FAEST: Algorithm Specifications

Version 1.0

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

[1] Aarhus University
[2] Technical University of Denmark
[3] imec-COSIC, KU Leuven
[4] Aalto University
[5] TU Graz
[6] AIT Austrian Institute of Technology
[7] Bocconi University

1 June 2023

Table of Contents

1 Introduction . . . 3
2 Overview of Algorithms and Main Parameters . . . 4
2.1 Main Parameters . . . 4
2.2 Overview of VOLE-in-the-Head and VOLE Commitments . . . 8
2.3 QuickSilver: a VOLE-based Zero-Knowledge Proof System . . . 11
3 Preliminaries . . . 14
3.1 Notation . . . 14
3.2 Data Types and Conversions . . . 14
3.3 Cryptographic Primitives . . . 16
3.4 Security Definitions . . . 17
4 Additional Building Blocks . . . 17
4.1 AES and Rijndael . . . 18
4.2 Universal Hashing . . . 23
5 VOLE-in-the-Head Functions . . . 26
5.1 All-but-One Vector Commitments . . . 26
5.2 Seed Expansion and Conversion to VOLE . . . 28
5.3 VOLEitH: Commitment and Reconstruction . . . 29
6 AES Functions . . . 32
6.1 Witness Extension . . . 32
6.2 Deriving Constraints for the Key Expansion Routine . . . 33
6.3 Deriving Constraints for the Encryption Routine . . . 38
6.4 Proving and Verifying AES Constraints . . . 42
7 Rijndael-EM Functions . . . 46
7.1 Witness Extension . . . 46
7.2 Deriving Constraints for the Encryption Routine . . . 46
7.3 Proving and Verifying Rijndael-EM Constraints . . . 51
8 The FAEST Signature Scheme . . . 54
8.1 Key Generation . . . 54
8.2 Signing . . . 54
8.3 Verification . . . 56
9 Performance Analysis . . . 58
10 Security Evaluation . . . 60
10.1 Provable Security . . . 60
10.2 Concrete Attacks . . . 79
10.3 Concrete Analysis of AES as a OWF . . . 81
11 Advantages and Limitations . . . 86
11.1 Advantages . . . 86
11.2 Limitations . . . 87
A Finite Field Generator Elements . . . 91

1 Introduction

This document describes and specifies 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 standard PRFs and PRGs for randomness derivation, 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, k is a secret key and x is a plaintext block 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)[8] [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 the QuickSilver information-theoretic proof system [YSWW21] is made non-interactive by the Fiat–Shamir transform [FS87] and proven secure in the random oracle model (ROM). (Evidence of FAEST's security in the quantum-accessible ROM (QROM) is also provided.)

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, and instantiating E with the standardized primitive of AES [AES01]. 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 block cipher 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.

[8] VOLE stands for vector oblivious linear evaluation.

Outline of this document. Section 2 provides a detailed overview of the VOLEitH tool and of the QuickSilver proof system; it also lists the specified parameter sets for the FAEST signature algorithm. Section 3 presents preliminaries for this document: the notation, the data types and their conversions, the elementary cryptographic primitives and the security definitions. Section 4 describes two additional components: the AES and Rijndael algorithms and universal hashing algorithms.

After these introductory sections, the document specifies the FAEST signature algorithm. Section 5 specifies the VOLEitH construction. Section 6 specifies the computation of the QuickSilver proof of the AES circuit in FAEST. Section 7 specifies the computation of the QuickSilver proof for the Even-Mansour variant of the one-way function. Section 8 specifies the key generation, signature and verification algorithms for FAEST.

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.

2 Overview of Algorithms and Main Parameters

This section presents the main algorithms used for the FAEST signature scheme to provide context for the parameter sets that we define.

2.1 Main Parameters

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 explain the meaning of the main parameters, we first give some background on the VOLEitH paradigm.

2.1.1 VOLE and VOLE-in-the-Head. FAEST uses *VOLE correlations* over a finite field $\mathbb{F}_{2^k}$, to use as a form of linearly homomorphic commitment scheme. A VOLE (vector oblivious linear evaluation) correlation of length ℓ is defined by a random global key $\Delta \in \mathbb{F}_{2^k}$, a set of random bits $u_i \in \mathbb{F}_2$, random VOLE tags $v_i \in \mathbb{F}_{2^k}$ and VOLE keys $q_i \in \mathbb{F}_{2^k}$ such that:[9]

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

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.

A VOLE correlation 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^{-k}. 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].

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

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, such that Equation (1) holds for Δ, 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.

2.1.2 Parameters for VOLEitH. To realise a single instance of VOLE which produces random values in the small field $\mathbb{F}_2$ and VOLE tags and keys in a larger field $\mathbb{F}_{2^k}$ using our VOLEitH construction requires the signer to perform $O(2^k)$ work. To achieve security in the zero-knowledge protocol, we require VOLE correlations with tags and keys in $\mathbb{F}_{2^\lambda}$ which is infeasible to realise with a single instance parameterised with $k = \lambda$.

Instead, we run several parallel instances of the VOLEitH protocol over smaller fields and concatenate the VOLE tags and keys that they produce. This creates VOLE correlations over the exponentially large field $\mathbb{F}_{2^\lambda}$ with only a polynomial amount of work. Namely, for each parameter set we select a repetition parameter $\tau \in \mathbb{N}$ and derive two length parameters $k_0, k_1 \in \mathbb{N}$ such that

$$k_0 := \lceil \lambda/\tau \rceil \qquad \text{and} \qquad k_1 := \lfloor \lambda/\tau \rfloor .$$

These will set the field sizes of our small VOLE instances. We also derive two repetition parameters $\tau_0 := (\lambda \bmod \tau)$ and $\tau_1 := \tau - \tau_0$, such that

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

ensuring that concatenating the outputs of τ_0 instances of VOLEitH for $\mathbb{F}_{2^{k_0}}$ and τ_1 instances for $\mathbb{F}_{2^{k_1}}$ produces VOLE correlations in $\mathbb{F}_{2^\lambda}$ exactly.

Selecting only a single k would have implied either choosing a value of k that divides λ and limiting a feasible k to $\{2, 4, 8, 16\}$, or choosing a more suitable value of k (around 11 or 12) but producing VOLE correlations over fields with bigger bit-length than λ, therefore leading to wasted work.

Letting τ be an arbitrary integer, as long as the resulting k_0 and k_1 still imply feasible runtimes for the signer, offers tradeoffs between signature size and speed. A small τ means computing fewer VOLEitH protocols and hence a smaller signature size (because signature size scales in the number of VOLE instances), but at the cost of larger values for k_0 and k_1 and hence more work for the signer and verifier. On the other hand, increasing τ implies both larger signature sizes as well as less work for the signer and verifier.

2.1.3 Parameters for the Signature Scheme. The key generation algorithm of FAEST uses a block-cipher one-way function family $\{F_k\}_k$ such that $F_k(x) = E_k(x)$, where $E_k(x)$ denotes the encryption of a plaintext block x under key k. The OWF relation $\mathcal{R}$ is defined as $((x, y), k) \in \mathcal{R} \iff E_k(x) = y$. Clearly, the security of the resulting OWF is based directly on the security of the block cipher E against

Scheme	OWF $F_k(x)$	ℓ	τ	k_0	k_1	B	sizes (bytes)	
							pk	sig.
FAEST-128s	$\mathsf{AES128}_k(x)$	1600	11	12	11	16	32	5 006
FAEST-128f		1600	16	8	8	16	32	6 336
FAEST-192s	$\mathsf{AES192}_k(x_0)\|\mathsf{AES192}_k(x_1)$	3264	16	12	12	16	64	12 744
FAEST-192f		3264	24	8	8	16	64	16 792
FAEST-256s	$\mathsf{AES256}_k(x_0)\|\mathsf{AES256}_k(x_1)$	4000	22	12	11	16	64	22 100
FAEST-256f		4000	32	8	8	16	64	28 400
FAEST-EM-128s	$\mathsf{AES128}_x(k) \oplus k$	1280	11	12	11	16	32	4 566
FAEST-EM-128f		1280	16	8	8	16	32	5 696
FAEST-EM-192s	$\mathsf{Rijndael192}_x(k) \oplus k$	2304	16	12	12	16	48	10 824
FAEST-EM-192f		2304	24	8	8	16	48	13 912
FAEST-EM-256s	$\mathsf{Rijndael256}_x(k) \oplus k$	3584	22	12	11	16	64	20 956
FAEST-EM-256f		3584	32	8	8	16	64	26 736

Table 2.1: 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_0, k_1 are the bit lengths of the small VOLEs; B is a padding parameter affecting security of the VOLE check

key recovery attacks with a single known plaintext/ciphertext pair. We refer to Section 10 for a more detailed security analysis.

We distinguish between the parameters for the FAEST signature scheme and the parameters for the used one-way function. In Table 2.1, we give parameters for FAEST for 3 security levels L1, L3, L5, corresponding to the security of AES-128, AES-192 and AES-256, respectively. For each security level, we give one set of parameters ending with 's'—achieving short signatures—and another ending with 'f'—for the speed-optimized variant. Note that Table 2.1 is divided in two parts according to the instantiation of the OWF used in the signature scheme.

We provide two instantiations of the OWF, the first—more standard one—with AES, that is $E_k(x) = \mathrm{AES}_k(x)$. This corresponds to the first part of the table. Due to the 128-bit block size of AES, we need to use $\beta = 2$ blocks of AES-192 and AES-256 encryptions to ensure security of the corresponding OWF of 192 and 256 bits, while for AES-128, $\beta = 1$ suffices. The second part of the table uses the single-key Even–Mansour (EM) construction, based on fixed-key AES. The EM variant uses a OWF F obtained from a public cryptographic permutation π by adding a key k both to the input x of the OWF and to the output of the permutation, i.e.,

$$F_k(x) = k + \pi(x + k),$$

where π is instantiated with AES by fixing a random key p_0, and making p_0 a public constant. This allows to avoid calculating the non-linear key schedule with a secret key in the zero-knowledge proof. We write $F : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^n$, where n is the block size, and λ is the key size and $\pi : \{0,1\}^n \to \{0,1\}^n$. We require that $n = \lambda$ and addition of n-bit strings is done with bitwise XOR. For security levels 3 and 5, there is no direct way to use AES as π directly for the EM variant, so we instead use the Rijndael cipher [DR98] (of which AES is a special case) as it offers a flexible block size that we set to be equal to the key length.

AES OWF Parameters. Table 2.2 describes parameters for the sets using AES directly to construct the OWF (i.e. not the EM variant). It includes for example the number of rounds and S-Boxes in the encryption routine, R and S_{enc}, and the number of S-boxes in the key expansion routine, S_{ke}, as well as an integer $\beta \in \{1,2\}$ used to indicate the number of AES blocks according to the security level.

Param. λ	Formula	Description	FAEST 128	FAEST 192	FAEST 256
N_{k}	$\frac{\lambda}{32}$	no. 32-bit words in an AES key	4	6	8
R	$N_{\mathsf{k}} + 6$	no. AES encryption rounds	10	12	14
S_{ke}	$-\frac{\lambda}{8} + 56 + 28\left\lfloor\frac{\lambda}{256}\right\rfloor$	no. S-Boxes in key expansion	40	32	52
S_{enc}	$16R$	no. S-Boxes in encryption	160	192	224
ℓ	$\ell_{\mathsf{ke}} + \beta\ell_{\mathsf{enc}}$	length of extended witness in bits	1 600	3 264	4 000
ℓ_{ke}	$\lambda + 8S_{\mathsf{ke}}$	no. witness bits for key expansion	448	448	672
ℓ_{enc}	$8(S_{\mathsf{enc}} - 16)$	no. witness bits for encryption	1 152	1 408	1 664
β	$\left\lceil\frac{\lambda}{128}\right\rceil$	no. message blocks	1	2	2
C	$S_{\mathsf{ke}} + \beta \cdot S_{\mathsf{enc}}$	no. $\mathbb{F}_{2^8}$ mult. constraints	200	416	500

Table 2.2: AES-specific parameters for the FAEST-λs/f instances

Rijndael-EM OWF Parameters. Table 2.3 describes parameters for the sets using the Even-Mansour variant of the OWF construction from AES128 and Rijndael. The main difference is the number N_{st} of 32-bit words in the state of the block cipher, which is fixed to $N_{\mathsf{st}} = 4$ for the AES algorithm but can vary for the Rijndael algorithm. This also includes the number of rounds R, the number of S-boxes in the encryption phase S_{enc}, and the witness length in bits ℓ.

Param. λ	Formula	Description	FAEST-EM 128	FAEST-EM 192	FAEST-EM 256
N_{k}	$\frac{\lambda}{32}$	no. 32-bit words in an EM variant key	4	6	8
N_{st}	$\frac{\lambda}{32}$	no. 32-bit words in an EM variant block	4	6	8
R	$\max(N_{\mathsf{k}}, N_{\mathsf{st}}) + 6$	no. encryption rounds	10	12	14
S_{enc}	$4 \cdot N_{\mathsf{st}} \cdot R$	no. S-Boxes in the EM variant OWF	160	288	448
ℓ	$\lambda + 8(S_{\mathsf{enc}} - 4N_{\mathsf{st}})$	length of extended witness in bits	1 280	2 304	3 584
C	S_{enc}	no. $\mathbb{F}_{2^8}$ mult. constraints	160	288	448

Table 2.3: Rijndael-EM specific parameters for the FAEST-EM-λs/f instances

Passing parameters. For this specification document, to avoid passing the fixed parameters explicitly to the algorithms that require them, we define the following shorthands. For the parameters related to the VOLEitH construction, we define

$$\texttt{param} := (\lambda, \ell, \tau, k_0, k_1, \tau_0, \tau_1, B, \beta).$$

For the parameter sets that construct the OWF directly from AES, we define

$$\texttt{param}_{\mathsf{OWF}} := (N_{\mathsf{k}}, R, S_{\mathsf{ke}}, S_{\mathsf{enc}}, \ell, \ell_{\mathsf{ke}}, \ell_{\mathsf{enc}}, \beta, C).$$

For the parameter sets that construct the OWF using AES in EM variant, we define

$$\mathtt{param}_{\mathsf{OWF}} := (N_{\mathsf{k}}, N_{\mathsf{st}}, R, S_{\mathsf{enc}}, \ell, C).$$

2.2 Overview of VOLE-in-the-Head and VOLE Commitments

Recall that the goal of VOLEitH is for the signer to be committed to a pair of random vectors $\mathbf{u} \in \{0,1\}^\ell$ and $\mathbf{v} \in \mathbb{F}_{2^\lambda}^\ell$, such that later on, after learning a random challenge $\Delta \in \mathbb{F}_{2^\lambda}$, the signer can open these vectors to the vector

$$\mathbf{q} = \mathbf{u} \cdot \Delta - \mathbf{v} \in \mathbb{F}_{2^\lambda}^\ell,$$

where $\cdot$ means that each entry of $\mathbf{u}$ is multiplied with Δ, component-wise.

As mentioned in the previous section, we will build up to the above by creating several smaller VOLEitH instances over finite fields $\mathbb{F}_{2^{k_0}}$ and $\mathbb{F}_{2^{k_1}}$, and concatenating these to build the final VOLE correlation in $\mathbb{F}_{2^\lambda}$. To ensure the same $\mathbf{u}$ vector is used across the different instances, we also add a consistency check. This approach is based on the SoftSpokenVOLE protocol [Roy22], which was adopted to VOLEitH in [BBD⁺23]. 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]. We model this as 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. The idea is to build a complete binary tree with N 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, each leaf value k_i is put into a hash function H_0, which outputs two new values sd_i and $\overline{\mathsf{com}}_i$. The seed sd_i is the i-th committed seed, which will later be expanded and converted into the VOLE values, while $\overline{\mathsf{com}}_i$ is an auxiliary commitment for the seed. The final commit to seeds $\mathsf{sd}_0, \ldots, \mathsf{sd}_{N-1}$ is computed as $h = \mathsf{H}_1(\overline{\mathsf{com}}_0, \ldots, \overline{\mathsf{com}}_{N-1})$, for another hash function H_1. We illustrate this for $N = 8$ in Figure 2.4.

To open all-but-one of the seeds, say, all except sd_j, the signer takes the siblings of all nodes on the path from the root to leaf j (excluding the root node) and sends these to the verifier, together with the commitment $\overline{\mathsf{com}}_j$. The verifier can then reconstruct every sd_i for $i \neq j$ using the sibling nodes, and also has enough information to compute the hash value h and check the original commitment. The formal pseudocode for these procedures is given in Section 5.1, and their security is shown in Section 10.1.1.

From Vector Commitments to VOLE. To obtain our desired VOLE commitments, the signer starts by committing to τ all-but-one vector commitments, of which τ_0 are of length $N_0 = 2^{k_0}$ and τ_1 of length $N_1 = 2^{k_1}$. Instead of sending a separate hash value h for every commitment, the signer hashes all τ of these into a single hash. The next step is to convert each vector commitment into a length-ℓ, small-field VOLE correlation in $\mathbb{F}_{2^k}$, where k is either k_0 or k_1. This involves first expanding each of

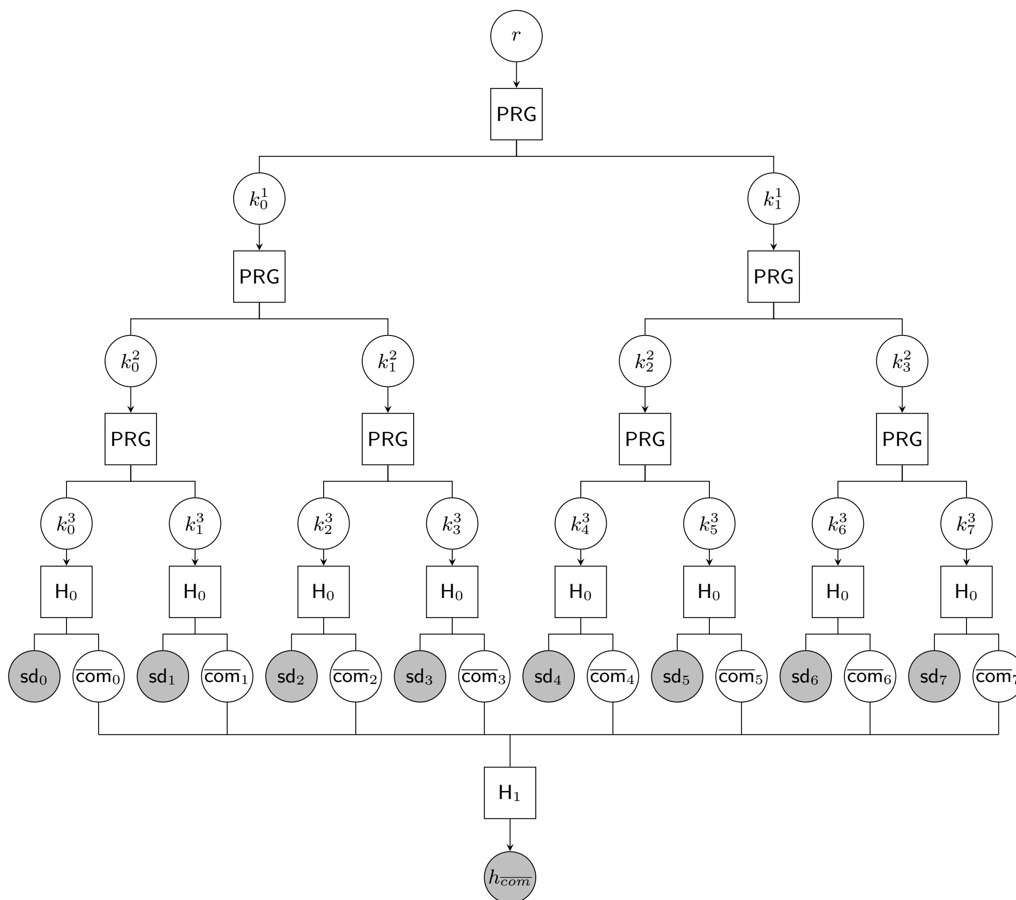

Fig. 2.4: The tree-based all-but-one vector commitment scheme. The committed messages are the seeds sd_i, and the commitment is the hash $h_{\overline{\mathsf{com}}}$

the seeds $\mathsf{sd}_0, \dots, \mathsf{sd}_{N-1}$ (committed in the vector commitment) using a PRG, to obtain N strings $\mathbf{r}_0, \dots, \mathbf{r}_{N-1} \in \{0,1\}^\ell$, and then computing, 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 $j^* \in \{0, \dots, N-1\}$ (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 correlation in $\mathbb{F}_{2^k}$. This step is specified in Section 5.2, which uses an optimized divide-and-conquer algorithm for computing $\mathbf{u}, \mathbf{v}$ and $\mathbf{q}$ with fewer XOR operations.

After doing this for each of the vector commitments, the signer has τ small, independent VOLE correlations. Denote these as $(\mathbf{u}_i, \mathbf{V}_i)$ for $i = 0, \dots, \tau - 1$, where we now view $\mathbf{V}_i$ as a matrix in $\{0,1\}^{\ell \times k}$, instead of a vector in $\mathbb{F}_{2^k}^\ell$. The verifier will eventually learn $(\Delta_i, \mathbf{Q}_i)$ such that

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

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

But first, 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 $\mathbf{u}_0$. Once this has been done, the parties can simply concatenate the τ VOLE instances, by forming the $\ell \times \lambda$ matrices

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

Viewing each row of the above as an element of $\mathbb{F}_{2^\lambda}$, these form a VOLE relation over $\mathbb{F}_{2^\lambda}$, where the new Δ value is formed from the bits of each Δ_i.

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

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

The verifier then computes $\widetilde{\mathbf{Q}} = \mathbf{H}\mathbf{Q}$ and checks that the VOLE relation still holds between $\widetilde{\mathbf{u}}, \widetilde{\mathbf{V}}$ and $\widetilde{\mathbf{Q}}$. One small optimization is that, instead of having the

signer send $\widetilde{\mathbf{V}}$ directly, we send a collision-resistant hash of this. This saves some communication as $\widetilde{\mathbf{V}}$ is quite large, and it still allows verification, since the verifier can simply compute what $\widetilde{\mathbf{V}}$ should be (from $\widetilde{\mathbf{Q}}, \varDelta$ and $\widetilde{\mathbf{u}}$) and check that its hash matches the collision-resistant hash sent by the signer. The details of the universal hash family we use are given in Section 4.2.

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 $\hat{\ell} = \ell + 2\lambda + B$, because the VOLE consistency check reveals a $(\lambda + B)$-bit linear function of $\mathbf{u}$ to the verifier, which needs to hide the underlying witness. After the check, we discard the additional bits and use the length $\ell + \lambda$ VOLE for the ZK proof stage. In the ZK proof, the first ℓ bits of $\mathbf{u}$ are used to commit to the extended witness, while the final λ bits will be used as a mask for a second consistency check.

2.3 QuickSilver: a VOLE-based Zero-Knowledge Proof System

The QuickSilver protocol of Yang, Sarkar, Weng and Wang [YSWW21] is an interactive zero-knowledge proof of knowledge protocol in the VOLE-hybrid model. It is based on the Line-Point Zero-Knowledge paradigm of Dittmer et al. [DIO21].

For the FAEST signature scheme, we use the QuickSilver protocol as specified for arithmetic circuits over any finite field [YSWW21, Section 4] to prove circuits derived from the Advanced Encryption Standard (AES) algorithm [AES01]. This section describes the QuickSilver proof system and provides motivation for the soundness and zero-knowledge guarantees that it provides.

As QuickSilver is an interactive proof system, 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.

2.3.1 Computing with Homomorphic Commitments. As described in Section 2.1.1, secure VOLE protocols provide a list of tuples (u_i, v_i, q_i) such that the VOLE relation of Equation (1) holds for the global key $\varDelta$. One such tuple is referred to as an *information-theoretic message authentication code (MAC)* on the value u_i under the global key $\varDelta$, since the binding property implies that u_i cannot be modified (for a fixed q_i) without knowledge of $\varDelta$.

Linear homomorphism. As mentioned, the information-theoretic MACs produced by the VOLE protocol under the same global key are additively homomorphic: given n such MACs (u_i, v_i, q_i) and $n+1$ constants $c, c_1, \ldots, c_n \in \mathbb{F}_2$, the prover and verifier can obtain a MAC on the value $u' = c + \sum_{i=1}^{n} c_i \cdot u_i$ by computing the VOLE tag v' and the VOLE key q' as follows:

$$v' = \sum_{i=1}^{n} c_i \cdot v_i \quad \text{and} \quad q' = c \cdot \varDelta + \sum_{i=1}^{n} c_i \cdot q_i.$$

Note the absence of c in the computation of v' for the prover, and the factor of $\varDelta$ multiplied to c for the verifier.

Authenticated constants. When it is necessary to generate a standalone deterministic VOLE MAC for a constant $c \in \mathbb{F}_2$ known to both parties, we let the prover set $v := 0$ and the verifier set $q := c \cdot \Delta$.

2.3.2 The QuickSilver Protocol for Circuits. Being defined in the VOLE-hybrid model, the QuickSilver protocol assumes access to a secure VOLE protocol. We instead use VOLEitH, and have the prover send all of the QuickSilver proof messages, before revealing the verifier's challenge Δ.

Let C denote an arithmetic circuit over $\mathbb{F}_2$, containing t multiplication gates, for which the prover knows an input (i.e. the witness) $\boldsymbol{w} \in \mathbb{F}_2^n$ of length n, such that $\mathsf{C}(\boldsymbol{w}) = 1$. To prove its knowledge of the witness, the prover first interacts with the VOLE protocol and the verifier in the following way.

1. The prover and verifier jointly request $n + t$ MACs from the VOLE protocol. This provides the prover with $(u_i, v_i)_{i \in [1..n+t]}$ and the verifier with $(q_i)_{i \in [1..n+t]}$ and Δ.
2. For every input element w_i, for $i \in [1..n]$, the prover computes $d_i := w_i - u_i$; it then sends $(d_i)_{i \in [1..n]}$ to the verifier. The verifier then computes $q_i := q_i + d_i \cdot \Delta$ for each i which means that the tuples $(w_i, v_i, q_i)_{i \in [1..n]}$ are now valid VOLE MACs for the witness $\boldsymbol{w}$, instead of being MACs on the random values u_i.
3. For every gate in the circuit C, with input values w_α, w_β, the prover and verifier proceed as follows.
   - Linear gate for $a, b, c \in \mathbb{F}_2$: the prover computes $w_\gamma := a \cdot w_\alpha + b \cdot w_\beta + c$ and $v_\gamma := a \cdot v_\alpha + b \cdot v_\beta$ while the verifier computes $q_\gamma := a \cdot q_\alpha + b \cdot q_\beta + c \cdot \Delta$. This does not require any communication between the parties.
   - i-th multiplication gate, for $i \in [1..t]$: the prover computes $w_\gamma := w_\alpha \cdot w_\beta$ and sends $d_{n+i} := w_\gamma - u_{n+1} \in \mathbb{F}_2$ to the verifier. The verifier then computes $q_{n+i} := q_{n+i} + d_{n+i} \cdot \Delta$ which means that the tuple $(w_\gamma, v_{n+i}, q_{n+i})$ is now a valid VOLE MAC for the output of the i-th multiplication gate.

At the issue of this interaction, the prover and the verifier now hold three valid VOLE MACs $(w_\alpha, v_\alpha, q_\alpha), (w_\beta, v_\beta, q_\beta)$ and $(w_\gamma, v_\gamma, q_\gamma)$ for the t multiplication gates $(\alpha, \beta, \gamma)_i$ contained in the execution of $\mathsf{C}(\boldsymbol{w})$. The verifier must now check that the prover did not behave maliciously when it sent the t values d_{n+i} which corrected the random VOLE outputs u_{n+i} and their keys q_{n+i} for the multiplication outputs w_γ.

Checking multiplications. The QuickSilver protocol performs a check of the multiplication values based on the observation that the verifier can compute a value $b_i \in \mathbb{F}_{2^\lambda}$ for each multiplication gate $(\alpha, \beta, \gamma)_i$, for $i \in [1..t]$, as follows:

$$\begin{aligned} b_i &:= q_\alpha \cdot q_\beta - q_\gamma \cdot \Delta \\ &= v_\alpha \cdot v_\beta + (w_\alpha \cdot v_\beta + w_\beta \cdot v_\alpha - v_\gamma) \cdot \Delta + (w_\alpha \cdot w_\beta - w_\gamma) \cdot \Delta^2 \end{aligned} \tag{2}$$

If the prover was honest in the computation of d_i, then the Δ^2 coefficient $w_\alpha \cdot w_\beta - w_\gamma$ disappears, and the verifier needs only to check that

$$b_i \stackrel{?}{=} a_{0,i} + a_{1,i} \cdot \Delta \tag{3}$$

$$\text{for} \quad a_{0,i} := v_\alpha \cdot v_\beta$$
$$\text{and} \quad a_{1,i} := w_\alpha \cdot v_\beta + w_\beta \cdot v_\alpha - v_\gamma.$$

To perform this check, the verifier requires the $a_{0,i}$ and $a_{1,i}$ values which the prover can compute, since they only depend on the w and v values for the multiplication gate $(\alpha, \beta, \gamma)_i$, and send to the verifier.

After receiving the t pairs $(a_{0,i}, a_{1,i})$ from the prover, the verifier accepts the proof if Equation (3) holds for all $i \in [1..t]$ and rejects if any of the tests fails. It is in this check that the secrecy of the global key Δ, or in other words the binding property of the VOLE MACs, guarantees the soundness of the proof: to cheat in the computation of the witness, the prover would need to modify values u and tags v such that the test of Equation (3) still passes; this requires guessing Δ.

Note. For the FAEST signature scheme, we obtain VOLE correlations for values $u \in \mathbb{F}_2$, but we will instead be checking correctness of multiplications in $\mathbb{F}_{2^8}$ since this is the field over which the AES S-box is defined (see Section 4.1). Since $\mathbb{F}_2$ is a subfield of $\mathbb{F}_{2^8}$, which is itself a subfield of $\mathbb{F}_{2^\lambda}$, we are able to combine VOLE MACS for 8 values in $\mathbb{F}_2$ into a VOLE MAC for a single value $\mathbb{F}_{2^8}$, with the corresponding tag and key still satisfying the VOLE relation for the original global key Δ.

The advantage of the QuickSilver protocol is that we can still commit to the witness bits using $d_i \in \mathbb{F}_2$, which costs only 1 bit of proof size (and therefore signature size) per bit of the witness, and then prove $\mathbb{F}_{2^8}$ multiplications at no extra cost, since $(a_{0,i}, a_{1,i})$ are already in $\mathbb{F}_{2^\lambda}$.

Optimising communication. Since the relation tested in Equation (3) is linear, the QuickSilver protocol optimises the checking procedure using a random linear combination. After receiving all the d_i values, the verifier sends a random challenge $\chi \in \mathbb{F}_{2^\lambda}$ to the prover. Instead of sending t pairs of values, the prover will then compute

$$a_0 := \sum_{i=1}^{t} a_{0,i} \cdot \chi^i \quad \text{and} \quad a_1 := \sum_{i=1}^{t} a_{1,i} \cdot \chi^i,$$

and send only a single pair (a_0, a_1) to the verifier, thereby not needing to communicate $2t$ $\mathbb{F}_{2^\lambda}$ field elements. Upon receiving this pair, the verifier will check

$$b := \sum_{i=1}^{t} b_i \cdot \chi^i \stackrel{?}{=} a_0 + a_1 \cdot \Delta.$$

Note. In the FAEST signature scheme, this communication optimisation is implemented differently, using a linear and universal hash function (see Section 4.2).

Adding zero-knowledge. The QuickSilver protocol adds one more component to this multiplication check in order to guarantee the zero-knowledge property of the protocol. Indeed, revealing a_0 and a_1 to the verifier leaks information about the circuit values used to compute them. To prevent this, the prover and the verifier jointly request λ further random VOLE MACs which they combine into a single MAC (a_1^*, a_0^*, b^*) such that $a_1^* \in \mathbb{F}_{2^\lambda}$ (as opposed to $u \in \mathbb{F}_2$ output by the VOLE protocol) and the VOLE relation of Equation (1) holds: $b^* = a_0^* + a_1^* \cdot \Delta$.

Using this additional MAC, the prover instead sends $\tilde{a}_0 := a_0 + a_0^*$ and $\tilde{a}_1 := a_1 + a_1^*$ and the verifier checks whether $\tilde{b} := b + b^* \stackrel{?}{=} \tilde{a}_0 + \tilde{a}_1 \cdot \Delta$.

3 Preliminaries

In this section, we describe notation, basic data types and conversion algorithms, the core cryptographic primitives (hash functions and PRGs) used in FAEST, as well as security definitions that are used later in the document.

3.1 Notation

If $b \in \mathbb{F}_2$ or $b \in \{0,1\}$, then $\bar{b} := 1 - b$ denotes the complement of b. When writing $+$ we mean addition in the underlying ring, while $\oplus$ is always the XOR operation. If we use either operation on vectors, then these will be component-wise operations. We use the following index sets:

- $[a..b] = \{a, \ldots, b-1, b\}$
- $[a..b) = \{a, \ldots, b-1\}$

Vectors are column vectors by default, and we denote them in bold font, $\mathbf{x} \in \{0,1\}^n$. For vectors of finite field elements (other than $\mathbb{F}_2$), we use bold italic $\boldsymbol{x} \in \mathbb{F}_q^n$. Matrices are given in capitals, $\mathbf{X}$. We use the following conventions for indexing elements of vectors and matrices:

- $\mathbf{x}[i]$: i-th element of a vector
- $\mathbf{x}[a..b]$: vector containing elements a through b (inclusive) of $\mathbf{x}$
- $\mathbf{X}|_i / \mathbf{X}|^i$: row/column i of a matrix
- $\mathbf{X}|_{[a..b]}$: rows a through b of a matrix
- $(\mathbf{x} \| \mathbf{y})$: concatenation of vectors $\mathbf{x}$ and $\mathbf{y}$
- $\begin{bmatrix}\mathbf{x} \; \mathbf{y}\end{bmatrix}$: two-column matrix made from $\mathbf{x}$ and $\mathbf{y}$

3.2 Data Types and Conversions

Finite field Arithmetic. FAEST uses finite field arithmetic over $\mathbb{F}_{2^8}$ (as part of AES), $\mathbb{F}_{2^{64}}$, $\mathbb{F}_{2^{128}}$, $\mathbb{F}_{2^{192}}$, and $\mathbb{F}_{2^{256}}$. 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.

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

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

- ToField$(\mathbf{x}; k)$: maps $\mathbf{x} \in \{0,1\}^{nk}$, for $n \geq 1$ into a (vector of) field element(s) $\boldsymbol{x} \in \mathbb{F}_{2^k}^n$ using little-endian ordering.
- ToBits$(\boldsymbol{x})$: maps $\boldsymbol{x} \in \mathbb{F}_{2^k}^n$, for $n \geq 1$, into a bit string $\mathbf{x} \in \{0,1\}^{nk}$.

