replacing the $\lceil \log(L) \rceil$ calls $\mathsf{PRG}(k_{\alpha'}, \mathsf{iv}, \alpha'; 2\lambda)$ in line 5 of BAVC.Commit along the path from the root to α by truly random values, exactly as in game G_1 of Lemma 10.24. Unlike there, the root call $\mathsf{PRG}(r, \mathsf{iv}, 0; 2\lambda)$ is replaced on its own rather than jointly with $\mathsf{PRG}(r, \mathsf{iv}, 2^{31}-1; n_{\mathsf{mask}} w_{\mathsf{grind}})$, since the latter is not computed in G_0' by Step 2. In G_1', the seed k_α is truly random, so we may let G_2' be obtained from G_1' by replacing the output of the LeafCommit call at α by a truly random value, as in game G_2 of Lemma 10.24. Again unlike there, only the leaf commitment $\mathsf{com}_{0,0}$ is needed, while the leaf seed $\mathsf{sd}_{0,0}$ and the ConvertToVOLE expansion $\mathbf{r}_{0,0}$ are not, so for FAEST-EM no joint generator is required either and the replaced call is $\mathsf{PRG}(k_\alpha, \mathsf{iv}, L-1; 2\lambda)$.

Each of these reductions simulates G_0' from its challenge value, which is possible because decom is never revealed and com is the only value the output depends on. Every replaced PRG call uses $T = 1$ query and outputs 2β blocks, except the leaf call, which outputs 4β blocks for FAEST and 2β blocks for FAEST-EM. As in the proof of Lemma 10.24, every replaced seed is fresh and independent of the preceding transcript, and each replacement is made with respect to the concrete IV $\mathsf{iv} = \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu)$. For FAEST, the argument of

game G_2 of Lemma 10.24 shows that $\mathsf{com}_{0,0}$ is uniform for *every* value of $\mathsf{uhash} = \mathsf{H}_0(\mathsf{iv})$, since $(\mathsf{sd}, \mathbf{x}_1) \mapsto (\mathsf{sd}, \mathsf{LeafHash}(\mathsf{uhash}, (\mathsf{sd}, \mathbf{x}_1)))$ is a permutation; together with Step 2 this is why no bound on the number of queries to H_0, H_4 or H_5 enters the lemma. We express the resulting bound in the quantities of Lemma 10.24. A distinguisher for one component of a joint generator extends to the joint generator by truncating its output, so the root step ($\mathsf{PRG}(\cdot,\cdot,0;2\lambda)$) has advantage at most $\mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}_{\mathcal{B}_{1.1}}[1]$, and likewise the FAEST-EM leaf step ($\mathsf{PRG}(\cdot,\cdot,L-1;2\lambda)$, a component of $\mathsf{PRG}^{\mathsf{pair}}_{\mathsf{EM},0}$) at most $\mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}}$; the FAEST leaf step has advantage $\mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}_{\mathcal{B}_2}[1] \le \mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}_{\mathcal{B}_2}[N_0] \le \mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}}$ by query monotonicity; the remaining $\lceil\log(L)\rceil - 1 = \lceil\log(L) - 1\rceil$ inner steps have advantage $\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1]$ each. For the exponent, the block counts above are bounded by $2\beta \le \beta_{\mathsf{root}}$ for the root step and, per variant, $4\beta \le a_{\mathsf{leaf}}$ (FAEST) resp. $2\beta \le a_{\mathsf{leaf}}$ (FAEST-EM) for the leaf step. By Corollary 10.10 and Remark 10.11 we arrive at

$$\Pr[\mathsf{G}'_0 = 1] \le e^{(\beta^2_{\mathsf{root}} + 4\lceil\log(L)-1\rceil\beta^2 + a^2_{\mathsf{leaf}})/2^{128}} \cdot \Big(\Pr[\mathsf{G}'_2 = 1] + \mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}_{\mathcal{B}_{1.1}}[1] + \lceil\log(L) - 1\rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1] + \mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}}\Big) \quad (11)$$

Step 4 (com is unpredictable in G'_2). In G'_2, the leaf commitment $\mathsf{com}_{0,0} \in \{0,1\}^{n_{\mathsf{leafcom}}\lambda}$ is truly random. Let $X_0 := \mathsf{com}_{0,0} \| \cdots \| \mathsf{com}_{0,N_0-1}$, then h_0 is uniformly random, unless H_1 was queried before on X_0, which happens with probability at most $Q_1/2^{n_{\mathsf{leafcom}}\lambda} \le Q_1/2^{2\lambda}$, since $n_{\mathsf{leafcom}} \ge 2$ (only the first stage of $\mathcal{A}$ runs in G'_2, so at most Q_1 queries to H_1 precede this one). In addition, let $Y := h_0 \| \cdots \| h_{\tau-1}$, then com in $\mathsf{BAVC.Commit}$ is uniformly random, unless H_1 was queried before on Y, which happens with probability at most $Q_1/2^{2\lambda}$. Finally, note that since com is uniformly random in $\{0,1\}^{2\lambda}$ and $\mathcal{L}_{\mathsf{com}}$ was output before r and $\mathsf{iv}^{\mathsf{pre}}$ were sampled, also the probability of $\mathsf{Fail}_{\mathsf{com}}$ is at most $Q_{\mathsf{com}}/2^{2\lambda}$. Hence

$$\Pr[\mathsf{G}'_2 = 1] \le Q_1 \cdot 2^{-n_{\mathsf{leafcom}}\lambda} + (Q_1 + Q_{\mathsf{com}}) \cdot 2^{-2\lambda} \le (2Q_1 + Q_{\mathsf{com}}) \cdot 2^{-2\lambda}$$

Plugging this into Equation (11), and the result into the inequality of Step 1, yields the claim. □

10.6 Properties of KOS Consistency Check

Here, we analyze the VOLEHash check, which is essentially a 4λ-bit output variant of the KOS OT extension [KOS15] consistency check.

Lemma 10.26. *Let $\mathcal{H} \subseteq \mathbb{F}^{k\times\ell}$ be an ϵ-almost universal hash family, and let $S \subseteq \mathbb{F}\times\mathbb{F}^\ell$ be a set (describing possible $(\Delta, \mathbf{q})$ openings). Then there exists a map $\mathcal{E}_{KOS,S} \colon \mathbb{F}^{k\times\ell}\times\mathbb{F}^k\times\mathbb{F}^k \to \mathbb{F}^\ell \times \mathbb{F}^\ell$ such that*

$$\Pr_{\mathbf{H}\leftarrow\mathcal{H}}\left[\begin{array}{r}\exists \widetilde{\mathbf{u}}, \widetilde{\mathbf{v}} \in \mathbb{F}^k, (\Delta, \mathbf{q}) \in S. \mathbf{Hq} = \widetilde{\mathbf{u}} \cdot \Delta + \widetilde{\mathbf{v}} \wedge \\ \mathbf{q} \neq \mathbf{u} \cdot \Delta + \mathbf{v}\end{array}\right] \le \frac{\epsilon\,|S|^3}{6},$$

where $(\mathbf{u}, \mathbf{v}) := \mathcal{E}_{KOS,S}(\mathbf{H}, \widetilde{\mathbf{u}}, \widetilde{\mathbf{v}})$.

As the security game is a bit unintuitive, let us quickly clarify it. In the game, the attacker first chooses the set S, whereupon $\mathbf{H}$ is chosen uniformly at random. An adversary can then only win if it produces $\widetilde{\mathbf{u}}, \widetilde{\mathbf{v}}$ which fool the extractor, i.e. lead it to reconstruct $\mathbf{u}, \mathbf{v}$ from S which are inconsistent with at least one line in S. Note that the existential quantifier implies that this holds against any, potentially computationally unbounded, attacker.

Proof. We present two bad events, then describe an extractor that always works whenever these bad events do not occur.

- Bad event 1: there exists distinct $(\Delta, \mathbf{q}_0), (\Delta, \mathbf{q}_1) \in S$ such that $\mathbf{H}\mathbf{q}_0 = \mathbf{H}\mathbf{q}_1$.
- Bad event 2: there exists 3 non-colinear points $(\Delta_0, \mathbf{q}_0), (\Delta_1, \mathbf{q}_1), (\Delta_2, \mathbf{q}_2) \in S$ with distinct Δ_i such that $(\Delta_0, \mathbf{H}\mathbf{q}_0)$, $(\Delta_1, \mathbf{H}\mathbf{q}_1)$, and $(\Delta_2, \mathbf{H}\mathbf{q}_2)$ are colinear in $\mathbb{F}^{1+k}$.

We bound bad event 1 by noticing that there are at most $\binom{|S|}{2}$ pairs, and for each pair the probability that $\mathbf{H}(\mathbf{q}_1 - \mathbf{q}_0) = 0$ is at most ϵ since $\mathbf{H}$ is sampled from a universal hash family. By the union bound, bad event 1 has probability at most $\epsilon\binom{|S|}{2}$. For bad event 2, we union bound over the at most $\binom{|S|}{3}$ non-colinear triplets, and show that each is bad only with probability ϵ. The 3 points being collinear in $\mathbb{F}^{1+k}$ is equivalent to $(\Delta_1 - \Delta_0, \mathbf{H}\mathbf{q}_1 - \mathbf{H}\mathbf{q}_0)$ and $(\Delta_2 - \Delta_0, \mathbf{H}\mathbf{q}_2 - \mathbf{H}\mathbf{q}_0)$ being linearly dependent, and since the Δ_i are distinct this is equivalent to

$$\begin{aligned}(\Delta_1 - \Delta_0)(\mathbf{H}\mathbf{q}_2 - \mathbf{H}\mathbf{q}_0) &= (\Delta_2 - \Delta_0)(\mathbf{H}\mathbf{q}_1 - \mathbf{H}\mathbf{q}_0)\\ \mathbf{H}\big((\Delta_1 - \Delta_0)(\mathbf{q}_2 - \mathbf{q}_0) - (\Delta_2 - \Delta_0)(\mathbf{q}_1 - \mathbf{q}_0)\big) &= 0.\end{aligned}$$

We have $(\Delta_1 - \Delta_0)(\mathbf{q}_2 - \mathbf{q}_0) \neq (\Delta_2 - \Delta_0)(\mathbf{q}_1 - \mathbf{q}_0)$ since the three points are not colinear in $\mathbb{F}^{1+\ell}$, so the event can only occur with probability ϵ by the universal hash property of $\mathbf{H}$. Therefore, bad event 2 has probability at most $\epsilon\binom{|S|}{3}$, and the total bad event probability is at most

$$\epsilon\binom{|S|}{3} + \epsilon\binom{|S|}{2} = \frac{\epsilon}{6}\big(|S|(|S|-1)(|S|-2) + 3|S|(|S|-1)\big) = \frac{\epsilon}{6}|S|(|S|-1)(|S|+1) \leq \frac{\epsilon|S|^3}{6}.$$

We now design the map $\mathcal{E}_{\mathrm{KOS},S}(\mathbf{H}, \widetilde{\mathbf{u}}, \widetilde{\mathbf{v}})$ so that it outputs a line in $\mathbb{F}^{1+\ell}$ that goes through at least the set of openings

$$S_{\mathbf{H},\widetilde{\mathbf{u}},\widetilde{\mathbf{v}}} = \{(\Delta, \mathbf{q}) \in S \mid \mathbf{H}\mathbf{q} = \widetilde{\mathbf{u}} \cdot \Delta + \widetilde{\mathbf{v}}\}$$

that would pass the KOS check. First, we can trivially handle any inputs where $|S_{\mathbf{H},\widetilde{\mathbf{u}},\widetilde{\mathbf{v}}}| = 1$ by picking $\mathbf{u} = 0$ and setting $\mathbf{v} = \mathbf{q}$ for the one $\mathbf{q}$; clearly this goes through at least the one point in $S_{\mathbf{H},\widetilde{\mathbf{u}},\widetilde{\mathbf{v}}}$.

Next, assume that $|S_{\mathbf{H},\widetilde{\mathbf{u}},\widetilde{\mathbf{v}}}| \geq 2$. We can also assume that all Δ in $S_{\mathbf{H},\widetilde{\mathbf{u}},\widetilde{\mathbf{v}}}$ are distinct, as otherwise bad event 1 would occur. Consider any two distinct points $(\Delta_0, \mathbf{q}_0), (\Delta_1, \mathbf{q}_1) \in S_{\mathbf{H},\widetilde{\mathbf{u}},\widetilde{\mathbf{v}}}$, and interpolate the line through them to get

$$\mathbf{u} = \frac{(\mathbf{q}_1 - \mathbf{q}_0)}{\Delta_1 - \Delta_0} \qquad \mathbf{v} = \frac{\mathbf{q}_0\Delta_1 - \Delta_0\mathbf{q}_1}{\Delta_1 - \Delta_0}.$$

If for all pair of points from $S_{\mathbf{H},\widetilde{\mathbf{u}},\widetilde{\mathbf{v}}}$ we get the same line $(\mathbf{u}, \mathbf{v})$, then clearly we can set $\mathcal{E}_{\mathrm{KOS},S}(\mathbf{H}, \widetilde{\mathbf{u}}, \widetilde{\mathbf{v}}) = (\mathbf{u}, \mathbf{v})$, as this line will contain all the points in $S_{\mathbf{H},\widetilde{\mathbf{u}},\widetilde{\mathbf{v}}}$. Otherwise, there would exist $\widetilde{\mathbf{u}}, \widetilde{\mathbf{v}}$ and 3 points $(\Delta_0, \mathbf{q}_0), (\Delta_1, \mathbf{q}_1), (\Delta_2, \mathbf{q}_2) \in S_{\mathbf{H},\widetilde{\mathbf{u}},\widetilde{\mathbf{v}}}$ that are not colinear in $\mathbb{F}^{1+\ell}$. As $S_{\mathbf{H},\widetilde{\mathbf{u}},\widetilde{\mathbf{v}}}$ is defined by a line in $\mathbb{F}^{1+k}$, $(\Delta_0, \mathbf{H}\mathbf{q}_0)$, $(\Delta_1, \mathbf{H}\mathbf{q}_1)$, and $(\Delta_2, \mathbf{H}\mathbf{q}_2)$ are colinear, so $(\Delta_0, \mathbf{q}_0), (\Delta_1, \mathbf{q}_1), (\Delta_2, \mathbf{q}_2)$ would violate bad event 2. □

The previous lemma applies to an arbitrary set S and therefore includes bad event 1. For the response set supplied by Lemma 10.19, this event is impossible because that set is the graph of a partial response function and two points with the same challenge necessarily have the same response.

Corollary 10.27 (Affine extraction). *Assuming the input-length condition of Lemma 4.6, namely $\ell + d_{\mathsf{zk}} - 1 \leq 2^{12}$, let $\varepsilon_{\mathsf{v}} := 2^{-4\lambda}(1 + 2^{-4})$ be the almost-universality error of* VOLEHash*. Given a commitment attempt t, let*

$$\mathbf{S}_t = \{(\Delta, \mathsf{Resp}_t(\Delta)) \mid \mathsf{Resp}_t(\Delta) \neq \bot\}$$

be the response graph supplied by Lemma 10.19. For a response

$$\mathbf{q} = (\mathbf{Q}, \bar{\mathbf{q}}_0, \ldots, \bar{\mathbf{q}}_{n_{\mathsf{mask}}-1}) \in \mathbb{K}^{\ell+n_{\mathsf{mask}}},$$

let $\mathbf{H}_{\mathsf{chall}_1}\mathbf{q}$ *denote the value obtained by applying* VOLEHash *with challenge* chall_1 *to the exact input split used by the verifier, namely*

$$\mathbf{H}_{\mathsf{chall}_1}\mathbf{q} := \mathsf{VOLEHash}\left(\mathsf{chall}_1, ((\mathbf{Q}[0], \ldots, \mathbf{Q}[\ell-1], \bar{\mathbf{q}}_0, \ldots, \bar{\mathbf{q}}_{d_{\mathsf{zk}}-2}), (\bar{\mathbf{q}}_{d_{\mathsf{zk}}-1}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}+1}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}+2}))\right).$$

After chall_1 *and* $(\widetilde{\mathbf{u}}, \mathbf{D})$ *have been fixed, set*

$$(\mathbf{u}_t^\star, \mathbf{v}_t^\star) := \mathcal{E}_{\mathrm{KOS},\mathbf{S}_t}(\mathbf{H}_{\mathsf{chall}_1}, \widetilde{\mathbf{u}}, \mathbf{D}).$$

Except with probability at most

$$\varepsilon_{\mathsf{KOS},t} \le \varepsilon_{\mathsf{v}} \binom{|\mathbf{S}_t|}{3}$$

over the choice of chall_1*, every* $(\Delta, \mathbf{q}) \in \mathbf{S}_t$ *satisfying*

$$\mathbf{H}_{\mathsf{chall}_1}\mathbf{q} = \mathbf{D} + \Delta\widetilde{\mathbf{u}}$$

also satisfies $\mathbf{q} = \mathbf{v}_t^\star + \Delta\mathbf{u}_t^\star$.

Proof. The graph $\mathbf{S}_t$ is fixed before chall_1 and satisfies $|\mathbf{S}_t| \le 2^{\lambda - w_{\mathsf{grind}}}$ by Lemma 10.19. Since $\mathbf{S}_t$ is a graph, bad event 1 of Lemma 10.26 is impossible. By Lemma 4.6, VOLEHash is ε_{v}-almost universal under the stated length condition, so only bad event 2 remains, with probability at most $\varepsilon_{\mathsf{v}}\binom{|\mathbf{S}_t|}{3}$. □

10.7 Overview of the security proof

We now argue security of the overall signature scheme, where we prove that it is EUF-CMA secure. We therefore begin by defining EUF-CMA security as well as the related EUF-KO notion.

Definition 10.28 (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 10.1). We denote by* $\mathsf{AdvEUFCMA}^{\mathsf{SIG}}_{\mathcal{A}} = \Pr[\mathsf{ExpEUFCMA}^{\mathsf{SIG}}_{\mathcal{A}} = 1]$ *(resp.* $\mathsf{AdvEUFKO}^{\mathsf{SIG}}_{\mathcal{A}} = \Pr[\mathsf{ExpEUFKO}^{\mathsf{SIG}}_{\mathcal{A}} = 1]$*) the advantage of an adversary* $\mathcal{A}$.

ExpEUFCMA / ExpEUFKO	OSign(msg)
1 : $\mathcal{M} := \emptyset$	1 : $\mathcal{M} := \mathcal{M} \cup \{\mathsf{msg}\}$
2 : $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}^{\mathsf{H}}(\mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$	2 : **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 : **return** $\mathsf{Verify}^{\mathsf{H}}(\mathsf{msg}, \mathsf{pk}, \sigma; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}) \wedge \mathsf{msg} \notin \mathcal{M}$	

Fig. 10.1: The EUF-CMA and EUF-KO security experiments in the random oracle model; H is a shorthand for any number of random oracles. Only for EUF-CMA does $\mathcal{A}$ get access to OSign.

Our EUF-CMA proof works in the following steps:

1. We first build a reduction from EUF-CMA security (which allows an attacker to query signatures) to EUF-KO security (where forgeries must be provided given only the public key).
2. We replace the public key with a public key that, with high probability, does not form a valid OWF instance.
3. We turn the resulting EUF-KO forgeries into an attack on the interactive proof (IP) underlying FAEST. Because the OWF instance is likely invalid, the attacker must break the soundness of the proof system.
4. We turn the IP into an IOP by using an inefficient extractor to obtain the underlying VOLE correlations (which become IOP strings).
5. Finally, we show Round-by-Round (RBR) soundness of this IOP.

We first give an overview and provide the full proof in Sections 10.8, 10.9, and 10.10.

EUF-CMA *to* EUF-KO. We begin with what is a standard technique, namely emulating a signing oracle without access to the signing key by using that FAEST is an honest-verifier zero-knowledge proof. Simulating a signature can be done by running an honest-verifier zero-knowledge simulator and programming the simulated challenges into the Fiat–Shamir random oracles $\mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$. By assumption, the resulting proofs are indistinguishable from real signatures except that programming of the random oracles may fail. We note here a technical detail in our proof, namely, that the fixed blocksize of 128 bits of AES necessitates a "divergence based" approach for hiding and simulatability, and relies on the upper bound of 2^{64} signature queries, independent of the security parameter. To address that programming of random oracles does not lead to aborts, we use the unpredictability of BAVC commitments: because the adversary will not have queried the input belonging to the programmed output beforehand if it is unpredictable, it can actually be programmed.

Aside from this, the reduction from EUF-CMA to EUF-KO is standard and is summarized in the following theorem, which is proven in Section 10.10.

Theorem 10.29. *Let ρ be chosen uniformly at random on every signing invocation. Let $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_4$ and H_5 be modeled as random oracles and let H_3 be a PRF. Then, for every PPT adversary $\mathcal{A}$ making at most $Q_{\mathsf{sig}} \leq 2^{64}$ signing queries and Q_0, Q_1, $Q_{2,0}$, $Q_{2,1}$, $Q_{2,2}, Q_{2,3}, Q_4, Q_5$ queries to the respective random oracles, there exists a PPT* EUF-KO *adversary $\mathcal{A}'$ such that*

$$\mathsf{AdvEUFCMA}_{\mathcal{A}}^{\mathsf{FAEST}}[Q_{\mathsf{sig}}] \leq 2\mathsf{AdvEUFKO}_{\mathcal{A}'}^{\mathsf{FAEST}} + \varepsilon_{\mathsf{CMA}},$$

where $\varepsilon_{\mathsf{CMA}}$ is the complete reduction loss stated in Theorem 10.42.

Moving to Unconditional Soundness. For FAEST and FAEST-EM, the key generation algorithm KeyGen excludes keys with bits $\mathbf{k}[0] = \mathbf{k}[1] = 1$. Thus, $\frac{\mathsf{dom}(\mathsf{OWF})}{\mathsf{cod}(\mathsf{OWF})} \leq \frac{3}{4}$ and $\left(1 - \frac{\mathsf{dom}(\mathsf{OWF})}{\mathsf{cod}(\mathsf{OWF})}\right)^{-1} \leq 4$.

Each signature is a Fiat-Shamir transformed VOLE-in-the-Head proof of knowledge of this secret key $\mathbf{k}$ to the public key $\mathsf{pk} = (\mathbf{x}, \mathbf{y})$. The proven relation asserts $\mathsf{OWF}_{\mathbf{x}}(\mathbf{k}) = \mathbf{y}$ and $\mathbf{k}[0] \cdot \mathbf{k}[1] = 0$, and hence at least $\frac{1}{4}$th of all $\mathbf{y}$ have no preimage.

Our reduction switches to such a pk, for which there exists no secret key and the statement is false. This allows us to reduce to *soundness instead of knowledge soundness*, permitting the use of an inefficient extractor in the security argument. We describe this approach in more detail in Section 10.8. This diverts from the original analysis of FAEST and follows an idea of Katz and Wang [KW03].

Proving EUF-KO *Security.* Our security reduction for EUF-KO, which we fully describe in Section 10.9, 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 interactive oracle proof (IOP):

1. In Lemma 10.36 we convert any attacker successfully generating EUF-KO forgeries into an attacker that wins in a state restoration-sound (SRS) interactive proof security game. Note that this conversion is standard as SRS can be seen as an interactive analogue of Fiat-Shamir NIZKs. The verifier of this SRS game is constructed as an interactive version of the FAEST signature verification algorithm. To build the attacker against the SRS game, we follow a known strategy and bound the probability that calls to the Fiat-Shamir random oracles $\mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$ made by the EUF-KO adversary are not occurring in the order in which an honest signer performs them, or that two distinct predecessor queries produce the same chaining value.
2. Since the proven statement is false, we can run the inefficient response extractor from Lemma 10.19, which fixes a partial response function before chall_1. Combining this response binding with the KOS consistency check, after chall_1 and $(\widetilde{\mathbf{u}}, \mathbf{D})$ have been fixed, yields effective affine coefficients $(\mathbf{u}^\star, \mathbf{v}^\star)$ such that every subsequently accepted reconstruction $(\Delta, \mathbf{q})$ satisfies $\mathbf{q} = \mathbf{v}^\star + \Delta\mathbf{u}^\star$. These effective correlations allow us to convert the SRS IP verifier into an SRS IOP verifier whose oracle is formed by $(\Delta, \mathbf{q})$.
3. Finally, we prove RBR soundness of the IOP verified by the verifier constructed in the previous step. This IOP verifier only performs information-theoretic checks. We use a consistency check in the style of [KOS15] (using Lemma 10.26) for VOLEHash, as part of the global affine-binding argument,use ZKHash (using Lemma 4.7) to compress the circuit constraints, and use the QuickSilver check [YSWW21] to check for correctness of circuit evaluation, similar to [BBD+23] (using Lemma 7.3). We then apply the theorem that round-by-round (RBR) soundness implies state-restoration soundness, finishing the proof.

The difference in FAEST and FAEST-EM for the security proof 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 10.13) on $\mathsf{PRG}_{2\lambda}$, which allows us to keep the 2λ output length, which is minimal by the birthday bound.

10.8 Switching to False Statements

In this as well as Section 10.9, we prove the following theorem:

Theorem 10.30 (FAEST is EUF-KO). *Let* H_0, H_1, $\mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_4$ *and* H_5 *be modeled as random oracles, and let* $(\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}, \mathtt{param}_{\mathsf{VOLE}})$ *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 *and* H_5, *respectively. Suppose that* $\ell + d_{\mathsf{zk}} - 1 \leq 2^{12}$, $C + 1 \leq 2^{13}$ *and* $d_{\mathsf{zk}} \in \{7, 14\}$ *and all other parameters are as required in in Lemmas 4.6 and 4.7. Set*

$$Q_{\mathsf{FS}} := Q_{2,0} + Q_{2,1} + Q_{2,2} + Q_{2,3} + 4, \quad Q_C := Q_{2,1} + 1.$$

Then

$$\mathbf{Adv}^{\mathsf{EUF\text{-}KO}}_{\mathcal{A}} \leq \left(1 - \tfrac{\mathsf{dom}(\mathsf{OWF})}{\mathsf{cod}(\mathsf{OWF})}\right)^{-1} \cdot \left(\mathsf{AdvSnd}^{\mathsf{SIG}}_{\mathcal{B}_2} + \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1}\right)$$

where OWF *is the OWF (or more precisely, PRG) defined by* $\mathtt{param}_{\mathsf{OWF}}$*, and*

$$\mathsf{AdvSnd}^{\mathsf{FAEST}}_{\mathcal{B}_2} \leq \frac{Q_{\mathsf{FS}}^2}{2^{2\lambda}} + \frac{(Q_1+Q_C)^2}{2^{2\lambda}} + \frac{\tau(Q_0+Q_C)}{2^{\lambda}} + (Q_{2,1}+Q_{2,2}+Q_{2,3}+8)\frac{d_{\mathsf{zk}}}{2^{\lambda}}$$

for $\mathsf{SIG} = \mathsf{FAEST}$ *as well as*

$$\mathsf{AdvSnd}^{\mathsf{FAEST\text{-}EM}}_{\mathcal{B}_2} \leq \frac{Q_{\mathsf{FS}}^2}{2^{2\lambda}} + \frac{(Q_1+Q_C)^2}{2^{2\lambda}} + Q_C \max_t \mathsf{AdvInj}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_t}[L] + (Q_{2,1}+Q_{2,2}+Q_{2,3}+8)\frac{d_{\mathsf{zk}}}{2^{\lambda}}$$

for $\mathsf{SIG} = \mathsf{FAEST\text{-}EM}$.

Remark 10.31. As in Lemma 10.19, 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.

To show EUF-KO security, we need the following technical fact.

Lemma 10.32. *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 10.32 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 10.9) 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 $\Pr[J=y] \geq \frac{1}{|\mathsf{dom}(F)|}$. Thus

$$\frac{\Pr[I=y]}{\Pr[J=y]} \leq \frac{|\mathsf{dom}(F)|}{|\mathsf{im}(F)|}$$

and the claim follows by Lemma 10.9. □

Proof of Theorem 10.30. We argue via game hops. We prove the result for FAEST first, and then explain what changes for $\mathsf{FAEST\text{-}EM}$. We write G_i for the output of the i-th game.

Game G_0**:** This is the real $\mathsf{EUF\text{-}KO}$ security game. As in the $\mathsf{EUF\text{-}KO}$ game, we let sk, pk be the generated challenge secret and public key, respectively. We have

$$\Pr[\mathsf{G}_0=1] = \mathsf{AdvEUFKO}^{\mathsf{FAEST}}_{\mathcal{A}}$$

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 $\mathsf{EUF\text{-}KO}$ game. We have

$$|\Pr[\mathsf{G}_1=1] - \Pr[\mathsf{G}_0=1]| \leq \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1}[1]$$

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 $F_{\mathbf{k}}(\mathbf{x}) = \mathbf{y}$.

Define G_1^* as G_1, except that it returns 0 if pk is valid, i.e., $\mathsf{pk} \in \mathcal{L}_{\mathcal{R}}$ is language of public keys. Observe that

$$p = \Pr[\mathsf{G}_1^* = 1] = \Pr[\mathsf{G}_1 = 1 \wedge \neg I]$$

and G_1^* is exactly the experiment in Corollary 10.41 and p the respective success probabilty. It follows that

$$\Pr[\mathsf{G}_1 = 1 \mid \neg I] = \frac{\Pr[\mathsf{G}_1^* = 1]}{\Pr[\neg I]},$$

by conditional probability and since the game outputs 0 if $\mathsf{pk} \in \mathcal{L}$, i.e., if I occurs. On the other hand, we have from Lemma 10.32

$$\Pr[\mathsf{G}_1 = 1 \mid I] \leq \frac{|\mathsf{dom}(\mathsf{OWF})|}{|\mathsf{im}(\mathsf{OWF}_{\mathbf{x}})|} \cdot \Pr[\mathsf{G}_0 = 1]$$

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

$$\begin{aligned}
\Pr[\mathsf{G}_1 = 1] &= \Pr[I] \cdot \Pr[\mathsf{G}_1 = 1 \mid I] + \Pr[\neg I] \cdot \Pr[\mathsf{G}_1 = 1 \mid \neg I] \\
&\leq \Pr[I] \cdot \frac{|\mathsf{dom}(\mathsf{OWF})|}{|\mathsf{im}(\mathsf{OWF}_{\mathbf{x}})|} \Pr[\mathsf{G}_0 = 1] + \Pr[\neg I] \cdot \frac{\Pr[\mathsf{G}_1^* = 1]}{\Pr[\neg I]} \\
&\leq \frac{|\mathsf{dom}(\mathsf{OWF})|}{|\mathsf{cod}(\mathsf{OWF})|} \Pr[\mathsf{G}_0 = 1] + \Pr[\mathsf{G}_1^* = 1]
\end{aligned}$$

where the last step uses $\Pr[I] = \frac{|\mathsf{im}(\mathsf{OWF}_{\mathbf{x}})|}{|\mathsf{cod}(\mathsf{OWF})|}$. (This 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

$$\Pr[\mathsf{G}_0 = 1] \leq \Pr[\mathsf{G}_1 = 1] + \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1} \leq \frac{\mathsf{dom}(\mathsf{OWF})}{\mathsf{cod}(\mathsf{OWF})} \Pr[\mathsf{G}_0 = 1] + \Pr[\mathsf{G}_1^* = 1] + \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1}$$

and resolving for $\Pr[\mathsf{G}_0 = 1]$ yields

$$\Pr[\mathsf{G}_0 = 1] \leq \frac{1}{1 - \frac{\mathsf{dom}(\mathsf{OWF})}{\mathsf{cod}(\mathsf{OWF})}} \cdot \left(\Pr[\mathsf{G}_1^* = 1] + \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1}\right) \tag{12}$$

which yields the claim after plugging in the advantage bound for $\Pr[\mathsf{G}_1^*]$ from Corollary 10.41. □

10.9 EUF-KO

Throughout this section, as in Section 10.5.1, H_5 is treated as a fixed public function that all algorithms and adversaries evaluate directly (rather than as a random oracle). No argument here relies on its randomness, and all statements also hold for every possible choice of a random oracle H_5.

$\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}((s_i)_{i\in[\mathsf{k}]}, \mathsf{rnd}, \tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}})$

$\mathsf{H} \leftarrow \mathcal{UF}(2\lambda)$
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 $\rho_i \leftarrow \mathsf{rnd}_i(\mathbb{x}, (\alpha_1, \ldots, \alpha_i), (\mathsf{st}_1, \ldots, \mathsf{st}_i))$
return $(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\mathsf{st}_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]})$

Fig. 10.2: The state-restoration game for IPs.

$\mathcal{V}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_4}_{\mathrm{IP}}(\mathbb{x}, (\alpha_1, \ldots, \alpha_4), (\rho_1, \ldots, \rho_4))$

1. **Parse the input and check grinding.**
   - Write $\mathbb{x} = (\mathsf{pk}, \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}, \mathtt{param}_{\mathsf{VOLE}})$, $\alpha_1 = (\mu \| \mathsf{com} \| \mathbf{c}_1 \| \ldots \| \mathbf{c}_{\tau-1} \| \mathbf{c}_{\mathsf{mult}} \| \mathsf{iv}^{\mathsf{pre}})$, $\alpha_2 = (\tilde{\mathbf{u}} \| \mathbf{D} \| \mathbf{d})$, $\alpha_3 = (\tilde{a}_0 \| \ldots \| \tilde{a}_{d_{\mathsf{zk}}-1})$, $\alpha_4 = (\mathsf{decom}_I)$.
   - Set $\mathsf{chall}_1 := \rho_1, \mathsf{chall}_2 := \rho_2, \mathsf{chall}_3 := \rho_3$.
   - Set $\mathsf{iv} \leftarrow \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu)$.
   - Check w_{grind} trailing bits of chall_3. If they are not all zero, return false.
2. **Reconstruct the VOLE Commitments.**
   - Compute $(\widetilde{\mathsf{com}}, \mathbf{Q}, (\bar{\mathbf{q}}_m)_{m\in[0..n_{\mathsf{mask}})}, \Delta) \leftarrow \mathsf{VOLEReconstruct}(\mathsf{chall}_3, \mathsf{decom}_I, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}; \mathtt{param}, \mathtt{param}_{\mathsf{VOLE}})$.
   - If $\mathsf{VOLEReconstruct}$ outputs $\perp$ or $\widetilde{\mathsf{com}} \neq \mathsf{com}$, return false.
3. **Hash the VOLE Tags.**
   - Set $\mathbf{Q}^{\mathsf{h}}_0 := (\mathbf{Q}[0], \ldots, \mathbf{Q}[\ell-1], \bar{\mathbf{q}}_0, \ldots, \bar{\mathbf{q}}_{d_{\mathsf{zk}}-2})$ and $\mathbf{Q}^{\mathsf{h}}_1 := (\bar{\mathbf{q}}_{d_{\mathsf{zk}}-1}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}+1}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}+2})$.
   - Compute $\widetilde{\mathbf{Q}} := \mathsf{VOLEHash}(\mathsf{chall}_1, (\mathbf{Q}^{\mathsf{h}}_0, \mathbf{Q}^{\mathsf{h}}_1)) \in \mathbb{F}^4_{2^\lambda}$
4. **Combine with $\tilde{\mathbf{u}}$.**
   If $\mathbf{D} \neq \widetilde{\mathbf{Q}} + \Delta \cdot \tilde{\mathbf{u}}$ then return false.
5. **Verify the QuickSilver (or AES) Proof.**
   Compute
   $$\widehat{a}_0 \leftarrow \mathsf{ZK.OWFVerify}(\mathbf{d}, \mathbf{Q}, (\bar{\mathbf{q}}_j)_{j\in[0..d_{\mathsf{zk}}-1)}, \mathsf{pk}, \mathsf{chall}_2, \Delta, \tilde{a}_1, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}).$$
6. **Check Consistency.**
   If $\widehat{a}_0 = \tilde{a}_0$ return true, otherwise false.

Figure 10.3: The FAEST interactive proof verifier.

10.9.1 Interactive proof induced by FAEST We present the IP derived from the FAEST signature scheme. For $k \in \mathbb{N}$, let denote the set of $\mathcal{UF}(k)$ of functions from $\{0,1\}^*$ to $\{0,1\}^k$, and sampling from $\mathcal{UF}(k)$ is implemented via lazy sampling.

Definition 10.33 ([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* $\mathsf{rnd} = (\mathsf{rnd}_i)_{i\in[\mathsf{k}]} \in (\mathcal{UF}(\mathsf{r}_i))_{i\in[\mathsf{k}]}$*, and IP state-restoration prover* $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ *is defined in Figure 10.2. 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 10.34 ([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* $(s_i)_{i\in[\mathsf{k}]} \in \mathbb{N}^{\mathsf{k}}$*, move budget* $t \in \mathbb{N}$*, and* t*-move malicious IP state-restoration prover* $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$*,*

$$\Pr\left[\begin{array}{l|l} \mathbb{x} \notin \mathcal{L}(\mathcal{R}) & \mathsf{rnd} = (\mathsf{rnd}_i)_{i\in[\mathsf{k}]} \leftarrow (\mathcal{UF}(\mathsf{r}_i))_{i\in[\mathsf{k}]} \\ \wedge\, \mathcal{V}_{\mathrm{IP}}(\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}) = 1 & (\mathbb{x}, (\alpha_i)_{i\in[\mathsf{k}]}, (\mathsf{st}_i)_{i\in[\mathsf{k}]}, (\rho_i)_{i\in[\mathsf{k}]}) \leftarrow \\ & \mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}((s_i)_{i\in[\mathsf{k}]}, \mathsf{rnd}, \tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}) \end{array}\right] \leq \epsilon^{\mathrm{sr}}_{\mathrm{IP}}((s_i)_{i\in[\mathsf{k}]}, t)$$

In Figure 10.3 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 10.33. This is, however, of no importance for the proof, which only considers a reduction to the state-restoration soundness of the FAEST IP.

10.9.2 From EUF-KO to SR soundness We begin with a proposition that we need in the construction of the reduction:

Proposition 10.35 (Random oracle list game). *Let $n \in \mathbb{N}$ and for $i \in \{0,\dots,n+1\}$ let $\mathcal{Y}_i, \mathcal{Y}'_i \subset \{0,1\}^*$ be finite sets. For each $i \in \{0,\dots,n\}$ 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 $i \in \{0,\dots,n\}$ 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 $i < n \wedge\ y \in L_{i+1}$ and $H_i(x,x')$ was not queried before, 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 $H_0,\dots,H_n$. Then the probability that $\mathcal{B}$ wins is bounded by

$$Q^2/(2\cdot Y).$$

Proof of Proposition 10.35. Clearly, $\mathcal{B}$ can only win by querying $H_0,\dots,H_{n-1}$. Moreover, it can only win for a *fresh* query. 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 $H_1,\dots,H_n$. 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

$$\sum_{i=1}^{Q} \frac{i-1}{Y} \leq \frac{Q^2}{2\cdot Y}. \qquad \square$$

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 10.36. *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 an 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 10.28 with random public key $\mathbf{x} \leftarrow \{0,1\}^{N_{\text{st-bits}}}$, $\mathbf{y} \leftarrow \{0,1\}^{\beta N_{\text{st-bits}}}$, $\mathsf{pk} := (\mathbf{x},\mathbf{y})$. Suppose $\mathcal{A}$ wins the EUF-KO game for $\mathsf{pk} \notin \mathcal{L}_{\mathcal{R}}$ with probability*

$$p = \Pr[\mathsf{ExpEUFRK}^{\mathsf{SIG}}_{\mathcal{A}}(\mathsf{pk}) = 1 \wedge \mathsf{pk} \notin \mathcal{L}_{\mathcal{R}} \mid \mathbf{x} \leftarrow \{0,1\}^{N_{\text{st-bits}}},\ \mathbf{y} \leftarrow \{0,1\}^{\beta N_{\text{st-bits}}},\ \mathsf{pk} := (\mathbf{x},\mathbf{y})].$$

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. Set $(s_1,s_2,s_3,s_4) := (2\lambda, 8\lambda+64, 3\lambda+96, 0)$. Then there exists a 4-round probabilistic algorithm $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ 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 10.34 with probability

$$\epsilon^{\mathrm{sr}}_{(\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}},\mathcal{V}_{\mathrm{IP}})}((s_i)_{i\in[4]}, Q_{2,1}+Q_{2,2}+Q_{2,3}+3) \geq p - (Q_{2,0}+Q_{2,1}+Q_{2,2}+Q_{2,3}+4)^2/2^{2\lambda}$$

against $\mathcal{V}_{\mathrm{IP}}$. *Moreover,* $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ *makes at most* $Q_{2,1}+Q_{2,2}+Q_{2,3}+3$ *moves in the SRS game and its running time is roughly that of* $\mathcal{A}$.

Proof. We define a malicious prover $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ 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. $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ is defined for an interactive game, where the messages from the SRS experiment will replace the responses from $\mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$.

Initially, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ samples $\mathbf{x} \leftarrow \{0,1\}^{N_{\mathsf{st\text{-}bits}}}$, $\mathbf{y} \leftarrow \{0,1\}^{\beta N_{\mathsf{st\text{-}bits}}}$, and $\mathsf{pk} := (\mathbf{x}, \mathbf{y})$, just as in EUF-KO with random public key. Then, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ starts $\mathcal{A}$ with the pk as well as $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}, \mathtt{param}_{\mathsf{VOLE}}$ as inputs. Further, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ runs the EUF-KO game with $\mathcal{A}$ and simulates access to the random oracles. Whenever $\mathcal{A}$ makes a query to any of $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_3, \mathsf{H}_4$, the reduction $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ 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 $\mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$, the reduction $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ runs the following *RO simulation*: For each of these oracles it maintains a table indexed by the complete query such that repeated complete queries are answered from the table and do not update any bad list or invoke the SRS experiment. The following rules therefore apply only to fresh queries, and every fresh answer is recorded together with its associated partial transcript, if one exists. Such an association is made only when the query's leading value was returned by a unique associated query at the preceding level, and outputs of H_2^0 form the base case. A query for which no such unique association exists is answered uniformly, and its leading value is added to the corresponding bad list.

- When $\mathcal{A}$ makes a query $\mu\|\mathsf{com}\|\mathbf{c}_1\|\ldots\|\mathbf{c}_{\tau-1}\|\mathbf{c}_{\mathsf{mult}}\|\mathsf{iv}^{\mathsf{pre}}$ to H_2^1,
  1. $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ checks if μ was previously returned to $\mathcal{A}$ as a response to a query to H_2^0; if not, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ returns a randomly sampled chall_1 to $\mathcal{A}$ and puts μ on the bad list.
  2. Otherwise, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ 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 $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$, where $\alpha_1 = \mu\|\mathsf{com}\|\mathbf{c}_1\|\ldots\|\mathbf{c}_{\tau-1}\|\mathbf{c}_{\mathsf{mult}}\|\mathsf{iv}^{\mathsf{pre}}$ and $\mathsf{st}_1 = \mu$ serves as salt. It receives ρ_1 from $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$ and passes $\mathsf{chall}_1 = \rho_1$ to $\mathcal{A}$ as the response from H_2^1.
- When $\mathcal{A}$ makes a query $\mathsf{chall}_1\|\tilde{\mathbf{u}}\|\mathbf{D}\|\mathbf{d}$ to H_2^2,
  1. $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ checks if chall_1 was previously returned to $\mathcal{A}$ as a response from H_2^1; if not, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ returns a randomly sampled chall_2 to $\mathcal{A}$ and puts chall_1 on the bad list.
  2. Otherwise, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ 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 $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$, 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 $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$ and returns $\mathsf{chall}_2 = \rho_2$ to $\mathcal{A}$ as the response from H_2^2.
- When $\mathcal{A}$ makes a query $\mathsf{chall}_2\|\tilde{a}_0\|\ldots\|\tilde{a}_{d_{\mathsf{zk}}-1}\|\mathsf{ctr}$ to H_2^3,
  1. $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ checks if chall_2 was previously returned to $\mathcal{A}$ as a response from H_2^2; if not, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ returns a randomly sampled chall_3 to $\mathcal{A}$ and puts chall_2 on the bad list.
  2. Otherwise, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ 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 $\alpha_3 = \tilde{a}_0\|\ldots\|\tilde{a}_{d_{\mathsf{zk}}-1}$ and $\mathsf{st}_3 = (\mathsf{chall}_2, \mathsf{ctr})$. It receives $\rho_3 \in \{0,1\}^{\lambda}$ from $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$ 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 $\mathsf{msg}, \sigma = ((\mathbf{c}_i)_{i\in[1..\tau)}, \mathbf{c}_{\mathsf{mult}}, \tilde{\mathbf{u}}, \mathbf{d}, \tilde{a}_1, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}, \mathsf{decom}_I, \mathsf{chall}_3, \mathsf{iv}^{\mathsf{pre}}, \mathsf{ctr})$. $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$, 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 $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ 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, applying the same table rule as above.

- When Verify makes the query $\mu\|\mathsf{com}\|\mathbf{c}_1\|\ldots\|\mathbf{c}_{\tau-1}\|\mathbf{c}_{\mathsf{mult}}\|\mathsf{iv}^{\mathsf{pre}}$ to H_2^1, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ checks if μ was put on the bad list during *RO simulation*. If yes, then $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ aborts. Otherwise it outputs

$(\mathbb{x}, (\alpha_1), (\mathsf{st}_1))$ to $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$, where $\alpha_1 = \mu \| \mathsf{com} \| \mathbf{c}_1 \| \dots \| \mathbf{c}_{\tau-1} \| \mathbf{c}_{\mathsf{mult}} \| \mathsf{iv}^{\mathsf{pre}}$ and $\mathsf{st}_1 = \mu$ serves as salt. It receives ρ_1 from $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$ and passes $\mathsf{chall}_1 = \rho_1$ to Verify as the response from H^1_2.

- When Verify makes the query $\mathsf{chall}_1 \| \widetilde{\mathbf{u}} \| \mathbf{D} \| \mathbf{d}$ to H^2_2, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ checks if chall_1 was put on the bad list during *RO simulation*. If yes, then $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ aborts. Otherwise it outputs $(\mathbb{x}, (\alpha_1, \alpha_2), (\mathsf{st}_1, \mathsf{st}_2))$ to $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$, where $\alpha_2 = \widetilde{\mathbf{u}} \| \mathbf{D} \| \mathbf{d}$ and $\mathsf{st}_2 = \mathsf{chall}_1$ serves as salt. It receives ρ_2 from $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$ and returns $\mathsf{chall}_2 = \rho_2$ to Verify as the response from H^2_2.
- When Verify makes the query $\mathsf{chall}_2 \| \tilde{a}_0 \| \dots \| \tilde{a}_{d_{\mathsf{zk}}-1} \| \mathsf{ctr}$ to H^3_2, $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ checks if chall_2 was put on the bad list during *RO simulation*. If yes, then $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ aborts. Otherwise it outputs $(\mathbb{x}, (\alpha_1, \alpha_2, \alpha_3), (\mathsf{st}_1, \mathsf{st}_2, \mathsf{st}_3))$, where $\alpha_3 = \tilde{a}_0 \| \dots \| \tilde{a}_{d_{\mathsf{zk}}-1}$ and $\mathsf{st}_3 = (\mathsf{chall}_2, \mathsf{ctr})$. It receives $\rho_3 \in \{0,1\}^\lambda$ from $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$ returns $\mathsf{chall}_3 = \rho_3$ to Verify as the response from H^3_2.

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 FAEST.Verify rejects the signature, then $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ aborts. Else, define $\alpha_1 = (\mu \| \mathsf{com} \| \mathbf{c}_1 \| \dots \| \mathbf{c}_{\tau-1} \| \mathbf{c}_{\mathsf{mult}} \| \mathsf{iv}^{\mathsf{pre}})$, $\alpha_2 = (\widetilde{\mathbf{u}} \| \mathbf{D} \| \mathbf{d})$, $\alpha_3 = (\tilde{a}_0 \| \dots \| \tilde{a}_{d_{\mathsf{zk}}-1})$ and $\alpha_4 = (\mathsf{decom}_I)$. Then exit the while loop in $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$ and output $(\mathsf{pk}, (\alpha_1, \alpha_2, \alpha_3, \alpha_4), (\mu, \mathsf{chall}_1, \mathsf{chall}_2 \| \mathsf{ctr}, \emptyset))$ to the experiment.

Note that the outputs of the ROs $\mathsf{H}^1_2, \mathsf{H}^2_2, \mathsf{H}^3_2$ which $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ 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 10.33, 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 $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ does not change the probability of $\mathcal{A}$ outputting a valid forgery when compared with the EUF-KO security experiment (for any choice of pk) . Also, note that $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ never aborts during the interaction with $\mathcal{A}$ that creates the forgery but potentially after obtaining the forgery attempt from $\mathcal{A}$.

Assume that $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ outputs a value to the experiment $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$ 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 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^3_2 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 $\mathbf{d}, \mathbf{Q}, (\overline{\mathbf{q}}_m)_{m \in [0..d_{\mathsf{zk}}-1)}, \mathsf{pk}, \mathsf{chall}_2, \Delta, \tilde{a}_1, \dots, \tilde{a}_{d_{\mathsf{zk}}-1}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$ 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 chall_2 and Δ deterministically and injectively computed from chall_3) and outputs to the experiment $\mathsf{Exp}^{\mathrm{sr}}_{\mathrm{IP}}$ is identical to the RO input to H^3_2 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, unless $\mathsf{pk} \in \mathcal{L}_{\mathcal{R}}$, 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 10.35: if the leading value μ of an input query $x = \mu\|\mathsf{com}\|\mathbf{c}_1\|\ldots\|\mathbf{c}_{\tau-1}\|\mathbf{c}_{\mathsf{mult}}\|\mathsf{iv}^{\mathsf{pre}}$ to H_2^1 was added to the bad list, then μ was not output by H_2^0 before (the same argument follows identically for $\mathsf{H}_2^2, \mathsf{H}_2^3$). If $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ aborts due to the bad list event for H_2^1 during Verify, then $\mathcal{A}$ must have output a forgery tuple msg, σ (for a predefined pk) such that $\mathsf{H}_2^0(\mathsf{pk}\|\mathsf{msg}) = \mu$ and where $x = \mu\|\ldots$ was queried to H_2^1 before $\mathsf{pk}\|\mathsf{msg}$ was queried to H_2^0 (if it was not queried by $\mathcal{A}$ during the EUF-KO game it will be queried by Verify). Let $Q_{\mathsf{FS}} := Q_{2,0} + Q_{2,1} + Q_{2,2} + Q_{2,3} + 4$, where the additional 4 count the queries that may be made during Verify. Then, by modeling $\mathsf{H}_2^0, \ldots, \mathsf{H}_2^3$ as the random oracles in Proposition 10.35, the abort probability due to the bad-list event is at most $Q_{\mathsf{FS}}^2/(2 \cdot 2^{2\lambda})$. In addition, recovery of a partial transcript can be ambiguous if two distinct fresh chained queries at one level return the same intermediate response, so the same union-bound calculation bounds this event by $Q_{\mathsf{FS}}(Q_{\mathsf{FS}} - 1)/(2 \cdot 2^{2\lambda})$. Their sum is at most $Q_{\mathsf{FS}}^2/2^{2\lambda}$, as claimed.