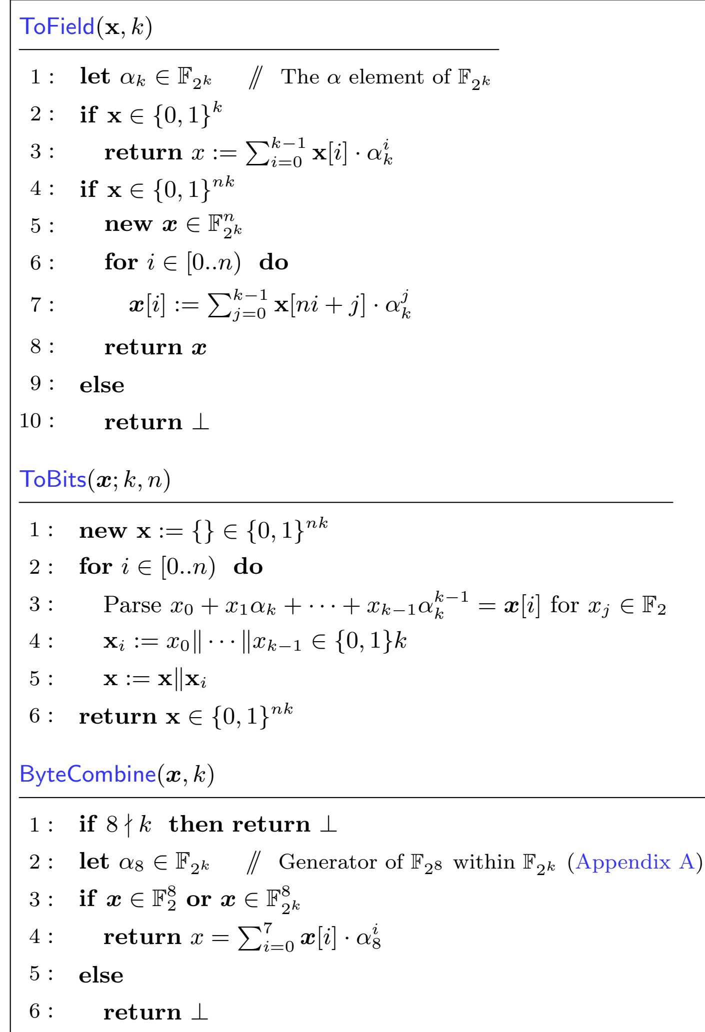

Fig. 3.1: Data conversion functions

- ByteCombine$(\boldsymbol{x}; k)$: takes a vector of exactly 8 elements in $\mathbb{F}_2$ or $\mathbb{F}_{2^k}$, where $8 \mid k$, and combines them into a single element in $\mathbb{F}_{2^k}$ using powers of an $\mathbb{F}_{2^8}$ generator within $\mathbb{F}_{2^k}$ and little-endian ordering. (The precise generators we use are specified in Appendix A.)
  This ordering, where the most significant bit of the byte indicates the highest power of the α_8 generator element matches the interpretation of bytes as polynomials in the AES standard [AES01].

In an implementation, depending on the representation of finite field elements in large binary fields, ToField and ToBits may not require any operations (e.g. if field elements are stored as 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 3.2, 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 3.2 contains the integer reconstruction al-

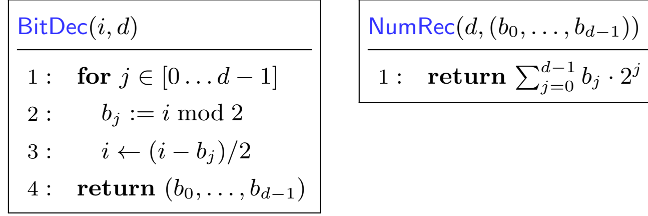

Fig. 3.2: Bit decomposition and reconstruction algorithms, little endian representation

gorithm, which maps a bit-string of length d uniquely into an integer in the interval $[0..2^d)$. Clearly, $\mathsf{NumRec}(d, \mathsf{BitDec}(i, d)) = i$ for all $i \in [0, 2^d)$.

3.3 Cryptographic Primitives

FAEST uses the following symmetric primitives:

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

PRG. We instantiate $\mathsf{PRG}(s, \mathsf{iv})$ using AES-λ in CTR mode with seed s and initialization vector iv. The counter is a 128-bit, big-endian integer initially set to iv and incremented by one after each block. To make the output length explicit, we write $\mathsf{PRG}(s, \mathsf{iv}; \ell)$ to indicate that ℓ output bits are required using the seed s. If ℓ is not a multiple of 128, we compute $\lceil \ell/128 \rceil$ 128-bit blocks of output and truncate the final block to $\ell \bmod 128$ bits. We use PRG with a random, per-signature iv, in order to prevent multi-target attacks when a large number of signatures are available (see Section 10.2.2).

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, defined as follows:

- $\mathsf{H}_i(m) := \mathrm{SHAKE}(m \| i, \ell)$, if $i \in \{0, 1, 3\}$
- $\mathsf{H}_2^j(m) := \mathrm{SHAKE}(m \| 2, \ell)$

Here, SHAKE is either SHAKE128 or SHAKE256, depending on λ. Note that we do not need to explicitly separate H_2^j for different j, since in our usage, the input is always a different length for each j, so we rely on SHAKE's built-in domain separation.

In our security proofs, we model $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^j$ as random oracles, while H_3 is modeled as a pseudo-random function (where the last λ input bits are the key). At one point in the proof, we also model H_0 as a secure pseudo-random generator.

3.4 Security Definitions

We define the notion of families of one-way functions (OWF) as follows. Note that in this document, we swap the key k and input x compared with the usual definition; in our case, the adversary is given the input/output pair x, y, and must attempt to recover the *key* k that selects the function f_k from the family. This approach corresponds naturally to how we build a one-way function from a block cipher, where the secret key is the hard-to-find OWF input.

Definition 1 (One-way function). *A family of functions* $\{f_k\}_{k \in K_\lambda}$ *with* $f_k : D_\lambda \to C_\lambda$ *is called* one-way*, if (1) there exists a PPT algorithm* F *such that for all* $\forall \lambda \in \mathbb{N}$, $\forall k \in K_\lambda$, $\forall x \in D_\lambda$: $\mathsf{F}(k, x) = f_k(x)$*, and (2) for every PPT algorithm* $\mathcal{A}$ *the advantage*

$$\mathsf{AdvOWF}^{f}_{\mathcal{A}} := \Pr\left[f_k(x) = f_{k^*}(x) \;\middle|\; k \leftarrow K_\lambda, x \leftarrow D_\lambda, k^* \leftarrow \mathcal{A}(1^\lambda, f_k(x))\right]$$

is negligible in λ.

We use the standard security notion for digital signature schemes, namely, existential unforgeability under adaptive chosen-message attacks (EUF-CMA). As a stepping stone to EUF-CMA security, we will first prove security of our scheme against key-only attacks (EUF-KO) where the adversary is given the public key but no access to a signing oracle.

Definition 2 (EUF-CMA security). *Given a signature scheme* $\mathtt{Sig} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *and security parameter* λ*, we say that* $\mathtt{Sig}$ *is* EUF-CMA-secure *if any PPT algorithm* $\mathcal{A}$ *has negligible advantage in the* EUF-CMA *game, defined as*

$$\mathbf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A}} = \Pr\left[\begin{array}{c}\mathsf{Verify}(\mathsf{pk}, \mathsf{msg}^*, \sigma^*) = 1 \\ \wedge\ \mathsf{msg}^* \notin Q\end{array} \;\middle|\; \begin{array}{c}(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda) \\ (\mathsf{msg}^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\mathsf{sk}, \cdot)}(\mathsf{pk})\end{array}\right],$$

where $\mathcal{A}^{\mathsf{Sign}(\mathsf{sk}, \cdot)}$ *denotes* $\mathcal{A}$*'s access to a signing oracle with private key* sk *and* Q *denotes the set of messages* msg *that were queried to* $\mathsf{Sign}(\mathsf{sk}, \cdot)$ *by* $\mathcal{A}$.

As a stepping stone to EUF-CMA security, we will first prove security of our scheme against key-only attacks (EUF-KO) where the adversary is given the public key but no access to a signing oracle.

Definition 3 (EUF-KO security). *Given a signature scheme* $\mathtt{Sig} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *and security parameter* λ*, we say that* $\mathtt{Sig}$ *is* EUF-KO-secure *if any PPT algorithms* $\mathcal{A}$ *has negligible advantage in the* EUF-KO *game, defined as*

$$\mathbf{Adv}^{\mathsf{EUF\text{-}KO}}_{\mathcal{A}} = \Pr\left[\mathsf{Verify}(\mathsf{pk}, \mathsf{msg}^*, \sigma^*) = 1 \;\middle|\; \begin{array}{c}(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda) \\ (\mathsf{msg}^*, \sigma^*) \leftarrow \mathcal{A}(\mathsf{pk})\end{array}\right].$$

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 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):

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

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:

$$\begin{array}{|c|c|c|c|} \hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ \hline s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ \hline s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|c|} \hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ \hline s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ \hline s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \\ \hline \end{array}$$

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:

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

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

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

where α generates $\mathbb{F}_{2^8}$.

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

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

where the $\bar{\mathbf{k}}[i]$ are the expanded key words, and $0 \leq \mathtt{round} \leq R$ 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 $[\mathtt{Rcon[i]} \| \{\mathtt{00}\} \| \{\mathtt{00}\} \| \{\mathtt{00}\}]$ is added at certain intervals during the key expansion (after every multiple of N_{k} words); the value $\mathtt{Rcon[i]}$ is computed as $\{\mathtt{02}\}^i = \alpha_8^i \in \mathbb{F}_{2^8}$. 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.

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.

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 $[\mathtt{Rcon[i/N_k} - \mathtt{1]} \| \{\mathtt{00}\} \| \{\mathtt{00}\} \| \{\mathtt{00}\}]$. 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.

```
KeyExpansion(k; param_OWF)
 1 :  new k̄ ∈ [{0,1}^32; N_st(R+1)]    // Empty expanded key, as an array of words

 2 :  for i ∈ [0..N_k)  do
 3 :    k̄[i] := k[i]    // Here k ∈ [{0,1}^32; N_k] is viewed as an array of words

 4 :  for i ∈ [N_k..N_st(R+1))  do
 5 :    tmp := k̄[i − 1]
 6 :    if i mod N_k = 0  then
 7 :      tmp := SubWord(RotWord(tmp)) + [Rcon[i/N_k − 1]||{00}||{00}||{00}]
 8 :    if N_k > 6 and i mod N_k = 4  then
 9 :      tmp := SubWord(tmp)
10 :    k̄[i] := k̄[i − N_k] + tmp

11 :  return k̄
```

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 **in** $\in \{0,1\}^{32\cdot N_{st}}$ 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 **out** $\in \{0,1\}^{32\cdot N_{st}}$.

4.1.3 Proving AES and Rijndael with QuickSilver

As described in Section 2 the FAEST algorithm is constructed from a QuickSilver zero-knowledge proof of the OWF relation $\mathcal{R}$ defined by $((x,y),k) \in \mathcal{R} \iff F_k(x) = y$, where F is constructed from the AES or Rijndael algorithm. Proving statements for this relation therefore requires expressing the key expansion and encryption routines as arithmetic circuits that are compatible with the QuickSilver system. Here we provide an overview of how to do this, and we refer the reader to Sections 6 and 7 for the detailed specification.

As presented in Section 2.3, the QuickSilver proof system is capable of proving multiplication constraints, i.e. proving that three wire values $w_\alpha, w_\beta, w_\gamma$ satisfy the constraint $w_\alpha \cdot w_\beta = w_\gamma$. To find such constraints in the AES and Rijndael routines, we look to the S-box, which contains the only non-$\mathbb{F}_2$-linear operation: the $\mathbb{F}_{2^8}$ field inversion. For every occurrence of this inversion, mapping $w_\alpha \in \mathbb{F}_{2^8}$ to $w_\gamma := w_\alpha^{-1}$ (assuming $w_\alpha \neq 0$), we can define the multiplicative constraint $w_\alpha \cdot w_\gamma = 1$. (Since inversion is a single-input–single-output gate, we do not require w_β.)

Given that there are no other non-linear operations in the routines, proving an instance of the relation $\mathcal{R}$ requires only describing how the bits of the inputs k and x are linearly transformed, using the AES operations, into the inputs of the S-boxes that occur during the computation of $F_k(x)$, and how the outputs of the F_{2^8}-inversions are linearly transformed into the inputs of the next S-boxes.

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

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

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

12 : return out := state
```

Fig. 4.3: The encryption routine

Given this, the QuickSilver proof that we compute in the FAEST algorithm proceeds in two stages:

1. *Witness extension.* The signer takes the secret input $\mathsf{sk} = k$, 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 $F_k(x)$. After each non-linear operation, the bits that depend on the previous S-box layer are recorded into an extended witness which then contains information about all the outputs of the F_{2^8} inversion operations. (The exact bits that are recorded are specified in Sections 6.1 and 7.1.) This extended witness is committed by the signer when it sends the d_i values in the QuickSilver proof.
2. *QuickSilver response.* For each F_{2^8} inversion, the signer computes the w_α input values (either from the inputs k and x or from the extended witness) and the w_γ output values (either from the extended witness or from the output y). As the inputs, output and extended witness are all represented as bits, this step requires combining 8 bits into single F_{2^8} elements in the same way as in the AES specification [AES01].
   With the (w_α, w_γ) derived for every inversion operation, the signer can then compute the $(\tilde{a}, \tilde{b})$ QuickSilver response using the VOLE tags and the universal linear hash information sent by the verifier (or derived from the Fiat–Shamir hash function). The computation of these constraint values is specified in the relevant subsections of Sections 6 and 7.

Note. This method of proving the execution of AES routines and operations with the QuickSilver proof system requires FAEST to ensure that no inversion operation ever has the value $w_\alpha = 0$ as input, otherwise the constraint $w_\alpha \cdot w_\gamma = 1$ would not hold. To guarantee this, we select the FAEST key pair such that this property is satisfied. This is described in more detail in Section 8.1 and the security implications of restricting FAEST keys in this manner are discussed in Section 10.3.

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. *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 $\mathbf{x} \in \mathbb{F}_q^n$,*

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

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

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

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

Our hashes must also satisfy the following hiding property.

Definition 5. *A matrix $\mathbf{H} \in \mathbb{F}_q^{r \times (n+h)}$ is $\mathbb{F}_q^n$-hiding if the distribution of $\mathbf{H}\mathbf{v}$ 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 1. *Let $\mathcal{H} \subseteq \mathbb{F}_q^{r \times n}$ be an ε-almost uniform hash family. Let $\mathcal{H}' \subseteq \mathbb{F}_q^{r \times (n+r)}$ be the family $\{[\mathbf{H}\ I_r] : \mathbf{H} \in \mathcal{H}\}$, where I_r is the $r \times r$ identity matrix. Then, it holds that (1) $\mathcal{H}'$ is ε-almost universal, and (2) $\mathcal{H}'$ is $\mathbb{F}_q^n$-hiding.*

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

4.2.1 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(sd, (x_0, x_1) ∈ {0,1}^{ℓ+λ} × {0,1}^{λ+B})
1 :  // λ-bit r_i, s; 64-bit t
2 :  Parse sd = (r_0||r_1||r_2||r_3||s||t) ∈ {0,1}^{5λ+64}
3 :  r_i := ToField(r_i, λ), for i ∈ [0..3]
4 :  s := ToField(s, λ)
5 :  t := ToField(t, 64)
6 :  ℓ' := λ · ⌈(ℓ + λ)/λ⌉
7 :  x_0 := x_0||0^{ℓ'−(ℓ+λ)}    // pad to multiple of λ
8 :  ŷ := ToField(x_0, λ)
9 :  ȳ := ToField(x_0, 64)
10 : h_0 := Σ_{i=0}^{ℓ'/λ−1} s^{ℓ'/λ−1−i} · ŷ[i] (in F_{2^λ})
11 : h_1 := Σ_{i=0}^{ℓ'/64−1} t^{ℓ'/64−1−i} · ȳ[i] (in F_{2^64})
12 : h'_1 := ToField(ToBits(h_1)||0^{λ−64}, λ)
13 : (h_2, h_3) := (r_0 h_0 + r_1 h'_1, r_2 h_0 + r_3 h'_1)
14 : h := (ToBits(h_2)||ToBits(h_3)[0..B)) ⊕ x_1
15 : return h
```

```
ZKHash(sd, (x_0, x_1) ∈ F_{2^λ}^ℓ × F_{2^λ})
1 :  // λ-bit r_i, s; 64-bit t
2 :  Parse sd = (r_0||r_1||s||t) ∈ {0,1}^{3λ+64}
3 :  r_i := ToField(r_i, λ), for i ∈ {0,1}
4 :  s := ToField(s, λ)
5 :  t := ToField(t||0^{λ−64}, λ)
6 :  h_0 := Σ_{i=0}^{ℓ−1} s^{ℓ−1−i} · x_0[i]
7 :  h_1 := Σ_{i=0}^{ℓ−1} t^{ℓ−1−i} · x_0[i]
8 :  h := ToBits(r_0 h_0 + r_1 · h_1 + x_1)
9 :  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 2. *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 3. *Let $\mathcal{H} \subseteq \mathbb{F}_q^{r' \times n}$ be ε-almost universal and $\mathcal{H}' \subseteq \mathbb{F}_q^{r \times r'}$ be ε'-almost uniform. Then, the product $\{\mathbf{H}'\mathbf{H} : \mathbf{H} \in \mathcal{H}, \mathbf{H}' \in \mathcal{H}'\}$ is $(\varepsilon + \varepsilon')$-almost uniform.*

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

Proposition 4. *Let $\delta \in \mathbb{N}$ and $\mathcal{H} \subseteq \mathbb{F}_q^{r \times n}$ be an ε-almost uniform family. Then, the truncated family $\{\mathbf{H}_{[0..r-\delta)} : \mathbf{H} \in \mathcal{H}\}$ is εq^δ-uniform.*

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

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

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

4.2.3 VOLE Universal Hash. The first check, to verify consistency of the VOLE correlations, requires a family that is linear over $\mathbb{F}_2$. It's evaluated on inputs in $\mathbb{F}_2^{\hat{\ell}}$, where $\hat{\ell} = \ell + 2\lambda + B$, ℓ is the witness length and $B = 16$ is a parameter chosen for security.[10]

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

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

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

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

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

We argue security of the construction below. Note that instead of aiming for $\varepsilon = 2^{\lambda}$, we aim for $2^{-\lambda - B}$, where $B = 16$ (Table 2.2). The extra few bits of security compensate for the $\binom{\tau}{2}$ security loss in the proof of the SoftSpokenVOLE protocol from [BBD+23].

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

Proof. We show ε_{v}-uniformity of the hash that outputs the first $\lambda + B$ bits of (h_2, h_3), i.e. without adding $\mathbf{x}_1$. By Proposition 1, this implies ε_{v}-universality of the final hash, as well as the hiding property.

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

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

The second part of the hash starts with a 2×2 matrix hash, which is $2^{-2\lambda}$-uniform. After truncation, the resulting hash is ε_1-uniform for $\varepsilon_1 = 2^{-\lambda-B}$, by Proposition 4. The final combined hash is the product of these two parts, so applying Proposition 3 and summing the probabilities, we get that for all $\hat{\ell} \le 2^{13}$, the hash is ε_{v}-uniform for $\varepsilon_{\mathsf{v}} = \varepsilon_0 + \varepsilon_1 \le 2^{-\lambda-B}(1 + 2^{B-50})$. □

4.2.4 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 $\ell' = \ell + 1$ field elements. We map the seed sd into $r_0, r_1, s, t \in \mathbb{F}_{2^\lambda}$, where r_0, r_1 and s are uniform, while t (viewed as an $\mathbb{F}_2$ polynomial) is zero in all its degree ≥ 64 coefficients (and uniform otherwise). Let

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

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

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

Proof. As with $\mathsf{VOLEHash}$, by Proposition 1 it suffices to show $\varepsilon_{\mathsf{zk}}$-uniformity of the hash defined by $\mathbf{r}^\top \mathbf{x}_0$.

This hash is a product of two separate hashes. The outer hash — multiplication by $(r_0 \; r_1)$ — is a standard matrix hash, which is $2^{-\lambda}$-almost uniform. The inner hash is a concatenation of two polynomial evaluations, the first of which defines an almost-universal hash for $\varepsilon_0 = (\ell' - 1)/2^\lambda$, and the second for $\varepsilon_1 = (\ell' - 1)/2^{64}$; together, this gives an $\varepsilon_0\varepsilon_1$-almost universal hash (via Proposition 2), where $\varepsilon_0\varepsilon_1 \le (\ell')^2/2^{\lambda+64} \le 2^{-\lambda-38}$, for all $\ell' \le 2^{13}$. Combining the inner and outer parts, from Proposition 3 we get that ZKHash is $\varepsilon_{\mathsf{zk}}$-almost uniform for $\varepsilon_{\mathsf{zk}} = 2^{-\lambda}(1+2^{-38})$. □

5 VOLE-in-the-Head Functions

In this section we describe the algorithms required to build the VOLEitH protocol. This is constructed from an all-but-one vector commitment scheme, presented in Section 5.1, whose outputs are transformed to VOLE correlations using the ConvertToVOLE algorithm described in Section 5.2. Finally, both of these are used to build the VOLEitH protocol, described in Section 5.3.

5.1 All-but-One Vector Commitments

The all-but-one vector commitment scheme VC is described in Figure 5.1. It consists of the four algorithms Commit, Open, Reconstruct and Verify for commitment, opening, reconstruction and verification respectively.

Note. The verification algorithm Verify is not used in the FAEST signature scheme, but it is included here for completeness.

Commit generates a vector commitment of length $N = 2^d$ based on a seed $r \in \{0,1\}^\lambda$ by expanding the seeds using PRG as a GGM tree. Each tree leaf k_j^d is then expanded using H_0 into two values, a seed sd_j and commitment $\overline{\mathsf{com}}_j$. A hash of $\overline{\mathsf{com}}_0, \ldots, \overline{\mathsf{com}}_{N-1}$ then serves as the commitment, while the sd_j are the committed messages. We additionally store all GGM tree keys k_j^i and the $\overline{\mathsf{com}}_j$ as decommitment information.

$\mathsf{VC}.\mathsf{Commit}(r, \mathsf{iv}; N = 2^d)$

1 : $k_0^0 := r$
2 : **for** $i \in [1..d]$, $j \in [0..2^{i-1})$ **do**
3 : $\quad (k_{2j}^i, k_{2j+1}^i) := \mathsf{PRG}(k_j^{i-1}, \mathsf{iv}; 2\lambda)$
4 : **for** $j \in [0..N)$ **do**
5 : $\quad (\mathsf{sd}_j, \overline{\mathsf{com}}_j) := \mathsf{H}_0(k_j^d \| \mathsf{iv})$
6 : $h := \mathsf{H}_1(\overline{\mathsf{com}}_0, \dots, \overline{\mathsf{com}}_{N-1})$
7 : $\mathsf{decom} := \left(\left((k_j^i)_{j \in [0..2^i)} \right)_{i \in [1..d]}, (\overline{\mathsf{com}}_j)_{j \in [0..N)} \right)$
8 : **return** $(\mathsf{com} := h, \mathsf{decom}, (\mathsf{sd}_j)_{j \in [0..N)})$

$\mathsf{VC}.\mathsf{Open}(\mathsf{decom}, (b_0, \dots, b_{d-1}))$

1 : $j^* := \mathsf{NumRec}(d, (b_0, \dots, b_{d-1}))$
2 : Parse $\mathsf{decom} := \left(\left((k_j^i)_{j \in [0..2^i)} \right)_{i \in [1..d]}, (\overline{\mathsf{com}}_j)_{j \in [0..N)} \right)$
3 : $a := 0$
4 : **for** $i \in [1..d]$ **do**
5 : $\quad \mathsf{cop}[i] := k_{2a+\overline{b_{d-i}}}^i$
6 : $\quad a := 2a + b_{d-i}$
7 : $\mathsf{pdecom} := (\mathsf{cop}, \overline{\mathsf{com}}_{j^*})$
8 : **return** pdecom

$\mathsf{VC}.\mathsf{Reconstruct}(\mathsf{pdecom}, (b_0, \dots, b_{d-1}), \mathsf{iv})$

1 : Parse $\mathsf{pdecom} = (\mathsf{cop}, \overline{\mathsf{com}}_{j^*})$
2 : $j^* := \mathsf{NumRec}(d, (b_0, \dots, b_{d-1}))$
3 : $a := 0$, $k_0^0 := \perp$
4 : **for** $i \in [1..d]$ **do**
5 : $\quad k_{2a+\overline{b_{d-i}}}^i := \mathsf{cop}[i]$
6 : $\quad k_{2a+b_{d-i}}^i := \perp$
7 : $\quad$ **for** $j \in [0..2^{i-1}) \setminus \{a\}$ **do**
8 : $\quad\quad$ // reconstruct other keys
9 : $\quad\quad (k_{2j}^i, k_{2j+1}^i) := \mathsf{PRG}(k_j^{i-1}, \mathsf{iv}; 2\lambda)$
10 : $\quad a := 2a + b_{d-i}$
11 : **for** $j \in [0..N) \setminus \{j^*\}$ **do**
12 : $\quad (\mathsf{sd}_j, \overline{\mathsf{com}}_j) := \mathsf{H}_0(k_j^d \| \mathsf{iv})$
13 : $h := \mathsf{H}_1(\overline{\mathsf{com}}_0, \dots, \overline{\mathsf{com}}_{N-1})$
14 : **return** $(h, (\mathsf{sd}_j)_{j \in [0..N), j \neq j^*})$

$\mathsf{VC}.\mathsf{Verify}(\mathsf{com}, \mathsf{pdecom}, (b_0, \dots, b_{d-1}), \mathsf{iv})$

1 : $j^* := \mathsf{NumRec}(d, (b_0, \dots, b_{d-1}))$
2 : $(\overline{\mathsf{com}}, (\mathsf{sd}_j)_{j \in [0..N), j \neq j^*})$
$\quad := \mathsf{Reconstruct}(\mathsf{pdecom}, (b_i)_{i \in [0..d)}, \mathsf{iv})$
3 : **if** $\mathsf{com} = \overline{\mathsf{com}}$
4 : $\quad$ **return** 1
5 : **else**
6 : $\quad$ **return** 0

Fig. 5.1: Commit, open, reconstruction and verification algorithms for all-but-one vector commitment

Open generates a partial decommitment pdecom which opens all but one message. The index j^* of message not to be opened is encoded as a bit string $b_0, \ldots, b_{d-1}$, where b_0 encodes the lowest bit of j^* and b_{d-1} is the highest bit of j^*. pdecom consists of all siblings of GGM keys on the path between k_0^0 and $k_{j^*}^d$, aggregated in the structure cop. This allows to reconstruct all messages except sd_{j^*}. pdecom also contains $\overline{\mathsf{com}}_{j^*}$ which can later be used to verify that cop is well-formed.

Reconstruct given index j^* encoded as $b_0, \ldots, b_{d-1}$ (identically as for Open) as well as the partial decommitment pdecom for the same index, reconstructs all GGM tree leafs k_j^d for $j \in [0..2^d)$ except j^*. It moreover recomputes $\overline{\mathsf{com}}_j$ for all indices except j^* (since $\overline{\mathsf{com}}_{j^*}$ is an input contained in pdecom) and hashes them. It outputs the hash as well as all the reconstructed leaf keys.

Verify verifies a commitment com (the hash of all the $\overline{\mathsf{com}}_0, \ldots, \overline{\mathsf{com}}_{N-1}$) against a partial decommitment pdecom allegedly opening all positions but j^* which is encoded via $b_0, \ldots, b_{d-1}$ as for Open. It therefore runs Reconstruct and compares the generated hash with com.

5.2 Seed Expansion and Conversion to VOLE

After committing to N random seeds using the all-but-one vector commitment, each seed is expanded to a longer vector of $\mathbb{F}_q$ elements using PRG. The set of N vectors is then converted into a VOLE correlation over $\mathbb{F}_N$, for $N = 2^d$, which we represent as d VOLE correlations over $\mathbb{F}_2$. Recall from Section 2.2 that this conversion requires the signer to compute, in $\mathbb{F}_N$

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

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

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

Instead of computing the above directly, the signer and verifier use a divide-and-conquer method [Roy22] to iteratively compute the vectors $\mathbf{v}_j$ and $\mathbf{q}_j$, which represent the j-th bits extracted from $\mathbf{v}, \mathbf{q}$. This algorithm is shown in Figure 5.2.

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

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

Proposition 5. *Let* $(\mathsf{sd}_0, \ldots, \mathsf{sd}_{N-1})$ *and* $(\mathsf{sd}'_0, \ldots, \mathsf{sd}'_{N-1})$ *be seeds such that* $\mathsf{sd}'_i = \mathsf{sd}_{i \oplus \Delta}$ *for all* $i > 0$*, for some* $\delta \in [0..N)$*, and* $\mathsf{sd}'_0 = \perp$*. Then, if* $(\delta_0, \ldots, \delta_{d-1}) :=$ $\mathsf{BitDec}(\Delta, d)$*,* $(\mathbf{u}, \mathbf{v}_0, \ldots, \mathbf{v}_{d-1}) := \mathsf{ConvertToVOLE}(\mathsf{sd}_0, \ldots, \mathsf{sd}_{N-1}; \hat{\ell})$*, and* $(\mathbf{u}', \mathbf{q}_0, \ldots, \mathbf{q}_{d-1}) := \mathsf{ConvertToVOLE}(\mathsf{sd}'_0, \ldots, \mathsf{sd}'_{N-1}; \hat{\ell})$*, it holds that*

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

```
ConvertToVOLE(sd_0, ..., sd_{N-1}, iv; ℓ̂)
 1 :  d := log N
 2 :  r_{0,0} := 0^ℓ̂ if sd_0 = ⊥ else PRG(sd_0, iv; ℓ̂)
 3 :  for i ∈ [1..N)  do
 4 :     r_{0,i} := PRG(sd_i, iv; ℓ̂)
 5 :  v_0 := ··· := v_{d-1} := 0^ℓ̂
 6 :  for j ∈ [0..d)  do
 7 :     for i ∈ [0..N/2^{j+1})  do
 8 :        v_j := v_j ⊕ r_{j,2i+1}
 9 :        r_{j+1,i} := r_{j,2i} ⊕ r_{j,2i+1}
10 :  u := r_{d,0}
11 :  return (u, v_0, ..., v_{d-1}) ∈ (F_2^ℓ̂)^{d+1}
```

Fig. 5.2: Seed expansion and conversion to VOLE

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

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

We now argue correctness by induction on d. With $d = 1$, the prover, on input $(\mathbf{r}_{0,0}, \mathbf{r}_{0,1})$, obtains output $\mathbf{v}_0 = \mathbf{r}_{0,1}$ and $\mathbf{u} = \mathbf{r}_{0,0} \oplus \mathbf{r}_{0,1}$. The verifier, on input $(0^{\hat{\ell}}, \mathbf{r}'_{0,1} = \mathbf{r}_{0,1\oplus\Delta})$, outputs $\mathbf{q}_0 = \mathbf{r}_{0,1\oplus\Delta}$, which equals $\mathbf{v}_0 \oplus \Delta \cdot \mathbf{u}$ as required. Suppose the algorithm is correct for inputs of length $N/2 = 2^{d-1}$. We first show that the $j = 0$ output is correct for inputs of length N. The prover's relevant output is $\mathbf{v}_0 = \bigoplus_{i=0}^{N/2-1} \mathbf{r}_{0,2i+1}$, the sum of all the odd-indexed $\mathbf{r}_0$ values. On the verifier's side, if $\delta_0 = 0$ then $\mathbf{q}_0 = \mathbf{v}_0$. Otherwise, if $\delta_0 = 1$ then $\mathbf{q}_0$ is the sum of all the even-indexed $\mathbf{r}_0$ values, because for all $i \in [0..N/2)$ we have $\mathbf{r}'_{2i+1} = \mathbf{r}_{(2i+1)\oplus\Delta}$, and $(2i+1) \oplus \Delta$ is even. It follows that $\mathbf{q}_0 \oplus \mathbf{v}_0 = \bigoplus_{i=0}^{N-1} \mathbf{r}_i = \delta_j \cdot \mathbf{u}$.

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

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

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

5.3 VOLEitH: Commitment and Reconstruction

The main FAEST algorithms use the vector commitments and ConvertToVOLE procedure to commit the signer to a batch of τ VOLE instances of length $\hat{\ell}$. These are

later verified, when the prover is challenged to open all-but-one of each set of vector commitment messages, allowing the verifier to reconstruct its VOLE output.

We describe this with the following three algorithms.

5.3.1 Challenge Decomposition Let chall be a challenge bit string of length λ and $0 \leq i < \tau_0 + \tau_1$ be an integer, the algorithm ChalDec (Figure 5.3) generates the challenge for the ith VOLE instance from chall.

Given that $\lambda = k_0\tau_0 + k_1\tau_1$, where $k_0 := \lceil \lambda/\tau \rceil$, $k_1 := \lfloor \lambda/\tau \rfloor$, $\tau_0 := (\lambda \bmod \tau)$, and $\tau_1 := \tau - (\lambda \bmod \tau)$, the input bit string chall is divided into τ_0 sub-strings of length k_0 and τ_1 sub-strings of length k_1. Equivalently, this means that the first τ_0 VOLE instances have a challenge of k_0 bits, while the other τ_1 VOLE instances use k_1 bits.

```
ChalDec(chall, i; param)
 1 :  If i ∉ [0 . . . τ0 + τ1)  then abort
 2 :  Parse chall ∈ {0,1}^(k0τ0+k1τ1)
 3 :  if i < τ0  then
 4 :    lo := i · k0
 5 :    hi := (i + 1) · k0 − 1
 6 :  else
 7 :    t := i − τ0
 8 :    lo := τ0k0 + t · k1
 9 :    hi := τ0k0 + (t + 1) · k1 − 1
10 :  return chall[lo . . . hi]
```

Fig. 5.3: Challenge decomposition algorithm

5.3.2 VOLE Commitment The algorithms VOLECommit and VOLEReconstruct use two of the three main sub-procedures from our all-but-one vector commitment from Section 5.1 to turn committed values into the sender values of VOLE correlations as well as to create the receiver parts of VOLE correlations from commitment openings, respectively.

VOLECommit(). The algorithm runs τ instances of VC.Commit and compresses each of these using ConvertToVOLE into a number of columns. The first τ_0 vector commitments have length 2^{k_0} while the other τ_1 have length 2^{k_1}, so ConvertToVOLE compresses the first τ_0 vector commitments each down to k_0 vectors while the others each become k_1 vectors. This generates, as output, a vector $\mathbf{u}$ and a matrix $\mathbf{V}$. Here $\mathbf{u}$ are the secrets of the first VOLE correlation, while $\mathbf{V} = [\mathbf{V}_1, \ldots, \mathbf{V}_\tau]$ column-wise contains the $\mathbf{v}$-vectors of the τ VOLE correlation. Each column vector in $\mathbf{V}$ has length $\hat{\ell}$. This implicitly generates $\tau \cdot \hat{\ell}$ VOLE correlations over small fields and with different secrets. VOLECommit therefore generates $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}$ to allow to correct the sender secrets of VOLE instances $1, \ldots, \tau - 1$ to the same secret vector $\mathbf{u}$ of VOLE instance 0. Using these corrections $\mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}$, τ VOLE instances with the same secret but different challenges can then be lifted row-wise to one VOLE correlation with a larger challenge. Finally, the function computes a hash of

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

1 : $(r_0 \| \dots \| r_{\tau-1}) := \mathsf{PRG}(r, \mathsf{iv}; \tau\lambda)$
2 : $\tau_0 := \lambda \bmod \tau$
3 : **for** $i \in [0..\tau)$ **do**
4 : $\quad b := 0$ **if** $i < \tau_0$ **else** $b := 1$
5 : $\quad (\mathsf{com}_i, \mathsf{decom}_i, (\mathsf{sd}_{i,0}, \dots, \mathsf{sd}_{i,N_b-1})) := \mathsf{VC}.\mathsf{Commit}(r_i, \mathsf{iv}; N_b = 2^{k_b})$
6 : $\quad (\boldsymbol{u}_i, \mathbf{v}_{i,0}, \dots, \mathbf{v}_{i,k_b-1}) := \mathsf{ConvertToVOLE}(\mathsf{sd}_{i,0}, \dots, \mathsf{sd}_{i,N_b-1}, \mathsf{iv}; \hat{\ell})$
7 : $\quad \mathbf{V}_i := [\mathbf{v}_{i,0} \cdots \mathbf{v}_{i,k_b-1}] \in \mathbb{F}_2^{\hat{\ell} \times k_b}$
8 : $\mathbf{V} := [\mathbf{V}_0 \cdots \mathbf{V}_{\tau-1}] \in \mathbb{F}_2^{\hat{\ell} \times \lambda}$ // stored in column major representation
9 : $\boldsymbol{u} := \boldsymbol{u}_0$
10 : **for** $i \in [1 \dots \tau)$ **do**
11 : $\quad \mathbf{c}_i := \boldsymbol{u} \oplus \boldsymbol{u}_i$
12 : $h_{\mathsf{com}} := \mathsf{H}_1(\mathsf{com}_0 \| \dots \| \mathsf{com}_{\tau-1})$
13 : **return** $(h_{\mathsf{com}}, \mathsf{decom}_0, \dots, \mathsf{decom}_{\tau-1}, \mathbf{c}_1, \dots, \mathbf{c}_{\tau-1}, \boldsymbol{u}, \mathbf{V})$

Fig. 5.4: FAEST-VOLE commitments

FAEST.VOLEReconstruct$(\mathsf{chall}, (\mathsf{pdecom}_i)_{i \in [0 \dots \tau)}, \mathsf{iv}, \hat{\ell}; \texttt{param})$

1 : **for** $i \in [0 \dots \tau)$ **do**
2 : $\quad b := 0$ **if** $i < \tau_0$ **else** 1
3 : $\quad (\delta_{i,0}, \dots, \delta_{i,k_b-1}) := \mathsf{ChalDec}(\mathsf{chall}, i; \texttt{param})$
4 : $\quad \Delta_i := \mathsf{NumRec}(k_b, (\delta_{i,0}, \dots, \delta_{i,k_b-1}))$
5 : $\quad (\mathsf{com}_i, (s_{i,j})_{j=[0..N_b), j \neq i}) := \mathsf{VC}.\mathsf{Reconstruct}(\mathsf{pdecom}_i, (\delta_{i,0}, \dots, \delta_{i,k_b-1}, \mathsf{iv}))$
6 : $\quad$ **for** $j \in [1 \dots N_b)$ let $\mathsf{sd}_{i,j} := s_{i,j \oplus \Delta_i}$ // j and Δ_i as bit-strings for $\oplus$
7 : $\quad (\boldsymbol{u}'_i, \mathbf{q}_{i,0}, \dots, \mathbf{q}_{i,k_b-1}) := \mathsf{ConvertToVOLE}(\bot, \mathsf{sd}_{i,1}, \dots, \mathsf{sd}_{i,N_b-1}, \mathsf{iv}; \hat{\ell})$
8 : $\quad \mathbf{Q}'_i := [\mathbf{q}_{i,0} \cdots \mathbf{q}_{i,k_b-1}] \in \mathbb{F}_2^{\hat{\ell} \times k_b}$ // stored in column major representation
9 : $h_{\mathsf{com}} := \mathsf{H}_1(\mathsf{com}_0 \| \dots \| \mathsf{com}_{\tau-1})$
10 : **return** $(h_{\mathsf{com}}, \mathbf{Q}'_0, \dots, \mathbf{Q}'_{\tau-1})$

Fig. 5.5: FAEST-VOLE reconstruction

all τ vector commitments, allowing to verify openings of these later more efficiently, and additionally outputs information decom_i allowing to efficiently decommit vector commitment i later.

5.3.3 VOLE Reconstruction Once the signer has committed to its execution of the VOLEitH protocol with VOLECommit, the verifier can use the commitments, together with the challenge that generates the Δ key, to reconstruct the matching VOLE keys using VOLEReconstruct.

VOLEReconstruct(). The algorithm decomposes the challenge chall into τ challenges $\Delta_0, \ldots, \Delta_{\tau-1}$ of the τ VOLE instances generated by VOLECommit. These Δ_i, along with partial decommitments $\mathsf{pdecom}_i, \overline{\mathsf{com}}_i$, are used to recompute corresponding vector commitment openings for all positions except Δ_i (as well as to recompute the actual com_i). The algorithm permutes the VC openings by Δ_i and compresses them by applying ConvertToVOLE, thus obtaining τ matrices of VOLE receiver messages $\mathbf{Q}'_i$ for each VOLE instance where the challenge is Δ_i. Note that these $\mathbf{Q}'_i$ are for possibly different messages and must later be corrected (using $\mathbf{c}_i$) outside of this function. Moreover, VOLEReconstruct during VC opening recomputes all com_i and their hash can then be used to check if the partial openings are valid decommitments for all vector commitments.

6 AES Functions

The principal FAEST algorithms use the following building blocks to perform AES-related operations to compute the QuickSilver proof of knowledge of the OWF for the FAEST-λ parameter sets.

- Section 6.1: FAEST.AES.ExtendWitness in Figure 6.1
- Section 6.2: FAEST.AES.KeyExpFwd in Figure 6.2.
- Section 6.2: FAEST.AES.KeyExpBkwd in Figure 6.3.
- Section 6.2: FAEST.AES.KeyExpCstrnts in Figure 6.4.
- Section 6.3: FAEST.AES.EncFwd in Figure 6.6.
- Section 6.3: FAEST.AES.EncBkwd in Figure 6.7.
- Section 6.3: FAEST.AES.EncCstrnts in Figure 6.8.
- Section 6.4: FAEST.AES.AESProve in Figure 6.9.
- Section 6.4: FAEST.AES.AESVerify in Figure 6.11.

As it does not make use of the Rijndael algorithm, this section is written assuming that $N_{\mathsf{st}} = 4$.

6.1 Witness Extension

The AES.ExtendWitness algorithm (Fig. 6.1), generates the extended witness $\mathbf{w} \in \{0,1\}^\ell$ for use with the AES.AESProve algorithm; it takes as input the AES key $\mathbf{k}$, the FAEST public key pk and the instance parameters `param` and $\mathtt{param}_{\mathsf{OWF}}$.

To derive the extended witness $\mathbf{w}$, ExtendWitness first executes the KeyExpansion routine for AES, as described in Section 4.1, to generate the expanded key $\bar{\mathbf{k}}$. It then saves the words of the expanded key which depend on the AES key $\mathbf{k}$ as `key_bits` (i.e. the first N_{k} words), and the words that depend on the SubWord operation as `non-lin_word_bits` (i.e. every 4 or 6 words thereafter, depending on λ).

```
FAEST.AES.ExtendWitness(k, pk; param, param_OWF)
1 :  Parse pk = (in, out) ∈ {0,1}^{128β} × {0,1}^{128β}   // decompose OWF key
2 :  new w ∈ {0,1}^{≤ℓ}   // New empty extended witness of length at most ℓ
       // Key expansion routine and saving witness bits
3 :  new k̄ ∈ [{0,1}^32; 4(R+1)]   // Array of words for expanded key
4 :  k̄ := KeyExpansion(k, N_k)
5 :  w := k̄[0..N_k − 1]   // Saving the first N_k words, which contain k
6 :  i_k := N_k
7 :  for j ∈ [0..S_ke/4)  do
8 :    w := w||k̄[i_k]   // Saving the word that depends on SubWord
9 :    if λ = 192  then i_k := i_k + 6
10 :   else i_k := i_k + 4
       // Encryption routine and saving witness bits
11 : for b ∈ [0..β)  do
12 :   new state ∈ {0,1}^128
13 :   state := in_b
14 :   AddRoundKey(state, k̄[0..3])
15 :   for j ∈ [1..R)  do
16 :     SubBytes(state)
17 :     ShiftRows(state)
18 :     w := w||state   // Saving the state bits, in column-major order
19 :     MixColumns(state)
20 :     AddRoundKey(state, k̄[4j..4j + 3])
       // Last round is not committed to, so not computed
21 : return w ∈ {0,1}^ℓ
```

Fig. 6.1: Extending the witness from the AES key and the input plaintext block(s).

For each block $b \in \{0,1\}$ of the plaintext input **in** (one block if $\lambda = 128$, two blocks if $\lambda \in \{192, 256\}$, extracted from pk), ExtendWitness next runs through the encryption routine operations by initializing an empty state and performing the AES round operations, as described in Section 4.1. After each ShiftRows operation (for the first $R-1$ rounds), the algorithm records the current bits of the state (*in column-major order*) denoted as $\texttt{shift_row_bits}_b$, into the extended witness $\mathbf{w}$. The state after the last round of ShiftRows is not recorded, since it can be derived linearly during AESProve using the FAEST public key output bits **out** and the expanded key.

It finally outputs the extended witness in the form:

$$\mathbf{w} := \begin{cases} \texttt{key_bits} \| \texttt{non-lin_word_bits} \| \texttt{shift_row_bits}_0 & \lambda = 128 \\ \texttt{key_bits} \| \texttt{non-lin_word_bits} \| \texttt{shift_row_bits}_0 \| \texttt{shift_row_bits}_1 & \lambda \neq 128 \end{cases}$$

6.2 Deriving Constraints for the Key Expansion Routine

The AES.KeyExpCstrnts algorithm, described in Figure 6.4, generates the first S_{ke} constraint values for the QuickSilver proof of the AES circuit computed by the AES.AESProve algorithm. To do so, it must derive the inputs and outputs of the

Inputs

m	$\{1, \lambda\}$	size of the field elements.
$\boldsymbol{x}$	$\mathbb{F}_{2^m}^{\ell_{\mathsf{ke}}}$	extended witness values, VOLE tags, or VOLE keys.
Mtag	$\{0, 1\}$	1 if $\boldsymbol{x}$ is VOLE tags, 0 otherwise.
Mkey	$\{0, 1\}$	1 if $\boldsymbol{x}$ is VOLE keys, 0 otherwise.
Δ	$\mathbb{F}_{2^\lambda} \cup \{\perp\}$	global VOLE key if $\mathtt{Mkey} = 1$, $\perp$ otherwise.

FAEST.AES.KeyExpFwd$(m, \boldsymbol{x}, \mathtt{Mtag}, \mathtt{Mkey}, \Delta; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$

```
1 :  if Mtag = 1 ∧ Mkey = 1 or Mkey = 1 ∧ Δ = ⊥  then return ⊥
2 :  new y ∈ F_{2^m}^{(R+1)·128}    // The output array
3 :  for i ∈ [0..λ)  do y[i] := x[i]    // First N_k words
4 :  i_wd := λ    // Index to read words from x
5 :  for j ∈ [N_k..4(R+1))  do    // Remaining key words, R+1 round keys, 4 words each
6 :    if j mod N_k = 0 or (N_k > 6 and j mod N_k = 4)
7 :      y[32j..32j+31] := x[i_wd..i_wd+31]    // Selecting expanded key word bits
8 :      i_wd := i_wd + 32
9 :    else
10 :     for i ∈ [0..32)  do   y[32j+i] = y[32(j−N_k)+i] + y[32(j−1)+i]
11 : return y ∈ F_{2^m}^{(R+1)·128}
```

Fig. 6.2: Transforming witness or VOLE values into $\mathbb{F}_{2^8}$-inverse input values for the key schedule routine.

$\mathbb{F}_{2^8}$ inversions within the S-boxes from the witness values and their VOLE tags, or the VOLE keys, passed by AESProve. This is the role of the AES.KeyExpFwd and AES.KeyExpBkwd algorithms, described in Figures 6.2 and 6.3.

KeyExpFwd. Given the security parameter λ, a dimension parameter m, a vector of witness values, their VOLE tags or their VOLE keys $\boldsymbol{x}$, two flags Mtag and Mkey, and optionally the global VOLE key Δ, this algorithm derives the $R+1$ round keys for the AES encryption routine.

KeyExpBkwd. Given the security parameter λ, a dimension parameter m, a vector of witness values, their VOLE tags or their VOLE keys $\boldsymbol{x}$, a vector of expanded key values, their VOLE tags or their VOLE keys $\boldsymbol{x_k}$, two flags Mtag and Mkey, and optionally the global VOLE key Δ, this algorithm derives the bits representing the S_{ke} outputs of the $\mathbb{F}_{2^8}$ inverse operations from the S-boxes in the key expansion routine.

KeyExpCstrnts. Given the security parameter λ, the extended witness values for the key expansion $\boldsymbol{w}$, their VOLE tags $\boldsymbol{v}$, a flag Mkey, optionally the VOLE keys for the witness values $\boldsymbol{q}$ and optionally the global VOLE key Δ, this algorithm derives S_{ke} constraint values for the QuickSilver proof of AES, either $(\boldsymbol{a}_0, \boldsymbol{a}_1)$ if $\mathtt{Mkey} = 0$, or $\boldsymbol{b}$ if $\mathtt{Mkey} = 1$.

Note. The KeyExpFwd and KeyExpBkwd perform the same arithmetic operations either on circuit values w, their VOLE tags v or their VOLE keys q. Since the homomorphic property of the VOLE MACs prescribes different operations for addition of constants in the case of tags and keys, we use the Mtag and Mkey Boolean flags

Inputs

m	$\{1, \lambda\}$	size of the field elements.
$\boldsymbol{x}$	$\mathbb{F}_{2^m}^{\ell_{\mathsf{ke}}}$	extended witness values, VOLE tags, or VOLE keys.
$\boldsymbol{x_k}$	$\mathbb{F}_{2^m}^{\ell_{\mathsf{ke}}}$	expanded key values, VOLE tags or VOLE keys.
Mtag	$\{0, 1\}$	1 if $\boldsymbol{x}$ is VOLE tags, 0 otherwise.
Mkey	$\{0, 1\}$	1 if $\boldsymbol{x}$ is VOLE keys, 0 otherwise.
Δ	$\mathbb{F}_{2^\lambda} \cup \{\bot\}$	global VOLE key if $\mathtt{Mkey} = 1$, $\bot$ otherwise.

FAEST.AES.KeyExpBkwd$(m, \boldsymbol{x}, \boldsymbol{x_k}, \mathtt{Mtag}, \mathtt{Mkey}, \Delta; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$

```
1 :  if Mtag = 1 ∧ Mkey = 1 or Mkey = 1 ∧ Δ = ⊥ return ⊥
2 :  new y ∈ F_{2^m}^{8 S_ke}    // The output array
3 :  i_wd := 0    // Index to read words from x_k
4 :  c := 0    // Counting S-boxes
5 :  rmvRcon := True, i_rcon := 0    // Handling round constant removal
6 :  for j ∈ [0..S_ke)  do    // Iterating S-box-wise
7 :    x̃ := x[8j..8j + 7] − x_k[i_wd + 8c..i_wd + 8c + 7]    // Removing XOR-ed byte
8 :    if ¬Mtag and rmvRcon and c = 0  then    // Removing Rcon[]
9 :      rcon := Rcon[i_rcon] ∈ {0, 1}^8, i_rcon := i_rcon + 1    // See Table 4.1 for Rcon[i]
10 :     new r ∈ F_{2^m}^8
11 :     for i ∈ [0..8)  do    // Lifting the bits of the round constant one by one
12 :       r[i] := rcon[i] · (Mkey · Δ + (1_{F_{2^m}} − Mkey))
13 :       x̃[i] := x̃[i] − r[i]
14 :   new ỹ ∈ F_{2^m}^8    // Array for the affine layer inverse
15 :   for i ∈ [0..8)  do
16 :     ỹ[i] := x̃[i − 1 mod 8] + x̃[i − 3 mod 8] + x̃[i − 6 mod 8]
17 :   ỹ[0] := ỹ[0] + (1_{F_{2^m}} − Mtag) · (Mkey · Δ + (1_{F_{2^m}} − Mkey))
18 :   ỹ[2] := ỹ[2] + (1_{F_{2^m}} − Mtag) · (Mkey · Δ + (1_{F_{2^m}} − Mkey))
19 :   y[8j..8j + 7] := ỹ[0..7]    // Storing the affine layer inverse
20 :   c := c + 1
21 :   if c = 4  then    // Move i_wd every 4 S-boxes
22 :     c := 0
23 :     if λ = 192  then i_wd := i_wd + 192
24 :     else
25 :       i_wd := i_wd + 128
26 :       if λ = 256  then rmvRcon := ¬rmvRcon
27 :  return y ∈ F_{2^m}^{8 S_ke}
```

Fig. 6.3: Transforming witness or VOLE values into $\mathbb{F}_{2^8}$-inverse output values for the key schedule routine.

Inputs listed in Table 6.5.

FAEST.AES.KeyExpCstrnts($\boldsymbol{w}, \boldsymbol{v}, \texttt{Mkey}, \boldsymbol{q}, \Delta; \texttt{param}, \texttt{param}_{\mathsf{OWF}}$)

1 : **if** $\texttt{Mkey} = 0$ // If computing wire values and tags
2 : $\boldsymbol{k}$ = KeyExpFwd($1, \boldsymbol{w}, 0, 0, \perp; \texttt{param}, \texttt{param}_{\mathsf{OWF}}$) // Expanded key
3 : $\boldsymbol{v}_{\boldsymbol{k}}$ = KeyExpFwd($\lambda, \boldsymbol{v}, \texttt{Mtag} = 1, 0, \perp; \texttt{param}, \texttt{param}_{\mathsf{OWF}}$) // Expanded key tags
4 : $\bar{\boldsymbol{w}}$ = KeyExpBkwd($1, \boldsymbol{w}[\lambda..], \boldsymbol{k}, 0, 0, \perp; \texttt{param}, \texttt{param}_{\mathsf{OWF}}$) // Inverse outputs
5 : $\boldsymbol{v}_{\bar{\boldsymbol{w}}}$ = KeyExpBkwd($\lambda, \boldsymbol{v}[\lambda..], \boldsymbol{v}_{\boldsymbol{k}}, \texttt{Mtag} = 1, 0, \perp; \texttt{param}, \texttt{param}_{\mathsf{OWF}}$) // Inverse output tags
6 : $i_{\mathsf{wd}} := 32(N_{\mathsf{k}} - 1)$ // Index of the first word to read
7 : $\texttt{doRotWord} = \texttt{True}$ // Handling performing RotWord for AES256
8 : **for** $j \in [0..S_{\mathsf{ke}}/4)$ **do** // Iterating word-wise
9 : **new** $\hat{\boldsymbol{k}}, \boldsymbol{v}_{\hat{\boldsymbol{k}}}, \hat{\boldsymbol{w}}, \boldsymbol{v}_{\hat{\boldsymbol{w}}} \in \mathbb{F}_{2^\lambda}^4$
10 : **for** $r \in [0..3]$ **do** // Combining $\mathbb{F}_2$ values and VOLE tags into $\mathbb{F}_{2^8}$ values and tags
11 : $r' := r$
12 : **if** $\texttt{doRotWord}$ **then** $r' := r + 3 \bmod 4$ // Rotating the index performs RotWord
13 : $\hat{\boldsymbol{k}}[r']$:= ByteCombine($\boldsymbol{k}[(i_{\mathsf{wd}} + 8r)..(i_{\mathsf{wd}} + 8r + 7)]; \lambda$) // Note writing index r'
14 : $\boldsymbol{v}_{\hat{\boldsymbol{k}}}[r']$:= ByteCombine($\boldsymbol{v}_{\boldsymbol{k}}[(i_{\mathsf{wd}} + 8r)..(i_{\mathsf{wd}} + 8r + 7)]; \lambda$) // Note writing index r'
15 : $\hat{\boldsymbol{w}}[r]$:= ByteCombine($\bar{\boldsymbol{w}}[32j + 8r..32j + 8r + 7]; \lambda$) // **Note** index counter j
16 : $\boldsymbol{v}_{\hat{\boldsymbol{w}}}[r]$:= ByteCombine($\boldsymbol{v}_{\boldsymbol{w}}[32j + 8r..32j + 8r + 7]; \lambda$)
17 : **if** $\lambda = 256$ **then** $\texttt{doRotWord} := \neg\texttt{doRotWord}$ // Only alternate RotWord for AES256
18 : **for** $r \in [0..3]$ **do** // Computing A_0, A_1, 4 S-boxes at a time
19 : $A_{0,4j+r} := \boldsymbol{v}_{\hat{\boldsymbol{k}}}[r] \cdot \boldsymbol{v}_{\hat{\boldsymbol{w}}}[r]$
20 : $A_{1,4j+r} := (\hat{\boldsymbol{k}}[r] + \boldsymbol{v}_{\hat{\boldsymbol{k}}}[r]) \cdot (\hat{\boldsymbol{w}}[r] + \boldsymbol{v}_{\hat{\boldsymbol{w}}}[r]) - 1_{\mathbb{F}_{2^8}} - A_{0,4j+r}$ // Tag of product is 0
21 : **if** $\lambda = 192$ **then** $i_{\mathsf{wd}} := i_{\mathsf{wd}} + 192$
22 : **else** $i_{\mathsf{wd}} := i_{\mathsf{wd}} + 128$
23 : **return** $(A_{0,0}, \ldots, A_{0,S_{\mathsf{ke}}-1}) \in \mathbb{F}_{2^\lambda}^{S_{\mathsf{enc}}}, (A_{1,0}, \ldots, A_{1,S_{\mathsf{ke}}-1}) \in \mathbb{F}_{2^\lambda}^{S_{\mathsf{enc}}}, \boldsymbol{k}, \boldsymbol{v}_{\boldsymbol{k}}$ // **Note** return of $\boldsymbol{k}, \boldsymbol{v}_{\boldsymbol{k}}$
24 : **else** // If computing tag keys
25 : $\boldsymbol{q}_{\boldsymbol{k}}$ = KeyExpFwd($\lambda, \boldsymbol{q}, 0, \texttt{Mkey} = 1, \Delta; \texttt{param}, \texttt{param}_{\mathsf{OWF}}$) // VOLE keys of the expanded key
26 : $\boldsymbol{q}_{\bar{\boldsymbol{w}}}$ = KeyExpBkwd($\lambda, \boldsymbol{q}[\lambda..], \boldsymbol{q}_{\boldsymbol{k}}, 0, \texttt{Mkey} = 1, \Delta; \texttt{param}, \texttt{param}_{\mathsf{OWF}}$) // Inverse output keys
27 : $i_{\mathsf{wd}} := 32(N_{\mathsf{k}} - 1)$ // Index of the first word to read
28 : $\texttt{doRotWord} := \texttt{True}$ // Handling performing RotWord for AES256
29 : **for** $j \in [0..S_{\mathsf{ke}}/4)$ **do** // Iterating word-wise
30 : **new** $\hat{\boldsymbol{q}}_{\boldsymbol{k}}, \hat{\boldsymbol{q}}_{\bar{\boldsymbol{w}}} \in \mathbb{F}_{2^\lambda}^4$
31 : **for** $r \in [0..3]$ **do** // Combining VOLE keys over $\mathbb{F}_2$ into VOLE keys over $\mathbb{F}_{2^8}$
32 : $r' := r$
33 : **if** $\texttt{doRotWord}$ **then** $r' := r + 3 \bmod 4$ // Rotating the index performs RotWord
34 : $\hat{\boldsymbol{q}}_{\boldsymbol{k}}[r']$:= ByteCombine($\boldsymbol{q}_{\boldsymbol{k}}[(i_{\mathsf{wd}} + 8r)..(i_{\mathsf{wd}} + 8r + 7)], \lambda$)
35 : $\hat{\boldsymbol{q}}_{\bar{\boldsymbol{w}}}[r]$:= ByteCombine($\boldsymbol{q}_{\bar{\boldsymbol{w}}}[(32j + 8r)..(32j + 8r + 7)], \lambda$) // **Note** index counter j
36 : **if** $\lambda = 256$ **then** $\texttt{doRotWord} := \neg\texttt{doRotWord}$ // Only alternate RotWord for AES256
37 : **for** $r \in [0..3]$ **do** // Computing B, 4 S-boxes at a time
38 : $B_{4j+r} := \hat{\boldsymbol{q}}_{\boldsymbol{k}}[r] \cdot \hat{\boldsymbol{q}}_{\bar{\boldsymbol{w}}}[r] - \Delta \cdot \Delta$ // Δ is VOLE key of expected result $\boldsymbol{k} \cdot \bar{\boldsymbol{w}} = 1$
39 : **if** $\lambda = 192$ **then** $i_{\mathsf{wd}} := i_{\mathsf{wd}} + 192$
40 : **else** $i_{\mathsf{wd}} := i_{\mathsf{wd}} + 128$
41 : **return** $(B_0, \ldots, B_{S_{\mathsf{ke}}-1}) \in \mathbb{F}_{2^\lambda}^{S_{\mathsf{enc}}}, \boldsymbol{q}_{\boldsymbol{k}}$ // **Note** return of $\boldsymbol{q}_{\boldsymbol{k}}$

Fig. 6.4: Deriving the constraint values for the AESλ key expansion routine.

to indicate what kind of values the algorithm is computing. In KeyExpCstrnts the signer and the verifier set these flags as follows, depending on the kind of values they wish to compute.

Party	Kind	`Mtag`	`Mkey`
Signer	Wire values	0	0
	VOLE tags	1	0
Verifier	VOLE keys	0	1

The same applies for the algorithms related to the encryption routine in Sections 6.3 and 7.2.

6.2.1 KeyExpFwd Description.

To compute the values for inputs of the S-boxes in the key expansion routine, this algorithm in fact computes the values for all of the required round keys, so that they may be used in the encryption routine algorithms; the KeyExpCstrnts algorithm will select which of these it requires to derive the constraint values.

Following the AES key expansion routine, the KeyExpFwd algorithm initialises the first λ values (equivalently the first N_{k} words) of the expanded key with the first λ values of the input vector $\boldsymbol{x}$. It then proceeds with one word of the expanded key at a time, starting from index N_{k}. If the word index j corresponds to a word derived from a SubWord operation (i.e. when $j \bmod N_{\mathsf{k}} = 0$ or when $N_{\mathsf{k}} > 6$ and $j \bmod N_{\mathsf{k}} = 4$) then it reads the word from the value vector $\boldsymbol{x}$ and places it in the expanded key. Otherwise it computes the XOR, represented here by field addition, of two previous words (as per the key expansion specification) and stores the result as the new word.

6.2.2 KeyExpBkwd Description.

To compute the values for outputs of the $\mathbb{F}_{2^8}$ inversions within the S-boxes in the key expansion routine, this algorithm selects the relevant values from the expanded key values $\boldsymbol{x}$, removes the byte that was XOR-ed into the key word and is contained in $\boldsymbol{x_k}$, removes the round constant if necessary (i.e. when not computing VOLE tags), and inverts the S-box $\mathbb{F}_2$-affine layer.

To simplify writing into the output array $\boldsymbol{y}$, 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 only one out of every two relevant words for AES256 have the round constant added into them, the boolean flag `rmvRcon` (which is always `true` for $\lambda = 128, 192$), is negated every 4 S-boxes for $\lambda = 256$.

6.2.3 KeyExpCstrnts Description.

The inputs for the KeyExpCstrnts algorithm are listed in Table 6.5. To derive the S_{ke} constraint values, this algorithm first calls KeyExpFwd to derive the extended key values and VOLE tags $\boldsymbol{k}$ and $\boldsymbol{v_k}$, or VOLE keys $\boldsymbol{q_k}$, depending on `Mkey`. It then calls KeyExpBkwd on the witness values and VOLE tags, or VOLE keys, together with the expanded key values and VOLE tags, or VOLE keys, to derive the corresponding elements for the outputs of the $\mathbb{F}_{2^8}$ inversions within the S-boxes.

Input	Type	Description
$\boldsymbol{w}$	$\mathbb{F}_2^{\ell_{\mathsf{ke}}} \cup \{\perp\}$	Witness values for the key expansion, or $\perp$ if computing keys.
$\boldsymbol{v}$	$\mathbb{F}_{2^\lambda}^{\ell_{\mathsf{ke}}} \cup \{\perp\}$	VOLE tags for the key expansion, or $\perp$ if computing keys.
`Mkey`	$\{0,1\}$	Whether this algorithm is operating on VOLE keys or not.
$\boldsymbol{q}$	$\mathbb{F}_{2^\lambda}^{\ell_{\mathsf{ke}}} \cup \{\perp\}$	VOLE keys for the key expansion, or $\perp$ if computing values or tags.
Δ	$\mathbb{F}_{2^\lambda} \cup \{\perp\}$	The VOLE global key if computing VOLE keys, $\perp$ otherwise.

Table 6.5: Inputs to the KeyExpCstrnts algorithm.

For each set of four S-boxes, the KeyExpCstrnts algorithm reads the previous expanded key elements (tracked with the i_{wd} index) and combine them to an element of the F_{2^8} (sub)field using ByteCombine in order to derive the inputs to the $\mathbb{F}_{2^8}$ inversions of the S-boxes; it similarly combines the outputs produced by KeyExpBkwd. Then, for each of the four S-boxes, it computes the A_0 and A_1 values, or B values for VOLE keys, as described in Section 2.3. Finally, it returns the vector(s) of constraint values, either $(\boldsymbol{a}_0, \boldsymbol{a}_1)$ or $\boldsymbol{b}$, together with the expanded key values and VOLE tags, or VOLE keys, depending on `Mkey`.

6.3 Deriving Constraints for the Encryption Routine

The AES.EncCstrnts algorithm (Fig. 6.8) generates a vector of S_{enc} constraint values for the QuickSilver proof of the AES circuit computed by the AES.AESProve algorithm. To do so, it must derive the inputs and outputs of the $\mathbb{F}_{2^8}$ inversions contained within the S-boxes from the witness values and their VOLE tags, or the VOLE keys, passed by AESProve. This is the role of the AES.EncFwd and AES.EncBkwd algorithms, described in Fig. 6.6 and Fig. 6.7.

EncFwd. Given the security parameter λ, a dimension parameter m, a vector of witness values, their VOLE tags or their VOLE keys $\boldsymbol{x}$, a vector of expanded key values, their VOLE tags or their VOLE keys $\boldsymbol{x_k}$, an AES plaintext block **in**, two flags `Mtag` and `Mkey`, and optionally the global VOLE key Δ, this algorithm derives the S_{enc} inputs of the $\mathbb{F}_{2^8}$ inversions in the S-boxes of one execution of the encryption routine.

EncBkwd. Given the security parameter λ, a dimension parameter m, a vector of witness values, their VOLE tags or their VOLE keys $\boldsymbol{x}$, a vector of expanded key values, their VOLE tags or their VOLE keys $\boldsymbol{x_k}$, an AES ciphertext block **out**, two flags `Mtag` and `Mkey`, and optionally the global VOLE key Δ, this algorithm derives the S_{enc} outputs of the $\mathbb{F}_{2^8}$ inversions in the S-boxes of one execution of the encryption routine.

EncCstrnts. Given the security parameter λ, the extended witness values for one execution of the encryption routine $\boldsymbol{w}$, their VOLE tags $\boldsymbol{v}$, a flag `Mkey`, optionally the VOLE keys for the witness values $\boldsymbol{q}$ and optionally the global VOLE key Δ, this algorithm derives S_{enc} constraint values for the QuickSilver proof of AES, either $(\boldsymbol{a}_0, \boldsymbol{a}_1)$ if `Mkey` $= 0$, or $\boldsymbol{B}$ if `Mkey` $= 1$.

6.3.1 EncFwd Description.

To compute the values for inputs of the S-boxes in the encryption routine, this algorithm selects the ShiftRows output bit values,

Inputs

m	$\{1, \lambda\}$	size of the field elements.
$\boldsymbol{x}$	$\mathbb{F}_{2^m}^{\ell_{\mathsf{enc}}}$	extended witness values, VOLE tags, or VOLE keys.
$\boldsymbol{x_k}$	$\mathbb{F}_{2^m}^{128(R+1)}$	expanded key values, VOLE tags, or VOLE keys.
in or **out**	$\{0,1\}^{128}$	AES plaintext or ciphertext block.
`Mtag`	$\{0,1\}$	1 if $\boldsymbol{x}$ and $\boldsymbol{x_k}$ are VOLE tags, 0 otherwise.
`Mkey`	$\{0,1\}$	1 if $\boldsymbol{x}$ and $\boldsymbol{x_k}$ are VOLE keys, 0 otherwise.
$\varDelta$	$\mathbb{F}_{2^\lambda} \cup \{\perp\}$	global VOLE key if `Mkey` $= 1$, $\perp$ otherwise.

FAEST.AES.EncFwd$(m, \boldsymbol{x}, \boldsymbol{x_k}, \mathbf{in}, \mathtt{Mtag}, \mathtt{Mkey}, \varDelta; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$

1 : **new** $\boldsymbol{y} \in \mathbb{F}_{2^\lambda}^{S_{\mathsf{enc}}}$ // The returned array of bytes
2 : **for** $i \in [0..16)$ **do** // First AddRoundKey operation
3 : **for** $j \in [0..8)$ **do**
4 : $\boldsymbol{x}_{\mathbf{in}}[j] = \mathbf{in}[8i+j] \cdot (1_{\mathbb{F}_{2^m}} - \mathtt{Mtag}) \cdot (\mathtt{Mkey} \cdot \varDelta + (1_{\mathbb{F}_{2^m}} - \mathtt{Mkey}))$
5 : $\boldsymbol{y}[i] := \mathsf{ByteCombine}(\boldsymbol{x}_{\mathbf{in}}[0..7]; \lambda) + \mathsf{ByteCombine}(\boldsymbol{x_k}[8i..8i+7]; \lambda)$
6 : **for** $j \in [1..R)$ **do** // Iterating round-wise
7 : **for** $c \in [0..3]$ **do** // Computing MixColumns and AddRoundKey column-wise
8 : $i_{\boldsymbol{x}} := 128(j-1) + 32c$ // Index to read from the witness or VOLE array
9 : $i_{\boldsymbol{k}} := 128j + 32c$ // Index to read from the round key array
10 : $i_{\boldsymbol{y}} := 16j + 4c$ // Index to write to the state array
11 : **new** $\hat{\boldsymbol{x}}, \hat{\boldsymbol{x}}_{\boldsymbol{k}} \in \mathbb{F}_{2^\lambda}^4$
12 : **for** $r \in [0..3]$ **do**
// Reading 8 elements and combining for $\mathbb{F}_{2^8}$ arithmetic within $\mathbb{F}_{2^\lambda}$
13 : $\hat{\boldsymbol{x}}[r] := \mathsf{ByteCombine}(\boldsymbol{x}[i_{\boldsymbol{x}} + 8r..i_{\boldsymbol{x}} + 8r + 7]; \lambda)$
14 : $\hat{\boldsymbol{x}}_{\boldsymbol{k}}[r] := \mathsf{ByteCombine}(\boldsymbol{x_k}[i_{\boldsymbol{k}} + 8r..i_{\boldsymbol{k}} + 8r + 7]; \lambda)$
// $\mathbf{1}, \mathbf{2}, \mathbf{3}$ are elements of the F_{2^8} sub-field of $\mathbb{F}_{2^\lambda}$
15 : $\mathbf{1} := \mathsf{ByteCombine}(\{\mathtt{01}\}; \lambda)$, $\mathbf{2} := \mathsf{ByteCombine}(\{\mathtt{02}\}; \lambda)$, $\mathbf{3} := \mathsf{ByteCombine}(\{\mathtt{03}\}; \lambda)$
16 : $\boldsymbol{y}[i_{\boldsymbol{y}} + 0] := \hat{\boldsymbol{x}}[0] \cdot \mathbf{2} + \hat{\boldsymbol{x}}[1] \cdot \mathbf{3} + \hat{\boldsymbol{x}}[2] \cdot \mathbf{1} + \hat{\boldsymbol{x}}[3] \cdot \mathbf{1} + \hat{\boldsymbol{x}}_{\boldsymbol{k}}[0]$
17 : $\boldsymbol{y}[i_{\boldsymbol{y}} + 1] := \hat{\boldsymbol{x}}[0] \cdot \mathbf{1} + \hat{\boldsymbol{x}}[1] \cdot \mathbf{2} + \hat{\boldsymbol{x}}[2] \cdot \mathbf{3} + \hat{\boldsymbol{x}}[3] \cdot \mathbf{1} + \hat{\boldsymbol{x}}_{\boldsymbol{k}}[1]$
18 : $\boldsymbol{y}[i_{\boldsymbol{y}} + 2] := \hat{\boldsymbol{x}}[0] \cdot \mathbf{1} + \hat{\boldsymbol{x}}[1] \cdot \mathbf{1} + \hat{\boldsymbol{x}}[2] \cdot \mathbf{2} + \hat{\boldsymbol{x}}[3] \cdot \mathbf{3} + \hat{\boldsymbol{x}}_{\boldsymbol{k}}[2]$
19 : $\boldsymbol{y}[i_{\boldsymbol{y}} + 3] := \hat{\boldsymbol{x}}[0] \cdot \mathbf{3} + \hat{\boldsymbol{x}}[1] \cdot \mathbf{1} + \hat{\boldsymbol{x}}[2] \cdot \mathbf{1} + \hat{\boldsymbol{x}}[3] \cdot \mathbf{2} + \hat{\boldsymbol{x}}_{\boldsymbol{k}}[3]$
20 : **return** $\boldsymbol{y} \in \mathbb{F}_{2^\lambda}^{S_{\mathsf{enc}}}$ // F_{2^8}-inverse inputs for the R S-box layers

Fig. 6.6: Transforming witness or VOLE values into $\mathbb{F}_{2^8}$-inverse inputs for the encryption routine.

Inputs

m	$\{1, \lambda\}$	size of the field elements.
$\boldsymbol{x}$	$\mathbb{F}_{2^m}^{\ell_{\mathsf{enc}}}$	extended witness values, VOLE tags, or VOLE keys.
$\boldsymbol{x_k}$	$\mathbb{F}_{2^m}^{128(R+1)}$	expanded key values, VOLE tags, or VOLE keys.
in or **out**	$\{0,1\}^{128}$	AES plaintext or ciphertext block.
$\mathtt{Mtag}$	$\{0,1\}$	1 if $\boldsymbol{x}$ and $\boldsymbol{x_k}$ are VOLE tags, 0 otherwise.
$\mathtt{Mkey}$	$\{0,1\}$	1 if $\boldsymbol{x}$ and $\boldsymbol{x_k}$ are VOLE keys, 0 otherwise.
Δ	$\mathbb{F}_{2^\lambda} \cup \{\perp\}$	global VOLE key if $\mathtt{Mkey} = 1$, $\perp$ otherwise.

FAEST.AES.EncBkwd$(m, \boldsymbol{x}, \boldsymbol{x_k}, \mathbf{out}, \mathtt{Mtag}, \mathtt{Mkey}, \Delta; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$