By assumption, the probability that $\mathsf{pk} \notin \mathcal{L}_{\mathcal{R}}$ and $\mathcal{A}$ outputs a valid forgery for a uniformly random pk is exactly p. Thus, the probability that $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$ wins the SRS experiment is at least

$$p - (Q_{2,0} + Q_{2,1} + Q_{2,2} + Q_{2,3} + 4)^2/2^{2\lambda}$$

which concludes the proof. □

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 $\tilde{\mathcal{P}}^{\mathrm{sr}}_{\mathrm{IP}}$.

10.9.3 From SRS IP to SRS IOP We now define a simplified notion of an IOP (following [BBD+23]), 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 10.37 (Interactive Oracle Proof (IOP)). *A (public-coin)* interactive oracle proof *(IOP) [BCS16, RRR16] for NP-relation $\mathcal{R}$ in the random oracle model 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 10.37 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 10.34, 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 10.37.

We now rewrite $\mathcal{V}_{\mathrm{IP}}$ as an IOP which we call $\mathcal{V}_{\mathrm{IOP}}$. We write $\mathsf{AdvSRSIOP}^{\mathcal{V}_{\mathrm{IOP}}}_{\mathcal{A}}$ for the advantage of an adversary $\mathcal{A}$ against state restoration soundness of the IOP with verifier $\mathcal{V}_{\mathrm{IOP}}$. The algorithm is defined in Figure 10.4.

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 $(\Delta, (\mathbf{Q}, \overline{\mathbf{q}}_1, \ldots, \overline{\mathbf{q}}_{n_{\mathsf{mask}}-1}))$ pairs corresponding to the committed inputs as supposedly generated by VOLECommit.

$\mathcal{V}_{\mathrm{IOP}}(\mathbb{x}, (\alpha_1, \dots, \alpha_4), (\rho_1, \dots, \rho_4))$

1. **Parse the input and check grinding.**
   - Write $\mathbb{x} = (\mathsf{pk}, \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$, $\alpha_1 = (g_1)$, $\alpha_2 = (\tilde{\mathbf{u}} \| \mathbf{D} \| \mathbf{d})$, $\alpha_3 = (\tilde{a}_0 \| \dots \| \tilde{a}_{d_{\mathsf{zk}}-1})$.
   - Set $\mathsf{chall}_1 := \rho_1, \mathsf{chall}_2 := \rho_2, \mathsf{chall}_3 := \rho_3$.
   - Check w_{grind} trailing bits of chall_3. If they are not all zero, return false.
2. **Reconstruct the VOLE Commitments from IOP Oracle.**
   - Call $(\mathbf{Q}, (\overline{\mathbf{q}}_m)_{m \in [0..n_{\mathsf{mask}})}, \Delta) \leftarrow \mathsf{VOLEIOPReconstruct}(g_1, \mathsf{chall}_3; \mathtt{param})$.
   - If VOLEIOPReconstruct outputs $\perp$ return false.
3. **Hash the VOLE Tags.**
   - Set $\mathbf{Q}_0^{\mathsf{h}} := (\mathbf{Q}[0], \dots, \mathbf{Q}[\ell - 1], \bar{\mathbf{q}}_0, \dots, \bar{\mathbf{q}}_{d_{\mathsf{zk}}-2})$ and $\mathbf{Q}_1^{\mathsf{h}} := (\bar{\mathbf{q}}_{d_{\mathsf{zk}}-1}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}+1}, \bar{\mathbf{q}}_{d_{\mathsf{zk}}+2})$.
   - Compute $\widetilde{\mathbf{Q}} := \mathsf{VOLEHash}(\mathsf{chall}_1, (\mathbf{Q}_0^{\mathsf{h}}, \mathbf{Q}_1^{\mathsf{h}})) \in \mathbb{F}_{2^\lambda}^4$

   If $\mathbf{D} \neq \widetilde{\mathbf{Q}} + \Delta \cdot \tilde{\mathbf{u}}$ then return false.
4. **Verify the QuickSilver (or AES) Proof.**
   Compute

   $$\widehat{a}_0 \leftarrow \mathsf{ZK.OWFVerify}(\mathbf{d}, \mathbf{Q}, (\bar{\mathbf{q}}_j)_{j \in [0..d_{\mathsf{zk}}-1)}, \mathsf{pk}, \mathsf{chall}_2, \Delta, \tilde{a}_1, \dots, \tilde{a}_{d_{\mathsf{zk}}-1}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}).$$

5. **Check Consistency.**
   If $\widehat{a}_0 = \tilde{a}_0$ return true, otherwise false.

Figure 10.4: The FAEST IOP verifier.

Therefore, no opening decom_I for the commitments must be sent in α_4, and $\mathsf{iv}^{\mathsf{pre}}$ and μ must also not be used. As the extractor Lemma 10.19, that we will later use to generate g_1, already applies all $\mathbf{c}_i$ and $\mathbf{c}_{\mathsf{mult}}$, these must also not be included. 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 10.5.

$\mathsf{VOLEIOPReconstruct}(g_1, \mathsf{chall}_3; \mathtt{param})$

INPUT: $g_1 \in (\mathbb{K}^{\ell + n_{\mathsf{mask}}} \cup \{\perp\})^{\mathcal{D}_\Delta}, \mathsf{chall}_3 \in \{0,1\}^\lambda$

OUTPUT: $\mathbf{Q} \in \mathbb{F}_{2^\lambda}^{\ell}, (\bar{\mathbf{q}}_m)_{m \in [0..n_{\mathsf{mask}})} \in \mathbb{F}_{2^\lambda}^{n_{\mathsf{mask}}}, \Delta \in \mathbb{F}_{2^\lambda}$

$I = (\Delta_0, \dots, \Delta_{\tau-1}) \leftarrow \mathsf{DecodeAllChall}_3(\mathsf{chall}_3; \mathtt{param})$
if $I = \perp$ **then return** $\perp$
else do
  Set $\Delta' := \mathsf{BitDec}(\Delta_0, d_0) \| \cdots \| \mathsf{BitDec}(\Delta_{\tau-1}, d_{\tau-1})$
  Compute $\Delta := \mathsf{ToField}(\mathbf{W}_{\mathrm{crt}} \cdot \Delta', \lambda)$
  if $g_1(\Delta) = \perp$ **then return** $\perp$
  Write $(\mathbf{Q}, \bar{\mathbf{q}}_0, \dots, \bar{\mathbf{q}}_{n_{\mathsf{mask}}-1}) := g_1(\Delta)$
  return $(\mathbf{Q}, \bar{\mathbf{q}}_0, \dots, \bar{\mathbf{q}}_{n_{\mathsf{mask}}-1}, \Delta)$

Fig. 10.5: VOLE reconstruction for the FAEST IOP.

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 $\Delta_0, \dots, \Delta_{\tau-1}$, and uses these, together with ConvertToVOLE, to compute the matrix $\mathbf{Q}$. This is an algorithm that can fail.

In comparison, VOLEIOPReconstruct simply takes $\mathbf{S}$ as input and extracts $\mathbf{Q}, \overline{\mathbf{q}}_m$. In addition, it also computes Δ from chall_3. Therefore, VOLEIOPReconstruct will succeed in computing $\mathbf{Q}$ as long as $g_1(\Delta) \neq \perp$.

Lemma 10.38. *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 an algorithm with access to random oracles $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_4$ which wins the SRS soundness game defined in Definition 10.34 for $\mathcal{R}$ against $\mathcal{V}_{\mathrm{IP}}$ with probability at least $p = \epsilon^{\mathrm{sr}}_{(\mathcal{A},\mathcal{V}_{\mathrm{IP}})}((s_i)_{i\in[4]}, t)$ and makes at most t moves. Let Q_C bound the number of distinct complete first messages submitted by $\mathcal{A}$, including the one in its final output. Suppose the composed execution of $\mathcal{A}$ and the complete final reconstruction 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 10.34 with probability at least

$$\mathsf{AdvSRSIOP}^{\mathcal{V}_{\mathrm{IOP}}}_{\tilde{\mathcal{P}}_{\mathrm{IOP}}} \geq p - \left(\frac{(Q_1+Q_C)^2}{2^{2\lambda}} + \frac{\tau(Q_0+Q_C)}{2^{\lambda}}\right)$$

against $\mathcal{V}_{\mathrm{IOP}}$. For FAEST-EM *, the success probability is*

$$\mathsf{AdvSRSIOP}^{\mathcal{V}_{\mathrm{IOP}}}_{\tilde{\mathcal{P}}_{\mathrm{IOP}}} \geq p - \left(\frac{(Q_1+Q_C)^2}{2^{2\lambda}} + Q_C \max_t \mathsf{AdvInj}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_t}[L]\right)$$

using the extractor from 10.19. Moreover, $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ makes $t+1$ moves.

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 10.38. 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 10.2. $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ 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 $\alpha_1 := (\mu, \mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}^{\mathsf{pre}})$, $\tilde{\mathcal{P}}_{\mathrm{IOP}}$ runs the extractor algorithm $\mathsf{Ext}(\mathcal{L}_0, \mathcal{L}_1, \mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}^{\mathsf{pre}}, \mu)$ from Lemma 10.19 to obtain $\mathbf{S}$. This is done only on the first occurrence of the complete message α_1. If $\mathbf{S} = \perp$, set $\mathbf{S} := \emptyset$, and define

$$g_1(\Delta) := \begin{cases} \mathbf{q}, & (\Delta, \mathbf{q}) \in \mathbf{S}, \\ \perp, & \text{otherwise.} \end{cases}$$

  Store the pair (α_1, g_1) and reuse the same g_1 whenever α_1 occurs again.of the same complete first message. It then sets $m_1 := (g_1)$, $\mathsf{st}_1 := \mathsf{st}'_1 \| \alpha_1$ and outputs $(\mathbb{x}, (m_1), (\mathsf{st}_1))$ to the IOP SRS experiment and returns to $\mathcal{A}$ what the experiment returns. The first salt size is increased by the fixed encoded length of α_1, which is permitted because state-restoration soundness is quantified over every salt-size vector.
- 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 $\mathsf{chall}_3 := \rho_3$, write $\alpha_4 = (\mathsf{decom}_I)$ and $\alpha_1 := (\mu, \mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}, \mathsf{iv}^{\mathsf{pre}})$, and let $\mathsf{iv} := \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre}} \| \mu)$ (computed using the simulated H_4). Additionally, compute

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

If $\widehat{\mathsf{rec}} \neq \bot$, parse $\widehat{\mathsf{rec}} = (\widehat{\mathsf{com}}, \widehat{\mathbf{Q}}, (\widehat{\mathbf{q}}_m)_{m \in [0..n_{\mathsf{mask}})}, \widehat{\Delta})$. If $\widehat{\mathsf{com}} = \mathsf{com}$ and

$$g_1(\widehat{\Delta}) \neq (\widehat{\mathbf{Q}}, \widehat{\mathbf{q}}_0, \ldots, \widehat{\mathbf{q}}_{n_{\mathsf{mask}}-1}),$$

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 α_1 part of st_1 used to generate the random function response ρ_1, which is necessary as an attacker might use different complete first messages 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.

On an accepted opening, the behaviour of $\mathcal{V}_{\mathrm{IP}}$ and $\mathcal{V}_{\mathrm{IOP}}$ differs only if the reconstructed response is not the value of the stored partial function g_1 at $\widehat{\Delta}$. For each distinct complete first message, the associated partial function g_1 was fixed when the extractor first ran, before its first challenge, and the same function is reused thereafter. If any accepted opening is inconsistent with the graph fixed for the corresponding first message, the reduction selects the corresponding indexed attempt in the multi-attempt game of Lemma 10.19; no additional union bound is needed. The probability of this event is at most $(Q_1 + Q_C)^2/2^{2\lambda} + \tau(Q_0 + Q_C)/2^\lambda$ for FAEST, and at most $(Q_1 + Q_C)^2/2^{2\lambda} + Q_C \max_t \mathsf{AdvInj}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_t}[L]$ for FAEST-EM. The reduction makes only one additional SRS move, to recover ρ_3. □

10.9.4 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.

Definition 10.39 (Round-by-round soundness, based on [CCH+18]). *For a IP Π for a language $\mathcal{L}$, let* RBRState *be a state function mapping an instance and a transcript prefix to a Boolean value, with the following properties. For* $\mathsf{inst} \notin \mathcal{L}$,

[23] https://github.com/hash-based-snargs-book/hash-based-snargs-book/blob/main/snargs-book.pdf

1. $\mathsf{RBRState}(\mathsf{inst}, \emptyset) = 0$, *where* $\emptyset$ *denotes the empty transcript prefix.*
2. *If a transcript prefix* trc *is empty or ends in a challenge and* $\mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}) = 0$, *then* $\mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}\|\mathsf{pmsg}) = 0$ *for every next prover message* pmsg.
3. *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.*
4. *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* $\mathsf{inst} \notin \mathcal{L}$ *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 10.40. *Let* $\mathsf{SIG} \in \{\mathsf{FAEST}, \mathsf{FAEST\text{-}EM}\}$ *and* $\mathcal{R}$ *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* SIG *signature scheme (with parameters* $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}, \mathtt{param}_{\mathsf{VOLE}}$ *which will be left implicit).*

Let $\ell + d_{\mathsf{zk}} - 1 \leq 2^{12}$, $C + 1 \leq 2^{13}$, $\varepsilon_{\mathsf{v}} := 2^{-4\lambda}(1 + 2^{-4})$, $\varepsilon_{\mathsf{zk}} := 2^{-\lambda}(1 + 2^{-38})$ *and* $d_{\mathsf{zk}} \in \{7, 14\}$. *The* SIG *IOP with verifier* $\mathcal{V}_{\mathrm{IOP}}$ *has round-by-round soundness with soundness error* $d_{\mathsf{zk}}/2^{\lambda}$, *where* $\mathbb{x}$ *is not a valid instance of* $\mathcal{R}$.

Proof of Lemma 10.40. For the first challenge we use Corollary 10.27, for the second Lemma 4.7 and for the last Lemma 7.3.

The SIG IOP is a 4-round IOP; in the first round, $\mathcal{P}_{\mathrm{IOP}}$ sends

$$\alpha_1 = (\emptyset, g_1), \qquad g_1 \in (\mathbb{K}^{\ell + n_{\mathsf{mask}}} \cup \{\bot\})^{\mathcal{D}_\Delta}.$$

From this message, $\mathsf{RBRState}$ obtains the graph

$$\mathbf{S}_{g_1} := \{(\Delta, g_1(\Delta)) : \Delta \in \mathcal{D}_\Delta,\ g_1(\Delta) \neq \bot\},$$

which is fixed before chall_1 and satisfies $|\mathbf{S}_{g_1}| \leq |\mathcal{D}_\Delta| = 2^{\lambda - w_{\mathsf{grind}}}$. The first message only fixes this graph and does not itself change $\mathsf{RBRState}$.

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. Let $\mathbf{H}_{\mathsf{chall}_1}$ be the linear map implemented by the exact VOLEHash input split in $\mathcal{V}_{\mathrm{IOP}}$, and let $\mathsf{Bad}_{\mathsf{KOS}}$ be the remaining KOS bad event of Corollary 10.27 for $\mathbf{S}_{g_1}$ and $\mathbf{H}_{\mathsf{chall}_1}$. Set $\mathsf{RBRState}$ to 1 after chall_1 exactly when $\mathsf{Bad}_{\mathsf{KOS}}$ occurs, and to 0 otherwise.

Claim 1. For every fixed response graph $\mathbf{S}_{g_1}$, the probability that $\mathsf{Bad}_{\mathsf{KOS}}$ occurs for an independently sampled VOLEHash challenge chall_1 is at most

$$\varepsilon_{\mathsf{v}}\binom{|\mathbf{S}_{g_1}|}{3} \leq \varepsilon_{\mathsf{v}}\binom{2^{\lambda - w_{\mathsf{grind}}}}{3}.$$

Proof. The proof of Corollary 10.27 applies verbatim to $\mathbf{S}_{g_1}$. □

Upon receiving chall_1, the prover responds with $m_2 = (\widetilde{\mathbf{u}} \| \mathbf{D} \| \mathbf{d})$. Conditioning on $\mathsf{Bad}_{\mathsf{KOS}}$ not occurring and settting

$$(\mathbf{u}^\star, \mathbf{v}^\star) := \mathcal{E}_{\mathrm{KOS}, \mathbf{S}_{g_1}}(\mathbf{H}_{\mathsf{chall}_1}, \widetilde{\mathbf{u}}, \mathbf{D}),$$

every $(\Delta, \mathbf{q}) \in \mathbf{S}_{g_1}$ satisfying $\mathbf{H}_{\mathsf{chall}_1}\mathbf{q} = \mathbf{D} + \Delta\widetilde{\mathbf{u}}$ also satisfies $\mathbf{q} = \mathbf{v}^\star + \Delta\mathbf{u}^\star$. Let

$$\mathbf{u}^\star = (\mathbf{u}^\star, \bar{\mathbf{u}}_0^\star, \ldots, \bar{\mathbf{u}}_{n_{\mathsf{mask}}-1}^\star), \qquad \mathbf{v}^\star = (\mathbf{V}^\star, \bar{\mathbf{v}}_0^\star, \ldots, \bar{\mathbf{v}}_{n_{\mathsf{mask}}-1}^\star),$$

where $\mathbf{u}^\star, \mathbf{V}^\star \in \mathbb{K}^\ell$. Using $\mathbf{d}$, we can reconstruct the witness $\mathbf{w}^\star := \mathbf{d} + \mathbf{u}^\star$ and run ZK.OWFProve inside RBRState (i.e. apply the computation to the committed witness). For every $i \in [0..\ell)$, the corresponding formal verifier-side witness commitment is

$$\rho_i(X) := \mathbf{V}^\star[i] + \mathbf{w}^\star[i]X.$$

We then apply the verifier side VOLE operations for the constraints, enforcing that every witness coordinate is in $\{0, 1\}$, and for OWFConstraints to these commitment polynomials and write the resulting vector of polynomials, each of degree at most d_{zk}, as

$$\sum_{j=0}^{d_{\mathsf{zk}}-1} \mathbf{A}_j X^j + \mathbf{Z}X^{d_{\mathsf{zk}}}, \quad \mathbf{A}_j, \mathbf{Z} \in \mathbb{K}^C.$$

The coefficient of $X^{d_{\mathsf{zk}}}$ is the vector consisting of the constraints enforcing that every coordinate of $\mathbf{w}^\star$ lies in $\{0, 1\}$ and the constraints returned by OWFConstraints, all evaluated on $\mathbf{w}^\star$. So we have that $\mathbf{Z} \neq 0$, otherwise the former constraints imply $\mathbf{w}^\star \in \mathbb{F}_2^\ell$, and the remaining constraints show that its first λ bits form a restricted signing key whose public image is pk, contradicting $\mathbb{x} \notin \mathcal{L}_{\mathcal{R}}$.

For the first $d_{\mathsf{zk}} - 1$ extracted mask correlations, we define

$$p_0 := \bar{\mathbf{v}}_0^\star, \quad p_j := \bar{\mathbf{u}}_{j-1}^\star + \bar{\mathbf{v}}_j^\star \quad (1 \le j \le d_{\mathsf{zk}} - 2), \text{ and } p_{d_{\mathsf{zk}}-1} := \bar{\mathbf{u}}_{d_{\mathsf{zk}}-2}^\star.$$

Then every response on the effective line satisfies

$$\sum_{j=0}^{d_{\mathsf{zk}}-1} p_j \Delta^j = \sum_{j=0}^{d_{\mathsf{zk}}-2} (\bar{\mathbf{v}}_j^\star + \Delta\bar{\mathbf{u}}_j^\star)\Delta^j = \sum_{j=0}^{d_{\mathsf{zk}}-2} \bar{\mathbf{q}}_j \Delta^j.$$

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 RBRState from 0 to 1.

Claim 2 (ZK Hash check failure). Let $\zeta :=$ ZKHash$(\mathsf{chall}_2, (\mathbf{Z}\|0))$ and define the bad event that ZK Hash check fails as $\mathbf{Z} \neq 0$ but $\zeta = 0$. The probability that the ZK Hash check fails for an independently chosen hash chall_2 is at most $\varepsilon_{\mathsf{zk}}$.

Proof. The input $(\mathbf{Z}\|0)$ is nonzero and fixed before chall_2. The claim follows from Lemma 4.7 showing the $\varepsilon_{\mathsf{zk}}$-almost universality of ZKHash under $C + 1 \le 2^{13}$. □

When the state after chall_1 is 0, define the state after chall_2 to be 1 exactly when $\zeta = 0$.

In the third round, $\mathcal{P}_{\mathrm{IOP}}$ sends

$$\alpha_3 = (\tilde{a}_0 \| \ldots \| \tilde{a}_{d_{\mathsf{zk}}-1})$$

For every $j \in [0..d_{\mathsf{zk}})$, set $a_j :=$ ZKHash$(\mathsf{chall}_2, (\mathbf{A}_j \| p_j))$. By linearity of ZKHash and the mask identity above, the value $\tilde{q}$ computed by OWFVerify at every passing response on the effective line is

$$\tilde{q} = \sum_{j=0}^{d_{\mathsf{zk}}-1} a_j \Delta^j + \zeta\Delta^{d_{\mathsf{zk}}}.$$

If the state is still 0, then $\zeta \neq 0$. The third message fixes all coefficients before chall_3, and acceptance of the QuickSilver check requires

$$E(\Delta) := \sum_{j=0}^{d_{\mathsf{zk}}-1} (\tilde{a}_j - a_j)\Delta^j - \zeta\Delta^{d_{\mathsf{zk}}} = 0.$$

Thus $E(X)$ is a nonzero polynomial of degree exactly d_{zk}.

To this, $\mathcal{V}_{\mathrm{IOP}}$ responds with $\mathsf{chall}_3 = \rho_3$ and rejects if the w_{grind} trailing bits are not all zero. For a valid challenge, the injective CRT map sends its first $\lambda - w_{\mathsf{grind}}$ bits to a distinct element $\Delta \in \mathcal{D}_\Delta$. Since E has at most d_{zk} roots, at most d_{zk} of the 2^λ raw challenges make the verifier reach a root. Thus the probability that $\mathsf{RBRState}$ flips in this round is at most

$$\frac{d_{\mathsf{zk}}}{2^\lambda}.$$

The fourth prover message is ignored, and the fourth verifier challenge is a fixed dummy value so its RBR error is zero. Leave $\mathsf{RBRState}$ unchanged on prover messages and retain the value 1 after any bad event. If a complete transcript on a false instance is accepted while $\mathsf{RBRState} = 0$, then its response belongs to $\mathbf{S}_{g_1}$ and passes the KOS equation, so it lies on the extracted effective line. The ZK bad event did not occur, so $\zeta \neq 0$, and the final QuickSilver equality gives $E(\Delta) = 0$, which is exactly the third bad event. This contradiction proves that every complete transcript with state 0 is rejected.

Since $\varepsilon_{\mathsf{zk}} = 2^{-\lambda}(1 + 2^{-38}) < 2^{-\lambda+1}$, we have that $d_{\mathsf{zk}}/2^\lambda$ is always larger than the soundness error of the second round. Moreover, we have

$$\varepsilon_{\mathsf{v}}\binom{2^{\lambda - w_{\mathsf{grind}}}}{3} \leq \frac{1 + 2^{-4}}{6}\, 2^{-\lambda - 3w_{\mathsf{grind}}} < 2^{-\lambda} \leq \frac{d_{\mathsf{zk}}}{2^\lambda},$$

and therefore the soundness error of the first round is also smaller than the soundness error of the third round.

□

Finally, let us gather the chain from SRS security to RBR security in one corollary.