```
1 :  new y ∈ F_{2^λ}^{S_enc}   // The returned array of bytes
2 :  for j ∈ [0..R)  do   // Iterating round-wise
3 :    for c ∈ [0..3]  do   // Iterating column-wise
4 :      for r ∈ [0..3]  do   // Iterating row-wise
5 :        i_rd := 128j + 32(c − r mod 4) + 8r   // c − r mod 4 inverts ShiftRows
6 :        if j < R − 1  then
7 :          x̃[0..7] := x[i_rd..i_rd + 7]
8 :        else
9 :          new x_out ∈ F_{2^m}^8
10 :         for i ∈ [0..8)  do
11 :           x_out[i] = out[i_rd − 128j + i] · (1_{F_{2^m}} − Mtag) · (Mkey · Δ + (1_{F_{2^m}} − Mkey))
12 :         x̃[0..7] := x_out[0..7] − x_k[128 + i_rd..128 + i_rd + 7]   // Inverting AddRoundKey
13 :       new ỹ ∈ F_{2^m}^8   // Array for the affine layer inverse
14 :       for i ∈ [0..8)  do
15 :         ỹ[i] := x̃[i − 1 mod 8] + x̃[i − 3 mod 8] + x̃[i − 6 mod 8]
16 :       ỹ[0] := ỹ[0] + (1_{F_{2^m}} − Mtag) · (Mkey · Δ + (1_{F_{2^m}} − Mkey))
17 :       ỹ[2] := ỹ[2] + (1_{F_{2^m}} − Mtag) · (Mkey · Δ + (1_{F_{2^m}} − Mkey))
18 :       y[16j + 4c + r] := ByteCombine(ỹ[0..7]; λ)
19 :  return y ∈ F_{2^λ}^{S_enc}   // F_{2^8}-inverse outputs for the R S-box layers
```

Fig. 6.7: Transforming witness or VOLE values into $\mathbb{F}_{2^8}$-inverse outputs for the encryption routine.

Inputs

in and **out**	$\{0,1\}^{128}$	AES plaintext or ciphertext blocks.
$\boldsymbol{w}$	$\mathbb{F}_2^{\ell_{\mathsf{enc}}} \cup \{\bot\}$	extended witness values, or $\bot$ if $\mathtt{Mkey} = 1$.
$\boldsymbol{v}$	$\mathbb{F}_{2^\lambda}^{\ell_{\mathsf{enc}}} \cup \{\bot\}$	extended witness VOLE tags, or $\bot$ if $\mathtt{Mkey} = 1$.
$\boldsymbol{k}$	$\mathbb{F}_2^{128(R+1)} \cup \{\bot\}$	expanded key values, or $\bot$ if $\mathtt{Mkey} = 1$.
$\boldsymbol{v_k}$	$\mathbb{F}_{2^\lambda}^{128(R+1)} \cup \{\bot\}$	expanded key VOLE tags, or $\bot$ if $\mathtt{Mkey} = 1$.
Mkey	$\{0,1\}$	1 if computing VOLE keys (for verifier), 0 otherwise.
$\boldsymbol{q}$	$\mathbb{F}_{2^\lambda}^{\ell_{\mathsf{enc}}} \cup \{\bot\}$	extended witness VOLE keys for encryption, or $\bot$ if $\mathtt{Mkey} = 0$.
$\boldsymbol{q_k}$	$\mathbb{F}_{2^\lambda}^{128(R+1)} \cup \{\bot\}$	expanded key VOLE keys, or $\bot$ if $\mathtt{Mkey} = 0$.
Δ	$\mathbb{F}_{2^\lambda} \cup \{\bot\}$	global VOLE key if $\mathtt{Mkey} = 1$, $\bot$ otherwise.

```
FAEST.AES.EncCstrnts(in, out, w, v, k, v_k, Mkey, q, q_k, Δ; param, param_OWF)
1 :  if Mkey = 0    // If computing wire values and tags
2 :    s := EncFwd(1, w, k, in, 0, 0, ⊥; param, param_OWF)    // Inverse inputs
3 :    v_s := EncFwd(λ, v, v_k, in, Mtag = 1, 0, ⊥; param, param_OWF)    // Inverse input tags
4 :    s̄ := EncBkwd(1, w, k, out, 0, 0, ⊥; param, param_OWF)    // Inverse outputs
5 :    v_s̄ := EncBkwd(λ, v, v_k, out, Mtag = 1, 0, ⊥; param, param_OWF)    // Inverse output tags
6 :    for j ∈ [0..S_enc) do    // Iterating S-box-wise
7 :      A_{0,j} := v_s[j] · v_s̄[j]
8 :      A_{1,j} := (s[j] + v_s[j]) · (s̄[j] + v_s̄[j]) − 1_{F_{2^8}} − A_{0,j}
9 :    return (A_{0,0}, ..., A_{0,S_enc−1}) ∈ F_{2^λ}^{S_enc}, (A_{1,0}, ..., A_{1,S_enc−1}) ∈ F_{2^λ}^{S_enc}
10 : else    // If computing tag keys
11 :   q_s := EncFwd(λ, q, q_k, in, 0, Mkey = 1, Δ; param, param_OWF)    // Inverse input keys
12 :   q_s̄ := EncBkwd(λ, q, q_k, out, 0, Mkey = 1, Δ; param, param_OWF)    // Inverse output keys
13 :   for j ∈ [0..S_enc) do    // Iterating S-box-wise
14 :     B_j := q_s[j] · q_s̄[j] − Δ · Δ    // Δ is VOLE key of expected result s · s̄ = 1
15 :   return (B_0, ..., B_{S_enc−1}) ∈ F_{2^λ}^{S_enc}
```

Fig. 6.8: Deriving the constraint values for the AES-λ encryption routine.

their VOLE tags or their VOLE keys from $\boldsymbol{x}$, combines them into $\mathbb{F}_{2^8}$ (sub-)field elements, performs the arithmetic for MixColumns and then adds the expanded key values, their VOLE tags or their VOLE keys, previously read as bit values from $\boldsymbol{x_k}$ and combined into $\mathbb{F}_{2^8}$ (sub-)field elements.

Depending on the round index $j \in [1..R)$ and state column index $c \in [0..3]$, the indices $i_{\boldsymbol{x}}$, $i_{\boldsymbol{k}}$ keep track of the reading positions for the witness array $\boldsymbol{x}$ and the expanded key array $\boldsymbol{x_k}$ respectively, and index $i_{\boldsymbol{y}}$ keeps track of the writing position for the return array $\boldsymbol{y}$ which contains the input values for the S-boxes, as $\mathbb{F}_{2^8}$ (sub-) field elements.

6.3.2 EncBkwd Description.

To compute the values for outputs of the inversion operations in the S-boxes of the encryption routine, this algorithm selects the ShiftRows output bit values, their VOLE tags or their VOLE keys from $\boldsymbol{x}$ and first removes the expanded key values, their VOLE tags or their VOLE keys read from $\boldsymbol{x_k}$. (In case it is computing values for the final layer of S-boxes, the values are read from the output block **out** instead and transformed into either VOLE tag or VOLE key values.) Undoing the ShiftRows operation (to work backwards towards the SubBytes operation) is done in the calculation of the i_{rd} index, using $c - r \bmod 4$, which ensures that the correct "un-shifted" value is read.

With the output of the S-box obtained, the $\mathbb{F}_2$-affine transformation is inverted with bit-wise operations before combining the 8 elements into a single $\mathbb{F}_{2^8}$ (sub-) field element and storing it in the return array.

6.3.3 EncCstrnts Description.

To derive a vector of S_{enc} constraints values, this algorithm first calls EncFwd and EncBkwd to derive the input and output values and VOLE tags, or VOLE keys, for each of the S-boxes in the encryption routine. Using these, it computes the A_0 and A_1 values, or B values, for each of the S-boxes, and returns them.

6.4 Proving and Verifying AES Constraints

The AES.AESProve (Fig. 6.9) and AES.AESVerify (Fig. 6.11) algorithms perform the operations of the QuickSilver zero-knowledge proof system instantiated for the AES algorithm.

AESProve. Given the security parameter λ, the extended witness bits $\mathbf{w}$, the VOLE masks $\mathbf{u}$, the VOLE tags $\mathbf{V}$, the FAEST public-key pk, and the QuickSilver challenge chall, this algorithm derives the QuickSilver response $(\tilde{a}, \tilde{b})$.

AESVerify. Given the security parameter λ, the masked witness bits $\mathbf{d}$, the VOLE keys $\mathbf{Q}$, the QuickSilver challenge chall_2, the VOLE challenge chall_3, the QuickSilver proof party $\tilde{a}$, and the FAEST public key pk this algorithm reconstructs parts of the QuickSilver response necessary for verifying the QuickSilver proof.

6.4.1 AESProve Description

In order to compute the QuickSilver response $(\tilde{a}, \tilde{b})$, this algorithm first embeds the witness and VOLE tag bit-strings $\mathbf{w}$ and $\mathbf{V}$ to $\mathbb{F}_2$ and $\mathbb{F}_{2^\lambda}$ respectively. With the first ℓ_{ke} elements of these, it then derives the first S_{ke} constraint values for $\boldsymbol{a}_0$ and $\boldsymbol{a}_1$ using the AES.KeyExpCstrnts algorithm; this also outputs the values and VOLE tags for the expanded AES key, $\boldsymbol{k}$ and $\boldsymbol{v_k}$.

Inputs

w	$\{0,1\}^{\ell}$	extended witness values, as bits.
u	$\{0,1\}^{\ell+\lambda}$	masking values for the QuickSilver response as bits.
V	$\{0,1\}^{(\ell+\lambda)\times\lambda}$	VOLE tags of the masking values, as bit-strings.
pk	$\{0,1\}^{2\cdot\beta\cdot128}$	FAEST public key
chall	$\{0,1\}^{3\lambda+64}$	ZKHash challenge to compute QuickSilver response $(\tilde{a}, \tilde{b})$.

FAEST.AES.AESProve($\mathbf{w}, \mathbf{u}, \mathbf{V}$, pk, chall; $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)

1 : **for** $i \in [0..\ell)$ **do** $\boldsymbol{w}[i] :=$ ToField($\mathbf{w}[i]; 1$)
2 : **for** $i \in [0..\ell+\lambda)$ **do** $\boldsymbol{v}[i] :=$ ToField($\mathbf{V}|_i; \lambda$)
3 : Parse pk $= (\mathbf{in}, \mathbf{out}) \in \{0,1\}^{128\beta} \times \{0,1\}^{128\beta}$ // decompose public key
4 : $\mathbf{in}_0 := \mathbf{in}[0..127]$, $\mathbf{out}_0 := \mathbf{out}[0..127]$
5 : **if** $\beta = 2$ **then** $\mathbf{in}_1 := \mathbf{in}[128..255]$, $\mathbf{out}_1 := \mathbf{out}[128..255]$
6 : **new** $\boldsymbol{a_0}, \boldsymbol{a_1} \in \mathbb{F}_{2^\lambda}^{\leq C}$
7 : $\tilde{\boldsymbol{w}} := \boldsymbol{w}[0..\ell_{\mathsf{ke}}-1]$, $\tilde{\boldsymbol{v}} := \boldsymbol{v}[0..\ell_{\mathsf{ke}}-1]$ // Selecting key expansion bits
8 : $(\tilde{\boldsymbol{a}}_0, \tilde{\boldsymbol{a}}_1, \boldsymbol{k}, \boldsymbol{v_k}) :=$ AES.KeyExpCstrnts($\tilde{\boldsymbol{w}}, \tilde{\boldsymbol{v}}, \mathtt{Mkey} = 0, \perp, \perp; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)
9 : $\boldsymbol{a}_0 := \tilde{\boldsymbol{a}}_0$, $\boldsymbol{a}_1 := \tilde{\boldsymbol{a}}_1$ // Saving values
10 : $\tilde{\boldsymbol{w}} := \boldsymbol{w}[\ell_{\mathsf{ke}}..\ell_{\mathsf{ke}}+\ell_{\mathsf{enc}}-1]$, $\tilde{\boldsymbol{v}} := \boldsymbol{v}[\ell_{\mathsf{ke}}..\ell_{\mathsf{ke}}+\ell_{\mathsf{enc}}-1]$ // Selecting encryption bits
11 : $(\tilde{\boldsymbol{a}}_0, \tilde{\boldsymbol{a}}_1) :=$ AES.EncCstrnts($\mathbf{in}_0, \mathbf{out}_0, \tilde{\boldsymbol{w}}, \tilde{\boldsymbol{v}}, \boldsymbol{k}, \boldsymbol{v_k}, \mathtt{Mkey} = 0, \perp, \perp, \perp; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)
12 : $\boldsymbol{a}_0 := \boldsymbol{a}_0 \| \tilde{\boldsymbol{a}}_0$, $\boldsymbol{a}_1 := \boldsymbol{a}_1 \| \tilde{\boldsymbol{a}}_1$ // Appending values
13 : **if** $\beta = 2$
14 : $\quad\tilde{\boldsymbol{w}} := \boldsymbol{w}[\ell_{\mathsf{ke}}+\ell_{\mathsf{enc}}..\ell-1]$, $\tilde{\boldsymbol{v}} := \boldsymbol{v}[\ell_{\mathsf{ke}}+\ell_{\mathsf{enc}}..\ell-1]$ // Selecting encryption bits
15 : $\quad(\tilde{\boldsymbol{a}}_0, \tilde{\boldsymbol{a}}_1) :=$ AES.EncCstrnts($\mathbf{in}_1, \mathbf{out}_1, \tilde{\boldsymbol{w}}, \tilde{\boldsymbol{v}}, \boldsymbol{k}, \boldsymbol{v_k}, \mathtt{Mkey} = 0, \perp, \perp, \perp; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)
16 : $\quad\boldsymbol{a}_0 := \boldsymbol{a}_0 \| \tilde{\boldsymbol{a}}_0$, $\boldsymbol{a}_1 := \boldsymbol{a}_1 \| \tilde{\boldsymbol{a}}_1$ // Appending values
17 : **for** $i \in [\ell..\ell+\lambda)$ **do**
18 : $\quad\boldsymbol{u}[i-\ell] :=$ ToField($\mathbf{u}[i], \lambda$)
19 : $u^* := \sum_{i\in[0..\lambda)} \boldsymbol{u}[i]\alpha_\lambda^i$, $v^* := \sum_{i\in[0..\lambda)} \boldsymbol{v}[\ell+i]\alpha_\lambda^i$ // Where $\alpha_\lambda \in \mathbb{F}_{2^\lambda}$
20 : $\tilde{a} :=$ ZKHash(chall, $(\boldsymbol{a}_1 \| u^*)$)
21 : $\tilde{b} :=$ ZKHash(chall, $(\boldsymbol{a}_0 \| v^*)$)
22 : **return** $(\tilde{a}, \tilde{b})$

Fig. 6.9: Proof of AES constraints

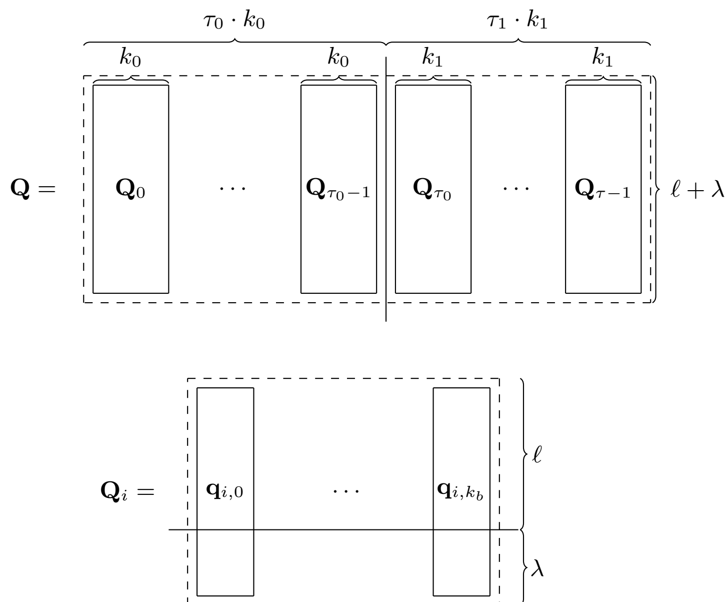

Fig. 6.10: Dividing $\mathbf{Q}$ into sub-matrices $\mathbf{Q}_i$ and then columns $\mathbf{q}_{i,j}$ in AESVerify lines 6–11.

Using these, as well as the next ℓ_{enc} elements of $\boldsymbol{w}$ and $\boldsymbol{v}$ and the (first) plaintext and ciphertext blocks **in** and **out** derived from pk, it computes the next S_{enc} constraint values for $\boldsymbol{a}_0$ and $\boldsymbol{a}_1$ using the AES.EncCstrnts algorithm. If the security level requires a second execution of the encryption routine, i.e. when $\lambda \in \{192, 256\}$, then AESProve selects the next ℓ_{enc} witness bits and their VOLE tags and computes the last S_{enc} constraint values for $\boldsymbol{a}_0$ and $\boldsymbol{a}_1$ using the AES.EncCstrnts algorithm, together with the expanded key values and VOLE tags and the second plaintext and ciphertext blocks.

With the constraint values $\boldsymbol{a}_0$ and $\boldsymbol{a}_1$ now computed, the algorithm next computes the zero-knowledge masking values u^* and v^* using the last λ elements of the VOLE masks $\mathbf{u}$ and tags $\mathbf{V}$. Finally, it calls the ZKHash algorithm with the challenge chall on $\boldsymbol{a}_1 \| u^*$ and $\boldsymbol{a}_0 \| v^*$ to compute the response $(\tilde{a}, \tilde{b})$.

6.4.2 AESVerify Description The algorithm first reconstructs the VOLE and QuickSilver challenge $\varDelta$ (observe that both use the same challenge). It then, as for AESProve, extracts out the AES input and output blocks from pk. As before, depending on the security level, these will either be one or two blocks of 128 bits.

Next, the algorithm adds $\mathbf{d}$ to the appropriate columns of $\mathbf{Q}_i$ depending on the challenge for the VOLE instance i. Since all VOLE key tags (i.e. all $\mathbf{Q}_i$) have been corrected to be tags for the same secret $(\mathbf{u})$ in the calling function, this will correct the key tags to be for the AES witness committed in $\mathbf{d}$. The division of the $\mathbf{Q}$ matrix specified in lines 6–11 is pictured in Figure 6.10. After this correction, all rows now encode the correct VOLE key tags of the verification operation, so we lift all rows $\mathbf{Q}|_i$ using ToField to the field used for verification computations of QuickSilver.

Following this, the AES operations are applied as follows: The first ℓ_{ke} VOLEs are used by KeyExpCstrnts to compute VOLE key tags $(\mathbf{b}_1)$ that allow to verify key scheduling operations for correctness, as well as VOLE key tags $\mathbf{q}_k$ corresponding to the key schedule output. EncCstrnts computes VOLE key tags corresponding to the actual evaluation of the cipher, where we either obtain one output block $\mathbf{b}_2$ if

Inputs

$\mathbf{d}$	$\{0,1\}^{\ell}$	masked extended witness, as bits.
$\mathbf{Q}$	$\{0,1\}^{(\ell+\lambda)\times\lambda}$	VOLE keys of the masking values $\mathbf{u}$, as bit-strings.
$\tilde{a}$	$\mathbb{F}_{2^\lambda}$	ZKHash of constraint values
chall_2	$\{0,1\}^{3\lambda+64}$	ZKHash challenge to compute $\tilde{q}$.
chall_3	$\{0,1\}^{\lambda}$	The VOLE challenge to correct $\mathbf{Q}$ with $\mathbf{d}$ according to Δ.
pk	$\{0,1\}^{2\cdot\beta\cdot 128}$	The public key consisting of AES inputs and outputs.

FAEST.AES.AESVerify($\mathbf{d}, \mathbf{Q}, \mathsf{chall}_2, \mathsf{chall}_3, \tilde{a}, \mathsf{pk}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$)

1 : $\Delta := \mathsf{ToField}(\mathsf{chall}_3; \lambda)$ // reconstruct VOLE and MAC challenge
2 : Parse $\mathsf{pk} = (\mathbf{in}, \mathbf{out}) \in \{0,1\}^{128\beta} \times \{0,1\}^{128\beta}$ // decompose public key
3 : $\mathbf{in}_0 := \mathbf{in}[0..127]$, $\mathbf{out}_0 := \mathbf{out}[0..127]$
4 : **if** $\beta = 2$ **then** $\mathbf{in}_1 := \mathbf{in}[128..255]$, $\mathbf{out}_1 := \mathbf{out}[128..255]$
5 : Write $\mathbf{Q} := \left[\mathbf{Q}_0 \cdots \mathbf{Q}_{\tau-1}\right]$ // $\mathbf{Q}_0, \ldots, \mathbf{Q}_{\tau_0-1}$ have k_0 columns, others k_1
6 : **for** $i \in [0..\tau)$ **do** // make commitments to witness
7 : $\quad b := 0$ **if** $i < \tau_0$ **else** 1
8 : $\quad (\delta_0, \ldots, \delta_{k_b-1}) := \mathsf{ChalDec}(\mathsf{chall}_3, i; \mathtt{param})$
9 : $\quad$ Write $\mathbf{Q}_i = [\mathbf{q}_{i,0}, \ldots, \mathbf{q}_{i,k_b-1}]$ // consider k_b columns separately
10 : $\quad$ **for** $j \in [0, k_b)$ **do**
11 : $\quad\quad \mathbf{q}_{i,j}[0..\ell-1] := \mathbf{q}_{i,j}[0..\ell-1] \oplus \delta_j \cdot \mathbf{d}$ // perform $\mathbf{q}[i] := \mathbf{q}[i] + \mathbf{d}[i] \cdot \Delta$ column-wise
12 : **for** $i \in [0..\ell+\lambda)$ **do** // create global MAC
13 : $\quad \boldsymbol{q}[i] := \mathsf{ToField}(\mathbf{Q}|_i; \lambda)$ // create VOLE keys row-wise (i.e. transpose $\mathbf{Q}$)
14 : **new** $\boldsymbol{b} \in \mathbb{F}_{2^\lambda}^{\leq C}$
15 : $(\boldsymbol{b}_1, \boldsymbol{q}_k) = \mathsf{KeyExpCstrnts}(\bot, \bot, \mathtt{Mkey} = 1, \boldsymbol{q}[0..\ell_{\mathsf{ke}}-1], \Delta; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
16 : $\boldsymbol{b}_2 = \mathsf{EncCstrnts}(\mathbf{in}_0, \mathbf{out}_0, \bot, \bot, \bot, \bot, \mathtt{Mkey} = 1, \boldsymbol{q}[\ell_{\mathsf{ke}}..\ell_{\mathsf{ke}} + \ell_{\mathsf{enc}}), \boldsymbol{q}_k, \Delta; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
17 : **if** $\beta = 1$ **then** $\boldsymbol{b} := [\boldsymbol{b}_1 \| \boldsymbol{b}_2]$ **else**
18 : $\quad \boldsymbol{b}_3 = \mathsf{EncCstrnts}(\mathbf{in}_1, \mathbf{out}_1, \bot, \bot, \bot, \bot, \mathtt{Mkey} = 1, \boldsymbol{q}[\ell_{\mathsf{ke}} + \ell_{\mathsf{enc}}..\ell), \boldsymbol{q}_k, \Delta; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
19 : $\quad \boldsymbol{b} := [\boldsymbol{b}_1 \| \boldsymbol{b}_2 \| \boldsymbol{b}_3]$
20 : $q^* := \sum_{i \in [0..\lambda)} \boldsymbol{q}[\ell + i] \alpha_\lambda^i$ // VOLE key for u^* and v^* in AESProve; $\alpha_\lambda \in \mathbb{F}_{2^\lambda}$
21 : $\tilde{q} := \mathsf{ZKHash}(\mathsf{chall}_2, (\boldsymbol{b} \| q^*))$
22 : **return** $\tilde{q} - \tilde{a} \cdot \Delta$

Fig. 6.11: Verification of AES constraints

we verify one AES instance or two outputs $\mathbf{b}_2, \mathbf{b}_3$ if there are 2 AES instances to be verified.

Before compressing the tags, we have to reconstruct the VOLE key tag from $\mathbf{q}$ for the u^* and v^* values that are used to make $\tilde{a}, \tilde{b}$ zero-knowledge. This tag is denoted as q^*. Then, all these VOLE instances, denoted as $\mathbf{b} \| q^*$, are linearly hashed using ZKHash using the ZKHash challenge chall_2. This yields a final VOLE key tag $\tilde{q}$ in $\mathbb{F}_{2^\lambda}$, and we return the value $\tilde{q} - \tilde{a}\Delta$ which equals $\tilde{b}$ if the proof was indeed correct.

7 Rijndael-EM Functions

The principal FAEST.Rijndael-EM algorithms use the following building blocks to perform AES and Rijndael operations to compute the QuickSilver proof of the OWF in the Even–Mansour variant for the FAEST-EM-λ parameter sets.

- Section 7.1: FAEST.Rijndael-EM.ExtendWitness in Figure 7.1
- Section 7.2: FAEST.Rijndael-EM.EncFwd in Figure 7.2.
- Section 7.2: FAEST.Rijndael-EM.EncBkwd in Figure 7.3.
- Section 7.2: FAEST.Rijndael-EM.EncCstrnts in Figure 7.4.
- Section 7.3: FAEST.Rijndael-EM.EMProve in Figure 7.5.
- Section 7.3: FAEST.Rijndael-EM.EMVerify in Figure 7.6.

7.1 Witness Extension

The Rijndael-EM.ExtendWitness algorithm (Fig. 7.1), generates the extended witness $\mathbf{w} \in \{0,1\}^\ell$ for use with the Rijndael-EM.EMProve algorithm; it takes as input the FAEST private key $\mathbf{k}$ (which is an AES or Rijndael input block), the FAEST public key pk (which is an AES key) and the instance parameters param and $\mathtt{param}_{\mathsf{OWF}}$.

In order to compute the encryption routine, ExtendWitness first executes the KeyExpansion routine for AES or Rijndael, as described in Section 4.1, to generate the expanded key $\mathbf{x}$. Unlike the non-EM variant, it does not save any state of this routine as the input pk is public.

ExtendWitness then runs through the encryption routine operations by initialising an empty state and performing the AES or Rijndael round operations, as described in Section 4.1 using $\mathbf{k}$ as input. It begins by recording the FAEST key $\mathbf{k}$ to the extended witness. Then, after each $\mathsf{ShiftRows}_{N_{\mathsf{st}}}$ operation (for the first $R-1$ rounds), the algorithm records the current bits of the state (*in column-major order*) denoted as shift_row_bits, into the extended witness $\mathbf{w}$. The state after the last round of $\mathsf{ShiftRows}_{N_{\mathsf{st}}}$ is not recorded, since it can be derived linearly during EMProve using the secret key $\mathbf{k}$, the FAEST public key output bits $\mathbf{y}$ and the public expanded key $\bar{\mathsf{pk}}$.

It finally outputs the extended witness in the form:

$$\mathbf{w} := \mathtt{key_bits} \| \mathtt{shift_row_bits}$$

7.2 Deriving Constraints for the Encryption Routine

The Rijndael-EM.EncCstrnts algorithm (Fig. 7.4) generates a vector of S_{enc} constraint values for the QuickSilver proof, computed by the Rijndael-EM.EMProve algorithm, of the circuit for the EM OWF based on AES. To do so, it must derive the inputs and outputs of the $\mathbb{F}_{2^8}$ inversions contained within the S-boxes from the witness values and their VOLE tags, or the VOLE keys, passed by EMProve. This is the role of the Rijndael-EM.EncFwd and Rijndael-EM.EncBkwd algorithms, described in Fig. 7.2 and Fig. 7.3.

EncFwd. Given the parameters param, $\mathtt{param}_{\mathsf{OWF}}$, a dimension parameter m, a vector of witness values, their VOLE tags or their VOLE keys $\boldsymbol{z}$, a vector of expanded key values, their VOLE tags or their VOLE keys $\boldsymbol{x}$, two flags Mtag and Mkey, and optionally the global VOLE key Δ, this algorithm derives the S_{enc} inputs of the $\mathbb{F}_{2^8}$ inversions in the S-boxes of one execution of the encryption routine.

```
FAEST.Rijndael-EM.ExtendWitness(k, pk; param, param_OWF)

      Parse pk = (in, out) ∈ {0,1}^λ × {0,1}^λ   // decompose public key
 1 :  new w ∈ {0,1}^{≤ℓ}   // New empty extended witness of length at most ℓ
        // Key expansion routine
 2 :  new x ∈ [{0,1}^32; N_st(R+1)]   // Array of words for expanded key
 3 :  x := KeyExpansion(in; param_OWF)
        // Encryption routine and saving witness bits
 4 :  w := k   // Saving the OWF key to the extended witness
 5 :  new state ∈ {0,1}^λ
 6 :  state := k
 7 :  AddRoundKey(state, x[0..N_st])
 8 :  for j ∈ [1..R)  do
 9 :    SubBytes(state)
10 :    ShiftRows_{N_st}(state)
11 :    w := w||state   // Appending the state bits, in column-major order
12 :    MixColumns(state)
13 :    AddRoundKey(state, x[N_st j..N_st(j+1) − 1])
        // Last round is not committed to, so not computed
14 :  return w ∈ {0,1}^ℓ
```

Fig. 7.1: Extending the witness from the FAEST-EM secret key $\mathbf{k}$ and the FAEST-EM public key pk.

EncBkwd. Given the parameters param, $\mathtt{param}_{\mathsf{OWF}}$, a dimension parameter m, a vector of witness values, their VOLE tags or their VOLE keys $\boldsymbol{z}$, a vector of expanded key values, their VOLE tags or their VOLE keys $\boldsymbol{x}$, values, VOLE tags or VOLE keys for an AES ciphertext block $\boldsymbol{z}_{\mathbf{out}}$, two flags Mtag and Mkey, and optionally the global VOLE key Δ, this algorithm derives the S_{enc} outputs of the $\mathbb{F}_{2^8}$ inversions in the S-boxes of one execution of the encryption routine.

EncCstrnts. Given the parameters param, $\mathtt{param}_{\mathsf{OWF}}$, the bits of the OWF output block **out**, the bits of the AES expanded key $\mathbf{x}$, the extended witness values $\boldsymbol{w}$, their VOLE tags $\boldsymbol{v}$, a flag Mkey, optionally the VOLE keys for the witness values $\boldsymbol{q}$ and optionally the global VOLE key Δ, this algorithm derives S_{enc} constraint values for the QuickSilver proof of AES-EM, either $(\boldsymbol{a}_0, \boldsymbol{a}_1)$ if $\mathtt{Mkey} = 0$, or $\boldsymbol{B}$ if $\mathtt{Mkey} = 1$.

7.2.1 Rijndael-EM.EncFwd Description. To compute the values for inputs of the S-boxes in the encryption routine, after performing the first AddRoundKey operation, this algorithm selects the $\mathsf{ShiftRows}_{N_{\mathsf{st}}}$ output bit values, their VOLE tags or their VOLE keys from $\boldsymbol{z}$, combines them into $\mathbb{F}_{2^8}$ (sub-)field elements, performs the arithmetic for MixColumns and then adds the expanded key values, their VOLE tags or their VOLE keys, previously read as bit values from $\boldsymbol{x}$ and combined into $\mathbb{F}_{2^8}$ (sub-)field elements.

Depending on the round index $j \in [1..R)$ and state column index $c \in [0..N_{\mathsf{st}}]$, the index i keeps track of the reading positions for the witness array $\boldsymbol{z}$ and the expanded key array $\boldsymbol{x}$, and index $i_{\boldsymbol{y}}$ keeps track of the writing position for the return array $\boldsymbol{y}$ which contains the input values for the S-boxes, as $\mathbb{F}_{2^8}$ (sub-) field elements.

Inputs

m	$\{1, \lambda\}$	The size of the field elements.
$\boldsymbol{z}$	$\mathbb{F}_{2^m}^{\ell}$	extended witness values, VOLE tags, or VOLE keys.
$\boldsymbol{x}$	$\mathbb{F}_{2^m}^{\lambda(R+1)}$	expanded key values, VOLE tags, or VOLE keys.
Mtag	$\{0, 1\}$	1 if $\boldsymbol{z}$ and $\boldsymbol{x}$ contain VOLE tags, 0 otherwise
Mkey	$\{0, 1\}$	1 if $\boldsymbol{z}$ and $\boldsymbol{x}$ contain VOLE keys, 0 otherwise
Δ	$\mathbb{F}_{2^\lambda} \cup \{\perp\}$	global VOLE key if Mkey $= 1$, $\perp$ otherwise

FAEST.Rijndael-EM.EncFwd($m, \boldsymbol{z}, \boldsymbol{x}, \texttt{Mtag}, \texttt{Mkey}, \Delta; \texttt{param}, \texttt{param}_{\mathsf{OWF}}$)

```
1 :  new y ∈ F_{2^λ}^{S_enc}   // The returned array of bytes
2 :  for j ∈ [0..4N_st) do   // First AddRoundKey operation
3 :    y[j] := ByteCombine(z[8j..8j + 7]; λ) + ByteCombine(x[8j..8j + 7]; λ)
4 :  for j ∈ [1..R) do   // Iterating round-wise
5 :    for c ∈ [0..N_st) do   // Computing MixColumns and AddRoundKey column-wise
6 :      i := 32N_st j + 32c   // Index to read from the z and x arrays
7 :      i_y := 4N_st j + 4c   // Index to write to the state array
8 :      new ẑ, x̂[r] ∈ F_{2^λ}^4
9 :      for r ∈ [0..3] do
             // Reading 8 elements and combining for F_{2^8} arithmetic within F_{2^λ}
10 :        ẑ[r] := ByteCombine(z[i + 8r..i + 8r + 7]; λ)
11 :        x̂[r] := ByteCombine(x[i + 8r..i + 8r + 7]; λ)
           // 1, 2, 3 are elements of the F_{2^8} sub-field of F_{2^λ}
12 :      1 := ByteCombine({01}; λ), 2 := ByteCombine({02}; λ), 3 := ByteCombine({03}; λ)
13 :      y[i_y + 0] := ẑ[0] · 2 + ẑ[1] · 3 + ẑ[2] · 1 + ẑ[3] · 1 + x̂[0]
14 :      y[i_y + 1] := ẑ[0] · 1 + ẑ[1] · 2 + ẑ[2] · 3 + ẑ[3] · 1 + x̂[1]
15 :      y[i_y + 2] := ẑ[0] · 1 + ẑ[1] · 1 + ẑ[2] · 2 + ẑ[3] · 3 + x̂[2]
16 :      y[i_y + 3] := ẑ[0] · 3 + ẑ[1] · 1 + ẑ[2] · 1 + ẑ[3] · 2 + x̂[3]
17 :  return y ∈ F_{2^λ}^{S_enc}   // F_{2^8}-inverse inputs for the R S-box layers
```

Fig. 7.2: Transforming witness or VOLE values into $\mathbb{F}_{2^8}$-inverse inputs for the encryption routine in EM mode.

Inputs

m	$\{1, \lambda\}$	The size of the field elements.
$\boldsymbol{z}$	$\mathbb{F}_{2^m}^{\ell}$	extended witness values, VOLE tags, or VOLE keys.
$\boldsymbol{x}$	$\mathbb{F}_{2^m}^{\lambda(R+1)}$	expanded key values, VOLE tags, or VOLE keys.
$\boldsymbol{z}_{\mathbf{out}}$	$\mathbb{F}_{2^m}^{\lambda}$	AES output block values, VOLE tags or VOLE keys
$\mathtt{Mtag}$	$\{0, 1\}$	1 if $\boldsymbol{z}$, $\boldsymbol{x}$ and $\boldsymbol{z}_{\mathbf{out}}$ contain VOLE tags, 0 otherwise
$\mathtt{Mkey}$	$\{0, 1\}$	1 if $\boldsymbol{z}$, $\boldsymbol{x}$ and $\boldsymbol{z}_{\mathbf{out}}$ contain VOLE keys, 0 otherwise
Δ	$\mathbb{F}_{2^\lambda} \cup \{\bot\}$	global VOLE key if $\mathtt{Mkey} = 1$, $\bot$ otherwise

FAEST.Rijndael-EM.EncBkwd$(m, \boldsymbol{z}, \boldsymbol{x}, \boldsymbol{z}_{\mathbf{out}}, \mathtt{Mtag}, \mathtt{Mkey}, \Delta; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$

1 : **new** $\boldsymbol{y} \in \mathbb{F}_{2^\lambda}^{S_{\mathsf{enc}}}$ // The returned array of bytes
2 : **for** $j \in [0..R)$ **do** // Iterating round-wise
3 : **for** $c \in [0..N_{\mathsf{st}}]$ **do** // Iterating column-wise
4 : **for** $r \in [0..3]$ **do** // Iterating row-wise
5 : $i_{\mathsf{col}} := c - r \bmod N_{\mathsf{st}}$ // $c - r \bmod N_{\mathsf{st}}$ and extra shift below inverts ShiftRows$_{N_{\mathsf{st}}}$
6 : **if** $N_{\mathsf{st}} = 8$ **and** $r \geq 2$ **then** $i_{\mathsf{col}} := i_{\mathsf{col}} - 1 \bmod N_{\mathsf{st}}$ // Extra shift when $N_{\mathsf{st}} = 8$
7 : $i_{\mathsf{rd}} := \lambda + 32N_{\mathsf{st}}j + 32i_{\mathsf{col}} + 8r$
8 : **if** $j < R - 1$ **then**
9 : $\tilde{\boldsymbol{z}}[0..7] := \boldsymbol{z}[i_{\mathsf{rd}}..i_{\mathsf{rd}} + 7]$
10 : **else**
11 : **new** $\tilde{\boldsymbol{z}}_{\mathbf{out}} \in \mathbb{F}_{2^m}^{8}$
12 : **for** $i \in [0..8)$ **do**
13 : $\tilde{\boldsymbol{z}}_{\mathbf{out}}[i] = \boldsymbol{z}_{\mathbf{out}}[i_{\mathsf{rd}} - 32N_{\mathsf{st}}(j + 1) + i]$
14 : $\tilde{\boldsymbol{z}}[0..7] := \tilde{\boldsymbol{z}}_{\mathbf{out}}[0..7] - \boldsymbol{x}[i_{\mathsf{rd}}..i_{\mathsf{rd}} + 7]$ // Inverting AddRoundKey
15 : **new** $\tilde{\boldsymbol{y}} \in \mathbb{F}_{2^m}^{8}$ // Array for the affine layer inverse
16 : **for** $i \in [0..8)$ **do**
17 : $\tilde{\boldsymbol{y}}[i] := \tilde{\boldsymbol{z}}[i - 1 \bmod 8] + \tilde{\boldsymbol{z}}[i - 3 \bmod 8] + \tilde{\boldsymbol{z}}[i - 6 \bmod 8]$
18 : $\tilde{\boldsymbol{y}}[0] := \tilde{\boldsymbol{y}}[0] + (1_{\mathbb{F}_{2^m}} - \mathtt{Mtag}) \cdot (\mathtt{Mkey} \cdot \Delta + (1_{\mathbb{F}_{2^m}} - \mathtt{Mkey}))$
19 : $\tilde{\boldsymbol{y}}[2] := \tilde{\boldsymbol{y}}[2] + (1_{\mathbb{F}_{2^m}} - \mathtt{Mtag}) \cdot (\mathtt{Mkey} \cdot \Delta + (1_{\mathbb{F}_{2^m}} - \mathtt{Mkey}))$
20 : $\boldsymbol{y}[4N_{\mathsf{st}}j + 4c + r] :=$ ByteCombine$(\tilde{\boldsymbol{y}}[0..7]; \lambda)$
21 : **return** $\boldsymbol{y} \in \mathbb{F}_{2^\lambda}^{S_{\mathsf{enc}}}$ // $\mathbb{F}_{2^8}$-inverse outputs for the R S-box layers

Fig. 7.3: Transforming witness or VOLE values into $\mathbb{F}_{2^8}$-inverse outputs for the encryption routine in EM mode.

Input

out	$\{0,1\}^\lambda$	FAEST OWF output block.
x	$\{0,1\}^{\lambda(R+1)}$	expanded key values.
$\boldsymbol{w}$	$\mathbb{F}_2^\ell \cup \{\perp\}$	extended witness values, or $\perp$ if $\texttt{Mkey} = 1$.
$\boldsymbol{v}$	$\mathbb{F}_{2^\lambda}^\ell \cup \{\perp\}$	extended witness VOLE tags, or $\perp$ if $\texttt{Mkey} = 1$.
Mkey	$\{0,1\}$	1 if computing VOLE keys (for verifier), 0 otherwise.
$\boldsymbol{q}$	$\mathbb{F}_{2^\lambda}^\ell \cup \{\perp\}$	extended Witness VOLE keys, or $\perp$ if $\texttt{Mkey} = 0$.
Δ	$\mathbb{F}_{2^\lambda} \cup \{\perp\}$	global VOLE key if $\texttt{Mkey} = 1$, $\perp$ otherwise.

FAEST.Rijndael-EM.EncCstrnts$(\mathbf{out}, \mathbf{x}, \boldsymbol{w}, \boldsymbol{v}, \texttt{Mkey}, \boldsymbol{q}, \Delta; \texttt{param}, \texttt{param}_{\mathsf{OWF}})$

```
1 :  if Mkey = 0   // If computing wire values and tags
2 :    new x ∈ F_2^{32N_st(R+1)}   // To hold the lifted bits of the public AES expanded key
3 :    for i ∈ [0..32N_st(R+1)) do
4 :      x[i] := ToField(x[i]; 1)   // Lifting the bits
5 :    new w_out ∈ F_2^λ, v_out ∈ F_{2^λ}^λ   // Holding values and VOLE tags for AES output
6 :    for i ∈ [0..λ) do w_out[i] = ToField(out[i]; 1) + w[i]   // Embedding AES output values
7 :    v_out[0..λ) := v[0..λ)   // Copying AES output tags from witness tags
8 :    s := EncFwd(1, w, x, 0, 0, ⊥; param, param_OWF)   // Inverse inputs
9 :    v_s := EncFwd(λ, v, 0, Mtag = 1, 0, ⊥; param, param_OWF)   // VOLE tags of inverse inputs
10 :   s̄ := EncBkwd(1, w, x, w_out, 0, 0, ⊥; param, param_OWF)   // Inverse outputs
11 :   v_s̄ := EncBkwd(λ, v, 0, v_out, Mtag = 1, 0, ⊥; param, param_OWF)   // Tags of inverse outputs
12 :   for j ∈ [0..S_enc) do   // Iterating S-box-wise
13 :     A_{0,j} := v_s[j] · v_s̄[j]
14 :     A_{1,j} := (s[j] + v_s[j]) · (s̄[j] + v_s̄[j]) − 1_{F_{2^8}} − A_{0,j}
15 :   return (A_{0,0}, ..., A_{0,S_enc−1}) ∈ F_{2^λ}^{S_enc}, (A_{1,0}, ..., A_{1,S_enc−1}) ∈ F_{2^λ}^{S_enc}
16 : else   // If computing tag keys
17 :   new x ∈ F_{2^λ}^{32N_st(R+1)}   // To hold the lifted bits of the public AES expanded key
18 :   for i ∈ [0..32N_st(R+1)) do
19 :     x[i] := ToField(x[i]; λ) · Δ   // Lifting the bits and multiplying with VOLE key
20 :   new q_out ∈ F_{2^λ}^λ   // Holding VOLE keys for AES output
21 :   for i ∈ [0..λ) do q_out[i] := ToField(out[i]; λ) · Δ + q[i]
22 :   q_s := EncFwd(λ, q, x, 0, Mkey = 1, Δ; param, param_OWF)   // VOLE keys of inverse inputs
23 :   q_s̄ := EncBkwd(λ, q, x, q_out, 0, Mkey = 1, Δ; param, param_OWF)   // Keys of inverse outputs
24 :   for j ∈ [0..S_enc) do   // Iterating S-box-wise
25 :     B_j := q_s[j] · q_s̄[j] − Δ · Δ   // Δ is VOLE key of expected result s · s̄ = 1
26 :   return (B_0, ..., B_{S_enc−1}) ∈ F_{2^λ}^{S_enc}
```

Fig. 7.4: Deriving the constraint values for the AESλ encryption routine in EM variant.

7.2.2 Rijndael-EM.EncBkwd Description.

To compute the values for outputs of the inversion operations in the S-boxes of the encryption routine, this algorithm selects the $\mathsf{ShiftRows}_{N_{\mathsf{st}}}$ output bit values, their VOLE tags or their VOLE keys from $\boldsymbol{z}$ and first removes the expanded key values, their VOLE tags or their VOLE keys read from $\boldsymbol{x}$. (In case it is computing values for the final layer of S-boxes, the values are read from the output block values, VOLE tags or VOLE keys $\boldsymbol{z}_{\mathbf{out}}$ instead.) Undoing the $\mathsf{ShiftRows}_{N_{\mathsf{st}}}$ operation (to work backwards towards the SubBytes operation) is done in the calculation of the i_{rd} index, using the i_{out} index, which ensures that the correct "un-shifted" value is read, correcting for the addition shift performed by the Rijndael algorithm when $N_{\mathsf{st}} = 8$.

With the output of the S-box obtained, the $\mathbb{F}_2$-affine transformation is inverted with bit-wise operations before combining the 8 elements into a single $\mathbb{F}_{2^8}$ (sub-) field element and storing it in the return array.

7.2.3 Rijndael-EM.EncCstrnts Description.

To derive a vector of S_{enc} constraints values, this algorithm first embeds the AES expanded key values $\mathbf{x}$ as field elements and computes the values and VOLE tags, or VOLE keys, for the AES encryption output based on the OWF output block bits **out** and the values, tags and keys of the OWF key k, stored in $\boldsymbol{w}[i]$, $\boldsymbol{v}[0..\lambda)$ and $\boldsymbol{q}[0..\lambda)$ respectively. Then it calls EncFwd and EncBkwd to derive the input and output values and VOLE tags, or VOLE keys, for each of the S-boxes in the encryption routine. Using these, it computes the A_0 and A_1 values, or B values, for each of the S-boxes, and returns them.

7.3 Proving and Verifying Rijndael-EM Constraints

The Rijndael-EM.EMProve (Fig. 7.5) and Rijndael-EM.EMVerify (Fig. 7.6) algorithms perform the operations of the QuickSilver zero-knowledge proof system instantiated for the Even-Mansour OWF construction based on the AES algorithm.

EMProve. Given the parameters $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$, the extended witness bits $\mathbf{w}$, the VOLE masks $\mathbf{u}$, the VOLE tags $\mathbf{V}$, the FAEST AES-EM public key pk, and the QuickSilver challenge chall, this algorithm derives the QuickSilver response $(\tilde{a}, \tilde{b})$.

EMVerify. Given the parameters $\mathtt{param}, \mathtt{param}_{\mathsf{OWF}}$, the masked witness bits $\mathbf{d}$, the VOLE keys $\mathbf{Q}$, the QuickSilver challenge chall_2, the VOLE challenge chall_3, part of the QuickSilver response $(\tilde{a})$, and the FAEST AES-EM public key pk, this algorithm recomputes the $\tilde{b}$ part of the QuickSilver response.

7.3.1 EMProve Description

In order to compute the QuickSilver response $(\tilde{a}, \tilde{b})$, this algorithm first embeds the witness and VOLE tag bit-strings $\mathbf{w}$ and $\mathbf{V}$ to $\mathbb{F}_2$ and $\mathbb{F}_{2^\lambda}$ respectively. It then computes the key expansion based on the publicly known AES key, derived from the FAEST public key input block **in**.

It then derives the constraint values $\boldsymbol{a}_0$ and $\boldsymbol{a}_1$ by calling Rijndael-EM.EncCstrnts and computes the zero-knowledge masking values u^* and v^* using the last λ elements of the VOLE masks $\mathbf{u}$ and tags $\mathbf{V}$. Finally, it calls the ZKHash algorithm with the challenge chall on $\boldsymbol{a}_1 \| u^*$ and $\boldsymbol{a}_0 \| v^*$ to compute the response $(\tilde{a}, \tilde{b})$.

Inputs

$\mathbf{w}$	$\{0,1\}^{\ell}$	extended witness values, as bits.
$\mathbf{u}$	$\{0,1\}^{\lambda}$	masking values for the QuickSilver response, as bits.
$\mathbf{V}$	$\{0,1\}^{(\ell+\lambda)\times\lambda}$	VOLE tags of the masking values, as bit-strings.
pk	$\{0,1\}^{2\cdot\lambda}$	FAEST public key
chall	$\{0,1\}^{3\lambda+64}$	ZKHash challenge to compute the QuickSilver response $(\tilde{a},\tilde{b})$.

FAEST.Rijndael-EM.EMProve$(\mathbf{w},\mathbf{u},\mathbf{V},\mathsf{pk},\mathsf{chall};\mathtt{param},\mathtt{param}_{\mathsf{OWF}})$

1 : Parse $\mathsf{pk} = (\mathbf{in},\mathbf{out}) \in \{0,1\}^{\lambda} \times \{0,1\}^{\lambda}$ // decompose public key
2 : **for** $i \in [0..\ell)$ **do** $\boldsymbol{w}[i] :=$ ToField$(\mathbf{w}[i];1)$
3 : **for** $i \in [0..\ell+\lambda)$ **do** $\boldsymbol{v}[i] :=$ ToField$(\mathbf{V}|_i;\lambda)$
4 : $\mathbf{x} :=$ KeyExpansion$(\mathbf{in};\mathtt{param}_{\mathsf{OWF}})$ // Compute key expansion, $32N_{\mathsf{st}}(R+1)$ bits
5 : **new** $\boldsymbol{a_0},\boldsymbol{a_1} \in \mathbb{F}_{2^\lambda}^{\leq C}$
6 : $(\boldsymbol{a}_0,\boldsymbol{a}_1) :=$ Rijndael-EM.EncCstrnts$(\mathbf{out},\mathbf{x},\boldsymbol{w},\boldsymbol{v},\mathtt{Mkey}=0,\bot,\bot;\mathtt{param},\mathtt{param}_{\mathsf{OWF}})$
7 : **for** $i \in [\ell..\ell+\lambda)$ **do**
8 : $\quad\boldsymbol{u}[i-\ell] :=$ ToField$(\mathbf{u}[i],\lambda)$
9 : $u^* := \sum_{i\in[0..\lambda)} \boldsymbol{u}[i]\alpha_\lambda^i,\ v^* := \sum_{i\in[0..\lambda)} \boldsymbol{v}[\ell+i]\alpha_\lambda^i$ // Where $\alpha_\lambda \in \mathbb{F}_{2^\lambda}$
10 : $\tilde{a} :=$ ZKHash$(\mathsf{chall},(\boldsymbol{a}_1\|u^*))$
11 : $\tilde{b} :=$ ZKHash$(\mathsf{chall},(\boldsymbol{a}_0\|v^*))$
12 : **return** $(\tilde{a},\tilde{b})$

Fig. 7.5: Proof of AES constraints in the EM variant

7.3.2 EMVerify Description Just as EMProve, the algorithm is very similar to AESVerify. The algorithm first reconstructs the VOLE and QuickSilver challenge Δ. Next, the algorithm adds $\mathbf{d}$ to the appropriate columns of $\mathbf{Q}_i$ depending on the challenge for the VOLE instance i. Since all VOLE key tags (i.e. all $\mathbf{Q}_i$) have been corrected to be tags for the same secret ($\mathbf{u}$) in the calling function, this will correct the key tags to be for the AES witness committed in $\mathbf{d}$. After this correction, all rows now encode the correct VOLE key tags of the verification operation, so we lift all rows $\mathbf{Q}|_i$ using ToField to the field used for verification computations of QuickSilver.

After this, the AES or Rijndael operations are applied as follows: We first use KeyExpansion to compute the bits of the public expanded key. We then run EncCstrnts to recompute $\mathbf{b}$, the MAC keys of the constraints.

Before compressing the tags, we have to reconstruct the VOLE key tag from $\mathbf{q}$ for the u^* and v^* values that are used to make $\tilde{a},\tilde{b}$ zero-knowledge. This tag is denoted as q^*. Then, all these VOLE instances, denoted as $\mathbf{b}\|q^*$, are linearly hashed using ZKHash using the ZKHash challenge chall_2. This yields a final VOLE key tag $\tilde{q}$ in $\mathbb{F}_{2^\lambda}$, and we return the value $\tilde{q}-\tilde{a}\Delta$ which equals $\tilde{b}$ if the proof was indeed correct.

Inputs

$\mathbf{d}$	$\{0,1\}^{\ell}$	masked extended witness, as bits.
$\mathbf{Q}$	$\{0,1\}^{(\ell+\lambda)\times\lambda}$	VOLE keys of the masking values $\mathbf{u}$, as bit-strings.
$\tilde{a}$	$\mathbb{F}_{2^\lambda}$	ZKHash of constraints values
chall_2	$\{0,1\}^{3\lambda+64}$	ZKHash challenge to compute $\tilde{q}$.
chall_3	$\{0,1\}^{\lambda}$	The VOLE challenge to correct $\mathbf{Q}$ with $\mathbf{d}$ according to Δ.
pk	$\{0,1\}^{2\cdot\lambda}$	The public key consisting of AES EM inputs and outputs.

$\mathsf{FAEST.Rijndael\text{-}EM.EMVerify}(\mathbf{d}, \mathbf{Q}, \mathsf{chall}_2, \mathsf{chall}_3, \tilde{a}, \mathsf{pk}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$

1 : Parse $\mathsf{pk} = (\mathbf{in}, \mathbf{out}) \in \{0,1\}^{\lambda} \times \{0,1\}^{\lambda}$ // decompose public key
2 : $\Delta := \mathsf{ToField}(\mathsf{chall}_3; \lambda)$ // reconstruct VOLE and MAC challenge
3 : Write $\mathbf{Q} := \left[\mathbf{Q}_0 \cdots \mathbf{Q}_{\tau-1}\right]$ // $\mathbf{Q}_0, \ldots, \mathbf{Q}_{\tau_0-1}$ have k_0 columns, others k_1
4 : **for** $i \in [0..\tau)$ **do** // make commitments to witness
5 : $\quad b := 0$ **if** $i < \tau_0$ **else** 1
6 : $\quad (\delta_0, \ldots, \delta_{k_b-1}) := \mathsf{ChalDec}(\mathsf{chall}_3, i; \mathtt{param})$
7 : $\quad$ Write $\mathbf{Q}_i = [\mathbf{q}_{i,0}, \ldots, \mathbf{q}_{i,k_b-1}]$ // consider k_b columns separately
8 : $\quad$ **for** $j \in [0, k_b)$ **do**
9 : $\quad\quad \mathbf{q}_{i,j}[0..\ell-1] := \mathbf{q}_{i,j}[0..\ell-1] \oplus \delta_j \cdot \mathbf{d}$ // perform column-wise
10 : **for** $i \in [0..\ell+\lambda)$ **do** // create global MACs
11 : $\quad \boldsymbol{q}[i] := \mathsf{ToField}(\mathbf{Q}|_i; \lambda)$ // create VOLE keys row-wise (i.e. transpose $\mathbf{Q}$)
12 : **new** $\boldsymbol{b} \in \mathbb{F}_{2^\lambda}^{\leq C}$
13 : $\mathbf{x} := \mathsf{KeyExpansion}(\mathbf{in}; \mathtt{param}_{\mathsf{OWF}})$ // Compute key expansion, $32N_{\mathsf{st}}(R+1)$ bits
14 : $\boldsymbol{b} = \mathsf{EncCstrnts}(\mathbf{out}, \mathbf{x}, \perp, \perp, \mathtt{Mkey} = 1, \boldsymbol{q}[0..\ell), \Delta; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
15 : $q^* := \sum_{i\in[0..\lambda)} \boldsymbol{q}[\ell+i]\alpha_\lambda^i$ // VOLE key for u^* and v^* in AESProve; $\alpha_\lambda \in \mathbb{F}_{2^\lambda}$
16 : $\tilde{q} := \mathsf{ZKHash}(\mathsf{chall}_2, (\boldsymbol{b} \| q^*))$
17 : **return** $\tilde{q} - \tilde{a} \cdot \Delta$

Fig. 7.6: Verification of AES constraints for the EM variant

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 2.1 via $\texttt{param}, \texttt{param}_{\mathsf{OWF}}$, which fixes the security parameter λ, the output length $128 \cdot \beta$ to 128 (i.e., $\beta = 1$) if $\lambda = 128$ and to 256 if $\lambda \in \{192, 256\}$, and the OWF.

The algorithm, given in Figure 8.1, is almost the same for all parameter sets. First, it samples $\mathbf{x} \leftarrow \{0,1\}^{\beta \cdot 128}$ and $\mathbf{k} \leftarrow \{0,1\}^{\lambda}$ until one pair (x, k) is found such that no occurrence of the AES S-box—in either SubBytes or SubWord in case of FAEST, or just in SubBytes in case FAEST-EM—receives 0 as input, by checking that $\mathsf{ZeroInpSB} = 0$. If $\mathsf{ZeroInpSB} = 1$, then it sets $\mathsf{sk} = \mathbf{k}$ and $\mathsf{pk} = (\mathbf{x}, \mathbf{y})$, and returns $(\mathsf{sk}, \mathsf{pk})$. Otherwise, it repeats the process until a key k with this property is found. A discussion on the implications of excluding secret keys with a 0 state byte for any SubBytes or SubWord is given in Section 10.3.

8.2 Signing

The FAEST signature algorithm Sign is given in Figure 8.2. It uses some of the procedures described in the previous section. The challenges $\mathsf{chall}_1, \mathsf{chall}_2, \mathsf{chall}_3$ for the signing are obtained using random oracle H_2 by generating outputs of different length. The signing algorithm runs in 4 phases, defined by the generation of the challenges.

In phase 1, the signer compresses the public key pk and the message msg into a string μ. This string, together with the secret key sk and unique randomness ρ is then used to derive the randomness r which seeds the VOLE generation. After this, the signer commits to the VOLEs. For this, it runs VOLECommit on input r and gets as a response the commitment h_{com} of all VOLE instances (or rather the vector commitments generating the VOLE instances), the τ *full* decommitments decom_i, the $\tau - 1$ correction values $\mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}$ as well as the VOLE secrets $\mathbf{u}$ and VOLE MAC tags $\mathbf{V}$. Here, $\mathbf{c}_i$ corrects the ith VOLE instance to have the same secret $\mathbf{u}$ as the 0th VOLE instance. Finally in this phase, the signer generates the first challenge chall_1 by hashing together the (hashed) message μ and the public outputs $h_{\mathsf{com}}, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\tau-1}$ of VOLECommit, returning a bit string chall_1 of length $5\lambda + 64$.

In phase 2, the signer prepares $\widetilde{\boldsymbol{u}}, \widetilde{\mathbf{V}}$ for the VOLE consistency check using the procedure VOLEHash on input $(\mathsf{chall}_1, \boldsymbol{u})$ and $(\mathsf{chall}_1, \mathbf{V})$, respectively, and hashes $\widetilde{\mathbf{V}}$ (for brevity). It then generates the extended AES witness $\mathbf{w}$ and creates $\mathbf{d}$, which is the witness $\mathbf{w}$ one-time padded using $\mathbf{u}$, thereby committing the signer in the proof to the witness. Finally in this phase, the signer will hash $\widetilde{\mathbf{u}}, h_V$ as well as $\mathbf{d}$ together with chall_1 to obtain the second challenge $\mathsf{chall}_2 \in \{0,1\}^{3\lambda+64}$.

In phase 3, the signer computes the QuickSilver proof $(\tilde{a}, \tilde{b})$ using AESProve. The proof algorithm uses $\mathbf{u}, \mathbf{V}$ as input VOLEs to generate the proof, as well as the witness $\mathbf{w}$ and the inputs and outputs of the AES instances. Here, the QuickSilver

```
FAEST.KeyGen(param, param_OWF)

1 :  ZeroInpSB := 0
2 :  while ZeroInpSB = 0  do
3 :    x ← {0,1}^{β·128}
4 :    k ← {0,1}^λ
5 :    y := F_k(x)
6 :    // In case the OWF is AES
7 :    if no SubBytes or SubWord has an input with a {00} byte  then ZeroInpSB ← 1
8 :    // In case the OWF is EM-AES
9 :    if no SubBytes has an input with a {00} byte  then ZeroInpSB ← 1
10 : sk := k
11 : pk := (x, y)
12 : return (sk, pk)
```

Fig. 8.1: FAEST key generation algorithm

```
FAEST.Sign(msg, sk, pk; param, param_OWF)