Corollary 10.41. *Consider the setting of Lemma 10.36 for adversary $\mathcal{A}$ against EUF-KO with random public key, and recall*

$$p = \Pr[\mathsf{ExpEUFRK}^{\mathsf{SIG}}_{\mathcal{A}}(\mathsf{pk}) = 1 \wedge \mathsf{pk} \notin \mathcal{L}_{\mathcal{R}} \mid \mathbf{x} \leftarrow \{0,1\}^{N_{\text{st-bits}}},\ \mathbf{y} \leftarrow \{0,1\}^{\beta N_{\text{st-bits}}},\ \mathsf{pk} := (\mathbf{x}, \mathbf{y})].$$

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 $\ell + d_{\mathsf{zk}} - 1 \leq 2^{12}$, $C + 1 \leq 2^{13}$ and $d_{\mathsf{zk}} \in \{7, 14\}$. Set

$$Q_{\mathsf{FS}} := Q_{2,0} + Q_{2,1} + Q_{2,2} + Q_{2,3} + 4, \quad Q_C := Q_{2,1} + 1,$$

and let $\widehat{Q}_0, \widehat{Q}_1$ include the complete reconstruction performed during final verification; when Q_0, Q_1 count only the adversary's queries, one may take $\widehat{Q}_0 := Q_0 + 1$ and $\widehat{Q}_1 := Q_1 + \tau + 1$. For FAEST, we have

$$p \leq \frac{Q_{\mathsf{FS}}^2}{2^{2\lambda}} + \frac{(\widehat{Q}_1 + Q_C)^2}{2^{2\lambda}} + \frac{\tau(\widehat{Q}_0 + Q_C)}{2^\lambda} + (Q_{2,1} + Q_{2,2} + Q_{2,3} + 8)\frac{d_{\mathsf{zk}}}{2^\lambda}.$$

For $\mathsf{FAEST\text{-}EM}$, we have instead

$$p \leq \frac{Q_{\mathsf{FS}}^2}{2^{2\lambda}} + \frac{(\widehat{Q}_1 + Q_C)^2}{2^{2\lambda}} + Q_C \max_t \mathsf{AdvInj}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_t}[L] + (Q_{2,1} + Q_{2,2} + Q_{2,3} + 8)\frac{d_{\mathsf{zk}}}{2^\lambda}$$

Proof. First, we apply Lemma 10.36 to turn $\mathcal{A}$ into an SRS adversary $\mathcal{A}_1$ with move budget $Q_{2,1}+Q_{2,2}+Q_{2,3}+3$ as an interactive proof (IP) system; success is reduced by at most $Q_{\mathsf{FS}}^2/2^{2\lambda}$. Outside the predecessor-collision event already accounted for in this loss, the simulator stores each commitment attempt under its first-round transcript. Every query to H_2^2 or H_2^3 that is forwarded to the SRS game therefore continues the unique commitment attempt created by a preceding query to H_2^1 and does not create a new first-message attempt. Hence the $Q_{2,1}$ queries to H_2^1 create at most $Q_{2,1}$ commitment attempts, and the final verification can create at most one additional attempt, so we can take $Q_C := Q_{2,1}+1$. Secondly, we apply Lemma 10.38 to turn the SRS IP adversary into an SRS IOP adversary $\mathcal{A}_2$ which requires one additional move in the SRS IOP game; success is reduced by at most $(\widehat{Q}_1+Q_C)^2/2^{2\lambda}+\tau(\widehat{Q}_0+Q_C)/2^{\lambda}$ for FAEST, and by at most $(\widehat{Q}_1+Q_C)^2/2^{2\lambda}+Q_C\max_t \mathsf{AdvInj}_{\mathcal{B}_t}^{\mathsf{PRG}_{2\lambda}}[L]$ for FAEST-EM. Finally, we apply [CY24, Theorem 31.2.1] to the SRS IOP adversary $\mathcal{A}_2$, which makes at most $Q_{2,1}+Q_{2,2}+Q_{2,3}+4$ moves. Only transcripts with $\mathsf{pk}\notin\mathcal{L}_{\mathcal{R}}$ contribute to the state-restoration soundness event. By Lemma 10.40 and the assumed bounds on $\ell+d_{\mathsf{zk}}-1$ and $C+1$, the RBR soundness error satisfies

$$\varepsilon_{\mathsf{IOP}}^{\mathsf{rbr}} \le \frac{d_{\mathsf{zk}}}{2^{\lambda}}.$$

Since the IOP has four rounds, [CY24, Theorem 31.2.1] bounds the success probability of $\mathcal{A}_2$ by

$$\big(Q_{2,1}+Q_{2,2}+Q_{2,3}+4+4\big)\frac{d_{\mathsf{zk}}}{2^{\lambda}} = \big(Q_{2,1}+Q_{2,2}+Q_{2,3}+8\big)\frac{d_{\mathsf{zk}}}{2^{\lambda}}.$$

This yields the claim. □

10.10 EUF-CMA

Theorem 10.42 (FAEST is EUF-CMA secure). *Let ρ be chosen uniformly at random on every signing invocation. Let $\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_2^0,\mathsf{H}_2^1,\mathsf{H}_2^2,\mathsf{H}_2^3,\mathsf{H}_4$ and H_5 be modeled as random oracles and H_3 be a PRF[24]. Then for any PPT adversary $\mathcal{A}$ which makes $Q_{\mathsf{sig}}\le 2^{64}$ queries to $\mathcal{O}_{\mathsf{sig}}$ and $Q_0,Q_1,Q_{2,0},Q_{2,1},Q_{2,2},Q_{2,3}$, Q_4 and Q_5 queries to $\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_2^0,\mathsf{H}_2^1,\mathsf{H}_2^2,\mathsf{H}_2^3,\mathsf{H}_4$ and H_5 respectively, there is a PPT adversary $\mathcal{A}'$ such that*

$$\begin{aligned}
\mathsf{AdvEUFCMA}_{\mathcal{A}}^{\mathsf{FAEST}}[Q_{\mathsf{sig}}] \le\ & 2\cdot \mathsf{AdvEUFKO}_{\mathcal{A}'}^{\mathsf{FAEST}} \\
&+ 2\cdot Q_{\mathsf{sig}}\cdot\Big(\mathsf{AdvPRF}^{\mathsf{H}_3}+\mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}[1]+\lceil\log(L)-1\rceil\cdot\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}[1]\Big) \\
&+ 2\cdot Q_{\mathsf{sig}}\cdot\Big(\mathsf{AdvPRP}_{\mathcal{B}_{2,3}}^{\mathsf{leaf}}\Big) \\
&+ 2\cdot Q_{\mathsf{sig}}\cdot\Big(\mathsf{AdvPRP}_{\mathcal{B}_{1,1}}^{\mathsf{PRG}_{\mathsf{root}}}[1]+\tau\cdot\Big(\lceil\log(L)-1\rceil\cdot\mathsf{AdvPRP}_{\mathcal{B}_1}^{\mathsf{PRG}_{2\lambda}}[1]+\mathsf{AdvPRP}_{\mathcal{B}_{2,3}}^{\mathsf{leaf}}\Big)\Big) \\
&+ 2\cdot\frac{Q_{\mathsf{sig}}(2Q_1+Q_{2,0}+Q_{2,1})+(2\tau+3)\binom{Q_{\mathsf{sig}}}{2}+\binom{Q_{2,0}}{2}}{2^{2\lambda}} \\
&+ 2\cdot Q_{\mathsf{sig}}\cdot\Big(\frac{Q_{\mathsf{sig}}+Q_{2,1}+Q_{2,2}}{2^{8\lambda+64}}+\frac{Q_{\mathsf{sig}}+Q_{2,2}+Q_{2,3}}{2^{3\lambda+64}}\Big) \\
&+ 2\cdot Q_{\mathsf{sig}}\,2^{-\lambda}+2\lambda\big(Q_{\mathsf{sig}}Q_5+2n_{\mathsf{mult}}Q_{\mathsf{sig}}^2\big)2^{-(128+\lambda)}+2\cdot\frac{8(Q_5+n_{\mathsf{mult}})+4}{2^{\lambda}}
\end{aligned}$$

where $\mathsf{AdvPRP}^{\mathsf{leaf}}$ (and, in the proof below, a_{leaf}) denote the variant-dependent leaf terms defined in Lemma 10.24, so the bound covers both FAEST and FAEST-EM.

Note that Theorem 10.42 only applies to randomized signing where ρ is chosen uniformly by the signer. After proving the statement, we will summarize how the proof has to be adjusted when setting $\rho=0^{\lambda}$ in Corollary 10.43.

[24] Specifically, $\mathsf{H}_3(\mathsf{sk}\|\mu\|\rho)=\mathsf{PRF}(\rho,\mathsf{sk}\|\mu)$ is a PRF keyed by ρ and applied to $\mathsf{sk}\|\mu$.

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:

$$\Pr[\mathsf{G}_1 = 1] = \mathsf{AdvEUFCMA}^{\mathsf{FAEST}}_{\mathcal{A}}[Q_{\mathsf{sig}}]. \tag{13}$$

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 $(r, \mathsf{iv}^{\mathsf{pre}})$ in every query to $\mathcal{O}_{\mathsf{sig}}$, we run a hybrid argument and obtain:

$$\Pr[\mathsf{G}_1] \le \Pr[\mathsf{G}_2] + Q_{\mathsf{sig}} \cdot \mathsf{AdvPRF}^{\mathsf{H}_3}. \tag{14}$$

G_3**:** $\mathcal{O}_{\mathsf{sig}}$ samples $\mathsf{chall}_1 \in \{0,1\}^{8\lambda+64}$ at random in every query. When $\mu\|\mathsf{com}\|\boldsymbol{c}_1\|\ldots\|\boldsymbol{c}_{\tau-1}\|\mathbf{c}_{\mathsf{mult}}\|\mathsf{iv}^{\mathsf{pre}}$ is queried to H_2^1, game G_3 aborts 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 10.25. 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 $\mu\|\mathsf{com}\|\boldsymbol{c}_1\|\ldots\|\boldsymbol{c}_{\tau-1}\|\mathbf{c}_{\mathsf{mult}}\|\mathsf{iv}^{\mathsf{pre}}$. If the i-th query repeats a pair (ρ,μ), the two adjacent hybrids are identical because both replay the response recorded in G_2. The i-th reduction $\mathcal{A}_{\mathsf{unpred},i}$ to unpredictability for a fresh pair plays unpredictability game in Lemma 10.25 such that it emulates $\mathsf{G}_{3,i}$, except that for the i-th $\mathcal{O}_{\mathsf{sig}}$-query, it outputs its current state, the IV label μ (the digest of the i-th signing query), and the list obtained from all previous correctly typed queries to H_2^1 whose first component is μ, after deleting that component from each query (where the attacker may have guessed com), receives the tuple C (containing $\mathsf{com}, \mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}, \mathbf{c}_{\mathsf{mult}}$, as well as decom, $\mathbf{u}$, $\mathbf{V}$ and the mask correlations needed to complete the signature) and $\mathsf{iv}^{\mathsf{pre}}$ from the unpredictability experiment (instead of running VOLECommit), and then continues from there. For such a fresh pair, by construction, $\mathsf{G}_{3,i-1} = 1 \iff \mathsf{ExpUnpred}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}}_{0,\mathcal{A}_{\mathsf{unpred},i}} = 1$ and $\mathsf{G}_{3,i} = 1 \iff \mathsf{ExpUnpred}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}}_{1,\mathcal{A}_{\mathsf{unpred},i}} = 1$. Before the i-th signing query, this reduction has made at most $Q_1 + (i-1)(\tau+1)$ queries to H_1, and the displayed list has size at most $Q_{2,1} + i - 1$. Thus, from Lemma 10.25 and a hybrid argument we get

$$\begin{aligned}\Pr[\mathsf{G}_2 = 1] \le \Pr[\mathsf{G}_3 = 1] + e^{(\beta^2_{\mathsf{root}}+4\lceil\log(L)-1\rceil\beta^2+a^2_{\mathsf{leaf}})/2^{128}} \cdot \Big(Q_{\mathsf{sig}} \cdot \Big(\mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}_{\mathcal{B}_{1.1}}[1] + \\ \lceil\log(L)-1\rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1] + \mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}}\Big) \\ + \frac{Q_{\mathsf{sig}}(2Q_1 + Q_{2,1}) + (2\tau+3)\binom{Q_{\mathsf{sig}}}{2}}{2^{2\lambda}}\Big),\end{aligned} \tag{15}$$

where we substituted the instance-dependent counts $Q_1 + (i-1)(\tau+1)$ and $Q_{2,1} + i - 1$ into the statistical term of Lemma 10.25 and used $\sum_{i=1}^{Q_{\mathsf{sig}}}(i-1) = \binom{Q_{\mathsf{sig}}}{2}$; since the bound of Lemma 10.25 is additive, the exponential factor does not compound over the hybrid steps.

G_4: $\mathcal{O}_{\mathsf{sig}}$ samples $\mathsf{chall}_2 \in \{0,1\}^{3\lambda+64}$ at random in every query. When $\mathsf{chall}_1 \| \tilde{\mathbf{u}} \| \widetilde{\mathbf{V}} \| \mathbf{d}$ 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:

$$\Pr[\mathsf{G}_3 = 1] \leq \Pr[\mathsf{G}_4 = 1] + Q_{\mathsf{sig}} \cdot \frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{8\lambda+64}}.$$

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, for at most 2^{32} iterations, until the loop ends with $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}}) = 0^{w_{\mathsf{grind}}}$ and $I = \mathsf{DecodeAllChall}_3(\mathsf{chall}_3; \mathtt{param})$ such that $\mathsf{decom}_I \neq \bot$; if no such iteration exists, it returns $\bot$, exactly as FAEST.Sign. When $\mathsf{chall}_2 \| (\tilde{a}_j)_{j \in [0..d_{\mathsf{zk}})} \| \mathsf{ctr}$ 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 10.24.

G_6: $\mathcal{O}_{\mathsf{sig}}$ replaces the call to VOLECommit by uniformly sampling $\mathbf{u}, (\boldsymbol{c}_i)_{i \in [1..\tau)}, \mathbf{c}_{\mathsf{mult}}$ as well as $(\overline{\mathbf{u}}_m)_{m \in [0..n_{\mathsf{mask}})}$, programming H_5 as described in Lemma 10.24 and computing $\mathbf{V}, \overline{\mathbf{v}}_m$ such that the VOLE relation holds with the Δ and $\mathbf{q}_i, \mathbf{q}_m$ values that result from chall_3.
In the reduction, we sample chall_3 as in game G_5, except that we replace the successful challenge chall_3 (where $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}}) = 0^{w_{\mathsf{grind}}}$ 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 $b = 1$. From the opening returned by the hiding oracle, the reduction runs VOLEReconstruct to obtain $\mathbf{Q}$ and $(\bar{\mathbf{q}}_m)_m$, and then sets $\mathbf{V} = \mathbf{Q} + \Delta\mathbf{u}$ and $\bar{\mathbf{v}}_m = \bar{\mathbf{q}}_m + \Delta\bar{\mathbf{u}}_m$. To see this, note that $\mathsf{decom}_I \neq \bot$ is derivable given only I (Remark 10.23), 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 $\mathsf{chall}_3[\lambda - w_{\mathsf{grind}}) = 0^{w_{\mathsf{grind}}}$ 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. The reduction invokes the hiding oracle on each fresh pair (ρ, μ^j) and supplies the current message digest μ^j, while a repeated pair is answered with the response recorded in G_2; if the bounded loop is exhausted, the hiding-oracle response is discarded and the signing query returns $\bot$. Hence there are at most Q_{sig} hiding-oracle invocations, their adaptively chosen labels are covered by the experiment, and $\mathsf{iv}^j = \mathsf{H}_4(\mathsf{iv}^{\mathsf{pre},j} \| \mu^j)$ matches the scheme. The H_5 queries made when the final forgery is verified occur after all hiding-oracle invocations and therefore do not enter the Q_5 programming-failure bound.
By Lemma 10.24, a hybrid argument, and applying $K_{\max} = \lambda$, the change in success is therefore bounded as

$$\begin{aligned}\Pr[\mathsf{G}_5 = 1] \leq e^{Q_{\mathsf{sig}} \cdot (\beta_{\mathsf{root}}^2 + 4\tau\lceil \log(L) - 1 \rceil \beta^2 + \tau a_{\mathsf{leaf}}^2)/2^{128}} \cdot \Big(\Pr[\mathsf{G}_6 = 1] + \\ Q_{\mathsf{sig}} \mathsf{AdvPRP}_{\mathcal{B}_{1,1}}^{\mathsf{PRG}_{\mathsf{root}}}[1] + Q_{\mathsf{sig}} \cdot \tau \cdot \Big(\lceil \log(L) - 1 \rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}_1}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}_{\mathcal{B}_{2,3}}^{\mathsf{leaf}} \Big) \\ + 2^{-\lambda} + Q_{\mathsf{sig}} 2^{-\lambda} + \lambda \big(Q_{\mathsf{sig}} Q_5 + 2 n_{\mathsf{mult}} Q_{\mathsf{sig}}^2 \big) 2^{-(128+\lambda)} \Big)\end{aligned}$$

After this change, the distribution of $\mathbf{u}, \mathbf{V}, (\overline{\mathbf{u}}_m, \overline{\mathbf{v}}_m)_{m \in [0..n_{\mathsf{mask}})}$ is uniform (conditioned on fulfilling the VOLE relation) and independent of the VOLE commitments $(\boldsymbol{c}_i)_{i \in [1..\tau)}$ as well as $\mathbf{c}_{\mathsf{mult}}$.

G_7**:** In every query, $\mathcal{O}_{\mathsf{sig}}$ samples $\tilde{\boldsymbol{u}}$ and $\tilde{\mathbf{V}}$ at random (while being consistent with the VOLE relation defined by $\tilde{\mathbf{Q}}$ and Δ) instead of computing VOLEHash during the signing. It then adjusts $(\overline{\mathbf{u}}_m, \overline{\mathbf{v}}_m)_{m\in[d_{\mathsf{zk}}-1..d_{\mathsf{zk}}+2]}$ to match $\tilde{\boldsymbol{u}}$ and $\tilde{\mathbf{V}}$ when VOLEHash is applied under challenge chall_1 to $\mathbf{u}_0^h, \mathbf{V}_0^h$.
Since $(\overline{\mathbf{u}}_m, \overline{\mathbf{v}}_m)_{m\in[d_{\mathsf{zk}}-1..d_{\mathsf{zk}}+2]}$ are uniformly random (conditioned on fulfilling the VOLE relation) because of G_6, and VOLEHash is $\mathbb{F}_{2^\lambda}^{\ell+d_{\mathsf{zk}}-1}$-hiding by Lemma 4.6, this game is perfectly indistinguishable from the previous one.

G_8**:** In every query, $\mathcal{O}_{\mathsf{sig}}$ samples $\tilde{a}_0, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}$ uniformly at random (while being consistent with the VOLE relation) instead of computing ZKHash. It then computes OWFProve, but adjusts $(\overline{\mathbf{u}}_m, \overline{\mathbf{v}}_m)_{m\in[0..d_{\mathsf{zk}}-1)}$ to match $\tilde{a}_0, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}$ when ZKHash is applied under challenge chall_2.
Since the respective elements $(\overline{\mathbf{u}}_m, \overline{\mathbf{v}}_m)_{m\in[0..d_{\mathsf{zk}}-1)}$ are uniformly random (conditioned on fulfilling VOLE relation) because of G_6, and ZKHash is $\mathbb{F}_{2^\lambda}^{\ell}$-hiding by Lemma 4.7, the distribution of $(\tilde{a}_0, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1})$ is unchanged and this game is perfectly indistinguishable from the previous one.

G_9**:** Now that the computation of $\tilde{a}_0, \ldots, \tilde{a}_{d_{\mathsf{zk}}-1}$ is independent of $\mathbf{w}$, $\mathcal{O}_{\mathsf{sig}}$ will sample $\mathbf{d}$ uniformly at random in every query instead of computing $\mathbf{d} := \mathbf{w} + \mathbf{u}$.
This does not change the distribution of $\mathbf{d}$ because $\mathbf{u}$ is uniform since G_6, 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 $\mathcal{A}$. However, we cannot yet reduce to EUF-KO as we must show that none of the points programmed into the RO by the simulator has aided the attacker in creating a forgery. For H_2^0 to be useful in a forgery we know that, as we only want to achieve weak EUF-CMA security, the input has to be different while still generating the same μ as msg must differ. Thus, any aid on a programmed point is due to a successful multi-instance second preimage attack on the outputs of H_2^0 generated during signature simulation. Assuming that the attack was not successful on H_2^0, an attacker might succeed with the same strategy on H_2^1 or H_2^2 with the inputs of the other hash functions being different. Lastly, a query to H_5 (either by the adversary, or during verification of the forgery) may hit a point programmed by the simulation, which we must also exclude. To address this, we add additional hybrids and aborts whenever the forgery uses RO queries that depend on RO points programmed by the simulation.

G_{10}**:** This game aborts if H_4 is not $K_{\max}$-preimage bounded. By Corollary 10.18, $\Pr[\mathsf{G}_9 = 1] \leq \Pr[\mathsf{G}_{10} = 1] + 2^{-\lambda}$. (Since this condition is not efficiently checkable, we will drop it before the EUF-KO game hop, to avoid an inefficient reduction.)

G_{11}**:** Let $\mathcal{L}_0$ be the list of all queries made to H_2^0. Equivalently, let $\mathcal{L}_1$ be the list of all queries made to H_2^1 and $\mathcal{L}_2$ be the list of all queries made to H_2^2. Lastly, let $\mathcal{L}_5$ be the list of query-response pairs $((\mathsf{iv}, e, L_e), y)$ to H_5. Note that these lists contain all queries, both added by the reduction and made by $\mathcal{A}$. In this hybrid, we abort whenever
- there is a collision $(x_0, y), (x_1, y) \in \mathcal{L}_0$ with $x_0 \neq x_1$ where either exactly one query was made by $\mathcal{O}_{\mathsf{sig}}$ or both queries were made by $\mathcal{A}$, or any such collision in $\mathcal{L}_1$ or $\mathcal{L}_2$ where one query was programmed by the reduction and the other generated by $\mathcal{A}$ or by the challenger; or
- any query $(\mathsf{iv}, e, L_e) \in \mathcal{L}_5$, made either by $\mathcal{A}$ or by the challenger when verifying the forgery, hits a point that was programmed by the reduction.

Towards bounding the abort probability, we first bound the probability that an abort occurs for H_2^0, H_2^1, or H_2^2 by

$$\frac{Q_{\mathsf{sig}} \cdot (Q_{2,0} + 1) + \binom{Q_{2,0}}{2}}{2^{2\lambda}} + \frac{Q_{\mathsf{sig}} \cdot (Q_{2,1} + 1)}{2^{8\lambda+64}} + \frac{Q_{\mathsf{sig}} \cdot (Q_{2,2} + 1)}{2^{3\lambda+64}}$$
$$< \frac{Q_{\mathsf{sig}} Q_{2,0} + \binom{Q_{2,0}}{2}}{2^{2\lambda}} + Q_{\mathsf{sig}} \cdot \left(\frac{Q_{2,1}}{2^{8\lambda+64}} + \frac{Q_{2,2}}{2^{3\lambda+64}} \right) + 2^{-\lambda}.$$

All terms trivially follow using standard bounds on multi-instance second preimage resistance, with the additional $\binom{Q_{2,0}}{2}$ term accounting for collisions between two adversarial H_2^0 queries, and the additional $+1$ accounting for the challenger's queries needed to verify the forgery, and the inequality using $Q_{\mathsf{sig}} \leq 2^{64}$. Next, we bound the probability that an abort occurs due to H_5 by

$$\frac{8(Q_5 + n_{\mathsf{mult}}) + 1}{2^\lambda}.$$

To achieve this bound without a Q_{sig} factor (which is particularly important here, since this is the only term with denominator 2^λ), we bound the *IV multiplicity* $M_{\max}$: the maximum over $v \in \{0,1\}^{128}$ of the number of distinct stored signing responses, generated for fresh pairs (ρ, μ), with $\mathsf{iv} = v$.
From G_{10}, H_4 is $K_{\max}$-preimage bounded, so every $x \mapsto \mathsf{H}_4(x\|\mu)$ is at most $K_{\max}$-to-one, for all $\mu \in \{0,1\}^{2\lambda}$. Since G_2, each fresh pair (ρ, μ^j) draws $\mathsf{iv}^{\mathsf{pre},j} \leftarrow \{0,1\}^{128}$ independently of H_4 and of the preceding transcript, so Lemma 10.17 applied to $F = \mathsf{H}_4(\cdot\|\mu^j)$ gives, for every $v \in \{0,1\}^{128}$ that is determined before the j-th response,

$$\Pr[\mathsf{iv}^j = v \mid \text{transcript before response } j] \leq K_{\max} 2^{-128}.$$

Now set $K := \lambda/56+3$ and order the stored responses by time. If $M_{\max} \geq K$, then some K of them, say $j_1 < \cdots < j_K$, share an IV; putting $v := \mathsf{iv}^{j_1}$, which is determined before response j_2, and chaining the above bound over $j_2, \ldots, j_K$ yields, using $K_{\max} = \lambda \leq 2^8$ and $Q_{\mathsf{sig}} \leq 2^{64}$,

$$\Pr[M_{\max} \geq K] \leq \binom{Q_{\mathsf{sig}}}{K} \left(\frac{K_{\max}}{2^{128}}\right)^{K-1} \leq 2^{64K} \cdot 2^{(8-128)(K-1)} = 2^{120-56K} \leq 2^{-\lambda}.$$

The event $M_{\max} \geq K$ and the preimage check from G_{10} depend only on H_4, the $\mathsf{iv}^{\mathsf{pre}}$ values, and $\mathcal{A}$'s messages, so both are independent of the offsets $\mathbf{u}^j[0..\lambda)$ that hide the programmed H_5 positions $(\mathsf{iv}, e, L_e^{\gamma_e \oplus 1})$.
We now bound the probability that any of the at most $Q_5 + n_{\mathsf{mult}}$ H_5 queries – by $\mathcal{A}$ or by the challenger in Verify – hits a programmed point, considering the first such query. After games G_6–G_9, fix all randomness except one offset $\mathbf{u}^j[0..\lambda)$. Conditioned on the preceding programming operations succeeding and on no earlier query hitting a programmed point, every condition involving this offset excludes at most one of its values. The equal-label condition excludes one value, the adversarial and final-verification queries exclude at most $Q_5 + n_{\mathsf{mult}}$ values, and the internal queries and programming operations of the other signatures exclude at most $2n_{\mathsf{mult}}(Q_{\mathsf{sig}}-1)$ values. Consequently, every offset has conditional point probability at most

$$\left(2^\lambda - 1 - Q_5 - n_{\mathsf{mult}} - 2n_{\mathsf{mult}}(Q_{\mathsf{sig}} - 1)\right)^{-1}.$$

A query (iv, e, x) can then match at most one programmed point per distinct stored signing response with the same (iv, e), of which there are up to $M_{\max} \leq K-1$. A union bound over the queries and the event $M_{\max} \geq K$ gives

$$\begin{aligned}
\Pr[\text{abort}_5] \leq & (Q_5 + n_{\mathsf{mult}}) \cdot \frac{K-1}{2^\lambda - 1 - Q_5 - n_{\mathsf{mult}} - 2n_{\mathsf{mult}}(Q_{\mathsf{sig}} - 1)} + 2^{-\lambda} \\
& \leq \frac{6(Q_5 + n_{\mathsf{mult}})}{2^\lambda - 1 - Q_5 - n_{\mathsf{mult}} - 2n_{\mathsf{mult}}(Q_{\mathsf{sig}} - 1)} + 2^{-\lambda} \\
& \leq \frac{8(Q_5 + n_{\mathsf{mult}}) + 1}{2^\lambda}
\end{aligned}$$

where the last inequality relies on $1 + Q_5 + n_{\mathsf{mult}} + 2n_{\mathsf{mult}}(Q_{\mathsf{sig}} - 1) \leq 2^{\lambda-2}$, which holds for the proposed parameters and $Q_{\mathsf{sig}} \leq 2^{64}$ whenever the final bound is nontrivial (for larger Q_5 the final bound holds trivially).