1 :  ρ ← {0,1}^λ    // Signature randomness — for deterministic signing, set ρ := 0^λ
2 :  μ := H_1(pk||msg)
3 :  (r, iv) := H_3(sk||μ||ρ)    // r ∈ {0,1}^λ, iv ∈ {0,1}^128
4 :  // Commitment and VOLE Derandomization
5 :  (h_com, decom_0, ..., decom_{τ−1}, c_1, ..., c_{τ−1}, u, V) := FAEST.VOLECommit(r, iv, ℓ + 2λ + B; param)
6 :  chall_1 := H_2^1(μ||h_com||c_1|| ... ||c_{τ−1}||iv; 5λ + 64)
7 :  // VOLE consistency check and witness commitment
8 :  ũ := VOLEHash(chall_1, u) ∈ {0,1}^{λ+B}
9 :  Ṽ := VOLEHash(chall_1, V) ∈ {0,1}^{(λ+B)×λ}    // hash column-wise
10 : h_V := H_1(Ṽ)    // hash in column-major order
11 : w := AES.ExtendWitness(sk, in; param, param_OWF) ∈ {0,1}^ℓ    // Extend witness with S-Box outputs
12 : d := w ⊕ u_{[0..ℓ)}    // Mask extended witness
13 : chall_2 := H_2^2(chall_1||ũ||h_V||d; 3λ + 64)
14 : // AES proof
15 : u := u_{[0..ℓ+λ)}
16 : V := V_{[0..ℓ+λ)}    // drop last λ + B rows (impl. may transpose to row-major order here)
17 : (ã, b̃) := FAEST.AES.AESProve(w, u, V, pk, chall_2; param, param_OWF)
18 : chall_3 := H_2^3(chall_2||ã||b̃; λ)
19 : // Build VOLE challenge and open Vector Commitments
20 : for i ∈ [0..τ)  do
21 :   s_i := ChalDec(chall_3, i; param)
22 :   (pdecom_i) := VC.Open(decom_i, s_i)
23 : return σ := ((c_i)_{i∈[1..τ)}, ũ, d, ã, (pdecom_i)_{i∈[0..τ)}, chall_3, iv)
```

Fig. 8.2: FAEST signing algorithm. For EM variant, replace AES.ExtendWitness and AES.AESProve by Rijndael-EM.ExtendWitness and Rijndael-EM.EMProve

proof consists of a VOLE tag $\tilde{a}$ and VOLE secret $\tilde{b}$, compressed using the challenge chall_2. chall_2 as well as the QuickSilver proof are then used to generate the last challenge $\mathsf{chall}_3 \in \{0,1\}^\lambda$. This challenge determines which index of each of the τ vector commitments will *not* be opened.

In phase 4, the signer for each vector commitment opens all except one index (namely $\mathbf{s}_i$) as specified by the challenge chall_3. To do so, it generates the partial decommitment pdecom_i for each of the τ vector commitments. Finally, the signer generates the signature σ which consists of:

- The $\tau - 1$ correction strings $\mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}$.
- The hashed VOLE secret $\widetilde{\mathbf{u}}$. This is used in the VOLE consistency check later.
- The commitment $\mathbf{d}$ to the witness.
- The QuickSilver proof part $\tilde{a}$.
- The partial decommitments pdecom_i for each of the τ vector commitment instances, opening all positions except $\mathbf{s}_i$.
- The last challenge chall_3.
- The initialization vector iv.

8.3 Verification

The FAEST verification algorithm Verify is given in Figure 8.3. Like the signing algorithm it can be separated into 4 phases. Initially, the verifier parses the signature σ into its individual components (mentioned in the previous subsection). If any of these parts are missing or σ is not well-formed, then verification aborts. Next, the verifier recomputes μ and extracts the AES inputs and outputs $\mathbf{in}, \mathbf{out}$ from pk.

In phase 2, the verifier computes the VOLE key tags based on chall_3 and the partial openings pdecom_i for all τ VOLE instances using the algorithm VOLEReconstruct. This generates all the key tags $\mathbf{Q}'_i$ for the VOLE instances. VOLEReconstruct also recomputes a hash h_{com} from the partial decommitments, which is the hash of all vector commitments. This h_{com} allows the verifier to recompute the first challenge chall_1.

In phase 3, the verifier adjusts all VOLE key tags to be for the same secret ($\mathbf{u}$ in the Sign algorithm) instead of different secrets. This is done by adding $\mathbf{c}_i$ to VOLE instance i where we add $\mathbf{c}_i$ iff the secret is present in the respective column of $\mathbf{Q}'_i$ based on Proposition 5 (observe that VOLEReconstruct internally calls ConvertToVOLE to generate the $\mathbf{Q}'_i$ key tags). This leaves $\mathbf{Q}'_0$ untouched, as it already has the correct secret. The verifier then wants to use the VOLEHash of $\mathbf{Q}_0, \ldots, \mathbf{Q}_{\tau-1}$ using challenge chall_1 to recompute the hash h_V of the VOLE MAC tags. To achieve this, the verifier corrects the VOLEHash $\widetilde{\mathbf{Q}}$ by the VOLEHash $\widetilde{\mathbf{u}}$ (as VOLEHash is linear), where it adds $\widetilde{\mathbf{u}}$ to a hashed column of $\mathbf{Q}'_i$ depending on the individual VOLE challenges (i.e. depending if the secret is present in the specific column of the VOLE instance i or not). h_V is in turn used by the verifier to reconstruct the second challenge chall_2.

Finally, in phase 4 the verifier has reconstructed all VOLE key tags in $\mathbf{Q}_0, \ldots, \mathbf{Q}_{\tau-1}$ for the same VOLE secrets, based on correct openings of the vector commitments. It then calls AESVerify to obtain the QuickSilver VOLE MAC $\tilde{b}$ based on the VOLE secret $\tilde{a}$ as outlined in Section 6.4. It uses this to compute chall'_3. If this disagrees with chall_3 of σ then it rejects the signature, otherwise it accepts it.

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

1 : Parse $\sigma = ((\mathbf{c}_i)_{i\in[1..\tau)}, \widetilde{\mathbf{u}}, \mathbf{d}, \tilde{a}, (\mathsf{pdecom}_i)_{i\in[0..\tau)}, \mathsf{chall}_3, \mathsf{iv})$
2 : $\mu := \mathsf{H}_1(\mathsf{pk}\|\mathsf{msg})$
3 : // Reconstruct VOLEs and check commitment
4 : $(h_{\mathsf{com}}, \mathbf{Q}'_0, \ldots, \mathbf{Q}'_{\tau-1}) :=$ FAEST.VOLEReconstruct$(\mathsf{chall}_3, (\mathsf{pdecom}_i)_{i\in[0..\tau)}, \mathsf{iv}, \ell + 2\lambda + B; \mathtt{param})$
5 : $\mathsf{chall}_1 := \mathsf{H}_2^1(\mu\|h_{\mathsf{com}}\|\mathbf{c}_1\|\ldots\|\mathbf{c}_{\tau-1}\mathsf{iv}; 5\lambda + 64)$
6 : // Apply the VOLE correction values and check consistency
7 : $\mathbf{Q}_0 := \mathbf{Q}'_0$
8 : **for** $i \in [0..\tau)$ **do**
9 : $\quad b := 0$ **if** $i < \tau_0$ **else** 1
10 : $\quad (\delta_0, \ldots, \delta_{k_b-1}) :=$ ChalDec$(\mathsf{chall}_3, i; \mathtt{param})$
11 : $\quad \widetilde{\mathbf{D}}_i := [\delta_0 \cdot \widetilde{\mathbf{u}} \cdots \delta_{k_b-1} \cdot \widetilde{\mathbf{u}}] \in \{0,1\}^{(\lambda+B)\times k_b}$
12 : $\quad$ **if** $i > 0$
13 : $\quad\quad \mathbf{Q}_i := \mathbf{Q}'_i \oplus \left[\delta_0 \cdot \mathbf{c}_i \cdots \delta_{k_b-1} \cdot \mathbf{c}_i\right]$
14 : Denote $\mathbf{Q} := \left[\mathbf{Q}_0 \cdots \mathbf{Q}_{\tau-1}\right]$
15 : $\widetilde{\mathbf{Q}} :=$ VOLEHash$(\mathsf{chall}_1, \mathbf{Q}) \in \{0,1\}^{(\lambda+B)\times\lambda}$
16 : $h_V := \mathsf{H}_1(\widetilde{\mathbf{Q}} \oplus [\widetilde{\mathbf{D}}_0 \cdots \widetilde{\mathbf{D}}_{\tau-1}])$ // Hash in column-major order
17 : $\mathsf{chall}_2 := \mathsf{H}_2^2(\mathsf{chall}_1\|\widetilde{\mathbf{u}}\|h_V\|\mathbf{d}; 3\lambda + 64)$
18 : // Compute AES consistency values
19 : $\tilde{b} :=$ FAEST.AES.AESVerify$(\mathbf{d}, \mathbf{Q}|_{[0..\ell+\lambda)}, \mathsf{chall}_2, \mathsf{chall}_3, \widetilde{a}, \mathsf{pk}; \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$
20 : Compute $\mathsf{chall}'_3 := \mathsf{H}_2^3(\mathsf{chall}_2\|\tilde{a}\|\tilde{b}; \lambda)$
21 : **return true if** $\mathsf{chall}_3 = \mathsf{chall}'_3$ **else false**

Fig. 8.3: FAEST verification algorithm. For EM variant, replace AES.AESVerify by Rijndael-EM.EMVerify

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 implementation with optimizations aimed at x86-64 architectures with the AVX2 and AES-NI instruction set extensions, as commonly seen in Intel and AMD CPUs. Note that the "optimized implementation" required by NIST is identical to the reference implementation. Both implementations were built using the eXtended Keccak Code Package (XKCP) and OpenSSL libraries.

We used a few techniques worth mentioning in the x86-64 implementation. For converting **V** and **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].[11]

Benchmarking Setup. We measured the performance of both implementations using a single core of a consumer notebook with an AMD Ryzen 7 5800H processor, with a base clock speed of 3.2 GHz and 16 GiB memory. Simultaneous Multi-Threading was enabled. The computer was running Linux 6.1.30, and the implementations were built with GCC 12.2.1.

Performance. Table 9.1 shows the performance of the AVX2 implementation, while Table 9.2 shows the reference implementation. Runtimes of the respective algorithms are given in milliseconds.

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

Scheme	Runtimes in ms			Sizes in B		
	KeyGen	Sign	Verify	sk	pk	Signature
FAEST-128s	0.001	8.097	8.097	32	32	5 006
FAEST-128f	0.001	0.869	0.869	32	32	6 336
FAEST-192s	0.002	19.311	19.311	56	64	12 744
FAEST-192f	0.002	1.962	1.962	56	64	16 792
FAEST-256s	0.003	26.763	26.763	64	64	22 100
FAEST-256f	0.003	3.104	3.104	64	64	28 400
FAEST-EM-128s	0.001	8.046	8.046	32	32	4 566
FAEST-EM-128f	0.001	0.857	0.857	32	32	5 696
FAEST-EM-192s	0.001	18.520	18.520	48	48	10 824
FAEST-EM-192f	0.001	1.869	1.869	48	48	13 912
FAEST-EM-256s	0.003	25.872	25.872	64	64	20 956
FAEST-EM-256f	0.003	3.029	3.029	64	64	26 736

Table 9.1: Benchmark results for the architecture specific implementation for x86-64 with AVX2.'

Scheme	Runtimes in ms			Sizes in B		
	KeyGen	Sign	Verify	sk	pk	Signature
FAEST-128s	0.029	54.685	54.685	32	32	5 006
FAEST-128f	0.029	8.688	8.688	32	32	6 336
FAEST-192s	0.133	130.200	130.200	56	64	12 744
FAEST-192f	0.132	22.125	22.125	56	64	16 792
FAEST-256s	0.218	172.761	172.761	64	64	22 100
FAEST-256f	0.219	38.640	38.640	64	64	28 400
FAEST-EM-128s	0.025	53.514	53.514	32	32	4 566
FAEST-EM-128f	0.025	8.195	8.195	32	32	5 696
FAEST-EM-192s	0.068	115.081	115.081	48	48	10 824
FAEST-EM-192f	0.070	18.476	18.476	48	48	13 912
FAEST-EM-256s	0.202	166.831	166.831	64	64	20 956
FAEST-EM-256f	0.210	37.475	37.475	64	64	26 736

Table 9.2: Benchmark results for the reference implementation.'

10 Security Evaluation

In this section, we evaluate the security of FAEST. In Section 10.1, we present a proof of security in the random oracle model, and provide evidence for security in the quantum-accessible random oracle model. In Section 10.2, we analyze the security of FAEST with respect to known attacks. In Section 10.3, we perform a thorough concrete analysis of the one-way functions based on AES whose security FAEST relies on.

10.1 Provable Security

We prove security of FAEST when the hash functions H_0, H_1, H_2^1, H_2^2, H_2^3 are modeled as random oracles. The proof is an adaptation of the proof of [BBD+23]: It follows the same reasoning, but uses concrete primitives, and not the abstract (compilation) steps of [BBD+23], so we get tighter bounds than in [BBD+23].

The section proceeds as follows. In Section 10.1.1, we start by proving that VOLECommit is extractable binding (for VOLE correlations) and that it is hiding, in Lemmas 3 and 5, respectively. In Section 10.1.2, we prove classical security of FAEST. Firstly, we prove existential unforgeability under key-only attacks (EUF-KO) of FAEST in Theorem 1. This proof relies (in a non-black-box manner) on the extractable binding property of VOLECommit, and mostly follows from the proof of [BBD+23]. Essentially, this proof exploits the straightline extractability of random oracles hashes to map adversarial executions to interactive executions, and implicitly relies on the round-by-round soundness of the underlying proof of knowledge, cf. [BBD+23]. Secondly, we extend EUF-KO security to full existential unforgeability under chosen message attacks (EUF-CMA) in Theorem 2. This proof handles signing queries by programming the random oracle.

In Section 10.1.3, we discuss QROM security. We formulate a conjecture about extractability of VOLECommit in the QROM, and then prove non-tight QROM security based on that conjecture. To that end we use a more modular (and less tight) approach to prove EUF-KO in the QROM, Theorem 5, and generalize Theorem 2 to the QROM in Theorem 3.

10.1.1 Security of Vector Commitments and VOLE Commitment We first argue that commitments are binding, by showing that there is an extractor algorithm that is allowed to observe random oracle queries, and will successfully extract the adversary's committed message with high probability. The analysis is similar to [BBD+23, Sec. 3.1], with the main differences being that instead of analyzing the vector commitments directly, we analyze the VOLECommit and VOLEReconstruct algorithms which use them. Thus, we only show the binding property on the signer's VOLE matrices, rather than the original vectors committed using VC (even though these are also still binding). By arguing security only for our specific construction, we also avoid the security loss incurred by generic reductions in [BBD+23].

Lemma 3 (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. $h_{\mathsf{com}} \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1}(\texttt{commit})$
2. $(\boldsymbol{u}_i^*, \mathbf{V}_i^*)_{i\in[0,\tau)} = \mathsf{Ext}(\mathcal{L}_0, \mathcal{L}_1, h_{\mathsf{com}})$*, 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*.*

$\mathsf{VOLEVerify}(h_{\mathsf{com}}, \mathsf{chall}, (\mathsf{pdecom}_i)_{i\in[0..\tau)}, \mathsf{iv}, \hat{\ell}; \mathtt{param})$
$1: \quad (h_{\overline{\mathsf{com}}}, \mathbf{Q}_0, \dots, \mathbf{Q}_{\tau-1}) := \mathsf{VOLEReconstruct}^{\mathsf{H}_1}(\mathsf{chall}, (\mathsf{pdecom}_i)_{i\in[0..\tau)}, \mathsf{iv}, \hat{\ell}; \mathtt{param})$
$2: \quad \mathbf{return}\ h_{\mathsf{com}} \neq h_{\overline{\mathsf{com}}}\ \mathbf{or}\ \ h_{\overline{\mathsf{com}}} = \perp$

Fig. 10.1: VOLECommit decommitment verification.

3. $(\mathsf{chall}, (\mathsf{pdecom}_i)_{i\in[0..\tau)}) \leftarrow \mathcal{A}^{\mathsf{H}_0,\mathsf{H}_1}(\mathtt{open})$

4. For $i \in [0..\tau)$*:*

  (a) $(\delta_{i,0}, \dots, \delta_{i,k_b-1}) := \mathsf{ChalDec}(\mathsf{chall}, i; \mathtt{param})$

  (b) $\Delta_i := \mathsf{NumRec}(k_b, (\delta_{i,0}, \dots, \delta_{i,k_b-1}))$

5. $(h_{\overline{\mathsf{com}}}, \mathbf{Q}_0, \dots, \mathbf{Q}_{\tau-1}) := \mathsf{VOLEReconstruct}^{\mathsf{H}_1}(\mathsf{chall}, (\mathsf{pdecom}_i)_{i\in[0..\tau)}, \mathsf{iv}, \hat{\ell}; \mathtt{param})$

6. Output 0 *(failure) if*

  (a) $h_{\overline{\mathsf{com}}} = \perp$ *or* $h_{\mathsf{com}} \neq h_{\overline{\mathsf{com}}}$ *(i.e., verifier rejects), or*

  (b) $\mathbf{Q}_i = \mathbf{V}_i^* \oplus \left[\delta_{i,0} \cdot \boldsymbol{u}_i^* \cdots \delta_{i,k_b} \cdot \boldsymbol{u}_i^*\right]$*, for all* $i \in [0, \dots, \tau)$ *(i.e., extraction correct).*

7. Output 1 *(success) otherwise.*

Suppose $\mathcal{A}$ *makes at most* Q_0, Q_1 *queries to the oracles* $\mathsf{H}_0, \mathsf{H}_1$ *respectively. Moreover, suppose that* $\mathcal{A}$ *checks if* $\mathsf{VOLEVerify}(h_{\mathsf{com}}, \mathsf{chall}, (\mathsf{pdecom}_i)_{i\in[0..\tau)}, \mathsf{iv}, \hat{\ell}; \mathtt{param}) = 1$ *for the purported decommitment, and otherwise outputs* $\perp$*. Let* Ext *be the straight-line extractor defined in the security proof. Then, the advantage of* $\mathcal{A}$ *in the above game is at most*

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

Intuitively, the lemma asserts that extraction succeeds unless $\mathcal{A}$ cheats by breaking collision or preimage resistance of the random oracle, which gives the advantage bound in the lemma. Note that $\mathcal{A}$ chooses the challenges Δ_i, hence it can cheat in the sense that only an opening for Δ_i is "known" to $\mathcal{A}$, i.e. the (extracted) commitment cannot be opened for any other challenge than Δ_i. However, as long as the extractor extracts that VOLE correlation, this does not constitute a success for the adversary — the adversary only wins if the extracted VOLE correlation disagrees with the opened VOLE correlation.

Remark 1. 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 $Q_0+\tau N$ (resp. $Q_1+\tau+1$). Thus, the advantage bound becomes

$$\frac{(Q_0+\tau N)^2 + (Q_1+\tau+1)^2}{2^{2\lambda}}$$

which for concrete security makes almost no difference, since $Q_0, Q_1 \gg \tau N$.

We use following lemma to simplify the proof of Lemma 3.

Lemma 4 (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 4. 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 $\exists x' \colon (x', y) \in E \land x \neq x'$). (The former is a preimage attack, the latter is a collision attack.) Clearly, the chances that the former happens are $\frac{|V'|}{|\mathcal{Y}|}$ and the chances that the latter happens are $\frac{|V''|}{|\mathcal{Y}|}$, and since V' and V'' are disjoint, the chances that either happens is $\frac{|V|}{|\mathcal{Y}|}$.

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

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

□

Proof of Lemma 3. We define the straightline extractor Ext, by looking through the lists of queries $\mathcal{L}_0, \mathcal{L}_1$ as follows.

1. Find a preimage $(\mathsf{com}_0 \| \dots \| \mathsf{com}_{\tau-1})$ of h_{com} under H_1. If there is no unique preimage, i.e., there are none or multiple preimages, immediately output $\bot$.
2. For each $i \in [0..\tau)$:
   (a) Find a preimage $(\overline{\mathsf{com}}_{i,0}, \dots, \overline{\mathsf{com}}_{i,N_i-1})$ of com_i under H_1, where $N_i = N_0$ if $i < \tau \bmod \lambda$ and N_1 otherwise. If there is no unique preimage, i.e., there are none or multiple preimages, immediately output $\bot$.
   (b) For each $j \in [0..N_i)$, find $\mathsf{sd}_{i,j}$ such that $(\mathsf{sd}_{i,j}, \overline{\mathsf{com}}_{i,j})$ has a preimage of some query to H_0. If there is no such preimage, set $\mathsf{sd}_{i,j} = \bot$. If there are multiple preimages, immediately output $\bot$.

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

– Case 1 (All preimages found): If Ext found all preimages $(\mathsf{sd}_{i,j})_{j \in [0,N)}$ for i, then it computes $\mathbf{V}_i^*$ and $\boldsymbol{u}_i'$ honestly (i.e., via $\mathsf{ConvertToVOLE}$) and sets $\Delta_i^* = \bot$.
– Case 2 (Missing one preimage): If a single preimage was not found, say $\mathsf{sd}_{i,j^*} = \bot$, then Ext sets $\Delta_i^* = j^*$ and, as in $\mathsf{VOLEReconstruct}$, computes $(\boldsymbol{u}_i^*, \mathbf{q}_{i,0}, \dots, \mathbf{q}_{i,k_b-1}) = \mathsf{ConvertToVOLE}(\bot, \mathsf{sd}_{i,1 \oplus \Delta_i^*}, \dots, \mathsf{sd}_{i,(N-1) \oplus \Delta_i^*}, \mathsf{iv}; \hat{\ell})$ and lets $\mathbf{Q}_i = (\mathbf{q}_{i,0}, \dots, \mathbf{q}_{i,k_b-1})$. With that, Ext defines $\mathbf{V}_i^* = \mathbf{Q}_i \oplus \left[\delta_{i,0} \cdot \boldsymbol{u}_i^* \cdots \delta_{i,k_b} \cdot \boldsymbol{u}_i^*\right]$, for $i \in [0, \tau)$.
– Case 3 (Missing multiple preimages): If two or more preimages were not found, say $\mathsf{sd}_{i,j_1^*} = \bot = \mathsf{sd}_{i,j_2^*}$ for $j_1^* \neq j_2^*$, then Ext outputs $\bot$, i.e., extraction fails.

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

- If extraction failed, since the preimage for h_{com} (resp., some preimage for com_i) was missing, in step 1 (resp. step 2), query $\mathsf{H}_1(h_{\mathsf{com}})$ (resp., pick an arbitrary, say random, com_i without preimage and query $\mathsf{H}_1(\mathsf{com}_i)$).
- If extraction failed since more than one preimage for some $\overline{\mathsf{com}}_{i,j}$ was missing, in step 2, pick an arbitrary, say random, $\overline{\mathsf{com}}_{i,j}$ without preimage and query it at $\mathsf{H}_0(\overline{\mathsf{com}}_{i,j})$.

Note that the cases are mutually exclusive, i.e., Ext will make at most one query to either H_1 or H_0.

To analyse Ext, we consider the (graph from) the game from Lemma 4 and define by Fail the (failure) event that, during the extractable binding game, a preimage attack or a collision attack succeeded in the graph for either H_0'' or H_1; here, we write $\mathsf{H}_0(x) = (\mathsf{H}_0'(x), \mathsf{H}_0''(x))$, i.e., we consider preimage and collision attacks w.r.t. to the second component of H_0 only. Note that since the second component H_0'' is used for the commitments $\overline{\mathsf{com}}_{i,j}$ to $\mathsf{sd}_{i,j}$, considering H_0'' instead of H_0 is necessary.

Consider the whole binding game as an adversary $\mathcal{A}'$ to the random oracle graph game in Lemma 4, where only the subroutine $\mathcal{A}$ of $\mathcal{A}'$ makes fresh random oracle queries. Observe that if the bad event Fail occurs, then if $\mathcal{A}'$ wins Lemma 4. This holds because, either the adversary finds a random oracle collision, or it finds a preimage for a value x which Ext could not extract. To cover the latter case in the random oracle game of Lemma 4, Ext makes a additional query $\mathsf{H}_0(x)$ or $\mathsf{H}_1(x)$ which adds x as a node to the random oracle game, so that if the adversary finds the missing preimage of x, then the game aborts.

As a consequence, failing to look up h_{com}, com_i, or $\overline{\mathsf{com}_{i,j}}$ can trigger an extra query to H_1 or to H_0 by Ext. By Lemma 4 and a union bound, we therefore find

$$\Pr[\mathsf{Fail}] \leq ((Q_0+1)^2 + (Q_1+1)^2)/2^{2\lambda} \leq ((Q_0+1)^2 + (Q_1+1)^2)/2^{2\lambda}.$$

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

- $\overline{\mathsf{com}}_{i,j}$ uniquely defines $\mathsf{sd}_{i,j}$.[12,13]
- h_{com} uniquely defines $(\mathsf{com}_0 \| \ldots \| \mathsf{com}_{\tau-1})$.
- com_i uniquely defines $(\overline{\mathsf{com}}_{i,0}, \ldots, \overline{\mathsf{com}}_{i,N_i-1})$ for $i \in [0,\tau)$.

Thus, it is only possible for $\mathcal{A}$ to open h_{com} successfully (i.e., $h_{\mathsf{com}} = h_{\overline{\mathsf{com}}} \neq \bot$) if

- Whenever $\Delta_i^* = \bot$, i.e., com_i could be fully extracted, then

$$\mathbf{Q}_i = \mathbf{V}_i^* \oplus \left[\delta_{i,0}\cdot\boldsymbol{u}_i^* \cdots \delta_{i,k_b}\cdot\boldsymbol{u}_i^*\right].$$

[12] If a preimage x exists under H_0'', it is unique and $\mathsf{sd}_{i,j} = \mathsf{H}_0'(x)$ by definition.

[13] 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 $\mathsf{VC.Reconstruct}$. The checks and extraction rely only on $(\mathsf{sd}_j, \overline{\mathsf{com}}_j) = \mathsf{H}_0(k_j^d)$, in the notation of $\mathsf{VC.Reconstruct}$.

This can be seen as follows: By the condition $\neg\mathsf{Fail}$, the reconstructed seeds $\mathsf{sd}'_{i,j}$ in VOLEReconstruct coincide with the (extracted) seeds $\mathsf{sd}_{i,j\oplus\Delta_i}$ for $j \in [0, N)$, except that $\mathsf{sd}'_{i,j} = \bot$. Proposition 5 asserts that in this case, the equality $\mathbf{Q}_i = \mathbf{V}^*_i \oplus \left[\delta_{i,0} \cdot \boldsymbol{u}^*_i \cdots \delta_{i,k_b} \cdot \boldsymbol{u}^*_i\right]$ holds, as required.

– Whenever $\Delta^*_i \neq \bot$, the choice Δ_i of $\mathcal{A}$ is Δ^*_i. Otherwise, $\mathcal{A}$ would lose: If $\Delta^*_i \neq \bot$, then at least one commitment $\overline{\mathsf{com}}_{i,j}$ could not be extracted, hence this challenge must be used by $\mathcal{A}$ otherwise it could not successfully decommit (due to condition $\neg\mathsf{Fail}$). If challenge Δ^*_i is used, then by condition $\neg\mathsf{Fail}$ and Proposition 5, we again find that the equality $\mathbf{Q}_i = \mathbf{V}^*_i \oplus \left[\delta_{i,0} \cdot \boldsymbol{u}^*_i \cdots \delta_{i,k_b} \cdot \boldsymbol{u}^*_i\right]$ holds, as required.

Thus, we have shown that if Fail does not occur, extraction always succeeds. This concludes the proof. □

Next, we show that VOLECommit is hiding. To do this, we model our use of random IVs in PRGs with the following multi-target security game. The batch size L denotes the number of times the PRG is used with the same IV, while the number of queries denotes the number of independent IVs used. In the AES-CTR construction we use, increasing L degrades security by a factor of up to L, while increasing Q does not lead to any practical loss.

Definition 6 (IV-based PRG and multi-challenge security). *Let* $\mathsf{PRG}_m \colon \{0,1\}^\lambda \times \{0,1\}^{128} \to \{0,1\}^m$ *be deterministic polynomial-time algorithm. Let $\mathcal{A}$ be a Q-query L-batch adversary in the following game:*

- *For* $i \in [1, Q]$ *sample* $\mathsf{iv}^i \leftarrow \{0,1\}^{128}$.
  - *For* $j \in [1, L]$ *sample* $r^i_j \leftarrow \{0,1\}^\lambda$
    - *Compute* $s^i_j = \mathsf{PRG}_m(r^i_j, \mathsf{iv}^i)$
- *Sample* $b \leftarrow \{0,1\}$
  - *If* $b = 0$, *set* $t^i_j = s^i_j$ *for all* i, j.
  - *If* $b = 0$, *set* $t^i_j \leftarrow \{0,1\}^m$ *for all* i, j.
- $b' \leftarrow \mathcal{A}((\mathsf{iv}^i, (t^i_j)_{j\in[1,L]})_{i\in[1,Q]})$.
- *Output* 1 *(win) if* $b = b'$, *else* 0 *(lose).*

Let p be the probability that $\mathcal{A}$ wins the game. The advantage $\mathsf{AdvPRG}^{\mathsf{PRG}_m}_{\mathcal{A}}[Q, M]$ *of Q-query L-batch adversary $\mathcal{A}$ against the PRG security is defined as*

$$\mathsf{AdvPRG}^{\mathsf{PRG}_m}_{\mathcal{A}}[Q, L] = 2p - 1.$$

Remark 2. By a standard hybrid argument, any 1-query 1-batch IV-based PRG PRG is Q-query L-batch secure with $\mathsf{AdvPRG}^{\mathsf{PRG}}[Q, L] \leq QL \cdot \mathsf{AdvPRG}^{\mathsf{PRG}}[1, 1]$.

Lemma 5 (VOLECommit is multi-hiding). *Consider the following multi-hiding experiment for VOLECommit and a Q-query adversary $\mathcal{A}$, defined for some arbitrary* $\hat{\ell}$, $\mathtt{param}$.

1. *Sample* $b^* \leftarrow \{0,1\}$
2. *For* $j \in [1, Q]$:
   - $r^j \leftarrow \{0,1\}^\lambda$, $\mathsf{iv}^j \leftarrow \{0,1\}^{128}$
   - $(h^j_{\mathsf{com}}, (\mathsf{decom}^j_i)_{i\in[0,\tau)}, (\boldsymbol{c}^j_i)_{i\in[1,\tau)}, \boldsymbol{u}^j, \mathbf{V}^j) := \mathsf{VOLECommit}(r^j, \mathsf{iv}^j, \hat{\ell}; \mathtt{param})$
   - $\mathsf{chall}^j_3 \leftarrow \{0,1\}^\lambda$
   - *For* $i \in [0..\tau)$, *let* $\mathbf{s}^j_i := \mathsf{ChalDec}(\mathsf{chall}^j_3, i; \mathtt{param})$.

- *For* $i \in [0..\tau)$, *let* $(\mathsf{pdecom}_i^j) := \mathsf{VC.Open}(\mathsf{decom}_i^j, \mathbf{s}_i^j)$
- *Define* $(\overline{\mathbf{u}}^j, \overline{\mathbf{c}}_1^j, \ldots, \overline{\mathbf{c}}_{\tau-1}^j) = \begin{cases} (\mathbf{u}^j, \mathbf{c}_1^j, \ldots, \mathbf{c}_{\tau-1}^j) & \text{if } b^* = 0 \\ \text{random from } (\mathbb{F}_2^{\hat{\ell}})^\tau & \text{if } b^* = 1 \end{cases}$

3. $b \leftarrow \mathcal{A}((\overline{\mathbf{u}}^j, (\mathsf{pdecom}_i^j)_{i\in[0..\tau)}, \mathsf{chall}^j, h_{\mathsf{com}}^j, \overline{\mathbf{c}}_1^j)_{j\in[1,N]})$
4. *Output* 1 *(success) if* $b = b^*$. *Else output* 0 *(failure).*

Write $\mathsf{PRG}_{\mathsf{len}(s)} = \mathsf{PRG}(s; \mathsf{len})$. *The advantage of* $\mathcal{A}$ *in the above game is at most*

$$\mathsf{AdvHide}_{\mathcal{A}}^{\mathsf{VOLECommit}}[Q] \leq \mathsf{AdvPRG}^{\mathsf{PRG}_{\tau\lambda}}[Q, 1] + k_0 \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_{2\lambda}}[Q, \tau] + \mathsf{AdvPRG}^{\mathsf{H}_0}[Q, \tau] + \mathsf{AdvPRG}^{\mathsf{PRG}_{\hat{\ell}}}[Q, \tau]$$

The same holds against quantum adversaries, in which case the advantage terms are with respect to quantum adversaries.

Note that we can sample the challenges $(\mathsf{chall}_3^j)_{j\in[1,Q]}$ before committing without changing the adversary's view. Hence, all decommitment indices for all VC commitments are known at commitment time. That is, we only assert selective security.

Proof. We define a sequence of hybrid experiments. For simplicity, we describe the reduction for $Q = 1$ challenges. Since all VOLECommit instances are independent, it is straightforward to generalize it to $Q > 1$.

First, we replace the $\tau\lambda$-bit PRG output $(r_0\|\ldots\|r_{\tau-1}) := \mathsf{PRG}(r; \tau\lambda)$ in line 1 of VOLECommit by sampling true randomness $(r_0\|\ldots\|r_{\tau-1}) \leftarrow \{0,1\}^{\tau\lambda}$. Note that we have an additional iv in our definition of PRG, so the reduction will first receive the PRG challenge (iv, s). Then, it uses s in place of $(r_0\|\ldots\|r_{\tau-1})$ and iv for the remainder of the computation.

Next, we modify each of the VC instances used in VOLECommit one by one. For each instance, we first rely on the GGM construction of a puncturable PRF from a length-doubling PRG, which allows us to successively replace PRG calls along the path of Δ_i to the root by truly random outputs. After $\log N_i \leq k_0$ hybrid steps, the leaf seeds k_j^i are replaced with random values. Next, using that H_0 is a PRG, we can replace the single hidden sd_j value with a random seed. Again, we receive an iv as part of the challenge during the reduction, and use that iv in computation. Finally, we replace the $\hat{\ell}$-bit PRG output of sd_j, computed in ConvertToVOLE, with a random string, with the same arguments as before.

After repeating these steps for each of the τ VC instances, every $\mathbf{u}_i$ value computed in VOLECommit is now uniformly random. Hence, $\mathbf{u}$ as well as $\mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1}$ are uniformly random (independent of the commitments), and no adversary can win the game with advantage better than 1/2. As described, we sequentially replace the τ instances of VC commitments in VOLECommit. However, using the multi-challenge PRG property, this can be done in parallel. Similarly, the reduction can handle Q instances of VOLECommit in parallel. This yields the advantage terms.

Observing that we did not make use of $\mathcal{A}$'s structure in any way yields the post-quantum security statement. □

10.1.2 Security of the Signature Scheme. Building on the previous analysis, we now argue security of the overall signature scheme. We first consider EUF-KO security, that is, a key-only attack where the adversary must output a forgery given only the public key. Then, we extend this to EUF-CMA security, where the adversary also has access to a signing oracle.

Theorem 1 (FAEST is EUF-KO). *Let* H_0, H_1 *and* $\mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$ *be modeled as random oracles and* $(\lambda, \tau, \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$ *be parameters of the* FAEST *signature. Let* $\mathcal{A}$ *be an adversary which, for simplicity, runs* FAEST.Verify *on its forgery before it outputs, and outputs* $\perp$ *if verification fails. Let* Q_0, Q_1, $Q_{2,i}$ *denote upper bounds on the number of queries of* $\mathcal{A}$ *to* H_0, H_1, H_2^i, *respectively, and let* $Q_2 = Q_{2,1} + Q_{2,2} + Q_{2,3}$. *Then*

$$\mathbf{Adv}_{\mathcal{A}}^{\mathsf{EUF\text{-}KO}} \leq 3 \cdot (Q_0 + Q_1 + Q_2)^2 / 2^{2\lambda} + Q_2 \cdot (\varepsilon_{\mathsf{v}} + \varepsilon_{\mathsf{zk}} + 2^{-\lambda}) + \mathbf{Adv}_{\mathcal{B}}^{\mathsf{OWF}}. \quad (4)$$

Remark 3. As in Lemma 3, the assumption that $\mathcal{A}$ checks if 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 $Q_0 + \tau N$, $Q_1 + \tau + 1$, $Q_{2,1} + 1$, $Q_{2,2} + 1$, $Q_{2,3} + 1$ for H_0, H_1, H_2^1, H_2^2, H_2^3 respectively. Since $Q_0, Q_1, Q_2 \gg \tau N$, the effect on concrete security is minimal.

Before we argue security, we sketch how the extraction of the witness from an adversary generating a signature works. Recall that the Fiat–Shamir transformation turns interactive public-coin proof systems into non-interactive ones. However, security is in general not preserved, and stronger round-by-round security notions are required [CCH+19, CMS19, BBD+23], because deriving the challenges via hashing allows the malicious prover to "reset" the verifier to a previous state. In [BBD+23], security was therefore argued in multiple steps:

1. The zero-knowledge proofs were designed in the VOLE-hybrid model and proven round-by-round secure.
2. A VOLE functionality was realized in the $n-1$-out-of-n-OT-hybrid model (based on the SoftSpokenVOLE protocol [Roy22]) and proven round-by-round secure.
3. A compiler for zero-knowledge proofs in the $n-1$-out-of-n-OT-hybrid model to a CRS and/or RO model based on computationally hiding, straightline extractable all-but-one vector commitments was designed and the security of the Fiat–Shamir transformation of the compiled protocol was shown.

In FAEST, all of these steps happen in the proof of Theorem 1. For extracting a forgery, we do these steps in reverse:

- Extract VOLECommit commitments, as in Lemma 3. Output $\perp$ if extraction fails.
- Run the simulator for [Roy22] (SSOT), with minor adaptations, to obtain the corrected VOLE. Output $\perp$ if chall_1 was "bad".
- Run the ZK proof protocol. Output $\perp$ if chall_2 or chall_3 were "bad".

To bound the probability that extraction during forgery yields $\perp$, we argue through the existence of bad events, analogous to [BBD+23]. As the "protocol execution" is driven through hash queries, we must take care of different threads of execution, i.e., the adversary may modify the hash query to obtain a different challenge. In [BBD+23], this was handled explicitly through round-by-round knowledge. In our proof, it is handled implicitly through the "bad events" $\mathsf{Fail}_{\mathsf{VOLEHash}}$, $\mathsf{Fail}_{\mathsf{ZKHash}}$, $\mathsf{Fail}_{\mathsf{QS}}$ which correspond to round-by-round knowledge of these steps.

Proof. We argue via game hops.

Game G_1**:** This is the real EUF-KO security game. As in the EUF-KO game, we let sk, pk be the generated challenge secret and public key, respectively.