G_{11} does not contain any dependency on H_2^3. To see why this is the case, notice that the input to H_2^3 contains chall_2 which itself depends on chall_1 and μ. By assumption, for a successful forgery, it must be that neither μ nor chall_1 or chall_2 in the forgery are outputs programmed by the simulator (otherwise this was already captured in the above hybrid). Thus, the forgery must contain an input chall_2 to H_2^3 that was not queried before. Hence, none of the Q_{sig} programmed points in H_2^3 can be used to create a valid forgery.

Taking together the intermediate advantages, we arrive at

$$\begin{aligned}
\Pr[\mathsf{G}_1 = 1] \le{}& \rho_{\mathsf{CMA}} \Pr[\mathsf{G}_{11} = 1] \\
&+ \rho_{\mathsf{CMA}} Q_{\mathsf{sig}} \Big(\mathsf{AdvPRF}^{\mathsf{H}_3} + \mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}[1] \\
&\qquad + \lceil \log(L) - 1 \rceil \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}} \Big) \\
&+ \rho_{\mathsf{CMA}} Q_{\mathsf{sig}} \Big(\mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}_{\mathcal{B}_{1,1}}[1] \\
&\qquad + \tau \Big(\lceil \log(L) - 1 \rceil \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1] + \mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}} \Big) \Big) \\
&+ \frac{\rho_{\mathsf{CMA}}}{2^{2\lambda}} \Big(Q_{\mathsf{sig}}(2Q_1 + Q_{2,0} + Q_{2,1}) + (2\tau + 3) \binom{Q_{\mathsf{sig}}}{2} + \binom{Q_{2,0}}{2} \Big) \\
&+ \rho_{\mathsf{CMA}} Q_{\mathsf{sig}} \left(\frac{Q_{\mathsf{sig}} + Q_{2,1} + Q_{2,2}}{2^{8\lambda+64}} + \frac{Q_{\mathsf{sig}} + Q_{2,2} + Q_{2,3}}{2^{3\lambda+64}} \right) \\
&+ \rho_{\mathsf{CMA}} \Bigg(Q_{\mathsf{sig}} 2^{-\lambda} + \lambda \Big(Q_{\mathsf{sig}} Q_5 + 2 n_{\mathsf{mult}} Q_{\mathsf{sig}}^2 \Big) 2^{-(128+\lambda)} \\
&\qquad + \frac{8(Q_5 + n_{\mathsf{mult}}) + 4}{2^{\lambda}} \Bigg).
\end{aligned}$$

where

$$\rho_{\mathsf{CMA}} = e^{Q_{\mathsf{sig}} \left(2\beta_{\mathsf{root}}^2 + 4(1+\tau)\lceil \log(L) - 1 \rceil \beta^2 + (1+\tau) a_{\mathsf{leaf}}^2 \right) / 2^{128}}$$

is bounded by 2 for all proposed parameter sets, covering both FAEST and FAEST-EM via the variant-dependent a_{leaf} and $\mathsf{AdvPRP}^{\mathsf{leaf}}$ from Lemma 10.24. Note that the final $+4$ constant in the overall bound accounts for four remaining $2^{-\lambda}$ terms, one from the G_6 bound, one from the G_{10} abort, one to absorb the challenger's verification queries in G_{11}, and the event $M_{\max} \ge K$ in G_{11}.

At this point, we reduce to EUF-KO security. Formally, game G_{11} still programs the random oracle, and thus does not fit into the EUF-KO experiment (which itself provides access to the random oracles and verifies EUF-KO w.r.t. its own oracles). We explain how to bridge this formal gap. After the changes in games G_1 through G_{11}, the reduction in G_{11} always simulates signing oracle queries and aborts if the adversary guessed a programmed query beforehand or finds a collision, or if any query to H_5 hits a programmed point. Thus, we can define an alternative adversary $\mathcal{A}'$ playing game a modified game G'_{11} as follows. By definition, $\mathcal{A}'$ never queries $\mathcal{O}_{\mathsf{sig}}$ and instead runs signature simulation and programming of the random oracle queries "in its head" (including the abort conditions, which are verifiable from the queries alone, hence by $\mathcal{A}'$). By construction, $\mathcal{A}'$ never makes queries to real programmed random oracle inputs. As a consequence, G'_{11} does not (need to) program its random oracles nor does it need to answer any signature queries. Apart from the preimage-boundedness check on H_4 in G_{10}, the probability that G_{11} with $\mathcal{A}$ outputs 1 is identical to the probability that G'_{11} with $\mathcal{A}'$ outputs 1. Since that check need not be efficient, $\mathcal{A}'$ simply omits it, which can only increase its success probability. Thus, from G'_{11} a direct reduction to EUF-KO security is possible at this point, and we have $\Pr[\mathsf{G}_{11} = 1] \le \Pr[\mathsf{G}'_{11} = 1] \le \mathsf{AdvEUFKO}^{\mathsf{FAEST}}_{\mathcal{A}'}$. Before the final verification, the EUF-KO

adversary $\mathcal{A}'$ makes at most $Q_0 + Q_{\mathsf{sig}}$ queries to H_0 for FAEST (and Q_0 for FAEST-EM), $Q_1 + (\tau+1)Q_{\mathsf{sig}}$ queries to H_1, $Q_{2,0} + Q_{\mathsf{sig}}$ queries to H_2^0, $Q_{2,i}$ queries to H_2^i for $i \in [1..4)$, and $Q_4 + Q_{\mathsf{sig}}$ queries to H_4. It also performs at most $Q_5 + n_{\mathsf{mult}}Q_{\mathsf{sig}}$ evaluations of the fixed public function H_5.

□

Now, we consider the deterministic version of FAEST and prove how its reduction from EUF-CMA to EUF-KO works. The proof of Theorem 10.42 breaks down in G_2 where we reduce to the pseudorandomness of H_3 based on the uniformity of ρ. This does not work if ρ is public and fixed. To salvage the proof, we rely on the randomness of sk and a stronger assumption on H_3, namely, that sk can serve as a PRF secret even when pk is known to the distinguisher. For simplicity, we model H_3 as a random oracle.[25] Observe that until $\mathcal{A}$ queries H_3 on input $\mathsf{sk}\|\mu\|0^\lambda$, programming $\mathsf{H}_3(\mathsf{sk}\|\mu\|0^\lambda)$ to provide a uniformly random output as done in Game G_2 still works. We argue that we can w.l.o.g. assume that $\mathcal{A}$ does not query H_3 at such a point. For this, we replace $\mathcal{A}$ with an adversary $\mathcal{A}'$ which acts as follows: It internally runs $\mathcal{A}$. However, for each query $\widehat{\mathsf{sk}}\|\mu\|0^\lambda$ to H_3 it checks if $\widehat{\mathsf{sk}}$ is a valid secret key for pk, i.e. if $\mathbf{y} = \mathsf{OWF}_{\mathbf{x}}(\widehat{\mathsf{sk}})$ and $\widehat{\mathsf{sk}}[0]\widehat{\mathsf{sk}}[1] = 0$ where $\mathsf{pk} := (\mathbf{x}, \mathbf{y})$. For any query $\widehat{\mathsf{sk}}\|\mu\|0^\lambda$ where $\widehat{\mathsf{sk}}$ is not a valid secret key for pk, $\mathcal{A}'$ implements H_3 honestly (i.e., returns randomness). However, if the first such secret key is queried by $\mathcal{A}$ to H_3, then $\mathcal{A}'$ uses $\widehat{\mathsf{sk}}$ to sign a random message (that was not signed by $\mathcal{O}_{\mathsf{sig}}$ previously) and outputs it as a forgery to the security game.

Computationally, the overhead of $\mathcal{A}'$ is small, i.e. $t_{\mathcal{A}'} \approx t_{\mathcal{A}}$ as it only additionally runs FAEST.Sign, and it only requires additional queries to the random oracles H_0, H_1, $\mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$, H_4, and H_5 proportional to one execution of FAEST.Sign. Moreover, the input distribution to $\mathcal{A}$ does not change. If $\mathcal{A}$ queries H_3 using a valid secret key, then $\mathcal{A}'$ uses that key to sign a fresh message and obtains a forgery whenever signing succeeds. On all other executions relevant to the bound, $\mathcal{A}$ makes no valid-secret-key query to H_3, and we can apply the same hybrid steps as in the proof of Theorem 10.42.

Corollary 10.43 (Deterministic FAEST is EUF-CMA secure). *Let $\rho := 0^\lambda$. Let $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_3, \mathsf{H}_4$ and H_5 be modeled as random oracles. Then for any PPT adversary $\mathcal{A}$ which makes $Q_{\mathsf{sig}} \le 2^{64}$ queries to $\mathcal{O}_{\mathsf{sig}}$ and $Q_0, Q_1, Q_{2,0}, Q_{2,1}, Q_{2,2}, Q_{2,3}, Q_3, Q_4$ and Q_5 queries to $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_3, \mathsf{H}_4$ and H_5 respectively, there is a PPT adversary $\mathcal{A}'$ with almost the same running time and a PPT EUF-KO adversary $\mathcal{A}''$, whose query bounds, after the one additional honest signing attempt, are $Q_0' = Q_0 + 1$ for FAEST and $Q_0' = Q_0$ for FAEST-EM, $Q_1' = Q_1 + \tau + 1$, $Q_{2,0}' = Q_{2,0} + 1$, $Q_{2,1}' = Q_{2,1} + 1$, $Q_{2,2}' = Q_{2,2} + 1$, $Q_{2,3}' = Q_{2,3} + 2^{32}$, $Q_3' = Q_3 + 1$, $Q_4' = Q_4 + 1$, and $Q_5' = Q_5 + 2n_{\mathsf{mult}}$, such that*

$$
\begin{aligned}
\mathsf{AdvEUFCMA}^{\mathsf{FAEST}}_{\mathcal{A}}[Q_{\mathsf{sig}}] \le\ & 2 \cdot \mathsf{AdvEUFKO}^{\mathsf{FAEST}}_{\mathcal{A}''} \\
& + 2 \cdot Q_{\mathsf{sig}} \cdot \Big(\mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}[1] + \lceil \log(L) - 1 \rceil \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}} \Big) \\
& + 2 \cdot Q_{\mathsf{sig}} \cdot \Big(\mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}_{\mathcal{B}_{1,1}}[1] + \tau \cdot \Big(\lceil \log(L) - 1 \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1] + \mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}} \Big) \Big) \\
& + 2 \cdot \frac{Q_{\mathsf{sig}}(2Q_1' + Q_{2,0}' + Q_{2,1}') + (2\tau + 3)\binom{Q_{\mathsf{sig}}}{2} + \binom{Q_{2,0}'}{2}}{2^{2\lambda}} \\
& + 2 \cdot Q_{\mathsf{sig}} \Big(\frac{Q_{\mathsf{sig}} + Q_{2,1}' + Q_{2,2}'}{2^{8\lambda+64}} + \frac{Q_{\mathsf{sig}} + Q_{2,2}' + Q_{2,3}'}{2^{3\lambda+64}} \Big) \\
& + 2Q_{\mathsf{sig}} 2^{-\lambda} + 2\lambda \big(Q_{\mathsf{sig}} Q_5' + 2n_{\mathsf{mult}} Q_{\mathsf{sig}}^2 \big) 2^{-(128+\lambda)} + 2 \cdot \frac{8(Q_5' + n_{\mathsf{mult}}) + 4}{2^\lambda} \\
& + \varepsilon_{\mathsf{sign}\text{-}\perp}(\mathcal{A})
\end{aligned}
$$

[25] It is possible to rely on a standard-model assumption with a slightly more complex proof. But we already use random oracles in several places.

where $\mathsf{AdvPRP}^{\mathsf{leaf}}$ *denotes the variant-dependent leaf advantage defined in Lemma 10.24, so the bound covers both* FAEST *and* FAEST-EM*, and* $\varepsilon_{\mathsf{sign}\text{-}\perp}(\mathcal{A})$ *is the signing completeness error from producing a forgery if* $\mathcal{A}$ *queries the secret key. The EUF-KO adversary* $\mathcal{A}''$ *additionally makes the signing-simulation queries listed at the end of the proof of Theorem 10.42, with each unprimed count there replaced by its primed counterpart here.*

10.11 QROM proof (NEW)

Here, we recall a minimal amount of notation relating to the compressed oracle technique [Zha19, CFHL21]. 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 $(\mathcal{Y} \cup \{\perp\})^{\mathcal{X}}$ has q_{P}-local witnesses if for all $D \in (\mathcal{Y} \cup \{\perp\})^{\mathcal{X}}$ 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 $D' \in (\mathcal{Y} \cup \{\perp\})^{\mathcal{X}}$ 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 $|D| = |\{x \in \mathcal{X} | D(x) \neq \perp\}|$.

We recall a lemma for QROM query bounds from [AHJ+23].

Lemma 10.44 (Special case of Lemma 1 in [AHJ+23]). *Let* $\mathsf{H} : \mathcal{X} \to \mathcal{Y}$ *be a random oracle and let* $\mathsf{P} : (\mathcal{Y} \cup \{\perp\})^{\mathcal{X}} \to \{0, 1\}$ *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*

$$\sqrt{\Pr_{z \leftarrow \mathcal{A}^{\mathsf{H}}}[\langle z, \mathsf{P}, \mathsf{H}\rangle]} \leq \sum_{k=1}^{q+q_{\mathsf{P}}} \sqrt{10 \max_{\substack{x, D: \\ |D| \leq k, \neg \mathsf{P}(D)}} \Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}(D[x \mapsto u])]}. \tag{16}$$

Here the maximum is taken over $x \in \mathcal{X}$ *and* $D \in (\mathcal{Y} \cup \{\perp\})^{\mathcal{X}}$*.*

We now present the techniques that facilitate a proof of tight EUF-CMA-security of VOLEitH-based Fiat-Shamir signatures. We will organize the proof along 3 challenges that arise in this security proof, and present the tools we develop to solve them. We describe general versions of the tools in Section 10.11.1 to 10.11.3. Lemmas tailored to FAEST and the its full QROM security proof for FAEST can be found in Section 10.11.4 to 10.11.12.

10.11.1 Challenge 1: No Tight Generic Reduction for Fiat-Shamir In the ROM, there is a security property of a public-coin interactive proof system (IP) IP that reduces tightly to the soundness of its Fiat-Shamir transformation FS[IP]: State-Restoration Soundness. This flavor of soundness is essentially just an interactive model of ROM security for FS[IP]. This is, unfortunately, unlikely to have any reasonable analogue in the QROM: the interaction for IP is *classical* (i.e., non-quantum) and the quantum-accessible random oracle is quantum. Any plausible generalization of the reduction would thus require techniques involving quantum measurement leading to unacceptable reduction losses.

One way around this problem would be to avoid a reduction proof altogether by exploiting a purely statistical property of the underlying IP, like round-by-round (RBR) soundness. For a large class of digital signature scheme (DSS) constructions that includes FAEST, this poses a problem: The natural IP IP_0 that yields the DSS upon Fiat-Shamir transformation is still in the (Q)ROM, and can thus not be RBR-sound. The culprit here are hash-based commitments modeled in the (Q)ROM. For efficiency, they are usually not statistically-binding, e.g., when tree commitments are used, like BAVC presented in Section 5.2.

A somewhat counterintuitive but conceptually simple solution to this problem is to reconceptualize the DSS as a (non-standard) FS transformation of a different IP IP, by viewing every QRO call as a Fiat-Shamir hash, including the ones for generating commitments. Whenever IP_0 has a *classical* "straightline extractor" that extracts a for the *honest* prover given a list of its RO queries, IP must have RBR soundness[26].

We now give a toy example of a signature scheme to illustrate the technique (see Section 10.11.6 for the case of FAEST). On a high level, the IP IP_0 underlying any VOLEitH and MPCitH signature schemes work as follows:

1. Create a secret sharing of the private key sk into shares $s_0, \dots, s_{L-1}$.
2. Commit to the shares in a way that allows opening certain subsets of them, producing a commitment com
3. Create a proof $\tilde{\pi}$ that the secret-shared sk is valid for the public key pk, that can be verified using certain subsets of the shares s_i
4. Send $(\mathsf{com}, \tilde{\pi})$ to the Verifier and receive challenge $\mathsf{chall} \subset [L]$.
5. Open $(s_i)_{i \in \mathsf{chall}}$

In general, there might be additional rounds of interaction, e.g., to sample keys for universal hash functions as in FAEST. For now, consider the simplest case where $\mathsf{com} = (\mathsf{com}_0, \dots, \mathsf{com}_{L-1})$, $\mathsf{com}_i = \mathsf{H}_1(s_i, r_i)$ is a hash-based commitment to s_i using a hash function H_1 modeled as a RO, and chall is the only verifier message. The Fiat-Shamir DSS based on this IP then computes $\mathsf{chall} = \mathsf{H}_2(\mathsf{pk}, \mathsf{msg}, \mathsf{com}, \tilde{\pi})$. If we are only interested in soundness[27], we can replace all calls to H_1 by a round of interaction with the verifier, i.e., Line 2 above is replaced with

2.' For $i = 0, \dots, L-1$, sample r_i. Send $(s_i, r_i)_{i \in [L]}$ to the verifier and receive challenge $(\mathsf{com}_i)_{i \in [L]}$.

This defines a 2-challenge IP IP. We now consider a non-standard variant of the Fiat-Shamir transformation where the first challenge is derived using parallel application of the RO H_1. This transformation recovers the original DSS.

For more efficient commitments like BAVC used in FAEST, more rounds have to be added as some of the RO queries need to happen sequentially, which requires replacing them with sequential rounds of interaction.

10.11.2 Challenge 2: Proving Tight Soundness of Fiat-Shamir from RBR Soundness After re-conceptualizing a DSS as the FS transformation of a RBR-sound IP, we face the next challenge: To the best of our knowledge, there is no theorem available in the literature tightly proving the soundness of the FS transformation of a RBR-sound IP. Fortunately, the framework for proving query complexity bounds in the QROM presented in [CFHL21] was used for a similar problem before [AHJ+23], and provides the necessary technical tool. In this section, we use this tool to prove soundness of the FS transformation of a RBR-sound IP.

We begin by formalizing the necessary properties of the non-standard FS transformation from Section 10.11.1. We define two properties of RO-based functions that make them fit for deriving challenges in a Fiat-Shamir transformation.

Definition 10.45. *A function $f^{\mathsf{H}} : \mathcal{X} \to \mathcal{Y}$ that can be computed using L queries to another function $\mathsf{H} : \mathcal{X}' \to \mathcal{Y}'$ is* collision resistance preserving*, if the following holds. For all $x_0, x_1 \in \mathcal{X}$ such that $x_0 \neq x_1$ and $f^{\mathsf{H}}(x_0) = f^{\mathsf{H}}(x_1)$ there are $x'_0, x'_1 \in \mathcal{X}'$ such that $x'_0 \neq x'_1$ and*

[26] In this generality, there might be some freedom in how to define the verifier of IP, so the claim here is to be understood as the existence of a Verifier for IP such that it has RBR soundness

[27] and if we in particular are fine with having no chance of any flavor of zero knowledge

1. H *is queried on input* x'_i *when* $f^{\mathsf{H}}(x_i)$ *is computed for* $i \in \{0,1\}$.
2. $\mathsf{H}(x'_0) = \mathsf{H}(x'_1)$.

For an input x*, let* $\mathsf{Qry}(f^{\mathsf{H}}, x)$ *denote the ordered list of oracle inputs queried in the deterministic evaluation of* $f^{\mathsf{H}}(x)$*. The function is* sensitive *if*

1. $|\{f^{\mathsf{H}_{x' \mapsto u}}(x) | u \in \mathcal{Y}'\}| \in \{1, |\mathcal{Y}'|\}$ *(SN1) and*
2. $\mathsf{Qry}(f^{\mathsf{H}}, x) = \mathsf{Qry}(f^{\mathsf{H}}, x') \implies x = x'$*. (SN2)*

Note that $f^{\mathsf{H}}(x_0, x_1, \ldots, x_{L-1}) = (\mathsf{H}(x_0), \mathsf{H}(x_1), \ldots, \mathsf{H}(x_{L-1}))$ has both properties. We go on to state and prove the main theorem of this section. For ease of presentation, we formulate the theorem for the FS transformation for proofs, rather than signatures, and denote problem instances by inst instead of pk.

Theorem 10.46 (RBR Soundness of IP $\Rightarrow$ **Soundness of** FS[IP]**).** *Let* IP *be an IP with* ℓ *challenges* $(\mathsf{chall}_i)_{i \in [\ell]}$*. Let* $S \subset [\ell]$*,* $|S| = \ell_1$*, such that* IP *is RBR sound with RBR soundness error 0 for challenges* chall_i *where* $i \in S$*, and RBR soundness error* ε *else. Let* f^{H_1} *be a collision-resistance-preserving and sensitive function making* L *queries to* H_1*, and let* FS[IP] *be the Fiat-Shamir transformation of* IP *where challenges* chall_i *for in* $i \in S$ *are derived using* f^{H_1}*, and challenges* chall_i *are derived using* H_2^i *for* $i \notin S$*. Denote by* $\mathsf{H}_2 = (\mathsf{H}_2^i)_{i \in [\ell] \setminus S}$ *an oracle that allows querying any of the* H_2^i*. For any static QROM adversary* $\mathcal{A}^{\mathsf{H}_1, \mathsf{H}_2}$ *against* FS[IP] *that makes a total of* $\tilde{Q}$ *quantum queries to its oracles,*

$$\mathsf{AdvSnd}_{\mathcal{A}}^{\mathsf{FS[IP]}} \leq 30 \cdot Q^3 \cdot 2^{-l_{\mathrm{out}}} + 10 \cdot Q^2 \cdot \varepsilon$$

where l_{out} *is the output length of* H_1 *and* H_2^i *and* $Q = \tilde{Q} + 2\ell + 2(L-1)\ell_1$

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 $\mathsf{H}_1^0 := \mathsf{H}_1$. For ease of notation, we define f_i^{H} as the function used to derive chall_i. Let T^{IP} be the set of transcripts for IP.

To use Lemma 10.44, we define the following predicate on pairs of partial transcripts $\mathsf{trc} = (\mathsf{pmsg}_1, \mathsf{chall}_1, \ldots, \mathsf{chall}_i)$,

$$\begin{aligned}\mathrm{pred}_{\mathsf{inst},\mathsf{H}}(\mathsf{trc}) =& \mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}) \wedge \neg \mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}_{i-1}) \\ & \wedge \left(\forall j = 1, \ldots, i : \mathsf{chall}_j = f_j^{\mathsf{H}}(\mathsf{chall}_{j-1} \| \mathsf{pmsg}_j) \right).\end{aligned}$$

Here we have defined the truncation $\mathsf{trc}_{i-1} = (\mathsf{pmsg}_1, \mathsf{chall}_1, \ldots, \mathsf{chall}_{i-1})$ and set $\mathsf{chall}_0 = \iota$, the empty string. If $\mathrm{pred}_{\mathsf{inst},\mathsf{H}}(\mathsf{trc})$, we say (trc) is *bad* for H. We further define a predicate $\mathsf{P} : (\mathcal{Y} \cup \{\perp\})^{\mathcal{X}} \to \{0,1\}$ by $\mathsf{P}(D) :\Leftrightarrow$ there exists bad (trc) for D. P has $(\ell + \ell_1 \cdot (L-1))$-local witnesses, as

the list of random oracle inputs queried when evaluating $\mathrm{pred}_{\mathsf{inst},\mathsf{H}}(\mathsf{trc})$ is a witness if $\mathrm{pred}_{\mathsf{inst},\mathsf{H}}(\mathsf{trc})$. We call i the length of the witness. In the following we denote by D_i^j the partial functions obtained in the same way as H_i^j are obtained from H. We also define the collision predicate P_C by setting $\mathsf{P}_C(D)$ iff there exist inputs $x \neq x'$ and indices i, j such that $D_i^j(x) = D_i^j(x') \neq \perp$. The collision predicate clearly has 2-local witnesses. We construct algorithm $\mathcal{B}^{\mathsf{H}}$ as follows. $\mathcal{B}^{\mathsf{H}}$ runs $\pi \leftarrow \mathcal{A}^{\mathsf{H}}$ and recomputes the challenges. Denote the completed transcript by trc. If there exists i such that $\mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}_i)$ and $\neg \mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}_{i-1})$, output the predicate witness z, else, output $\perp$. By RBR soundness we have

$$\Pr_{\pi \leftarrow \mathcal{A}^{\mathsf{H}}(\mathsf{inst})}[\text{verifier accepts } \pi] \leq \Pr_{z \leftarrow \mathcal{B}^{\mathsf{H}}}[\langle z, \mathsf{P}, \mathsf{H} \rangle] \leq \Pr_{z \leftarrow \mathcal{B}^{\mathsf{H}}}[\langle z, \mathsf{P} \vee \mathsf{P}_C, \mathsf{H} \rangle].$$

We now aim to bound the quantity

$$\max_{\substack{x,D:\\ |D|\le k,\neg \mathsf{P}'(D)}} \Pr_{u\leftarrow\mathcal{Y}}[\mathsf{P}'(D[x\mapsto u])]$$

for $\mathsf{P}' = \mathsf{P} \vee \mathsf{P}_C$. In the following, we omit the domain of the maximum. Here, D is a partial function with the same domain and range as H. We distinguish different cases depending on which domain x is from. We derive bound assuming the query does not derive the last challenge. If it derives the last challenge, the bounds also hold as a strict subset of the options for making the database go bad exist in this case.

1. x is in the domain of H_1. As the RBR soundness error of the challenges in whose derivation H_1 is involved in is 0, so there are two ways to fulfill $\mathsf{P}'(D[x\mapsto u])$, i) if a collision occurs, i.e., $\exists x' : D_1^0(x) = D_1^0(x') \neq \perp$, or ii) if there exists x, y such that $f^{D_1^0[x\mapsto u]}(x) = y$ and y is the challenge part of some x' such that $\exists j : D_2^j(x') \neq \perp$, or $f^{D_1^0}(x') \neq \perp$. By the sensitivity of f, we have

$$\Pr_{u\leftarrow\mathcal{Y}}[\mathsf{P}'(D[x\mapsto u])] \le 3|D|2^{-l_{\mathrm{out}}}.$$

2. x is in the domain of H_2^j for some j. For D with $\neg\mathsf{P}'(D)$, there are 3 ways to fulfill $\mathsf{P}'(D[x\mapsto u])$. The first option is that there exists x' in the domain of H_2^j, $x' \neq x$ 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 $f_{i+1}(u\|\mathsf{pmsg}_{i+1}) \neq \perp$. 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 $\neg\mathsf{P}_C(D)$ and the collision-resistance-preservingness of f)
   with the following properties. Let $\mathsf{chall}_{i-1} = f_{i-1}^D(z_{\mathrm{last}})$, 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) $\mathsf{RBRState}(\mathsf{inst}, \mathsf{trc}_{i-2}) = 0$
   (c) there exists pmsg_i such that $x = 2\|i\|\mathsf{chall}_{i-1}\|\mathsf{pmsg}_i$ and $\mathsf{RBRState}(\mathsf{inst}, (\mathsf{pmsg}_1, \mathsf{chall}_1, \ldots, \mathsf{pmsg}_i, u)) = 1$.
   In summary, we have

$$\Pr_{u\leftarrow\mathcal{Y}}[\mathsf{P}'(D[x\mapsto u])] \le 2|D|\cdot 2^{-l_{\mathrm{out}}} + \varepsilon.$$

   where the term $2|D|\cdot 2^{-l_{\mathrm{out}}}$ is from the first and second option, we have used RBR soundness to bound
   the probability

$$\Pr_{u\leftarrow\mathcal{Y}}[\mathsf{RBRState}(\mathsf{inst}, (\mathsf{pmsg}_1, \mathsf{chall}_1, \ldots, \mathsf{pmsg}_i, u))] \le \varepsilon.$$

In conclusion, we have

$$\max \Pr_{u\leftarrow\mathcal{Y}}[\mathsf{P}'(D[x\mapsto u])] \le \max\left(3|D|\cdot 2^{-l_{\mathrm{out}}}, 2|D|\cdot 2^{-l_{\mathrm{out}}} + \varepsilon\right) \le \frac{3\tilde{Q}}{2^{l_{\mathrm{out}}}} + \varepsilon := \delta.$$

Using Equation (17), and applying Lemma 10.44 to the $(\tilde{Q}+\ell+(L-1)\ell_1)$-query algorithm $\mathcal{B}$, recalling that P has $(\tilde{Q}+\ell+(L-1)\ell_1)$-local witnesses,
we obtain for any invalid instance $\mathsf{inst} \notin \mathcal{L}$,

$$
\begin{aligned}
&\Pr_{\pi\leftarrow\mathcal{A}^{\mathsf{H}_1,\mathsf{H}_2}(\mathsf{inst})}[\mathcal{V}_{\mathsf{FS[IP]}}(\mathsf{inst},\pi)=1]\\
&\leq\left(\sum_{k=1}^{\tilde{Q}+2\ell+2(L-1)\ell_1}\sqrt{10\max_{u\leftarrow\mathcal{Y}}\Pr[\mathsf{P}(D[x\mapsto u])]}\right)^2\leq\left(\sum_{k=1}^{\tilde{Q}+2\ell+2(L-1)\ell_1}\sqrt{10\delta}\right)^2\\
&=10(\tilde{Q}+2\ell+2(L-1)\ell_1)^2\delta\\
&=30(\tilde{Q}+2\ell+2(L-1)\ell_1)^3 2^{-l_{\mathrm{out}}}+10(\tilde{Q}+2\ell+2(L-1)\ell_1)^2\varepsilon.
\end{aligned}
$$
□

10.11.3 Challenge 3: Reprogramming a QRO at a Computationally Unpredictable Input The third challenge arises when upgrading EUF-KO security to EUF-CMA security. For FS signatures, reductions of breaking EUF-KO to breaking EUF-CMA are based on the following rough strategy. The IP underlying the FS DSS has some form of simulability, usually honest-verifier zero knowledge (HVZK). In the following, we assume the IP has HVZK. To show that an adversary with access to a signing oracle can be simulated without using the private key, simulated proofs are generated using the HVZK simulator. The random oracle(s) are then re-programmed to be consistent with the simulated proofs being honest signatures. This needs to happen adaptively – the outputs to be re-programmed are only revealed to the reduction once a query to the signing oracle is made. To show that this re-programming is indistinguishable, it is a necessary ingredient to show that it is hard to predict the inputs where the RO will be re-programmed, and thus the adversary will not have queried these inputs in advance. In the QROM, this step is facilitated by the adaptive reprogramming lemma (also known as resampling lemma) [GHHM21]. Here, it is exploited that the inputs where the re-programming happens are *statistically* unpredictable in the sense that they have min-entropy conditioned on the adversary's view.

In the corresponding step in the ROM security proof of FAEST, however, it is used that the BAVC commitment in a FAEST signature is *computationally* unpredictable. The reduction is to an unpredictability game where the adversary has to output a list of guesses, and wins if the commitment generated by the challenger is in the list. Such an argument cannot be generalized to the QROM setting using the techniques of [GHHM21] directly. It is natural to instantiate the QRO using a compressed oracle for the reduction, but to reduce to the list-unpredictability game a quantum measurement needs to be performed.

We chose a different way to solve this problem. To retain as much tightness as possible, while also achieving some modularity in the security proof, we define quantum-list unpredictability, where only a two-outcome measurement is performed. This two-outcome measurement checks whether the commitment generated by the challenger is in a quantum register the adversary has submitted. Quantum-list unpredictability is a clean security property that can be evaluated for any cryptographic algorithm (not just commitments), and it facilitates a tight bound on the distinguishability between a security game and a hybrid that is aborted if a reprogrammed input was previously queried.

An additional advantage of this technique is, that a (small) multiplicative loss, like the one arising in proofs using Rényi-divergence arguments, need not be converted into an additive loss. This allows for better parameters. To facilitate this type of reduction, our definition of quantum-list unpredictability has a slightly more complicated syntax than it would appear necessary.

The natural definition of quantum-list unpredictability is as follows.

Definition 10.47 (Quantum-List Unpredictability). *Let $\mathcal{B}$ be an algorithm, possibly with access to one or more (Q)ROs. The quantum-list unpredictability game* $\mathsf{ExpQUnpred}^{\mathcal{B}}_{b,\mathcal{A}}$

between an adversary $\mathcal{A}$ (with access to the same oracles as $\mathcal{B}$, if any) and a challenger is as follows.

1. $(L, \mathsf{inp}, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda)$, *where L is a quantum register holding a (superposition of) list(s) $\mathcal{L}_{\mathsf{com}}$ of guesses for the output of $\mathcal{B}$.*
2. $C = (C_{\mathsf{pub}}, C_{\mathsf{aux}}) \leftarrow \mathcal{B}(\mathsf{inp})$.
3. *Set $b' = 1$. If $b = 1$*
   (a) *apply the binary measurement to L that tests membership of C_{pub} in the list $\mathcal{L}_{\mathsf{com}}$ the register L contains (in superposition) to obtain output b'' ($b'' = 1$ indicates "found").*
   (b) *If $b'' = 1$, set $b' = 0$.*
4. *If $b' = 1$, set $b' \leftarrow \mathcal{A}(\mathsf{state}, L, C)$.*
5. *Output b'*

We say $\mathcal{B}$ is (γ, ε)-quantum-list unpredictable if

$$\Pr[\mathsf{ExpQUnpred}^{\mathcal{B}}_{0,\mathcal{A}} = 1] \leq \gamma \left(\Pr[\mathsf{ExpQUnpred}^{\mathcal{B}}_{1,\mathcal{A}} = 1] + \varepsilon\right).$$

We prove the quantum-list unpredictability of VOLECommit used in FAEST in Section 10.11.11, using the adaptive reprogramming lemma from [GHHM21] and a gentle measurement argument to analyze the measurement in Line 3.a.

10.11.4 QROM Security of FAEST Using these techniques, we obtain a EUF-CMA security bound for FAEST in the QROM providing meaningful concrete post-quantum security for FAEST.

The security bound we obtain has a multiplicative loss of ≈ 8 with respect to the PRG security of AES-CTR, and the additive loss terms are all at most a Grover's search advantage larger than their counterparts in Theorems 10.29 and 10.30. See Corollary 10.54 for a bound for FAEST-EM.

10.11.5 Proof Overview In the following sections, 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 10.53) 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") [GHHM21]. 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 10.52).

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 10.50).

10.11.6 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. $\mathsf{pmsg}_{-1} = \mathsf{com}_{0,0} \| \mathsf{com}_{0,1} \| \dots \mathsf{com}_{0,N_0-1} \| \mathsf{com}_{1,0} \| \dots \mathsf{com}_{\tau-1,N_{\tau-1}-1}$ 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 $\mathsf{iv}^{\mathsf{pre}}$ removed, i.e. $\mathsf{pmsg}_1 = \mathbf{c}_1 \| \dots \| \mathbf{c}_{\tau-1} \| \mathbf{c}_{\mathsf{mult}}$
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. $\mathsf{chall}_{-2} = \mathsf{uhash}$
3. $\mathsf{chall}_{-1} = h_0 \| \dots \| h_{\tau-1}$
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, $i = 1, 2, 3$. Additionally, replace H_5 by an arbitrary fixed function. 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 4,5,13 and 18, and replacing Lines 21 and 22. The verifier parses pmsg_{-1} as

$$(\mathsf{com}_{i,j})_{i \in [0..\tau),\, j \in [0..N_i)}.$$

When executing the reconstruction algorithm, it uses chall_{-2} in place of $\mathsf{H}_0(\mathsf{chall}_{-3})$. Whenever reconstruction recomputes a leaf commitment, the verifier rejects unless it equals the corresponding component of pmsg_{-1}.

The verifier does not call H_1 to derive the values $(h_i)_{i \in [0..\tau)}$ or com; it uses chall_{-1} and chall_0, respectively, and verifies that they are consistent with the explicit leaf commitments as prescribed by the protocol.

Instead of Lines 21 and 22, the verifier compares the result $\tilde{a}_0'$ of Line 20 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, H_5 by an arbitrary function 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. $\mathsf{chall}_{-3} = \mathsf{H}_4(\mathsf{pmsg}_{-3} \| H_2^0(\mathsf{pk} \| \mathsf{msg}))$
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. $\mathsf{chall}_0 \coloneqq \mathsf{H}_1(\mathsf{chall}_{-1} \| \mathsf{pmsg}_0)$
5. $\mathsf{chall}_1 \coloneqq \mathsf{H}_2^1(H_2^0(\mathsf{pk} \| \mathsf{msg}) \| \mathsf{chall}_0 \| \mathsf{pmsg}_1 \| \mathsf{iv}^{\mathsf{pre}})$
6. $\mathsf{chall}_2 \coloneqq \mathsf{H}_2^2(\mathsf{chall}_1 \| \mathsf{pmsg}_2)$
7. $\mathsf{chall}_3 \coloneqq \mathsf{H}_2^3(\mathsf{chall}_2 \| \mathsf{pmsg}_3)$

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. 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 arguments for deriving chall_{-3} and 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 (FAEST-EM$_r^{\mathrm{elc}}$ of FAEST-EM$_r$) that is equivalent from a security point of view, and differs only in efficiency (see Lemma 10.50)

10.11.7 RBR Soundness of FAEST-IP$^+$.

We prove RBR soundness for the expanded IP.

Lemma 10.48. FAEST-*IP*$^+$ *has RBR soundness with passive prefix up to* chall_{-3}*, where challenge* chall_{-2} *is isolated, with RBR soundness errors*

$$\varepsilon_{-3} = \varepsilon_{-1} = \varepsilon_0 = 0 \tag{17}$$

$$\varepsilon_{-2} = \tau \cdot 2^{-\lambda} \tag{18}$$

$$\varepsilon_1 = \varepsilon_2 = \varepsilon_3 = \varepsilon \coloneqq d_{\mathsf{zk}} \cdot 2^{-\lambda} \tag{19}$$

Proof. Define the state function RBRState to output 1 if uhash in chall_{-2} defines a non-injective $\mathcal{H}_{\mathsf{uhash}}$. 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 uhash in chall_{-2} defines an injective $\mathcal{H}_{\mathsf{uhash},i}$ for each tweak i, RBRState is computed by extracting all commitments that are actually extractable as in Lemma 10.38 and applying the RBR state function from Lemma 10.40. Now the expression for ε_{-2} follows by the analysis in the proof of Lemma 10.19, and the expression for $\varepsilon_1, \varepsilon_2, \varepsilon_3$ is due to Lemma 10.40. 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. □

10.11.8 $\mathsf{FAEST}_r^{\mathrm{elc}}$ Soundness to $\mathsf{FAEST\text{-}IP}^+$ RBR Soundness. We prove the soundness of FAEST based on the round-by-round soundness of $\mathsf{FAEST\text{-}IP}^+$, using Lemma 10.44 and the notation introduced in Section 10.11.2. 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 10.49. *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,*

$$\mathsf{AdvSnd}_{\mathcal{A}}^{\mathsf{FAEST}_r^{\mathrm{elc}}} \leq 10(\tau+1)\cdot Q^3\cdot 2^{-2\lambda} + 10\cdot Q^2\cdot(\tau+d_{\mathsf{zk}})\cdot 2^{-\lambda}.$$

For FAEST-EM, the advantage is instead bounded as

$$\mathsf{AdvSnd}_{\mathcal{A}}^{\mathsf{FAEST\text{-}EM}_r^{\mathrm{elc}}} \leq 10(\tau+1)\cdot Q^3\cdot 2^{-2\lambda} + 10\cdot Q^2\cdot d_{\mathsf{zk}}\cdot 2^{-\lambda}.$$

Here,

$$Q = Q_0 + Q_1 + Q_{2,0} + Q_{2,1} + Q_{2,2} + Q_{2,3} + Q_4 + 2\tau + d_{\mathsf{zk}} + 12.$$

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 10.44, we define the following predicate on pairs of partial transcripts $\mathsf{trc} = (\mathsf{msg}, \mu, \mathsf{pmsg}_{-3}, \mathsf{chall}_{-3}, \ldots, \mathsf{chall}_i)$,

$$\begin{aligned}
\mathrm{pred}_{\mathsf{pk},\mathsf{H}}(\mathsf{trc},\mathsf{msg}) = &\mathsf{RBRState}(\mathsf{pk},\mathsf{trc})\\
&\wedge \neg\mathsf{RBRState}(\mathsf{pk},\mathsf{trc}_{i-1})\\
&\wedge \Big(\forall j\in[-3:i]\setminus\{-1,1\} : \mathsf{chall}_j = \hat{\mathsf{H}}_j(\mathsf{chall}_{j-1}\|\mathsf{pmsg}_j)\Big)\\
&\wedge \Big(i<-1 \vee \mathsf{chall}_{-1} = \hat{\mathsf{H}}_{-1}(\mathsf{pmsg}_{-1})\Big)\\
&\wedge \Big(i<1 \vee \mathsf{chall}_1 = \hat{\mathsf{H}}_1(\mu\|\mathsf{chall}_0\|\mathsf{pmsg}_1\|\mathsf{pmsg}_{-3})\Big)\\
&\wedge \mathsf{H}_2^0(\mathsf{pk},\mathsf{msg}) = \mu
\end{aligned}$$

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 $\mathrm{pred}_{\mathsf{pk},\mathsf{H}}(\mathsf{trc},\mathsf{msg})$, we say (trc,μ) is *bad* for H. We further define a predicate $\mathsf{P} : (\mathcal{Y}\cup\{\perp\})^{\mathcal{X}} \to \{0,1\}$ 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 $x\neq x'$ in the domain of the same hash function H_1 or H_2^i, $i=0,1,2,3$ such that $D(x)=D(x')\neq\perp$ (to be understood with respect to the appropriate truncated output). 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 $\perp$. By RBR soundness (Lemma 10.48) we have

$$\begin{aligned}\Pr_{\pi \leftarrow \mathcal{A}^{\mathsf{H}}(\mathsf{pk})}[\text{verifier accepts } \pi] &\leq \Pr_{z \leftarrow \mathcal{B}^{\mathsf{H}}}[\langle z, \mathsf{P}, \mathsf{H}\rangle] \\ &\leq \Pr_{z \leftarrow \mathcal{B}^{\mathsf{H}}}[\langle z, \mathsf{P} \vee \mathsf{P}_C, \mathsf{H}\rangle] \qquad (20)\end{aligned}$$

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 $D(x) \neq \perp$. 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., $\exists x' : \hat{D}_{-2}(x) = \hat{D}_{-2}(x') \neq \perp$, or ii) if a u is chosen such that $\mathsf{RBRState}(\mathsf{pk}, \tau | x | u) = 1$. By Lemma 10.48, 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 10.48.
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. $\exists x' : D_1^0(x) = D_1^0(x') \neq \perp$. 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 $D_1^0(x') \neq \perp$ and there exists $i \in [0, \tau)$ such that $u = h_i$, or $x' = \mu \| \mathsf{com} \| \mathsf{pmsg}_1 \mathsf{pmsg}_{-3} \|$ 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 $\neg \mathsf{P}_C(D)$)

$$z = \Big(0\|0\|\mathsf{iv}, 1\|1\|\mathsf{com}_{0,0}\| \dots \|\mathsf{com}_{0,N_0}, \dots, 1\|1\|\mathsf{com}_{\tau-1,0}\| \dots \|\mathsf{com}_{\tau-1,N_{\tau-1}},$$
$$1\|1\|\mathsf{chall}_{-1}, 2\|1\|\mu\|\mathsf{chall}_0\|\mathsf{pmsg}_1\|\mathsf{pmsg}_{-3}\Big)$$

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 $\mathsf{RBRState}(\mathsf{pk}, (\mathsf{pmsg}_1, \mathsf{chall}_1, \dots, \mathsf{pmsg}_i, u)) = 1$.

In summary, we have

$$\Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}'(D[x \mapsto u])] \le \frac{|\hat{D}_i| + |\hat{D}_{i+1}|}{\hat{M}_i} + \varepsilon =: \hat{\delta}_i,$$

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 10.48 to bound the probability

$$\Pr_{u \leftarrow \mathcal{Y}}[\mathsf{RBRState}(\mathsf{pk}, (\mathsf{pmsg}_1, \mathsf{chall}_1, \dots, \mathsf{pmsg}_i, u))] \le \varepsilon$$

in the third option, and ε is defined in Lemma 10.48.

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

$$\Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}'(D[x \mapsto u])] \le \varepsilon =: \hat{\delta}_3$$

in this case

In conclusion, we have

$$\max_{\substack{x,D:\\ |D|\le k\\ \neg \mathsf{P}'(D)}} \Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}'(D[x \mapsto u])] \le \max(\hat{\delta}_{-2}, \delta_1, \delta_{2,0}, \hat{\delta}_1, \hat{\delta}_2, \hat{\delta}_3) := \delta(k).$$

Using Equation (20), and applying Lemma 10.44 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

$$\begin{aligned}
&\Pr_{(\mathsf{pk},\pi) \leftarrow \mathcal{A}^{\mathsf{H}_1, \mathsf{H}_2^1, \dots, \mathsf{H}_2^\ell}}[\mathsf{pk} \notin \mathcal{L} \wedge \text{verifier accepts}] \\
&\le \left(\sum_{k=1}^{\tilde{Q}+2\tau+12} \sqrt{10 \max_{\substack{x,D:\\ |D|\le k\\ \neg \mathsf{P}(D)}} \Pr_{u \leftarrow \mathcal{Y}}[\mathsf{P}(D[x \mapsto u])]} \right)^2 \\
&\le \left(\sum_{k=1}^{\tilde{Q}+2\tau+12} \sqrt{10\delta(k)} \right)^2 \\
&\le 10(\tilde{Q} + 2\tau + 12) \sum_{k=1}^{\tilde{Q}+2\cdot\tau+12} \delta(k)
\end{aligned}$$

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 $\delta(k) \leq (\tau+1)(\tilde{Q}+2\tau+12)2^{-2\lambda} + (\tau+d_{\mathsf{zk}})\cdot 2^{-\lambda}$ and thus

$$\begin{aligned}&\Pr_{(\mathsf{pk},\pi)\leftarrow\mathcal{A}^{\mathsf{H}_1,\mathsf{H}_2^1,\ldots,\mathsf{H}_2^\ell}}[\mathsf{pk}\notin\mathcal{L}\wedge\text{verifier accepts}]\\ &\leq 10(\tau+1)(\tilde{Q}+2\tau+12)^3 2^{-2\lambda} + 10(\tilde{Q}+2\tau+12)^2(\tau+d_{\mathsf{zk}})\cdot 2^{-\lambda}.\end{aligned}$$

For FAEST-EM, chall_{-2} does not exist and the RBRState function can only flip after chall_1 at the earliest. The term $10(\tilde{Q}+2\tau+12)^2(\tau+d_{\mathsf{zk}})\cdot 2^{-\lambda}$ thus becomes Q^2, $10(\tilde{Q}+2\tau+12)^2\cdot d_{\mathsf{zk}}\cdot 2^{-\lambda}$. □

10.11.9 FAEST EUF-KO ⇐ FAEST$_r^{\mathrm{elc}}$ Soundness

Lemma 10.50. *Let $\mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_2^0,\mathsf{H}_2^1,\mathsf{H}_2^2,\mathsf{H}_2^3,\mathsf{H}_3,\mathsf{H}_4,\mathsf{H}_5}$ be a QROM* EUF-KO *adversary against* FAEST *that makes a total of $Q_0, Q_1, Q_{2,0}, Q_{2,1}, Q_{2,2}, Q_{2,3}, Q_3, Q_4, Q_5$ queries to its oracles. Then there exist a quantum PRG adversary $\mathcal{B}_1$ against* OWF *and a QROM adversary $\mathcal{B}^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_2^0,\mathsf{H}_2^1,\mathsf{H}_2^2,\mathsf{H}_2^3,\mathsf{H}_4,\mathsf{H}_5}$ against* FAEST$_r^{\mathrm{elc}}$ *that makes at most a total of $Q_0+1, Q_1+\tau+1, Q_{2,0}+1, Q_{2,1}+1, Q_{2,2}+1, Q_{2,3}+1, Q_4+1, Q_5+n_{\mathsf{mult}}$ queries to its oracles such that*

$$\mathsf{AdvEUFKO}^{\mathsf{FAEST}}_{\mathcal{A}} \leq \frac{1}{1-\frac{\mathsf{dom(OWF)}}{\mathsf{cod(OWF)}}}\cdot\left(\mathsf{AdvSnd}^{\mathsf{FAEST}_r^{\mathrm{elc}}}_{\mathcal{B}} + \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1}\right)$$

For FAEST-EM*, there exists an adversary $\mathcal{C}$ against the almost injectivity (Definition 10.13) of* PRG *such that the following similar bound holds.*

$$\mathsf{AdvEUFKO}^{\mathsf{FAEST\text{-}EM}}_{\mathcal{A}} \leq \frac{1}{1-\frac{\mathsf{dom(OWF)}}{\mathsf{cod(OWF)}}}\cdot\left(\mathsf{AdvSnd}^{\mathsf{FAEST\text{-}EM}_r^{\mathrm{elc}}}_{\mathcal{B}} + \mathsf{AdvPRG}^{\mathsf{OWF}}_{\mathcal{B}_1} + \mathsf{AdvOivInj}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{C}}[L]\right).$$