Game G_2: As the random oracle graph game in Lemma 4, this game now tracks random oracle queries in a graph (separately for each of the oracles H_0'', H_1, H_3^1, H_3^2), and aborts (that is, immediately outputs 0) when the graph game (Lemma 4) would be won. Here, $\mathsf{H}_0(x) = (\mathsf{H}_0'(x), \mathsf{H}_0''(x))$, where $\mathsf{H}_0''(x)$ is the $\overline{\mathsf{com}}$ part of H_0, cf. Lemma 3. Let Fail_0, Fail_1, $\mathsf{Fail}_{2,1}$, $\mathsf{Fail}_{2,2}$ denote the event that the graph for H_0'', H_1, $\mathsf{H}_{2,1}$, $\mathsf{H}_{2,2}$, goes bad, respectively. Observe that unless a bad event happens, G_2 is identical to G_1. Since the following games will change the (number of) random oracle queries which are made throughout the game, we postpone to bound the probabilities of the bad events to the end of the proof.

We denote by $\overline{Q}_i$ and $\overline{Q}_{2,j}$ a bound on the number of queries to H_i and H_2^j, respectively. In this game, we have $\overline{Q}_i = Q_i$ and $\overline{Q}_{2,j} = Q_{2,j}$ for $i = 0, 1$, $j = 1, 2, 3$.

Game G_3: When $\mu \| h_{\mathsf{com}} \| \mathbf{c}_1 \| \ldots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}$ is queried to H_2^1 by $\mathcal{A}$, the game applies the extractor Ext from Lemma 3 to h_{com} to extract the base VOLEs $(\mathbf{u}_i, \mathbf{V}_i)_{i \in [0,\tau)}$ and remembers the result. If the adversary's forgery is valid, but was not extracted before, the game aborts.

Observe that, by the changes introduced in G_2, if extraction failed for h_{com}, then h_{com} cannot be part of a successful forgery. This holds, because $\mathsf{FAEST.Verify}$ checks the decommitment, and exactly as in Lemma 3, the decommitment can only be valid if a bad event Fail_0 or Fail_1 occurred. But due to the changes in G_2, this leads to an abort, contradicting a success of $\mathcal{A}$. This shows that the change is just conceptual, and we have

$$\Pr[\mathsf{G}_2 = 1] = \Pr[\mathsf{G}_3 = 1].$$

As in Lemma 3, the bound on random oracle queries of the game changes to

$$\overline{Q}_0 = Q_0 + Q_{2,1} \qquad \overline{Q}_1 = Q_1 + Q_{2,1}.$$

because at most $Q_{2,1}$ runs of Ext will be triggered, and at most one query to H_0 or H_1 will be made by a (failing) run of Ext.

Game G_4: When $\mu \| h_{\mathsf{com}} \| \mathbf{c}_1 \| \ldots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}$ is queried to H_2^1 by $\mathcal{A}$ and extraction succeeds (i.e., does not output $\perp$, but $(\mathbf{u}_i, \mathbf{V}_i)_{i \in [0,\tau)}$), the game samples chall_1 from $\{0,1\}^{5\lambda+64}$ and checks if the VOLE consistency check for $\mathbf{u} = \mathbf{u}_0$ with purported correction $(\mathbf{c}_1, \ldots, \mathbf{c}_{\tau-1})$ was bad, and aborts if that happens. We define and bound the event for a bad consistency check in the following.

This stage of our protocol is a special case of SoftSpokenVOLE, with the difference that we use all-but-one vector commitments instead of all-but-one oblivious transfer. So to define the bad event, we define an extractor Ext_2, which corresponds to the extractor/simulator of the subspace VOLE from [BBD+23, Figure 5]. That is, we explain how the protocol parameters and simulation match up; a tedious hurdle is to account the fact that our VOLEs are built up using two different fields $\mathbb{F}_{2^{k_0}}$ and $\mathbb{F}_{2^{k_1}}$ (with $k_0 \geq k_1$), whose elements are then concatenated together. We handle this by relying on the feature of the VOLE functionality in [BBD+23] that the random challenge Δ does not need to be sampled from the full vector space, but rather, its individual elements can be restricted to lie in a smaller subset of the field.

In more detail, our VOLE setup and consistency check are based on [BBD+23, Figure 4], instantiated with the repetition code over $\mathbb{F}_2$, where $\mathbf{G}_{\mathcal{C}} = (1, \ldots, 1)$, with $k_{\mathcal{C}} = 1, n_{\mathcal{C}} = \tau$, $p = 2$ and $q = 2^{k_0}$. Define the set $S_\Delta \subset \mathbb{F}_q^\tau$ to be the set of vectors $(\Delta_0, \ldots, \Delta_{\tau-1})$, where for each i, if $i < \tau_0$ then $\Delta_i \in \{0,1\}^{k_0}$ is arbitrary, and otherwise, $\Delta_i \in \{0,1\}^{k_1} \| 0^{k_0 - k_1}$ (viewing bit-strings as field elements in the usual way).

We define the extractor Ext_2 to first compute $(\mathbf{u}_i^*, \mathbf{V}_i^*)_{i\in[0..\tau)}$ using Ext, and abort if this fails. Recall that $\mathbf{V}_i^* \in \mathbb{F}_2^{\hat{\ell}\times k_b}$, where $b = 0$ if $i < \tau_0$ and 1 otherwise. To ensure that all $\mathbf{V}_i^*$'s have the same dimensions, append $k_0 - k_1$ columns of zeroes to $\mathbf{V}_i^*$ for every i where $i \geq \tau_0$. Now, encode each row of $\mathbf{V}_i^*$ as an element of $\mathbb{F}_q$, translating $\mathbf{V}_i^*$ into a vector $\mathbf{v}_i^* \in \mathbb{F}_q^\tau$. Define

$$\mathbf{U}' = \left[\mathbf{u}_0^* \cdots \mathbf{u}_{\tau-1}^*\right] \in \mathbb{F}_2^{\hat{\ell}\times\tau}, \quad \mathbf{V} = \left[\mathbf{v}_0^* \cdots \mathbf{v}_{\tau-1}^*\right] \in \mathbb{F}_q^{\hat{\ell}\times\tau}$$

Additionally define the correction matrix $\mathbf{C} = [\mathbf{c}_1 \cdots \mathbf{c}_{\tau-1}] \in \mathbb{F}_p^{\hat{\ell}\times(\tau-1)}$. This covers step 1 of the simulator $\mathcal{S}$ in [BBD+23, Figure 5] (receiving $\mathbf{U}', \mathbf{V}$ from $\mathcal{P}^*$), and step 2 (receiving $\mathbf{C}$ from $\mathcal{P}^*$). In step 4, $\mathcal{S}$ samples the hash $\mathbf{H}$ for the consistency check. This corresponds to the matrix associated with $\mathsf{VOLEHash}(\mathsf{chall}_1, \cdot)$, a linear hash. Observe that since we describe how the first query $\mu \| h_{\mathsf{com}} \| \mathbf{c}_1 \| \ldots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}$ is handled, the game samples chall_1 and hence is free to program it (to whatever hash $\mathcal{S}$ sampled). At this point, the bad event for the VOLE consistency can be defined exactly as in the proof of [BBD+23, Theorem 2]. However, to make sense of the bad event, we need to explain the simulator/extractor $\mathcal{S}$ of [BBD+23] further. As the next step, step 5, $\mathcal{S}$ receives the purported hash values $\widetilde{\mathbf{u}} \in \{0,1\}^{\lambda+B}$, $\widetilde{\mathbf{V}} \in \{0,1\}^{(\lambda+B)\times\lambda}$. In our case, these will be extracted from preimages, namely from a query $\mathsf{chall}_1 \| \widetilde{\mathbf{u}} \| h_V \| \mathbf{d}$ to H_2^2 (to obtain chall_2), and a query $\widetilde{\mathbf{V}}$ to H_1 (to obtain h_V). At that point, our extraction proceeds exactly as the simulator (steps 6–9) to compute the corrected VOLE inputs $\mathbf{u}^*, \mathbf{V}^*$. In particular, it computes a set $G \subseteq [0, \tau)$ of guessed columns and challenge values Δ_i^* for these columns, which are inconsistent with the correction. If the adversary's guesses do not occur, the verifier will eventually reject.

Going back to the VOLEHash and its bad event, we import following claim from [BBD+23, Theorem 2].

Claim 1. We say the consistency check is *bad*, if the set G which $\mathcal{S}$ computes is not guaranteed to cover all cheating challenges. It holds that: For all $(\mathbf{u}_i^*, \mathbf{V}_i^*)_{i\in[0..\tau)}$, $\mathbf{C} = [\mathbf{c}_1 \cdots \mathbf{c}_{\tau-1}] \in \mathbb{F}_p^{\hat{\ell}\times(\tau-1)}$, the probability that the consistency check is *bad* for independently sampled VOLEHash is at most $\varepsilon_{\mathsf{v}}\binom{\tau}{2}$.

Proof. See proof of [BBD+23, Theorem 2]. □

Let $\mathsf{Fail}_{\mathsf{VOLEHash}}$ be event that (extraction succeeds but) the VOLE check fails for some query of $\mathcal{A}$ to H_2^1. By Claim 1 and a union bound,

$$\Pr[\mathsf{Fail}_{\mathsf{VOLEHash}}] \leq Q_{2,1}\binom{\tau}{2}\varepsilon_{\mathsf{v}}$$

Game G_5: We modify the handling of a query $\mathsf{chall}_1 \| \widetilde{\mathbf{u}} \| h_V \| \mathbf{d}$ to H_2^2 by $\mathcal{A}$ as follows:

- If chall_1 has no preimage under H_2^1, i.e., is not in $\mathcal{L}_2^1$, then query chall_1 under H_2^1 to add it as a node in the graph (from game G_2).
- If h_V has no preimage under H_1, i.e., is not in $\mathcal{L}_1$, then query h_V under H_1 to add it as a node in the graph (from game G_2).

This increases the bound on the random oracle queries of the game to

$$\overline{Q}_1 = Q_1 + Q_{2,1} + Q_{2,2} \qquad \overline{Q}_{2,1} = Q_{2,1} + Q_{2,2}.$$

Game G_6: In the following, we always consider the interaction associated to some chall_1, which also by assumption, is a preimage of H_2^1 where VOLE extraction was successful; other cases are not modified. For such a chall_1, we modify the handling of a query $\mathsf{chall}_1 \| \tilde{\mathbf{u}} \| h_V \| \mathbf{d}$ to H_2^2 by $\mathcal{A}$ as follows:

– Sample $\mathsf{chall}_2 \in \{0,1\}^{3\lambda+64}$ and check if the ZK soundness check was bad (as defined below), and abort if that happens.
– Look up a preimage $\widetilde{\mathbf{V}}$ of h_V under H_1. If a preimage does not exist, the extraction aborts.

Building on game G_4, which asserts that the VOLE relation corresponding to chall_1 is extracted, and the SSOT simulator (also described in game G_4), which is perfect unless the bad event excluded in game G_4 occurs, we obtain the corrected VOLE parts $\mathbf{u}$ and $\mathbf{v}$. Thus, we can extract the purported witness $\mathbf{w}$ from $\mathbf{d}$ in the query, by setting $\mathbf{w} := \mathbf{d} \oplus \mathbf{u}_{[0,\ldots,\ell)}$, undoing the masking in FAEST.Sign. If the extracted witness violates the quadratic constraints $(f_i)_{i=1}^{C}$ of the OWF that it should satisfy, but the random linear combination of the ZKHash does not, then this is the bad event. More precisely:

Claim 2 (ZK Hash check failure). Let $\mathbf{h} = [\mathbf{h}_1, h_2] \in \mathbb{F}_{2^\lambda}^{1\times\ell} \times \mathbb{F}_{2^\lambda}$ denote the matrix associated with the linear hash ZKHash, i.e., $\mathsf{ZKHash}(\mathsf{chall}_2, (\mathbf{x}_0, x_1)) = \mathbf{h}_1\mathbf{x}_0 + h_2 x_1$. Let $\mathbf{e} \in \mathbb{F}_{2^\lambda}$ be the degree 2 coefficient of $f_i(\xi \cdot \mathbf{w} + \mathbf{v})$ in the variable ξ, which is $\mathbf{0}$ if and only if $\mathbf{w}$ satisfies all $(f_i)_{i=1}^{C}$, where C is the number of multiplication constraints, cf. Table 2.2.

Define the bad event ("ZK Hash check fails") as $\mathbf{e} \neq 0$ but $\mathbf{h}_0 \cdot \mathbf{e} = 0$, i.e. the event where ZK Hash corrects an error into a valid VOLE correlation. The probability that the ZK Hash check fails for an independent chosen hash chall_2 is at most $\varepsilon_{\mathsf{zk}}$.

Proof. This follows from the $\varepsilon_{\mathsf{zk}}$-universality of ZKHash. □

Let $\mathsf{Fail}_{\mathsf{ZKHash}}$ be the event that any ZK Hash check fails for a query of $\mathcal{A}$ to H_2^2 (which is modified in this game). By Claim 2 and a union bound, we find

$$\Pr[\mathsf{Fail}_{\mathsf{ZKHash}}] \leq Q_{2,2}\varepsilon_{\mathsf{zk}}.$$

Game G_7: We modify the handling of a query $\mathsf{chall}_2 \| \tilde{a} \| \tilde{b}$ to H_2^3 by $\mathcal{A}$ as follows: If chall_2 has no preimage under H_2^2, i.e., is not in $\mathcal{L}_2^2$, then query chall_2 under H_2^2 to add it as a node in the graph (from game G_2). This increases the bound on the random oracle queries of the game to

$$\overline{Q}_{2,2} = Q_{2,2} + Q_{2,3}.$$

Game G_8**:** In the following, we further modify the handling of a query $\mathsf{chall}_2 \| \tilde{a} \| \tilde{b}$ to H_2^3 by $\mathcal{A}$. We always consider the interaction associated to some chall_2 and chall_1 of the same partial transcripts, such that the VOLE extraction was success for chall_1; other cases are not modified. The game's changes are as follows

1. The game looks up the preimage of chall_2 to find h_V and then looks up the preimage $\widetilde{\mathbf{V}}$ of h_V. If a preimage does not exist, the extraction aborts.
2. As in game G_6, obtain $\mathbf{u}$, $\mathbf{v}$ and define the witness $\mathbf{w}$.
3. The game samples $\mathsf{chall}_3 \in \{0,1\}^\lambda$.
4. If the bad event described below occurs, abort the game.

Claim 3 (QuickSilver check). Suppose that the witness $\mathbf{w}$ does not satisfy the constraints $(f_i)_{i=1}^C$. Then neither does the random linear combination via ZKHash, due to the abort introduced in G_6. The bad event is: for uniformly random chall_3, Δ and $\tilde{q}$ together with $\tilde{a}$ are such that $\mathsf{chall}'_3 = \mathsf{chall}_3$, where chall'_3 is computed using the $\tilde{b}$ queried by the adversary. The probability of this event is bounded by $2/2^\lambda$.

Proof. If the quadratic equation is not satisfied, then the "error polynomial" is a quadratic polynomial, which must agree with a linear polynomial evaluated at the random point Δ. This can occur in at most two points, hence 2 out of 2^λ challenges. □

Let $\mathsf{Fail}_{\mathsf{QS}}$ be the event that the QuickSilver check fails for any query of $\mathcal{A}$ to H_2^3 (which is modified in this game). By Claim 3 and a union bound, we find

$$\Pr[\mathsf{Fail}_{\mathsf{QS}}] \leq Q_{2,3} 2^{-\lambda+1}.$$

Claim 4. If a forgery is valid, then a valid witness for the OWF specified by pk is extracted during game G_8.

Proof. If the adversary wins, then by the changes in games until G_8,

- the VOLE is extracted and a witness $\mathbf{w}$ computed, so that
- the witness satisfies all constraints if and only if it satisfies the constraint obtained by the random linear combination (by G_6), and
- the Quicksilver check was not bad.

Since the Quicksilver check was not bad, the input to the check was a quadratic relation which holds. That relation is the ZKHash of the OWF relations $(f_i)_{i=1}^C$. By the second item, it follows that if the linearly combined constraint is satisfied, then all constraints are satisfied. In other words, the witness $\mathbf{w}$ satisfies all $(f_i)_{i=1}^C$, as required. □

Probability of an abort due to G_2. In game G_2, we introduced aborts whenever a preimage or collision attack succeeded on a random oracle. Lemma 4 bounds this probability in the total number of queries. Thus, we get the following probability bounds for the bad events:

$$\Pr[\mathsf{Fail}_0] \leq \overline{Q}_0^2/2^{2\lambda} \qquad \Pr[\mathsf{Fail}_1] \leq \overline{Q}_1^2/2^{2\lambda}$$
$$\Pr[\mathsf{Fail}_{2,1}] \leq \overline{Q}_{2,1}^2/2^{5\lambda+64} \qquad \Pr[\mathsf{Fail}_{2,2}] \leq \overline{Q}_{2,2}^2/2^{3\lambda+64}$$

where

$$\overline{Q}_0 = Q_0 + Q_{2,1} \qquad \overline{Q}_1 = Q_1 + Q_{2,1} + Q_{2,2}$$
$$\overline{Q}_{2,1} = Q_{2,1} + Q_{2,2} \qquad \overline{Q}_{2,2} = Q_{2,2} + Q_{2,3}$$

Overall, by a union bound and some rough upper bounds, we find

$$\begin{aligned}\Pr[F_0 \vee F_1 \vee F_{2,1} \vee F_{2,2}] &\leq (\overline{Q}_0 + \overline{Q}_1 + \overline{Q}_2)^2/2^{2\lambda} \\ &\leq 3 \cdot (Q_0 + Q_1 + Q_2)^2/2^{2\lambda}\end{aligned}$$

where $Q_2 = Q_{2,1} + Q_{2,2} + Q_{2,3}$.

Reduction to OWF. Finally, we give a reduction $\mathcal{B}$ to the OWF security. Given a OWF challenge $(\mathbf{x}, \mathbf{y})$ for uniformly random $\mathbf{x}$, $\mathbf{k}$ and $f_{\mathbf{k}}(\mathbf{x}) = \mathbf{y}$, define $\mathsf{pk} = (\mathbf{x}, \mathbf{y})$. Run game G_8 with $\mathcal{A}$ and if $\mathcal{A}$ wins, then by Claim 4 obtain a witness which, in

particular, contains a preimage $\mathbf{k}^*$ with $f_{\mathbf{k}^*}(\mathbf{x}) = \mathbf{y}$. The reduction $\mathcal{B}$ outputs $\mathbf{k}^*$, and thus wins whenever $\mathcal{A}$ wins game G_8. Moreover, if $\mathcal{A}$ wins EUF-KO, i.e., game G_1, it also wins G_8 unless an abort occurred. Taking all abort probabilities in game G_8 together, we find

$$\begin{aligned}\mathbf{Adv}_{\mathcal{A}}^{\mathsf{EUF\text{-}KO}} \leq\ & 3 \cdot (Q_0 + Q_1 + Q_2)^2 \cdot 2^{-2\lambda} \\ & + Q_{2,1} \cdot \binom{\tau}{2} \cdot \varepsilon_{\mathsf{v}} \\ & + Q_{2,2} \cdot \varepsilon_{\mathsf{zk}} \\ & + Q_{2,3} \cdot 2^{-\lambda+1} \\ & + \mathbf{Adv}_{\mathcal{B}}^{\mathsf{OWF}}.\end{aligned}$$

This concludes the proof. □

Theorem 2 **(FAEST is EUF-CMA secure).** *Let* $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^1, \mathsf{H}_2^2$ *and* H_2^3 *be modeled as random oracles and* H_3 *be a PRF, then for any PPT adversary* $\mathcal{A}$ *which makes* Q_{sig} *queries to* $\mathcal{O}_{\mathsf{sig}}$ *and* $Q_0, Q_1, Q_{2,1}, Q_{2,2}$ *and* $Q_{2,3}$ *queries to* $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2^1$*,* H_2^2 *and* H_2^3 *respectively,*

$$\mathbf{Adv}_{\mathcal{A}}^{\mathsf{EUF\text{-}CMA}} \leq Q_{\mathsf{sig}} \cdot \Bigg(\mathsf{AdvPRF}^{\mathsf{H}_3} + \frac{Q_{\mathsf{sig}} + Q_{2,1}}{2^{2\lambda+128}} + \frac{((N+2)\tau + 2)Q_{\mathsf{sig}} + Q_1 + Q_0}{2^{2\lambda}} \tag{5}$$

$$\begin{aligned}& + k_0 \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_{2\lambda}} + \frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{5\lambda+64}} + \frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}}\Bigg) \\ & + \mathsf{AdvPRG}^{\mathsf{PRG}_{\tau\lambda}}[Q_{\mathsf{sig}}, 1] + k_0 \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_{2\lambda}}[Q_{\mathsf{sig}}, \tau] \\ & + \mathsf{AdvPRG}^{\mathsf{H}_0}[Q_{\mathsf{sig}}, \tau] + \mathsf{AdvPRG}^{\mathsf{PRG}_{\hat{\ell}}}[Q_{\mathsf{sig}}, \tau] \\ & + \mathbf{Adv}_{\mathcal{B}}^{\mathsf{EUF\text{-}KO}}\end{aligned} \tag{6}$$

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.

G_2: $\mathcal{O}_{\mathsf{sig}}$ samples $r \in \{0,1\}^\lambda$ and $\mathsf{iv} \in \{0,1\}^{128}$ at random instead of computing $\mathsf{H}_3(\mathsf{sk}\|\mu\|\rho)$.
The difference between this game and the previous one reduces to the PRF security of H_3 with secret key ρ. Since we need to change (r, iv) in every query to $\mathcal{O}_{\mathsf{sig}}$, we run a hybrid argument and obtain:

$$|\Pr[\mathcal{A} \text{ wins } \mathsf{G}_1] - \Pr[\mathcal{A} \text{ wins } \mathsf{G}_2]| \leq Q_{\mathsf{sig}} \cdot \mathsf{AdvPRF}^{\mathsf{H}_3}. \tag{7}$$

G_3: $\mathcal{O}_{\mathsf{sig}}$ samples $\mathsf{chall}_1 \in \{0,1\}^{5\lambda+64}$ at random in every query. When $\mu\|h_{\mathsf{com}}\|\boldsymbol{c}_1\| \ldots \|\boldsymbol{c}_{\tau-1}\|\mathsf{iv}$ is queried to H_2^1, abort if this query has already been made during the game; otherwise program H_2^1 to output chall_1 on this input. As iv is random and h_{com} is sufficiently pseudorandom, we get the following

Claim.

$$\Pr[\mathsf{G}_3 \text{ aborts}] \leq Q_{\mathsf{sig}} \cdot \Big(\frac{Q_{\mathsf{sig}} + Q_{2,1}}{2^{2\lambda+128}} + \frac{((N+2)\tau + 2)Q_{\mathsf{sig}} + Q_1 + Q_0}{2^{2\lambda}} + d \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_{2\lambda}} \Big).$$

Proof. We proceed via an additional sequence of hybrids where we replace intermediate values by randomness in the VC commitment com_0, which is part of $h_{\mathsf{com}} = \mathsf{H}_1(\mathsf{com}_0, \ldots, \mathsf{com}_{\tau-1})$. More concretely, the PRG evaluations of the leftmost path of the GGM tree in com_0 are replaced by randomness, and the further hashes $(\mathsf{sd}_0, \overline{\mathsf{com}}_0) = \mathsf{H}_0(k_0^{k_0} \| \mathsf{iv})$ and $h = \mathsf{H}_1(\overline{\mathsf{com}}_0, \ldots, \overline{\mathsf{com}}_{N-1})$ in VC.Commit, as well as $h_{\mathsf{com}} = \mathsf{H}_1(\mathsf{com}_0, \ldots, \mathsf{com}_{\tau-1})$ in VOLECommit are replaced by randomness as well. For $i' = 0, \ldots, d$ (where $d = k_0$), define

$\mathsf{G}_{3,i'}$: When $\mathcal{O}_{\mathsf{sig}}$ calls VC.Commit in Line 5 of VOLECommit for loop index $i = 0$, then for loop indices $j = 0$ and $i \leq i'$ in Line 3 of the PRG calls of VC.Commit the PRG outputs (k_0^i, k_1^i) are replaced by uniformly random strings.

In addition, we define 3 more games,

$\mathsf{G}_{3,d+1}$: In addition, the output $(\mathsf{sd}_0, \overline{\mathsf{com}}_0)$ of H_0 in Line 5 of VC.Commit (again only for the VC.Commit call in Line 5 of VOLECommit for loop index $i = 0$) is replaced with a uniformly random string for loop index $j = 0$.

$\mathsf{G}_{3,d+2}$: In addition, the output h of H_1 in Line 6 of VC.Commit (again only for the VC.Commit call in Line 5 of VOLECommit for loop index $i = 0$) is replaced by a random string.

$\mathsf{G}_{3,d+3}$: In addition, the output h_{com} of Line 12 of VOLECommit is replaced with a random string.

In $\mathsf{G}_{3,d+3}$, the input $\mu \| h_{\mathsf{com}} \| \boldsymbol{c}_1 \| \ldots \| \boldsymbol{c}_{\tau-1} \| \mathsf{iv}$ to H_2^1 has min-entropy at least $2\lambda + 128$, so abort happens with probability at most

$$\Pr[\mathsf{G}_{3,d+3} \text{ aborts}] \leq Q_{\mathsf{sig}} \cdot \frac{Q_{\mathsf{sig}} + Q_{2,1}}{2^{2\lambda+128}}.$$

It remains to bound the differences in abort probability of consecutive hybrids. $H_1(\cdot \| \mathsf{com}_1 \| \ldots \| \mathsf{com}_{\tau-1})$ is a random-oracle-based PRG with seed length 2λ. In $\mathsf{G}_{3,d+2}$, it is seeded by a random string to produce h_{com} in Line 12 of VOLECommit, which is uniformly random in $\mathsf{G}_{3,d+3}$. We thus get

$$|\Pr[\mathsf{G}_{3,d+3} \text{ aborts}] - \Pr[\mathsf{G}_{3,d+2} \text{ aborts}]| \leq Q_{\mathsf{sig}} \frac{(\tau+1)Q_{\mathsf{sig}} + Q_1}{2^{2\lambda}}.$$

By a similar argument we get

$$|\Pr[\mathsf{G}_{3,d+2} \text{ aborts}] - \Pr[\mathsf{G}_{3,d+1} \text{ aborts}]| \leq Q_{\mathsf{sig}} \frac{(\tau+1)Q_{\mathsf{sig}} + Q_1}{2^{2\lambda}}$$

and

$$|\Pr[\mathsf{G}_{3,d+1} \text{ aborts}] - \Pr[\mathsf{G}_{3,d} \text{ aborts}]| \leq Q_{\mathsf{sig}} \frac{N\tau Q_{\mathsf{sig}} + Q_0}{2^{2\lambda}}.$$

For $i \in \{0, \ldots, d-1\}$, the difference between $\mathsf{G}_{3,i}$ and $\mathsf{G}_{3,i+1}$ is that for each generated signature, one output of PRG is real in the former and random in the latter. Hence

$$|\Pr[\mathsf{G}_{3,i} \text{ aborts}] - \Pr[\mathsf{G}_{3,i+1} \text{ aborts}]| \leq Q_{\mathsf{sig}} \cdot \mathsf{AdvPRG}_{\mathcal{A}}^{\mathsf{PRG}_{2\lambda}}.$$

The triangle inequality now yields

$$\Pr[\mathsf{G}_3 \text{ aborts}] \leq Q_{\mathsf{sig}} \cdot \Big(\frac{Q_{\mathsf{sig}} + Q_{2,1}}{2^{2\lambda+128}} + \frac{((N_0+2)\tau + 2)Q_{\mathsf{sig}} + Q_1 + Q_0}{2^{2\lambda}} + k_0 \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_{2\lambda}} \Big).$$

where we inserted $d = k_0$ and $N = N_0$. □

G_4: $\mathcal{O}_{\mathsf{sig}}$ samples $\mathsf{chall}_2 \in \{0,1\}^{3\lambda+64}$ at random in every query. When $\mathsf{chall}_1 \| \tilde{\mathbf{u}} \| h_V \| \mathbf{d}$ is queried to H_2^2, abort if this query 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 chall_1 has been previously returned by H_2^1; since chall_1 is fresh because of G_3, by a hybrid argument, the abort probability in this game is:

$$\Pr[\mathsf{G}_4 \text{ aborts}] \leq Q_{\mathsf{sig}} \cdot \frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{5\lambda+64}}.$$

G_5: $\mathcal{O}_{\mathsf{sig}}$ samples $\mathsf{chall}_3 \in \{0,1\}^{\lambda}$ at random in every query. When $\mathsf{chall}_2 \| \tilde{a} \| \tilde{b}$ is queried to H_2^3, abort if this query 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 chall_2 has been previously returned by H_2^2; since chall_2 is fresh because of G_4, by a hybrid argument, the abort probability of this game is:

$$\Pr[\mathsf{G}_5 \text{ aborts}] \leq Q_{\mathsf{sig}} \cdot \frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}}.$$

Observe that the relevant elements of this game are now the same as the hiding experiment for VOLECommit described in Lemma 5.

G_6: $\mathcal{O}_{\mathsf{sig}}$ replaces the call to VOLECommit by uniformly sampling $\mathbf{u} \in \mathbb{F}_2^{\hat{\ell}}$ and $(\boldsymbol{c}_i)_{i\in[1..\tau)}$ and computing $\mathbf{V} \in \mathbb{F}_2^{\hat{\ell}\times\lambda}$ such that the VOLE relation holds with the $\varDelta$ and $\mathbf{q}_i$ values that result from chall_3; it does so in every signature query. By Lemma 5, the difference between this game and the previous one is

$$|\Pr[\mathcal{A} \text{ wins } \mathsf{G}_5] - \Pr[\mathcal{A} \text{ wins } \mathsf{G}_6]| \leq \mathsf{AdvPRG}^{\mathsf{PRG}_{\tau\lambda}}[Q_{\mathsf{sig}}, 1] + k_0 \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_{2\lambda}}[Q_{\mathsf{sig}}, \tau] + \mathsf{AdvPRG}^{\mathsf{H}_0}[Q_{\mathsf{sig}}, \tau] + \mathsf{AdvPRG}^{\mathsf{PRG}_{\hat{\ell}}}[Q_{\mathsf{sig}}, \tau].$$

After this change, the distribution of $(\mathbf{u}, \mathbf{V})$ is uniform (modulo the VOLE relation) and independent of the VOLE commitments $(\boldsymbol{c}_i)_{i\in[1..\tau)}$.

G_7: In every query, $\mathcal{O}_{\mathsf{sig}}$ samples $\tilde{\boldsymbol{u}}$ and $\tilde{\mathbf{V}}$ at random (modulo the VOLE relation with $\tilde{\mathbf{Q}}$ and $\varDelta$) instead of computing VOLEHash during the signing. It then adjusts the last $\lambda + B$ elements of $\mathbf{u}$ and rows of $\mathbf{V}$ to match $\tilde{\boldsymbol{u}}$ and $\tilde{\mathbf{V}}$ when VOLEHash is applied under challenge chall_1 to $\mathbf{u}, \mathbf{V}$.
Since the last $\lambda + B$ elements of $\mathbf{u}$ and rows of $\mathbf{V}$ are uniformly random (modulo the VOLE relation) because of G_6, and VOLEHash is $\mathbb{F}_2^{\ell+\lambda}$-hiding by Lemma 1, this game is perfectly indistinguishable from the previous one.

G_8: In every query, $\mathcal{O}_{\mathsf{sig}}$ samples $\tilde{a}$ and $\tilde{b}$ at random (modulo the VOLE relation) instead of computing ZKHash. It then computes AESProve, but adjusts the $\mathbf{u}, \mathbf{V}$ in rows $[\ell..\ell+\lambda)$ to match $\tilde{a}$ and $\tilde{b}$ when ZKHash is applied under challenge chall_2 to $\mathbf{a}_1 \| u^*, \mathbf{a}_0 \| v^*$.
Since the respective rows of $\mathbf{u}$ and $\mathbf{V}$ are uniformly random (modulo the VOLE relation) because of G_6, and ZKHash is $\mathbb{F}_{2^\lambda}^{\ell}$-hiding by Lemma 2, the distribution of $(\tilde{a}, \tilde{b})$ is unchanged and this game is perfectly indistinguishable from the previous one.

G_9**:** Now the computation of $\tilde{a}$ and $\tilde{b}$ is independent of $\mathbf{w}$, so $\mathcal{O}_{\mathsf{sig}}$ samples $\mathbf{d}$ uniformly at random in every query instead of computing $\mathbf{d} := \mathbf{w} + \mathbf{u}_{[0..\ell)}$.
This does not change the distribution of $\mathbf{d}$ because $\mathbf{u}$ is uniform since G_6 and the first ℓ elements were not used in games G_7 and G_8 to produce their outputs, hence this game is perfectly indistinguishable from the previous one.

In G_9, we see that the distribution of σ produced by $\mathcal{O}_{\mathsf{sig}}$ no longer depends on the secret key sk and the success probability of $\mathcal{A}$ in this final game can be reduced to the EUF-KO security of FAEST using the $\mathcal{B}$ reduction defined above. □

10.1.3 QROM security. In this section, we provide evidence (short of a full proof) for the security of FAEST in the quantum-accessible random oracle model (QROM). More precisely, we give a non-tight security proof that relies on conjectural straightline-extractability of the VOLE commitment.

We begin with a QROM version of Theorem 2. To this end, we need to reprogram quantum-accessible random oracles (QROs). The re-sampling game REPRO_b for $b \in \{0,1\}$ and an adversary $\mathcal{A}^{\mathsf{H}} = (\mathcal{A}_0^{\mathsf{H}}, \mathcal{A}_1^{\mathsf{H}})$ with quantum access to a random oracle $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\gamma$ is defined as follows.

1. Adversary outputs an internal state and a re-sampling point distribution $(st, \mathsf{p}) \leftarrow \mathcal{A}_0^{\mathsf{H}}$
2. Sample re-sampling point $x \leftarrow \mathsf{p}$, $y \leftarrow \{0,1\}^\gamma$. Set $\mathsf{H}_0 = \mathsf{H}$ and $\mathsf{H}_1(x) = y$, $\mathsf{H}_1(x') = \mathsf{H}(x')$ for $x' \neq x$.
3. Adversary gets access to H_b, outputs guess $b' \leftarrow \mathcal{A}_1^{\mathsf{H}_b}(st, x)$.

We need the following

Lemma 6 (Special case of Theorem 1 in [GHHM21]). *Let $\mathcal{A}^{\mathsf{H}} = (\mathcal{A}_0^{\mathsf{H}}, \mathcal{A}_1^{\mathsf{H}})$ be an adversary for the reprogramming games Repro_b, making Q queries to H. Then*

$$\left|\Pr[b' = 1|\mathsf{REPRO}_0] - \Pr[b' = 1|\mathsf{REPRO}_1]\right| \leq \frac{3}{2}\sqrt{Qp_{\max}},$$

where $p_{\max} = \mathbb{E}_{\mathsf{p}}(\max_x \mathsf{p}(x))$, the maximum probability of any element under the probability distribution output by $\mathcal{A}_0$.

Theorem 3 (EUF-CMA security from EUF-KO security in the QROM). *Let $H_0, H_1, \mathsf{H}_2^1, \mathsf{H}_2^2$ and H_2^3 be modeled as quantum-accessible random oracles and H_3 be modeled as a post-quantum PRF, then for any QPT adversary $\mathcal{A}$ which makes Q_{sig} queries to $\mathcal{O}_{\mathsf{sig}}$ and $Q_0, Q_1, Q_{2,1}, Q_{2,2}$ and $Q_{2,3}$ queries to $H_0, H_1, \mathsf{H}_2^1, \mathsf{H}_2^2$ and H_2^3 respectively,*

$$\begin{aligned}\mathbf{Adv}_{\mathcal{A}}^{\mathsf{EUF\text{-}CMA}} \leq Q_{\mathsf{sig}} \cdot \Bigg(&\mathsf{AdvPRF}^{\mathsf{H}_3} + \frac{3}{2}\sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,1}}{2^{2\lambda+128}}} + 2\frac{((N+2)\tau + 2)Q_{\mathsf{sig}} + Q_1 + Q_0}{2^{2\lambda}} \\ &+ 2k_0 \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_{2\lambda}} + \frac{3}{2}\sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{5\lambda+64}}} + \frac{3}{2}\sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}}}\Bigg) \\ &+ \mathsf{AdvPRG}^{\mathsf{PRG}_{\tau\lambda}}[Q_{\mathsf{sig}}, 1] + k_0 \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_{2\lambda}}[Q_{\mathsf{sig}}, \tau] \\ &+ \mathsf{AdvPRG}^{\mathsf{H}_0}[Q_{\mathsf{sig}}, \tau] + \mathsf{AdvPRG}^{\mathsf{PRG}_{\hat{\ell}}}[Q_{\mathsf{sig}}, \tau] \\ &+ \mathsf{AdvColRes}^{\mathsf{H}_1} + \mathbf{Adv}_{\mathcal{B}}^{\mathsf{EUF\text{-}KO}}.\end{aligned} \quad (8)$$

Proof. The proof follows almost the same series of games as the proof of Theorem 2. G_1 and G_2 are defined identically to the ones in the proof of Theorem 2, and Equation (7) holds with the PRF advantage defined with respect to quantum computing adversaries. G_3 is identical except that it doesn't abort. Instead, we define hybrid H_i reprogramming H_2^1 for the first i calls of $\mathcal{O}_{\mathsf{sig}}$. For each i, we construct an adversary $\mathcal{B}^i$ for the reprogramming games REPRO_b. $\mathcal{B}_0^i$ simulates H_i for $\mathcal{A}$ until right before the $i+1$st call to $\mathcal{O}_{\mathsf{sig}}$. Then it outputs the distribution of $\mu \| h_{\mathsf{com}} \| \boldsymbol{c}_1 \| \dots \| \boldsymbol{c}_{\tau-1} \| \mathsf{iv}$ for the $i+1$st query. $\mathcal{B}_1^i$ uses its input x in place of $\mu \| h_{\mathsf{com}} \| \boldsymbol{c}_1 \| \dots \| \boldsymbol{c}_{\tau-1} \| \mathsf{iv}$ to generate the reply to the $i+1$st $\mathcal{O}_{\mathsf{sig}}$ query and finishes simulating H_i. If the simulated $\mathcal{A}$ outputs a valid forgery output 0, otherwise output 1. Using the same sequence of hybrids as in the proof of the claim in G_3 in the proof of Theorem 2, the games REPRO_b are successively transformed until h_{com} is random. Compared to the classical hybrid sequence, we incur each loss twice, once for changing REPRO_0 and once for changing REPRO_1. Using Lemma 6 we conclude

$$\begin{aligned}&|\Pr[\mathcal{A} \text{ succeeds in } \mathsf{G}_3] - \Pr[\mathcal{A} \text{ succeeds in } \mathsf{G}_2]| \\ &\le Q_{\mathsf{sig}}\Bigg(\frac{3}{2}\sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,1}}{2^{\lambda+128}}} + 2\frac{((N+2)\tau+2)Q_{\mathsf{sig}} + Q_1 + Q_0}{2^{2\lambda}} \\ &\quad + 2d \cdot \mathsf{AdvPRG}^{\mathsf{PRG}_{2\lambda}}\Bigg).\end{aligned}$$

The transitions G_3 to G_4 and G_4 to G_5 are obtained in the same way by constructing re-sampling game adversaries instead of aborting, yielding

$$|\Pr[\mathcal{A} \text{ succeeds in } \mathsf{G}_4] - \Pr[\mathcal{A} \text{ succeeds in } \mathsf{G}_3]| \le \frac{3}{2}Q_{\mathsf{sig}}\sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,2}}{2^{5\lambda+64}}}$$

and

$$|\Pr[\mathcal{A} \text{ succeeds in } \mathsf{G}_4] - \Pr[\mathcal{A} \text{ succeeds in } \mathsf{G}_3]| \le \frac{3}{2}Q_{\mathsf{sig}}\sqrt{\frac{Q_{\mathsf{sig}} + Q_{2,3}}{2^{3\lambda+64}}}.$$

The remaining game transitions are identical to the proof of Theorem 2, except that all adversarial advantages have to be defined with respect to quantum computing adversaries. □

It remains to bound $\mathbf{Adv}_{\mathcal{B}}^{\mathsf{EUF\text{-}KO}}$. FAEST.Sign can be interpreted as a mild tweak of the Fiat Shamir transformation of the 7-round identification scheme Π_{FAEST} defined as follows, referring to the lines in algorithm FAEST.Sign.

1. Prover runs line 5 with uniformly random r and sends the arguments of H_2^1 in line 6 except μ.
2. Verifier sends uniformly random chall_1.
3. Prover runs lines 6-12 and sends the arguments of H_2^2 in line 13 except chall_1.
4. Verifier sends uniformly random chall_2.
5. Prover runs lines 14-17 and sends the arguments of H_2^3 in line 18 except chall_2.