Proof. First, define $\mathcal{A}'^{\mathsf{H}_0,\mathsf{H}_1,\mathsf{H}_2^0,\mathsf{H}_2^1,\mathsf{H}_2^2,\mathsf{H}_2^3,\mathsf{H}_4,\mathsf{H}_5}$ 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

$$\mathsf{AdvSnd}^{\mathsf{FAEST}}_{\mathcal{A}} = \mathsf{AdvSnd}^{\mathsf{FAEST}_r}_{\mathcal{A}} = \mathsf{AdvSnd}^{\mathsf{FAEST}_r}_{\mathcal{A}'}.$$

Now define a prover $\mathcal{B}$ for FAEST$_r^{\mathrm{elc}}$ 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 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 13, 18, and 21 of FAEST.Verify, respectively. By contsruction, if FAEST.Verify accepts, then the resulting proof is accepted by FAEST$_r^{\mathrm{elc}}$ verification. We have thus shown that

$$\mathsf{AdvSnd}^{\mathsf{FAEST}}_{\mathcal{A}} = \mathsf{AdvSnd}^{\mathsf{FAEST}_r^{\mathrm{elc}}}_{\mathcal{B}}.$$

For FAEST-EM, the same argument shows that

$$\mathsf{AdvSnd}^{\mathsf{FAEST\text{-}EM}}_{\mathcal{A}} = \mathsf{AdvSnd}^{\widetilde{\mathsf{FAEST\text{-}EM}}_r^{\mathrm{elc}}}_{\mathcal{B}},$$

where $\widetilde{\mathsf{FAEST\text{-}EM}}_r^{\mathrm{elc}}$ is identical to FAEST-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 10.13 as follows. $\mathcal{C}$ runs $\mathcal{A}$ while simulating oracles locally. It then outputs the list of $\mathsf{com}_{i,j}$ computed from the signature submitted by $\mathcal{A}$, together with the challenge iv output by H_4 when queried on $\mathsf{iv}^{\mathsf{pre}}$. We thus have

$$\mathsf{AdvSnd}^{\widetilde{\mathsf{FAEST\text{-}EM}^{\mathrm{elc}}_r}}_{\mathcal{B}} - \mathsf{AdvSnd}^{\mathsf{FAEST\text{-}EM}^{\mathrm{elc}}_r}_{\mathcal{B}} \leq \mathsf{AdvOivInj}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{C}}[L].$$

The claim then follows by applying Equation (12) proven in the proof of Theorem 10.30, which holds for any computational model. □

10.11.10 Multi-hiding

Lemma 10.51 (VOLECommit is multi-hiding). *Consider the multi-hiding experiment from Lemma 10.24 but with an adversary that has quantum access to its oracles.*

Then, with the same definitions of adversaries as in Lemma 10.24,

$$\begin{aligned}\Pr[\mathsf{G}^*_0 = 1] \leq e^{Q\cdot(\beta^2_{\mathsf{root}}+4\tau\lceil\log(L)-1\rceil\beta^2+\tau a^2_{\mathsf{leaf}})/2^{128}} \cdot \Big(&\Pr[\mathsf{G}^*_1 = 1] \\ &+ Q\cdot\mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}_{\mathcal{B}_{1.1}}[1] + Q\cdot\tau\lceil\log(L)-1\rceil\cdot\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1] \\ &+ Q\cdot\tau\cdot\mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}} \\ &+ 2^{-\lambda} + \frac{3}{2}Q2^{-\lambda} + Q\cdot\sqrt{K_{\max}\cdot n_{\mathsf{mult}}\cdot Q_5\cdot 2^{-(128+\lambda)}} + 2n_{\mathsf{mult}}Q^2K_{\max}2^{-(128+\lambda)}.\Big)\end{aligned}$$

with all quantities defined as in Lemma 10.51.

Proof. The proof is identical to the case without quantum oracle access, except for the hybrid transition where H_5 is programmed. Here, the distinguishability of the programmed and un-programmed games is

$$2^{-\lambda} + \frac{3}{2}\sqrt{n_{\mathsf{mult}}\cdot K_{\max}Q_5\cdot 2^{-(128+\lambda)}} + 2n_{\mathsf{mult}}(j-1)K_{\max}\cdot 2^{-(128+\lambda)}$$

instead, by an application of the adaptive reprogramming lemma [GHHM21]. Summing over j yields the desired bound.

□

10.11.11 Quantum-Guess Unpredictability of VOLECommit.

We start by proving a quantum generalization of Lemma 10.25.

Lemma 10.52 (VOLECommit is quantum-list unpredictable). *Fix admissible* $\mathtt{param}$, $\mathtt{param}_{\mathsf{VOLE}}$. *Let* $\beta = \frac{\lambda}{128}$. *Define algorithm* $\mathcal{B}$ *as follows: On input* μ, *run Lines 2–5 of* $\mathsf{ExpUnpred}$ *defined in Lemma 10.25 and output* $C = (C_{\mathsf{pub}}, C_{\mathsf{aux}})$, *where* C_{pub} *is defined as in Lemma 10.25 and* C_{aux} *consists of the remaining output values.*

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}, *define*

$$\gamma = e^{(4\lceil\log(L)\rceil\beta^2+16\beta^2)/2^{128}},$$
$$\varepsilon = 3\sqrt{\lceil\log(L)\rceil\cdot\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1] + 3\sqrt{Q_1}\cdot 2^{-\lambda} + Q_{\mathsf{com}}\cdot 2^{-2\lambda}}.$$

Then $\mathcal{B}$ *is* (γ,ε)*-quantum-list unpredictable. Here we write* $\mathsf{AdvPRP}^{\mathsf{LeafCommit}}$ *for* $\mathsf{AdvPRP}^{\mathsf{PRG}_{4\lambda}}$ *in* FAEST, *and* $\mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}$ *in* $\mathsf{FAEST\text{-}EM}$.

Proof. Denote

$$\mathsf{G}_b^{\mathsf{unpred}} = \mathsf{ExpQUnpred}_{b,\mathcal{A}}^{\mathcal{B}}.$$

We first add the membership measurement to $\mathsf{G}_0^{\mathsf{unpred}}$, before making any changes to the distribution of VOLECommit.

Define $\widetilde{\mathsf{G}}_0^{\mathsf{unpred}}$ to be identical to $\mathsf{G}_0^{\mathsf{unpred}}$, except that after C has been generated, the binary membership measurement from Line 3(a) of the quantum-list unpredictability experiment is applied to L. Its outcome is ignored, and the second invocation of $\mathcal{A}$ is performed regardless of the outcome.

Let p denote the probability that this measurement returns 1. By the gentle measurement lemma,

$$\left|\Pr[\mathsf{G}_0^{\mathsf{unpred}} = 1] - \Pr[\widetilde{\mathsf{G}}_0^{\mathsf{unpred}} = 1]\right| \leq 2\sqrt{p}.$$

Moreover, $\widetilde{\mathsf{G}}_0^{\mathsf{unpred}}$ and $\mathsf{G}_1^{\mathsf{unpred}}$ perform exactly the same measurement on exactly the same state. Conditioned on the measurement returning 0, the two games proceed identically, while conditioned on outcome 1, $\mathsf{G}_1^{\mathsf{unpred}}$ outputs 0. Hence

$$\Pr[\widetilde{\mathsf{G}}_0^{\mathsf{unpred}} = 1] \leq \Pr[\mathsf{G}_1^{\mathsf{unpred}} = 1] + p.$$

Consequently,

$$\Pr[\mathsf{G}_0^{\mathsf{unpred}} = 1] \leq \Pr[\mathsf{G}_1^{\mathsf{unpred}} = 1] + 2\sqrt{p} + p \leq \Pr[\mathsf{G}_1^{\mathsf{unpred}} = 1] + 3\sqrt{p}. \tag{21}$$

It remains to bound p. We do so using an event-only game, so the second stage of $\mathcal{A}$ will play no role in the remainder of the proof.

For each entry in L, consider only its com-component, and let the corresponding binary measurement test whether at least one of these components equals the generated com. Since membership of the complete C_{pub} in the list implies equality of its com-component, the projector for the original membership measurement is upper-bounded in the Loewner order by the projector for this coarser measurement. Let $\mathsf{G}_0^{\mathsf{hit}}$ be the event-only game which runs the first stage of $\mathcal{A}$, generates com according to the real commitment algorithm, applies this coarser measurement, and outputs its outcome. Then

$$p \leq \Pr[\mathsf{G}_0^{\mathsf{hit}} = 1].$$

Importantly, $\mathsf{G}_0^{\mathsf{hit}}$ depends on the output of the challenger only through com. We may therefore discard all the other values generated by VOLECommit, and in particular need only consider the computation of the BAVC commitment.

We now apply the same GGM-path and LeafCommit game hops as in the proof of Lemma 10.25, randomizing only a single leaf, say $(i, j) = (0, 0)$. Let $\mathsf{G}_2^{\mathsf{hit}}$ denote the resulting event-only game. The same PRG and divergence argument gives

$$\Pr[\mathsf{G}_0^{\mathsf{hit}} = 1] \leq \gamma\Big(\Pr[\mathsf{G}_2^{\mathsf{hit}} = 1] + \lceil\log(L)\rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1]\Big). \tag{22}$$

We next randomize the two H_1 outputs used to obtain com. Let

$$X_0 := \mathsf{com}_{0,0} \| \cdots \| \mathsf{com}_{0,N_0-1}.$$

In $\mathsf{G}_2^{\mathsf{hit}}$, the value $\mathsf{com}_{0,0}$ is uniformly random, so X_0 has at least 2λ bits of min-entropy. Define $\mathsf{G}_2'^{,\mathsf{hit}}$ by sampling h_0 independently uniformly at random and reprogramming H_1 at X_0 consistently. By Proposition 1 of [GHHM21],

$$\left|\Pr[\mathsf{G}_2^{\mathsf{hit}} = 1] - \Pr[\mathsf{G}_2'^{,\mathsf{hit}} = 1]\right| \leq \frac{3}{2}\sqrt{Q_1}\, 2^{-\lambda}.$$

Let

$$Y := h_0 \| \cdots \| h_{\tau-1}.$$

In $\mathsf{G}_2^{\prime,\mathsf{hit}}$, h_0 is uniformly random and independent, so Y likewise has 2λ bits of min-entropy. Define $\mathsf{G}_2^{\prime\prime,\mathsf{hit}}$ by additionally sampling com independently uniformly at random and reprogramming H_1 at Y consistently. Again by Proposition 1 of [GHHM21],

$$\left|\Pr[\mathsf{G}_2^{\prime,\mathsf{hit}} = 1] - \Pr[\mathsf{G}_2^{\prime\prime,\mathsf{hit}} = 1]\right| \leq \frac{3}{2}\sqrt{Q_1}\, 2^{-\lambda}.$$

It remains to bound the hit probability in $\mathsf{G}_2^{\prime\prime,\mathsf{hit}}$. Write

$$L = L_0 L_1 \ldots L_{Q_{\mathsf{com}}-1},$$

and write L_i^{com} for the subregister containing the com-component of the i-th guess. For fixed com, the projector corresponding to the 1 outcome of the coarse measurement is upper-bounded in the Loewner order as[28]

$$P_{1,\mathsf{com}}^{\mathsf{com}} \leq \sum_{i=0}^{Q_{\mathsf{com}}-1} |\mathsf{com}\rangle\langle\mathsf{com}|_{L_i^{\mathsf{com}}} \otimes \mathbb{1}_{(L_i^{\mathsf{com}})^c}.$$

Taking the expectation over the uniformly random com gives

$$\mathbb{E}_{\mathsf{com}\leftarrow\{0,1\}^{2\lambda}} P_{1,\mathsf{com}}^{\mathsf{com}} \leq \frac{Q_{\mathsf{com}}}{2^{2\lambda}}\mathbb{1}.$$

Therefore,

$$\Pr[\mathsf{G}_2^{\prime\prime,\mathsf{hit}} = 1] \leq Q_{\mathsf{com}} \cdot 2^{-2\lambda},$$

and hence

$$\Pr[\mathsf{G}_2^{\mathsf{hit}} = 1] \leq 3\sqrt{Q_1} \cdot 2^{-\lambda} + Q_{\mathsf{com}} \cdot 2^{-2\lambda}.$$

Combining this with (22) yields

$$p \leq \Pr[\mathsf{G}_0^{\mathsf{hit}} = 1] \leq \gamma(\varepsilon/3)^2.$$

Finally, inserting this bound into (21), and using $\gamma \geq 1$, gives

$$\begin{aligned}\Pr[\mathsf{G}_0^{\mathsf{unpred}} = 1] &\leq \Pr[\mathsf{G}_1^{\mathsf{unpred}} = 1] + 2\sqrt{\gamma\delta} + \gamma\delta \\ &\leq \gamma\left(\Pr[\mathsf{G}_1^{\mathsf{unpred}} = 1] + 2\sqrt{\delta} + \delta\right) \\ &\leq \gamma\left(\Pr[\mathsf{G}_1^{\mathsf{unpred}} = 1] + \varepsilon\right),\end{aligned}$$

which proves the claim. □

10.11.12 **FAEST** EUF-CMA ⇐ **FAEST** EUF-KO

Theorem 10.53 (FAEST is EUF-CMA secure). *Let* $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_4$ *and* H_5 *be modeled as quantum-accessible random oracles and* H_3 *be a PRF. Then for any QPT adversary* $\mathcal{A}$ *which makes* $Q_{\mathsf{sig}} \leq 2^{64}$ *queries to* $\mathcal{O}_{\mathsf{sig}}$ *and* $Q_0, Q_1, Q_{2,1}, Q_{2,2}, Q_{2,3}$ *and* Q_5

[28] The operator on the right-hand side counts the number of entries whose com-component equals com.

queries to $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$ *and* H_5*, respectively, there exists an* EUF-KO *adversary* $\mathcal{B}$ *such that*

$$
\begin{aligned}
\mathsf{AdvEUFCMA}^{\mathsf{FAEST}}_{\mathcal{A}}[Q_{\mathsf{sig}}] \leq\ & 2 \cdot \mathsf{AdvEUFKO}^{\mathsf{FAEST}}_{\mathcal{A}'} \\
& + 2 \cdot Q_{\mathsf{sig}} \cdot \Big(\mathsf{AdvPRF}^{\mathsf{H}_3} + \mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}[1] + \lceil \log(L) - 1 \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}[1] \\
& + 2 \cdot Q_{\mathsf{sig}} \cdot \Big(\mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}} \Big) \\
& + 2 \cdot Q_{\mathsf{sig}} \cdot \Big(\mathsf{AdvPRP}^{\mathsf{PRG}_{\mathsf{root}}}_{\mathcal{B}_{1,1}}[1] + \tau \cdot \Big(\lceil \log(L) - 1 \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}_1}[1] + \mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}} \Big) \Big) \\
& + 2 \cdot Q_{\mathsf{sig}} \cdot 3 \sqrt{\lceil \log(L) \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}}[1] + \mathsf{AdvPRP}^{\mathsf{leaf}}_{\mathcal{B}_{2,3}}} + 3\sqrt{Q_1} \cdot 2^{-\lambda} + Q_{\mathsf{com}} \cdot 2^{-2\lambda} \\
& + \frac{(Q_2^0 + Q_2^1 + 3)^2}{2^{2\lambda}} \\
& + \frac{3}{2} \cdot \sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{8\lambda+64}}} + \frac{3}{2} \sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}}} \Big) \\
& + 2 n_{\mathsf{mult}} \cdot 8 \cdot \left(2^{-\lambda} + \frac{3}{2} \sqrt{\frac{Q_5 + n_{\mathsf{mult}} Q_{\mathsf{sig}}}{2^{128+\lambda}}} \right) \\
& + 2 \left(2^{-\lambda} + Q_{\mathsf{sig}} 2^{-\lambda} + Q_{\mathsf{sig}} \cdot \sqrt{\lambda \cdot n_{\mathsf{mult}} \cdot Q_5 \cdot 2^{-(128+\lambda)}} + \lambda (2 n_{\mathsf{mult}} \cdot Q_{\mathsf{sig}}^2) 2^{-(128+\lambda)} \right)
\end{aligned}
$$

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 10.42, excluding game G_{11}. We only prove bounds for game hops where they differ from the classical ones beyond advantage terms now being defined with respect to quantum algorithms, and show how to finish the reduction without G_{11}.

G_3: In G_3, the quantum-accessible random oracle is simulated using the compressed oracle technique [Zha19]. Upon a query to $\mathcal{O}_{\mathsf{sig}}$, $\mathsf{chall}_1 \in \{0,1\}^{8\lambda+64}$ is sampled at random. When $\mathcal{O}_{\mathsf{sig}}$ is about to query H_2^1 on $\mu \| \mathsf{com} \| \boldsymbol{c}_1 \| \dots \| \boldsymbol{c}_{\tau-1} \| \mathbf{c}_{\mathsf{mult}} \| \mathsf{iv}^{\mathsf{pre}}$, 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 10.52. 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 10.52 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 $(\mathsf{com}, \mathbf{c}_1, \dots, \mathbf{c}_{\tau-1})$ and $\mathsf{iv}^{\mathsf{pre}}$ from the unpredictability experiment (instead of running VOLECommit and sampling $\mathsf{iv}^{\mathsf{pre}}$ itself), and then continues from there. By construction, $\mathsf{G}_{3,i-1} = 1 \iff \mathsf{ExpQUnpred}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}}_{0,\mathcal{A}_{\mathsf{unpred}}} = 1$ and $\mathsf{G}_{3,i} = 1 \iff \mathsf{ExpQUnpred}^{\mathsf{VOLECommit},\hat{\ell},\mathtt{param},\mathtt{param}_{\mathsf{VOLE}}}_{1,\mathcal{A}_{\mathsf{unpred},i}} = 1$. Thus, from Lemma 10.52 and a hybrid argument we get

$$
\begin{aligned}
\Pr[\mathsf{G}_3 = 1] \leq & e^{Q_{\mathsf{sig}} \cdot (4\lceil \log(L) \rceil \beta^2 + 16\beta^2)/2^{128}} \cdot \Big(\Pr[\mathsf{G}_1 = 1] \\
& + Q_{\mathsf{sig}} \cdot 3 \sqrt{\lceil \log(L) \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}_{\mathcal{B}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1]} + 3\sqrt{Q_1} \cdot 2^{-\lambda} + Q_{\mathsf{com}} \cdot 2^{-2\lambda} \Big)
\end{aligned}
\tag{23}
$$

G_4,G_5&G_6: For G_4, G_5, and G_6, instead of adding an abort condition, we just reprogram H_2^2 and H_2^3 to the randomly sampled values for chall_2 and chall_3, and the reprogramming in Equation (9) is also done without abort condition. By the adaptive reprogramming lemma (Proposition 1 in [GHHM21]) we get

$$\Pr[\mathsf{G}_3 = 1] \leq \Pr[\mathsf{G}_4 = 1] + \frac{3}{2} \cdot Q_{\mathsf{sig}} \cdot \sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{8\lambda+64}}},$$

$$\Pr[\mathsf{G}_4 = 1] \leq \Pr[\mathsf{G}_5 = 1] + \frac{3}{2} Q_{\mathsf{sig}} \cdot \sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}}},$$

and

$$\begin{aligned}\Pr[\mathsf{G}_5 = 1] \leq e^{Q_{\mathsf{sig}} \cdot (\beta_{\mathsf{root}}^2 + 4\tau\lceil \log(L) - 1\rceil \beta^2 + \tau a_{\mathsf{leaf}}^2)/2^{128}} \cdot \Big(&\Pr[\mathsf{G}_6 = 1] + \\ &Q_{\mathsf{sig}} \mathsf{AdvPRP}_{\mathcal{B}_{1,1}}^{\mathsf{PRG}_{\mathsf{root}}}[1] + Q_{\mathsf{sig}} \cdot \tau \cdot \Big(\lceil \log(L) - 1\rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}_1}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}_{\mathcal{B}_{2,3}}^{\mathsf{leaf}}\Big) \\ &+ 2^{-\lambda} + Q_{\mathsf{sig}} 2^{-\lambda} + Q_{\mathsf{sig}} \cdot \sqrt{K_{\max} \cdot n_{\mathsf{mult}} \cdot Q_5 \cdot 2^{-(128+\lambda)}} + \lambda\big(2n_{\mathsf{mult}} \cdot Q_{\mathsf{sig}}^2\big) 2^{-(128+\lambda)}\Big)\end{aligned}$$

Now consider the candidate forgery that the adversary outputs in game G_9. We now define two adversaries, an $\mathsf{EUF\text{-}KO}$ adversary $\mathcal{B}$ and a multi-target pre-image finding adversary $\mathcal{C}$ as follows. Both adversaries simulate $\mathcal{A}$ playing game G_9. $\mathcal{B}$ simulates the reprogramming performed by G_9 internally. $\mathcal{C}$ also simulates the reprogramming performed by G_9 internally, but uses pre-image challenges in place of randomly sampled values for reprogramming. Once $\mathcal{A}$ has finished, $\mathcal{C}$ runs $\mathsf{FAEST.Verify}$ on $\mathcal{A}$'s candidate forgery and outputs any preimage it finds among the queries $\mathsf{FAEST.Verify}$ makes. $\mathcal{D}$ proceeds as $\mathcal{C}$, but checks if the queries made by $\mathsf{FAEST.Verify}$ to H_4 collide with any query in the compiled lists. Now suppose $\mathcal{B}$ outputs a candidate forgery that is a successful forgery under the reprogrammed oracle simulated by $\mathcal{B}$ but invalid under the un-reprogrammed oracle that $\mathcal{B}$ has access to. Then one of H_2^0, H_2^1, H_2^2 and H_5 is queried by $\mathsf{FAEST.Verify}$ on an input that was reprogrammed by the simulated signature oracle. The input for H_2^0 includes the message, i.e., it has to be different from all reprogrammed inputs for a valid forgery. There are now three cases left to consider.

1. Suppose $\mathsf{FAEST.Verify}$ queries H_2^1 on a reprogrammed input. Then the forgery message leads to the same value μ as one of the signing oracle queries.
2. Suppose $\mathsf{FAEST.Verify}$ queries H_2^1 on a fresh input but H_2^2 on a reprogrammed input. Then the value chall_1 recomputed by $\mathsf{FAEST.Verify}$ using the fresh H_2^1-query coincides with one generated in a signing query.
3. Suppose $\mathcal{C}$ fails but $\mathsf{FAEST.Verify}$ queries H_5 on a reprogrammed input. After G_3, the values $\mathbf{u}^j$ used in the jth signing query are uniformly random for all j. The proof of the adaptive reprogramming lemma of [GHHM21] actually shows a bound on the trace distance of the joint state of the adversary and the compressed oracle with and without reprogramming. From this bound, it is easy to conclude that from the point of view of the adversary, $\mathbf{u}^j$ is still within $\frac{3}{2}\sqrt{\frac{Q_5 + n_{\mathsf{mult}} Q_{\mathsf{sig}}}{2^{128+\lambda}}}$ total variational distance from uniform and thus has at least $-\log\left(2^{-\lambda} + \frac{3}{2}\sqrt{\frac{Q_5 + n_{\mathsf{mult}} Q_{\mathsf{sig}}}{2^{128+\lambda}}}\right)$ bits of min-entropy. Additionally, by the analysis of G_{11} in the proof of Theorem 10.42, the maximum number of signing queries that have the same iv as the forgery is upper-bounded by 7 except with probability $2^{-\lambda}$. By a union bound, this event thus happens with at most probability $n_{\mathsf{mult}} \cdot 7 \cdot \left(2^{-\lambda} + \frac{3}{2}\sqrt{\frac{Q_5 + n_{\mathsf{mult}} Q_{\mathsf{sig}}}{2^{128+\lambda}}}\right) + 2^{-\lambda} \leq n_{\mathsf{mult}} \cdot 8 \cdot \left(2^{-\lambda} + \frac{3}{2}\sqrt{\frac{Q_5 + n_{\mathsf{mult}} Q_{\mathsf{sig}}}{2^{128+\lambda}}}\right)$ as the value e in the input of H_5 separates the domains of the different queries made in one call of $\mathsf{VOLECommit}$ or $\mathsf{VOLEReconstruct}$.

As the pre-image finding challenges are uniformly random, we can consider $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ in the same probability space. Due to the considerations above we get the equation

$$\Pr[\mathsf{G}_9 = 1] \leq \mathsf{AdvEUFKO}_{\mathcal{B}} + \Pr[\mathcal{C} \text{ succeeds}] + n_{\mathsf{mult}} \cdot 8 \cdot \left(2^{-\lambda} + \frac{3}{2}\sqrt{\frac{Q_5 + n_{\mathsf{mult}} Q_{\mathsf{sig}}}{2^{128+\lambda}}}\right).$$

Using a standard bound [HRS16], we have

$$\Pr[\mathcal{C} \text{ succeeds}] \leq Q_{\mathsf{sig}} \frac{(Q_2^0 + Q_2^1 + 3)^2}{2^{2\lambda}}.$$

Here we have added 1 to the query numbers for H_2^0, H_2^1 to account for the fact that $\mathcal{C}$ runs FAEST.Verify. Collecting all loss terms, including the ones from the games hops that are identical to Theorem 10.42, we get the desired bound. □

We can now combine the above to get the EUF-CMA security of FAEST in the QROM. The part of the corollary below that is about FAEST is a restatement of **??**.

Corollary 10.54 (FAEST is EUF-CMA secure in the QROM). *Let* $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_4$ *and* H_5 *be modeled as quantum-accessible random oracles and* H_3 *be a PRF. Then for any QPT adversary* $\mathcal{A}$ *which makes* $Q_{\mathsf{sig}} \leq 2^{64}$ *queries to* $\mathcal{O}_{\mathsf{sig}}$ *and* $Q_0, Q_1, Q_{2,0}, Q_{2,1}, Q_{2,2}, Q_{2,3}, Q_4$ *and* Q_5 *queries to* $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^0, \mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3, \mathsf{H}_4$ *and* H_5*, respectively, there exists an* EUF-KO *adversary* $\mathcal{B}$ *such that*

$$\begin{aligned}
\mathsf{AdvEUFCMA}_{\mathcal{A}}^{\mathsf{FAEST}}[Q_{\mathsf{sig}}] \leq\ & 2 \cdot \mathsf{AdvEUFKO}_{\mathcal{B}}^{\mathsf{FAEST}} + 2 \cdot Q_{\mathsf{sig}} \cdot \mathsf{AdvPRF}^{\mathsf{H}_3} \\
& + 2 \cdot Q_{\mathsf{sig}} \cdot 3\sqrt{\lceil \log(L) \rceil \cdot \mathsf{AdvPRP}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}^{\mathsf{LeafCommit}}[1] + 3\sqrt{Q_1} \cdot 2^{-\lambda} + Q_{2,1} \cdot 2^{-2\lambda}} \\
& + 2 \cdot Q_{\mathsf{sig}} \cdot \left(\mathsf{AdvPRP}_{\mathcal{B}_{1,1}}^{\mathsf{PRG}_{\mathsf{root}}}[1] + \tau \cdot \left(\lceil \log(L) - 1 \rceil \cdot \mathsf{AdvPRP}_{\mathcal{B}_1}^{\mathsf{PRG}_{2\lambda}}[1] + \mathsf{AdvPRP}_{\mathcal{B}_{2,3}}^{\mathsf{leaf}}\right)\right) \\
& + 2 \cdot Q_{\mathsf{sig}} \cdot \left(\frac{(Q_{2,0} + Q_{2,1} + 3)^2}{2^{2\lambda}} + \frac{3}{2}\sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{8\lambda+64}}} + \frac{3}{2}\sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}}}\right) \\
& + 2 \cdot n_{\mathsf{mult}} \cdot 8 \cdot \left(2^{-\lambda} + \frac{3}{2}\sqrt{\frac{Q_5 + n_{\mathsf{mult}} Q_{\mathsf{sig}}}{2^{128+\lambda}}}\right) \\
& + 2 \cdot \left(2^{-\lambda} + Q_{\mathsf{sig}}\, 2^{-\lambda} + \frac{3}{2} Q_{\mathsf{sig}} \sqrt{\lambda \cdot n_{\mathsf{mult}} \cdot Q_5 \cdot 2^{-(128+\lambda)}} + 2\lambda \cdot n_{\mathsf{mult}} \cdot Q_{\mathsf{sig}}^2 2^{-(128+\lambda)}\right).
\end{aligned}$$

Furthermore,

$$\mathsf{AdvEUFKO}_{\mathcal{B}}^{\mathsf{FAEST}} \leq \frac{1}{1 - \frac{\mathsf{dom(OWF)}}{\mathsf{cod(OWF)}}} \cdot \left(10(\tau+1)Q^3 \cdot 2^{-2\lambda} + 10Q^2(\tau + d_{\mathsf{zk}}) \cdot 2^{-\lambda} + \mathsf{AdvPRG}_{\mathcal{B}_1}^{\mathsf{OWF}}\right).$$

For FAEST-EM*, the same* EUF-CMA *bound holds with* FAEST *replaced by* FAEST-EM *and with the corresponding variant-dependent leaf advantages. Moreover, there exists a QROM adversary* $\mathcal{C}$ *against the almost injectivity of* PRG *such that*

$$\mathsf{AdvEUFKO}_{\mathcal{B}}^{\mathsf{FAEST\text{-}EM}} \leq \frac{1}{1 - \frac{\mathsf{dom(OWF)}}{\mathsf{cod(OWF)}}} \cdot \left(10(\tau+1)Q^3 \cdot 2^{-2\lambda} + 10Q^2\tau \cdot 2^{-\lambda} + \mathsf{AdvPRG}_{\mathcal{B}_1}^{\mathsf{OWF}} + \mathsf{AdvOivInj}_{\mathcal{C}}^{\mathsf{PRG}_{2\lambda}}[L\right.$$

Here,

$$Q = Q_0 + Q_1 + Q_{2,0} + Q_{2,1} + Q_{2,2} + Q_{2,3} + Q_4 + (\tau+4)Q_{\mathsf{sig}} + 3\tau + d_{\mathsf{zk}} + 19,$$

and 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*; the variant-dependent* $\mathsf{AdvPRP}^{\mathsf{leaf}}$ *term is as defined in Lemma 10.51.*

Remark 10.55. We have chosen to instantiate all random oracles via domain separation to facilitate the proof of Lemma 10.49 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 10.44 to multiple distinct quantum-accessible random oracles and applying it to improve Lemma 10.49.

11 Advantages and Limitations

11.1 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, just 3498 bytes for FAEST-EM-128s, smaller than 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.

11.2 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+23. 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.

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.

BDEZ12. Razvan Barbulescu, Jérémie Detrey, Nicolas Estibals, and Paul Zimmermann. Finding optimal formulae for bilinear maps. In *Arithmetic of Finite Fields (WAIFI 2012)*, volume 7369 of *LNCS*, pages 168–186. Springer, 2012.

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.

Dia22. Benjamin E. Diamond. On the security of KOS. Cryptology ePrint Archive, Report 2022/1371, 2022.

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.

HRS16. Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part I*, volume 9614 of *LNCS*, pages 387–416. Springer, Berlin, Heidelberg, March 2016.

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.

KOS15. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Berlin, Heidelberg, August 2015.

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.

KS08. Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Berlin, Heidelberg, July 2008.

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.

Mon05. Peter L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362–369, 2005.

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 $\sum_{i=0}^{7} \zeta_i x^{2^i} + \zeta_8$.

B Deriving the CRT Multiplication Tables

The algorithms in Section 6 rely on precomputed tables $\mathbf{F}, \mathbf{G}, \mathbf{W}_{\text{tree}}, \mathbf{W}_{\text{gate}}, \mathbf{W}_{\text{crt}}$ and the moduli $M_0, \ldots, M_{s-1} \in \mathbb{F}_2[X]$ (all fixed for each parameter set) to perform efficient bilinear multiplication in $\mathbb{F}_{2^\lambda}$. We now explain how these tables are derived, for two purposes:

1. To allow regenerating or auditing the tables from scratch using only the parameters $(\lambda, \tau, w_{\mathsf{grind}})$.
2. As an alternative to using the tables, it's possible to implement the multiplication directly from the moduli. This reduces storage costs and may allow additional optimizations.

In the submission package, we include a reference generator, `vole_mult_tables.py`, which follows these steps to produce the tables. To *verify correctness* of a set of tables, the machinery of this section is not needed. Instead, it suffices to run the self-contained check of Appendix B.6.

B.1 Conventions

As in the ToPoly/ToBits correspondence of Figure 3.4, we identify a polynomial in $\mathbb{F}_2[x]_{<n}$ with its coefficient vector in $\mathbb{F}_2^n$, bit i being the coefficient of x^i. This is how all moduli and matrix rows are stored in the tables (as hexadecimal integers). An $\mathbb{F}_2$-linear form on $\mathbb{F}_2^n$ is likewise a vector $r \in \mathbb{F}_2^n$, applied as the inner product $\langle r, v \rangle$. We write $n_\Delta := \lambda - w_{\mathsf{grind}}$ for the number of non-zero challenge bits, $N := \lambda + n_\Delta - 1 = 2\lambda - w_{\mathsf{grind}} - 1$ for the number of coefficients of the polynomial product $\hat{u}\hat{\Delta}$, where $\deg(\hat{u}) < \lambda$ and $\deg(\hat{\Delta}) < n_\Delta$, and we write $P_\lambda(x)$ for the FAEST field polynomial.

We want to build a *bilinear algorithm* for the map $(\bar{\mathbf{u}}, \Delta') \mapsto \bar{\mathbf{u}} \cdot \Delta$: a triple $(\mathbf{F}, \mathbf{G}, W)$ such that

$$\mathsf{ToBits}(\bar{\mathbf{u}} \cdot \Delta; \lambda) = W \cdot \big((\mathbf{F} \cdot \mathsf{ToBits}(\bar{\mathbf{u}}; \lambda)) * (\mathbf{G} \cdot \Delta')\big),$$

where $*$ is component-wise AND. Recall, Δ is the CRT lift of $\Delta' \in \{0,1\}^{n_\Delta}$ with respect to the moduli $\{M_i\}_i$.

Each pair of rows $(\mathbf{F}[e], \mathbf{G}[e])$ defines one AND gate; the number of such gates is the only quantity that costs signature bytes, since every gate needs one correction bit per

mask, while the $\mathbb{F}_2$-linear maps $\mathbf{F}, \mathbf{G}, W$ are applied for free. The construction below splits W into $\mathbf{W}_{\text{tree}}$ and $\mathbf{W}_{\text{gate}}$: the first τ places are served by the GGM trees at zero gate cost, so their contribution is separated out.

B.2 Places, residues, and the degree budget

A *place* is either a power $q(x)^e$ of a monic irreducible $q \in \mathbb{F}_2[x]$, or the place at infinity taken with some multiplicity m_∞. Each place has a degree and an $\mathbb{F}_2$-linear residue map, which reduces a degree $< N$ polynomial f.

- *Finite place* q^e, degree $e \deg q$: the residue is $f \bmod q^e$. Its matrix over the N input coefficients is obtained by iterating $x^{j+1} = x \cdot x^j \bmod q^e$ for $j \in [0..N)$ and recording, for each j, which residue coordinates the input coefficient f_j feeds.
- *Place at infinity with multiplicity* m_∞, degree m_∞: the residue is the top m_∞ coefficients $(f_{N-1}, \ldots, f_{N-m_\infty})$.

The places used must be distinct and their degrees must sum to N, as in Section 6; the recovery of f from its residues is then the bijection proved there. After fixing the τ tree places from the GGM tree, which contribute $\sum_i d_i = n_\Delta$, the *degree deficit* that the gates must cover is

$$D \;=\; N - n_\Delta \;=\; (2\lambda - w_{\mathsf{grind}} - 1) - (\lambda - w_{\mathsf{grind}}) \;=\; \lambda - 1,$$

independent of w_{grind}. This is why n_{mult} only mildy varies between parameter sets of the same λ: the only difference is the available places left after fixing the τ tree places.

B.3 Cost of a place, and the choice of places

Let $\mu(n)$ be the number of bit multiplications used to multiply two n-coefficient polynomials over $\mathbb{F}_2$ without reduction (giving $2n-1$ coefficients). Let $\mu^{\mathsf{sh}}(n)$ be the count for the same product reduced modulo x^n. For small values of n, we compute each of these using dynamic programming over the constructions below. These are constructive upper bounds on bilinear complexity, and not all optimal: for instance, at $n = 6$, Montgomery also gives a 17-multiplication formula we did not implement, as it did not seem to improve any FAEST parameter set.

Base. $\mu(1) = 1$.

Block split. Viewing an n-coefficient polynomial as an s-coefficient polynomial in $y = x^b$ with $b = \lceil n/s \rceil$-coefficient blocks gives $\mu(n) \le \mu(s) \cdot \mu(b)$, for any $2 \le s < n$. This is Karatsuba/Toom applied recursively.

Montgomery's five-term formula [Mon05]. $\mu(5) \le 13$, using as the linear forms on both sides the 13 index sets over the five input coefficients

$$01234,\ 0234,\ 0124,\ 0134,\ 023,\ 124,\ 34,\ 01,\ 04,\ 4,\ 3,\ 1,\ 0$$

(each denoting the sum of the listed coefficients). This formula is optimal [BDEZ12].

CRT over small places. The full product of two n-coefficient inputs may itself be recursively computed by CRT decomposition using sub-places, with target degree $2n-1$.

$\mathbb{F}_4$ tower. This construction does not give a full product algorithm, but we use it directly for places of even degrees in $\{6, 8\}$. For an irreducible q of even degree $d = 2m$, write $\mathbb{F}_{2^d} = \mathbb{F}_4[x]/Q(x)$ with $\deg Q = m$ and multiply over $\mathbb{F}_4$: a full product of m-term polynomials over $\mathbb{F}_4$ needs only $2m-1$ $\mathbb{F}_4$-multiplications for $m \le 3$ (evaluation at $0, 1, \omega, \omega^2, \infty$) and 8 for $m = 4$ (those five places plus one degree-2 place of the projective line). Each $\mathbb{F}_4$-multiplication is 3 bit multiplications, and the remaining $\mathbb{F}_4$-linear parts are all $\mathbb{F}_2$-linear and hence free. The residue modulo q therefore costs 15 gates at degree 6 and 24 at degree 8. Note that this does *not* give $\mu(6) = 15$, since we only obtain the product modulo q and not the full unreduced result.

Short products. $\mu^{\mathsf{sh}}(n) \le \mu(n)$ by truncation, and splitting $A = A_0 + x^m A_1$ with any $m \ge \lceil n/2 \rceil$ kills the $A_1 B_1$ term modulo x^n, giving $\mu^{\mathsf{sh}}(n) \le \mu(m) + 2\mu^{\mathsf{sh}}(n-m)$; the generator minimises over m.

The resulting values are

n	1	2	3	4	5	6	7	8	9	10	11	12
$\mu(n)$	1	3	6	9	13	18	22	26	30	35	39	44
$\mu^{\mathsf{sh}}(n)$	1	3	5	8	11	15	19	23	28	32	36	40

and the cost of a place is then

$$\mathsf{cost}(q^e) = \begin{cases} \mu^{\mathsf{sh}}(e) & q \in \{x,\, x+1\}, \\ 15 & \deg q = 6,\ e = 1, \\ 24 & \deg q = 8,\ e = 1, \\ \mu(e \deg q) & \text{otherwise}, \end{cases} \qquad \mathsf{cost}(\infty^{m_\infty}) = \mu^{\mathsf{sh}}(m_\infty).$$

The two linear places and the place at infinity are cheap because their residues are short products: for x^e the residue is the bottom e coefficients; for ∞^{m_∞} the top m_∞, by the reversal identity of Section 6; and for $(x+1)^e$ the substitution $y = x + 1$ turns the residue into the bottom e coefficients in the shifted basis.

Choosing the extra places is then an exact multiple-choice knapsack: pick at most one multiplicity per place, so that the degrees sum to at least the deficit D, minimising the total cost. Two properties of the places found are worth noting. Firstly, a power q^e is never preferable to e distinct irreducibles of the same degree, since already at $d = 2$, we have $\mu(ed) > e\,\mu(d)$; higher multiplicities are therefore only ever taken when we run out of distinct places at the necessary degree, which happens for x, $x+1$, ∞, and at $x^2 + x + 1$, the unique irreducible of degree 2, whose square appears in every portfolio. Secondly, the tree moduli must be excluded from the pool, which is why two parameter sets with the same λ can end up with different gate counts.

B.4 The linear maps

We build the following three families of $\mathbb{F}_2$-linear maps.

Residue maps. For a modulus q and an input width n, the matrix of $f \mapsto f \bmod q$ is obtained by the loop already described in Appendix B.2. These are used both to feed a place algorithm from the full-width inputs (widths λ and n_Δ) and to express a place's residue as a function of the N product coefficients.

The CRT lift $\mathbf{W}_{\mathrm{crt}}$. For each tree modulus M_i, let $Q_i := M_{\mathrm{tree}}/M_i$ and let $E_i := Q_i \cdot (Q_i^{-1} \bmod M_i) \bmod M_{\mathrm{tree}}$ be the corresponding CRT idempotent, so that $E_i \equiv 1 \pmod{M_i}$ and $E_i \equiv 0 \pmod{M_j}$ for $j \ne i$. The column of $\mathbf{W}_{\mathrm{crt}}$ belonging to residue bit b of tree i is $E_i \cdot x^b \bmod M_{\mathrm{tree}}$. By construction $\mathbf{W}_{\mathrm{crt}} \cdot \Delta'$ is the unique representative of degree less than n_Δ with the prescribed residues, which is the definition of Δ used in Section 6.

Interpolation. Stack the residue maps of all places — tree places first, then the extra places in a fixed order — as rows over the N product coefficients. This stack has rank N, so admits a left inverse X computed by Gaussian elimination, where row c of X expresses coefficient c of the product as an XOR of residue coordinates. Composing X with the reduction map modulo $P_\lambda(x)$ finishes the job.

B.5 Assembling F, G, $\mathbf{W}_{\text{tree}}$ and $\mathbf{W}_{\text{gate}}$

Given λ, w_{grind}, the field polynomial $p := P_\lambda$, the tree moduli $M_0, \dots, M_{\tau-1}$ and the chosen extra places $\mathcal{P} = (P_0, \dots, P_{s-\tau-1})$ (in portfolio order, see Remark B.1), the tables are built as follows. Write $n_{\mathsf{tree}} := \sum_i \deg M_i = n_\Delta$.

1. *Evaluation rows.* Initialise the list ev with the residue rows of $M_0, \dots, M_{\tau-1}$ over N columns, in tree order. These are the first n_{tree} residue coordinates and cost no gates.
2. *Place algorithms.* For each extra place $P \in \mathcal{P}$ in order, build its bilinear algorithm on inputs of widths (λ, n_Δ): take the underlying full or short product algorithm of Appendix B.3 and pre-compose its left and right linear forms with the residue maps of P at those two widths. Append its gates to the global gate list, record its output rows as masks over the global gate indices, and append its residue rows over N columns to ev.
3. *Interpolate and reduce.* Compute the left inverse X of ev and the reduction rows R of $f \mapsto f \bmod p$ over N columns. For each output coordinate $r \in [0..\lambda)$, form the residue selector $\sigma_r := \bigoplus_{c \in R[r]} X[c]$ and split it:

$$\mathbf{W}_{\text{tree}}[r] := \sigma_r \cap [0..n_{\mathsf{tree}}),$$
$$\mathbf{W}_{\text{gate}}[r] := \bigoplus_{j \in \sigma_r,\ j \geq n_{\mathsf{tree}}} \big(\text{output mask of residue coordinate } j\big).$$

   Thus $\mathbf{W}_{\text{tree}}$ merely *selects* tree residue bits, while $\mathbf{W}_{\text{gate}}$ expands the extra places' residue coordinates into gate outputs.
4. *Left rows.* $\mathbf{F}[e]$ is the left linear form of gate e, already over the λ coefficients of the mask.
5. *Right rows.* The gates' right forms live over the n_Δ coefficients of the *lifted* Δ. Compose them with $\mathbf{W}_{\text{crt}}$ to obtain rows acting on the residue bits Δ' directly: $\mathbf{G}[e][j] := \langle \text{right form of gate } e,\ \mathbf{W}_{\text{crt}}[\cdot, j] \rangle$.
6. *Prune.* Discard gates whose left or right form is zero, merge gates with identical $(\mathbf{F}[e], \mathbf{G}[e])$ into their first occurrence, discard gates referenced by no row of $\mathbf{W}_{\text{gate}}$, and renumber the survivors keeping their relative order. For every FAEST parameter set this step removes nothing, so the gate order is exactly the concatenation over places in portfolio order.

B.6 Verification

Both sides of (3) are bilinear in $(\bar{\mathbf{u}}_m, \Delta')$. To completely verify the equation, it therefore suffices to check it on the $\lambda \cdot (\lambda - w_{\mathsf{grind}})$ basis pairs $(\bar{\mathbf{u}}_m, \Delta') = (x^i, e_j)$ using standard $\mathbb{F}_2[x]$ arithmetic. Concretely, for each pair, compute $\Delta := \mathsf{ToField}(\mathbf{W}_{\text{crt}} \cdot e_j, \lambda)$, form $x^i \cdot \Delta \bmod P_\lambda(x)$ directly, and compare against $\mathbf{W}_{\text{tree}} \cdot \mathsf{res}_\tau(x^i, \Delta) \oplus \mathbf{W}_{\text{gate}} \cdot ((\mathbf{F} \cdot x^i) * (\mathbf{G} \cdot e_j))$. Alongside this, one should also check (1) that $\mathbf{W}_{\text{crt}}$ really is the CRT lift of the tree moduli (column j reduces to x^b modulo its own M_i and to 0 modulo every other), and (2) that $M_{\text{tree}} = \prod_i M_i$. Our generator script performs these verification steps for every parameter set.

B.7 Implementing the product without the tables

Instead of using the $\mathbf{F}$ and $\mathbf{G}$ matrices, an implementation can directly compute the two vectors of n_{mult} bits to be ANDed together: the signer needs $(f_e(\bar{\mathbf{u}}_m))_e$ and the verifier needs $(g_e(\Delta))_e = (\mathbf{G}[e] \cdot \Delta')_e$. By following the recursive structure of the bilinear algorithm, they can be evaluated with $O(\lambda \log \lambda)$ word operations instead of $O(\lambda n_{\mathsf{mult}})$ bit operations for the dense matrix product. Concretely, for each mask:

1. *Signer.* For each extra place P in portfolio order, reduce $\hat{u}_m$ modulo P, then run the left half of P's bilinear algorithm on the residue, emitting one bit per gate in the algorithm's own order. Concatenating over places gives $(f_e(\bar{\mathbf{u}}_m))_e$.
2. *Verifier.* Lift Δ' to $\Delta = \mathbf{W}_{\mathrm{crt}} \cdot \Delta'$ once — either using $\mathbf{W}_{\mathrm{crt}}$ or directly from the formula — and run the same loop with the right halves to obtain $(\gamma_e)_e$.
3. *Recombination.* Both parties feed their gate shares and their tree residue shares through $\mathbf{W}_{\mathrm{tree}}, \mathbf{W}_{\mathrm{gate}}$. These matrices are smaller ($\lambda \times n_\Delta$ and $\lambda \times n_{\mathsf{mult}}$) and may be worth precomputing, but we can alternatively run the interpolation of Appendix B.4 on the shares directly.

Remark B.1 (Ordering in the portfolio). One aspect that cannot change is the ordering of the AND gates. We fix this by: places in portfolio order (the place at infinity, then x^e, then $(x+1)^e$, then irreducibles by increasing degree and, within a degree, lexicographically by their coefficient vector, skipping those used as tree moduli); within a place, the gate order of the recursive construction; and then the pruning rule of Appendix B.5, which for every FAEST parameter set is the identity. An implementation that derives $\mathbf{F}$ and $\mathbf{G}$ on the fly must reproduce this order exactly, but this can be easily verified by comparing its output against the tables offline.

B.8 Parameter set data

Table B.1 lists, for each parameter set, the data that fixes the construction. The tree degrees are $d_i = k$ for $i < \tau_1$ and $k-1$ otherwise, with $k = \lfloor (\lambda - w_{\mathsf{grind}})/\tau \rfloor + 1$ and $\tau_1 = (\lambda - w_{\mathsf{grind}}) \bmod \tau$; the tree moduli are the lexicographically smallest distinct irreducibles of those degrees. Note that the eight sets fall into only four cost profiles: 312 gates at $\lambda = 128$, 504 at $\lambda = 192$, and 696 or 704 at $\lambda = 256$. FAEST-EM shares its tables with the corresponding FAEST variant except at $\lambda = 192$, where both EM variants have their own parameters.

set	λ	τ	w_{grind}	tree degrees	n_Δ	m_∞	#places	n_{mult}	$\bar{n}_{\mathsf{mult}}/8$
128s	128	11	7	11×11	121	3	25	312	39
128f	128	17	8	$1 \times 8 + 16 \times 7$	120	3	25	312	39
192s	192	16	12	$4 \times 12 + 12 \times 11$	180	3	33	504	63
192f	192	24	8	$16 \times 8 + 8 \times 7$	184	3	33	504	63
192s_em	192	16	8	$8 \times 12 + 8 \times 11$	184	3	33	504	63
192f_em	192	25	8	$9 \times 8 + 16 \times 7$	184	3	33	504	63
256s	256	22	6	$8 \times 12 + 14 \times 11$	250	3	41	696	87
256f	256	33	8	$17 \times 8 + 16 \times 7$	248	3	41	704	88

Table B.1: Places and gate counts per parameter set. "#places" counts the extra places only, excluding the τ tree places; the last column is the per-mask correction row of $\mathbf{c}_{\mathsf{mult}}$ in bytes.

For concreteness, the portfolio at $\lambda = 128$ consists of the place at infinity ∞^3 at 5 gates, the linear places x^5 and $(x+1)^5$ at 11 gates each, the squared place $(x^2+x+1)^2$ at 9 gates, and 21 irreducibles used once: $2, 3, 6, 9$ of degrees $3, 4, 5, 6$ and one of degree 8, costing

$$5 \;+\; 2 \cdot 11 \;+\; 9 \;+\; 2 \cdot 6 \;+\; 3 \cdot 9 \;+\; 6 \cdot 13 \;+\; 9 \cdot 15 \;+\; 24 \;=\; 312$$

gates in total.