6. Verifier sends uniformly random chall_3.
7. Prover runs lines 19-23 and sends σ except chall_3, together with $\tilde{b}$ computed in line 17.
8. Verifier runs FAEST.Verify, skipping lines 5, 17 and 20 and using the values for chall_i $i = 1, 2, 3$ it sent.

We denote the verifier of Π_{FAEST} by $V_{\Pi_{\mathsf{FAEST}}}$. The signature scheme FAEST is now obtained from Π_{FAEST} by i) applying the hash-then-sign construction (μ in Line 2 of FAEST.Sign is the message hash), ii) applying the multi-round Fiat-Shamir transformation, and iii) replacing some data that can be recomputed by the verifier ($h_{\mathsf{com}}, h_V, \tilde{b}$) by (random-oracle-based) commitments to them. For the following QROM security considerations, we will ignore i) and iii) as the post-quantum security of Hash-then-Sign up to collision finding is well-known, and a signature for the scheme without iii) can be converted by anybody into a FAEST signature, and vice versa.

A natural strategy to prove knowledge soundness of Π_{FAEST} in the QROM is to show extractability of the VOLE commitment in the QROM, i.e. to prove a QROM analogue of Lemma 3. We formulate it here as a conjecture and intend to prove it in the near future.

Conjecture 1. Consider the extractable binding game for algorithms (VOLECommit, VOLEReconstruct), a stateful adversary $\mathcal{A}$ and a straightline extractor $\mathsf{Ext}_{\mathsf{VC}}$ described in Lemma 3.

Suppose $\mathcal{A}$ is any adversary making at most Q_0, Q_1 *quantum* queries to the oracles $\mathsf{H}_0, \mathsf{H}_1$ respectively. Moreover, suppose that $\mathcal{A}$ checks if $\mathsf{VOLEVerify}(h_{\mathsf{com}}, \mathsf{chall}, (\mathsf{pdecom}_i)_{i\in[0..\tau)}, \mathsf{iv}, \hat{\ell}; \mathtt{param}) = 1$ for the purported decommitment, and otherwise outputs $\perp$. There exists a straightline extractor Ext such that the advantage of $\mathcal{A}$ in the game is at most

$$\frac{\mathsf{poly}(Q_0, Q_1)}{2^{\gamma\lambda}}.$$

for some polynomial poly and a constant $\gamma > 0$

We present some evidence towards this conjecture.

- Plain hash-based commitments are known to be straightline extractable in the QROM [DFMS22].
- The proof strategy for Lemmas 3 and 4 seems amenable to generalization to the QROM via the compressed oracle method [Zha19, CFHL21]. The graph game in Lemma 4 includes finding pre-images of existing input values. In the compressed oracle framework, finding such a pre-image cannot be witnessed based just on the internal oracle register's state (just as it requires comparing the query graph before the query and the present query's input-output pair in the classical case). It can, however, be witnessed by applying a combined query/check operator acting on the internal oracle register and the adversary's query register.

Based on the conjectural straightline extractor $\mathsf{Ext}_{\mathsf{VC}}$ for the VOLE commitment, we can construct a straightforward extractor Ext_{Π} for Π_{FAEST}. Given a malicious prover $\mathcal{A}$ against Π_{FAEST}, the extractor $\mathsf{Ext}_{\Pi}^{\mathcal{A}}(\mathsf{pk})$ lets $\mathsf{Ext}_{\mathsf{VC}}$ simulate the QROs for $\mathcal{A}(\mathsf{pk})$ to obtain the extraction result $(\boldsymbol{u}^*, \mathbf{V}^*)$ and a transcript

$$T = (h_{\mathsf{com}} \| \mathbf{c}_1 \| \dots \| \mathbf{c}_{\tau-1} \| \mathsf{iv}, \mathsf{chall}_1, \widetilde{\mathbf{u}} \| h_V \| \mathbf{d}, \mathsf{chall}_2, \tilde{a} \| \tilde{b}, \mathsf{chall}_3, (\mathsf{pdecom}_i)_{i\in[0..\tau)}),$$

computes $\mathbf{w} = \mathbf{d} \oplus \mathbf{u}^*_{[0,\dots,\ell)}$ and outputs $\mathsf{sk} = \mathbf{w}$.

Theorem 4. *Assuming Conjecture 1, Π_{FAEST} is a quantum proof of knowledge with extractor Ext_{Π}. More precisely, we have*

$$\begin{aligned}\Pr&\left[(\mathsf{sk}',\mathsf{pk}) \text{ is valid} \,\middle|\, \begin{array}{c}(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ \mathsf{sk}' \leftarrow \mathsf{Ext}_{\Pi}^{\mathcal{A}}(\mathsf{pk})\end{array}\right]\\ &\geq \Pr[\langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{A}(\mathsf{pk})\rangle = \textit{accept} \mid (\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)]\\ &- \frac{\mathsf{poly}(Q_0,Q_1)}{2^{\gamma\lambda}} - \varepsilon_{\mathsf{v}}\binom{\tau}{2} - \varepsilon_{\mathsf{zk}} - 2/2^\lambda.\end{aligned}$$

Proof. The verifier of Π_{FAEST} performs the VOLE commitment check as part of its verification. We thus have by Conjecture 1, that

$$\begin{aligned}\Pr&\left[\mathsf{bad}_1 \wedge \langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{A}(\mathsf{pk})\rangle = \text{accept} \,\middle|\, \begin{array}{c}(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ \mathsf{sk}' \leftarrow \mathsf{Ext}_{\Pi}^{\mathcal{A}}(\mathsf{pk})\end{array}\right]\\ &\leq \frac{\mathsf{poly}(Q_0,Q_1)}{2^{\gamma\lambda}},\end{aligned}$$

where bad_1 is the event that the pair $(\mathbf{u}^*, \mathbf{V}^*)$ extracted by the instance of $\mathsf{Ext}_{\mathsf{VC}}$ run by Ext_{Π} does not yield VOLE instances.

What is left is to analyze the probability with which any of the probabilistic checks the verifier performs, facilitated by the challenges chall_i, fails. The proof system only uses random oracles for the VOLE commitment, so its soundness given successfully extracted VOLE correlations is proven in the same way classically and in the post-quantum case. It was analyzed in [BBD+23] and in the proof of Theorem 1, so we only give a sketch here.

Suppose first that there exists an $i \in \{1, \ldots, \tau-1\}$ such that $\mathbf{u}_i \neq \mathbf{u}_0 \oplus \mathbf{c}_i$. Call this event bad_2. According to Claim 1, the verifier accepts with probability at most $\varepsilon_{\mathsf{v}}\binom{\tau}{2}$.

Next, suppose that the extracted VOLE instances $\mathbf{u}, \mathbf{V}$ allow to extract a candidate witness $\mathbf{w}$ but which is incorrect for the OWF, as it violates a multiplication constraint as expressed in Equation Equation (2). Consider these constraints as equations in X, which for every inconsistency must be of degree 2. Let bad_3 be the event that the output of ZKHash on all constraints lets the degree 2 term of the linear combination of the constraints vanish. This happens with probability at most $\varepsilon_{\mathsf{zk}}$ by the universality of the hash function.

Finally, suppose again that the extracted witness violates a multiplication constraint as expressed in Equation Equation (2). This means that in a consistency test, the relation checked by the verifier has a non-zero term in X^2. At the same time, the verifier compares by evaluating a degree 1 relation defined by $\tilde{a}, \tilde{b}$ in the point Δ to test equality. Let bad_4 be the event that both the degree 2 and degree 1 constraints coincide when evaluated in Δ. According to Claim 3 this happens with probability at most $2/2^\lambda$.

We can now bound

$$\begin{aligned}&\Pr[(\mathsf{sk},\mathsf{pk}) \text{ is valid}]\\ &\quad\geq \Pr[(\mathsf{sk},\mathsf{pk}) \text{ is valid} \wedge \langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{A}(\mathsf{pk})\rangle = \text{accept}]\\ &\quad\geq \Pr[\langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{A}(\mathsf{pk})\rangle = \text{accept}] - \Pr[(\mathsf{sk},\mathsf{pk}) \text{ is invalid} \wedge \langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{A}(\mathsf{pk})\rangle = \text{accept}]\end{aligned}$$

Here and in the following, all probabilities are taken over $(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$ and $\mathsf{sk} \leftarrow \mathsf{Ext}_{\Pi}^{\mathcal{A}}(\mathsf{pk})$. If the verifier accepts, a good VOLE pair is extracted by $\mathsf{Ext}_{\mathsf{VC}}$ and

neither of the bad events occurs, the extracted key pair is valid. We thus have

$$\begin{aligned}
&\Pr\left[(\mathsf{sk},\mathsf{pk}) \text{ is invalid} \wedge \langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{A}(\mathsf{pk})\rangle = \text{accept}\right] \\
&\quad \le \Pr\left[(\mathsf{bad}_1 \vee \mathsf{bad}_2 \vee \mathsf{bad}_3 \vee \mathsf{bad}_4) \wedge \langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{A}(\mathsf{pk})\rangle = \text{accept}\right] \\
&\quad \le \Pr\left[\mathsf{bad}_1 \wedge \langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{A}(\mathsf{pk})\rangle = \text{accept}\right] + \Pr\left[\mathsf{bad}_2 \wedge \langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{A}(\mathsf{pk})\rangle = \text{accept}\right] \\
&\qquad + \Pr\left[\mathsf{bad}_3 \wedge \langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{A}(\mathsf{pk})\rangle = \text{accept}\right] + \Pr\left[\mathsf{bad}_4 \wedge \langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{A}(\mathsf{pk})\rangle = \text{accept}\right] \\
&\quad \le \frac{\mathsf{poly}(Q_0, Q_1)}{2^{\gamma\lambda}} + \varepsilon_{\mathsf{v}}\binom{\tau}{2} + \varepsilon_{\mathsf{zk}} + 2/2^{\lambda}.
\end{aligned}$$

□

We can now use the fact that Π_{FAEST} is a quantum proof of knowledge to show that its Fiat-Shamir transform is a quantum proof of knowledge as well (Corollary 15 in [DFM20]). This allows inverting the underlying AES-based OWF by reduction to breaking EUF-KO-security of FAEST′, the digital signature scheme where (pk, m) is used in place of μ in algorithms FAEST.Sign and FAEST.Verify.

Theorem 5. *Let H_0, H_1 and $\mathsf{H}_2^1, \mathsf{H}_2^2, \mathsf{H}_2^3$ be modeled as quantum-accessible random oracles and $(\lambda, \tau, \mathtt{param}, \mathtt{param}_{\mathsf{OWF}})$ be parameters of the FAEST′ signature. Let $\mathcal{A}$ be an adversary and let Q_0, Q_1, $Q_{2,i}$ denote upper bounds on the number of quantum queries of $\mathcal{A}$ to H_0, H_1, H_2^i, respectively, and let $Q_2 = Q_{2,1} + Q_{2,2} + Q_{2,3}$. Then, Assuming Conjecture 1,*

$$\mathbf{Adv}_{\mathcal{A}}^{\mathsf{EUF\text{-}KO}-\mathrm{QROM}} \le \frac{(2Q_2+4)^6}{6}\left(\mathbf{Adv}_{\mathcal{D}}^{\mathsf{OWF}} + 6\cdot 2^{-\lambda} + \frac{\mathsf{poly}(Q_0,Q_1)}{2^{\gamma\lambda}} + \varepsilon_{\mathsf{v}}\binom{\tau}{2} + \varepsilon_{\mathsf{zk}} + 2^{-\lambda+1}\right) \tag{9}$$

Proof. Π_{FAEST}, an identification scheme, is an interactive proof of knowledge of a secret key. In other words, the set of statements to be proven is the set of possible public keys, and the secret key corresponding to a given public key is a witness. Let Π'_{FAEST} be the proof system obtained from Π_{FAEST} by augmenting the set of statements by appending messages to the public keys (for the details of this formalism, see, e.g., Section 8 of [Unr17]). We can regard FAEST as the Fiat-Shamir transformation for public-coin interactive proof systems, applied to Π'_{FAEST}, and a successful malicious prover in the QROM is exactly an EUF-KO−QROM-adversary: It receives the public key, and convinces the verifier by generating a proof (i.e., by forging a signature).

By Corollary 13 in [DFM20], we can use $\mathcal{A}$ in a black-box way to construct an adaptive adversary $\mathcal{B}'$ against Π'_{FAEST} such that

$$\Pr[\langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{B}'(\mathsf{pk})\rangle = \text{accept} | (\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{Gen}(1^{\lambda})] \ge \frac{6}{(2Q_2+4)^6}\mathbf{Adv}_{\mathcal{A}}^{\mathsf{EUF\text{-}KO}-\mathrm{QROM}} - 6\cdot 2^{-\lambda}.$$

$\mathcal{B}'$ is a partially adaptive adversary in that it takes the public key as input, but outputs a message. From $\mathcal{B}'$ we build an adversary $\mathcal{B}$ against Π_{FAEST}: $\mathcal{B}$ runs $\mathcal{B}'$ but discards the message it outputs. They have the same success probability. Assuming

Conjecture 1, we can use Theorem 4 to bound

$$\begin{aligned}\Pr\left[(\mathsf{sk},\mathsf{pk}) \text{ is valid} \;\middle|\; \begin{matrix}(\mathsf{sk}',\mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ \mathsf{sk}' \leftarrow \mathsf{Ext}_\Pi^{\mathcal{B}}(\mathsf{pk})\end{matrix}\right] \\ \geq \Pr[\langle V_{\Pi_{\mathsf{FAEST}}}, \mathcal{B}(\mathsf{pk})\rangle = \text{accept} | (\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)] \\ - \frac{\mathsf{poly}(Q_0,Q_1)}{2^{\gamma\lambda}} - \varepsilon_{\mathsf{v}}\binom{\tau}{2} - \varepsilon_{\mathsf{zk}} - 2^{-\lambda+1}\end{aligned}$$

The algorithm $\mathsf{Ext}_\Pi^{\mathcal{B}}$ can now be used to construct an adversary $\mathcal{D}$ inverting the one-way function. Given a OWF challenge $(\mathbf{x},\mathbf{y})$ for uniformly random $\mathbf{x}$, $\mathbf{k}$ and $f_{\mathbf{k}}(\mathbf{x}) = \mathbf{y}$, define $\mathsf{pk} = (\mathbf{x},\mathbf{y})$, a public key identically distributed to the ones output by key generation. Then, run $\mathsf{sk}' \leftarrow \mathsf{Ext}_\Pi^{\mathcal{B}}(\mathsf{pk})$ and output sk'. Putting the inequalities together we get

$$\begin{aligned}&\mathbf{Adv}_{\mathcal{A}}^{\mathsf{EUF\text{-}KO}-\mathrm{QROM}} \\ &\leq \frac{(2Q_2+4)^6}{6}\left(\mathbf{Adv}_{\mathcal{D}}^{\mathsf{OWF}} + 6\cdot 2^{-\lambda} + \frac{\mathsf{poly}(Q_0,Q_1)}{2^{\gamma\lambda}} + \varepsilon_{\mathsf{v}}\binom{\tau}{2} + \varepsilon_{\mathsf{zk}} + 2^{-\lambda+1}\right)\end{aligned}$$

□

10.2 Concrete Attacks

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

The cryptographic primitives used in FAEST are the one-way functions 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). In Section 10.3, 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.

10.2.1 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 translates into a key recovery attack on AES, given one or two ciphertext blocks. A slight difference, in our case, is that KeyGen uses rejection sampling to sample a key such that the inputs to all S-Boxes in the OWF computation are non-zero. As analyzed in Section 10.3, this reduces the effective size of the keyspace by around 2–3x. We highlight that even an ordinarily sampled AES key will have all its S-Box inputs be non-zero with some probability anyway, sampling keys as we do cannot lead to any significant weaknesses; otherwise, a large fraction of AES keys in use today would be vulnerable to the same attack. As shown in Table 10.2, this keyspace leads to estimated attack complexities of approximately 2^{127}, 2^{191} and 2^{254} AES-λ evaluations, for $\lambda = 128, 192$ and 256, respectively.

10.2.2 Brute Force the PRG. Another way to try to recover the signing key is to attack the pseudo-random generator PRG, used in the VC.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. Suppose an attacker is attempting to recover the secret key, given just one signature. Since the same IV 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 first consider the length-doubling PRGs used in the tree-based vector commitment algorithm, VC.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[14] 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 PRG 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.

Combining these two stages, given any guess g and the first 2λ output bits of AES-CTR under g, an adversary can compare g with up to $k_0 \leq 12$ PRG candidate keys from VC.Commit, plus one more for the PRG in ConvertToVOLE. Since these steps are repeated τ times, there are a total of $\tau_0 k_0 + \tau_1 k_1 + \tau = \lambda + \tau$ values that can be tested with a guess g. Assuming each test can be done with a cheap lookup table, this attack requires around $2^\lambda/(\lambda+\tau)$ guesses, or approximately $2^{121}, 2^{184}$ and 2^{248} AES-λ evaluations for $\lambda = 128, 192$ and 256 respectively. These estimates are slightly smaller than, but still comparable to, the 2^λ cost of a brute-force attack on AES. As discussed in Section 11, this attack could be prevented by using unique IVs for each PRG.

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 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.

10.2.3 Attack Soundness of the ZK Protocol. Instead of directly recovering the signing key, an attacker may attempt to forge a signature by violating the sound-

[14] 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.

ness 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 1. 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 advantage term in the theorem corresponds to finding a collision in a random oracle with $2^{2\lambda}$ 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 second advantage term is $Q_{2,1} \cdot \binom{\tau}{2}\varepsilon_{\mathsf{v}}$, where $Q_{2,1}$ is a number of random oracle queries performed by the adversary. Since $\varepsilon_{\mathsf{v}} \leq 2^{-\lambda-B}$ and our parameters always ensure that $B \geq 2\log\tau$, the term is bounded by $Q_{2,1} \cdot 2^{-\lambda}$, so has the same cost as a generic preimage attack on SHAKE with λ bits of output. The third and fourth advantage terms of $Q_{2,2}\varepsilon_{\mathsf{zk}}$ and $Q_{2,3}2^{-\lambda-1}$ correspond to cheating in the QuickSilver zero-knowledge proof check; since $\varepsilon_{\mathsf{zk}} \leq 2^{-\lambda}$, these attacks again require close to 2^{λ} SHAKE evaluations to succeed. Note that, since SHAKE is more costly to evaluate than AES, none of these attacks are worthwhile for an attacker compared with directly inverting the one-way function in the public key to recover the secret key (which is the final advantage term in Theorem 1).

10.2.4 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.

10.3 Concrete Analysis of AES as a OWF

It is well-known that OWFs can be used as hard instances generator for key pairs in signature schemes. At the core, in FAEST this one-way function is built using the AES block cipher. To justify this design choice, we divide our discussion into two parts considering AES with and without its key expansion routine, respectively. In the latter case, the use of a public constant for the AES key removes the need to calculate the AES key expansion routine as part of the protocol, reducing the number of S-boxes (and therefore inversions) from 200 to 160 for AES-128, which leads to smaller signatures. Using AES as a OWF is very conservative as AES was designed to resist cryptanalysis in a setting giving much more freedom to an attacker.

10.3.1 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 an OWF. Hence, in our case, the OWF security of F_k could be simply reduced to the PRF security of AES [CDG+17].

However, when aiming to prove statements related to the AES algorithm using zero-knowledge proof systems working with arithmetic circuits, the behaviour of the AES S-box is problematic. While its use of the inverse function is relatively easy to handle, the fact that this function has the special case of mapping the 0 input to the 0 output complicates matters because a change in behaviour of the proof system in that special case would leak information about the input to the S-box. Previous MPC-in-the-Head work, using AES [DDOS19] with proof systems for arithmetic circuits, instead select a pair (x, k) such that no S-box, whether in KeyExpansion or in Encrypt, ever has an input value of 0.

When first proposing AES as an alternative block cipher for MPCitH constructions, Delpech de Saint Guilhem et al. estimated that imposing such conditions on the (x, k) pair would result in a space of possible values for k, for a given value of x, that is 45.7% smaller than the full set of $\{0, 1\}^{128}$ [DDOS19].

More formally, let $\mathcal{K}'$ be the subset of keys such that there is no zero as input to any S-box in the computation of $F_k(x)$ for a given x, [DDOS19] proved the following.

Lemma 7. *Let $\{F_{k,x}\}$ be the AES function family AES_k-$\lambda(x)$ with $\lambda \in \{128, 192, 256\}$ and $k \in \mathcal{K}'$ (as described above) indexed by x (i.e. x selects the concrete instance from the family of one-way function), then $\{F_{k,x}\}$ is also a family of OWF with respect to k for each input x.*

In addition, the authors of [DDOS19] quantify how many AES keys k do not have a 0 state byte for any SubBytes or SubWord operation given a fixed input x to the AES instance. Assuming independence of the S-box evaluations, then $(255/256)^s$ is the probability that an OWF with s S-box evaluations has the desired property [DDOS19]. For AES128 we have $s = 200$, while $s = 224$ for AES192 and $s = 276$ for AES256. The authors of [DDOS19] empirically validate that this bound is indeed correct. For AES256 this translates into 34% of the choices of k being accepted, while the percentage is higher for AES128 and AES192 due to the lower number of S-boxes, see also Table 10.2.

Since the OWF for $\lambda \in \{192, 256\}$ applies two parallel AES instances for the same key but different inputs, approximately 11.5% of the keys for FAEST-256s, FAEST-256f have the respective property given $x = (x_0 \| x_1) \in \{0, 1\}^{256}$, while it is higher for the other variants of the signature scheme. Therefore, the upper bound on the expected number of keys which must be sampled in FAEST.KeyGen before success is 9. [DDOS19, Section 3.2] moreover argues that the reduction in key space does not come at an additional loss in security of the signature scheme.

	AES-128	AES-192	AES-256
Number of S-boxes	200	224	276
Probability of no zeroes	45.7%	41.6%	34.0%
Bit security	126.9	190.7	254.4

Table 10.2: Probability of no zero-inputs to S-boxes, and estimated corresponding bit security

10.3.2 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 x and to the output of the permutation, i.e.,

$$F_k(x) = k + \pi(x + k)\,. \tag{10}$$

In practice we can instantiate π with a block cipher such as AES, as described in Section 2.1.3. As in the previous case, F must be a one-way function of the key: given some (x, y) such that $y = F_k(x)$, it should be difficult to find a k' such that $F_{k'}(x) = y$. More formally, F is a secure OWF if the following probability

$$\Pr[x \leftarrow \{0,1\}^\lambda, k \leftarrow \{0,1\}^\lambda, k' \leftarrow A^\pi(x, F_k(x)) : F_{k'}(x) = F_k(x)] \tag{11}$$

is negligible in λ for all polynomial-time adversaries $\mathcal{A}$, with oracle access to π.

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. Dobrauning et al. [DKR+22], showed the following result.

Theorem 6. *The single-key Even–Mansour construction (Equation (10)) is a secure one-way function, when the permutation π is an ideal random permutation.*

Proof. The attacker $\mathcal{A}$ is initialized with (x, y) and has oracle access to π. We must show that the probability in Equation (11) is negligible in λ (the key size and block size in bits).

We say that a key k is *consistent* with the pair (x, y) if $y = F_k(x)$, i.e., $y = k + \pi(x + k)$.

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

Therefore, each query X_i has the form $X_i = x + k_i$ for a distinct key k_i. In one case, the query either gives $\mathcal{A}$ the correct key, and we have $y_i + k_i = y$. In the other case, the query does not give $\mathcal{A}$ the correct key, but reveals that $k'_i = y - y_i$, a second key distinct from k_i, cannot be consistent with (x, y). First we note that k'_i is distinct from k_i since $y_i + k_i \neq y$, then $k_i \neq y - y_i$. To see that k'_i cannot be consistent if k_i is not consistent, assume that k'_i was consistent, which implies that

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

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

Now we argue that the remaining $2^\lambda - 2q$ keys are equally likely to be consistent with (x, y), given the information $\mathcal{A}$ has. Consider k^*, one of the remaining keys, not equal to k_i or k'_i. If $\mathcal{A}$ knew that k^* was not consistent with (x, y), then $\mathcal{A}$ knows $y - k^*$, and that

$$y - k^* \neq \pi(x + k^*) \tag{12}$$

But since π is a random permutation, $\pi(x+k^*)$ is uniformly selected from the $2^\lambda - 2q$ remaining possible values, so Equation (12) holds with probability $1 - 1/(2^\lambda - 2q)$ and all remaining keys are equally likely.

Therefore, after q queries $\mathcal{A}$ succeeds with probability not more than $2q/2^n$. Alternatively, $\mathcal{A}$ succeeds after $\Omega(2^\lambda)$ queries. □

When compared to the upper bound given by the complexity of a brute-force attack (where $\mathcal{A}$ succeeds with probability $q/2^\lambda$), this lower bound is off by a factor of two. As described in the proof, each query to π can be used to rule out two keys, improving the brute force attack, to succeed with probability $2q/2^\lambda$. Therefore the lower bound is tight, since it matches the complexity of the attack.

Effect to Zero-inputs to S-boxes. Similar to the case of using AES with a key expansion routine in Section 10.3.1 also here zero-inputs to S-boxes are relevant, however there are fewer (e.g. 160 instead of 200 for AES-128) leading to slightly better numbers, see Table 10.3.

	AES-128	Rijndael-192	Rijndael-256
Number of S-boxes	160	288	448
Probability of no zeroes	53.5%	32.4%	17.3%
Bit security	127.1	190.4	253.5

Table 10.3: Probability of no zero-inputs to S-boxes, and estimated corresponding bit security

10.3.3 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 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 7 (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*

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

where F_k 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.

There is an easy reduction from PRP security to OWF security of single-key Even Mansour, which yields the following

Corollary 1. *Let $\mathcal{A}$ be an adversary making q quantum queries to its oracle. Let π be a uniformly random n-bit permutation, and $x, k \leftarrow \{0,1\}^\lambda$. Then*

$$\Pr\left[\mathcal{A}^\pi(x, k+\pi(k+x)) = k\right] \leq 10\sqrt{3}\cdot 2^{-\lambda/2}\left(q+2+\sqrt{3(q+2)}\right)+\frac{1}{2^\lambda-1},$$

where it is understood that $\mathcal{A}$ has forward and inverse access to its oracle.

Proof. We construct an adversary $\mathcal{B}$ for the PRP security game for single-key Even-Mansour. $\mathcal{B}$ samples random $x, k \leftarrow \{0,1\}^\lambda$ and queries its first oracle on x to obtain y. Then it runs $k' \leftarrow \mathcal{A}^\pi(x, y)$, instantiating the oracle π using its second oracle. Then it samples $x_0 \leftarrow \{0,1\}^\lambda$ and $x_1 \leftarrow \{0,1\}^\lambda \setminus \{x_0\}$, computes $y_i = k' + \pi(k' + x_i)$ and queries its first oracle on x_i to obtain y'_i, for $i = 0, 1$. Finally it outputs 1 if $y_i = y'_i$ for $i = 0, 1$, and 0 else.

First, we observe that $\mathcal{B}$ makes $q + 2$ queries to its first oracle and 3 query to its second oracle. We thus have

$$\left|\Pr\left[\mathcal{B}^{F_k,\pi}(1^\lambda) = 1\right] - \Pr\left[\mathcal{B}^{\sigma,\pi}(1^\lambda) = 1\right]\right| \leq 10 \cdot 2^{-\lambda/2}\left(3\sqrt{q+2} + \sqrt{3}(q+2)\right). \tag{13}$$

If $\mathcal{B}$ has oracles σ and π, $\mathcal{B}$ runs $\mathcal{A}$ on input $(x, \sigma(x))$, and $y'_i = \sigma(x_i)$. We bound

$$\begin{aligned}\Pr\left[\mathcal{B}^{\sigma,\pi}(1^\lambda) = 1\right] &= \Pr\left[\sigma(x_i) = \pi(k' + x_i) + k', i = 0, 1\right] \\ &\leq \Pr\left[\exists \hat{k} : \sigma(x_i) = \pi(\hat{k} + x_i) + \hat{k}, i = 0, 1\right] \\ &\leq \frac{1}{2^\lambda - 1}.\end{aligned}$$

The last inequality follows from the observation that for any fixed $\hat{k}$,

$$\Pr\left[\sigma(x_i) = \pi(\hat{k} + x_i) + \hat{k}, i = 0, 1\right] = \frac{1}{2^\lambda(2^\lambda - 1)}, \tag{14}$$

together with a union bound over $\hat{k}$.

Now consider the case where $\mathcal{B}$ has oracles F_k and π, so $\mathcal{A}$ is run on input $(x, k + \pi(k + x))$. If $\mathcal{A}$ outputs k then $y_i = y'_i$, therefore we have

$$\Pr\left[\mathcal{A}^\pi(x, k + \pi(k + x)) = k\right] \leq \Pr\left[\mathcal{B}^{F_k,\pi}(1^\lambda) = 1\right].$$

Using Equations (13) and (14) we thus obtain

$$\begin{aligned}&\Pr\left[\mathcal{A}^\pi(x, k + \pi(k + x)) = k\right] \\ &\leq \Pr\left[\mathcal{B}^{F_k,\pi}(1^n) = 1\right] \\ &\leq \Pr\left[\mathcal{B}^{\sigma,\pi}(1^\lambda) = 1\right] + 10 \cdot 2^{-\lambda/2}\left(3\sqrt{q+2} + \sqrt{3}(q+2)\right) \\ &\leq \frac{1}{2^\lambda - 1} + 10 \cdot 2^{-\lambda/2}\left(3\sqrt{q+2} + \sqrt{3}(q+2)\right).\end{aligned}$$

□

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

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

When cryptanalysts can not 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].

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 pseudo-randomness 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 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, 5038 bytes at category 1. 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, via a reduction to the security of the underlying primitives. Although the proof is not fully tight, due to a multiplicative factor in the number of signing queries, this provides strong evidence of security for FAEST. We also give a natural conjecture that would lead to a proof of security in the QROM.

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 could make it 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, Heidelberg, 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.

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 *CRYPTO*. Springer, 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.

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, 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, Heidelberg, 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, 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, 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, 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, Heidelberg.

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, Heidelberg, 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, Heidelberg, 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, Heidelberg, December 2013.

CCH+19. Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, Ron D. Rothblum, and Daniel Wichs. Fiat-Shamir: from practice to theory. In Moses Charikar and Edith Cohen, editors, *51st ACM STOC*, pages 1082–1090. ACM Press, June 2019.

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, Heidelberg, October 2021.

CMS19. Alessandro Chiesa, Peter Manohar, and Nicholas Spooner. Succinct arguments in the quantum random oracle model. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 1–29. Springer, Heidelberg, December 2019.

CW79. Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

DDOS19. Cyprien Delpech de Saint Guilhem, Lauren De Meyer, Emmanuela Orsini, and Nigel P. Smart. BBQ: Using AES in picnic signatures. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019*, volume 11959 of *LNCS*, pages 669–692. Springer, Heidelberg, August 2019.

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, Heidelberg, May 2013.

DFM20. Jelle Don, Serge Fehr, and Christian Majenz. The measure-and-reprogram technique 2.0: Multi-round fiat-shamir and more. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 602–631. Springer, Heidelberg, August 2020.

DFMS22. Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Online-extractability in the quantum random-oracle model. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 677–706. Springer, Heidelberg, May / June 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, 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, Heidelberg, May 2019.

DR98. Joan Daemen and Vincent Rijmen. The block cipher rijndael. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*, volume 1820 of *Lecture Notes in Computer Science*, pages 277–284. Springer, 1998.

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, 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, Heidelberg, 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, Heidelberg, April / May 2017.

IKNP03. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

IKOS07. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

KPTZ13. Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013.

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.

MS04. Alfred Menezes and Nigel Smart. Security of signature schemes in a multi-user setting. *Designs, Codes and Cryptography*, 33(3):261–274, 2004.

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, Heidelberg, August 2022.

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, 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.

Unr17. Dominique Unruh. Post-quantum security of Fiat-Shamir. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 65–95. Springer, Heidelberg, December 2017.

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, Heidelberg, August 2019.

A 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}
```

$$z^{250} + z^{247} + z^{246} + z^{243} + z^{240} + z^{239} + z^{237} + z^{235} + z^{231} + z^{230} + z^{227} + z^{226} + z^{225} + z^{224} + z^{221} + z^{215} + z^{214} + z^{211} + z^{208} + z^{206} + z^{204} + z^{203} + z^{197} + z^{196} + z^{193} + z^{192} + z^{191} + z^{190} + z^{188} + z^{187} + z^{186} + z^{185} + z^{184} + z^{181} + z^{177} + z^{175} + z^{172} + z^{171} + z^{166} + z^{162} + z^{160} + z^{159} + z^{158} + z^{157} + z^{156} + z^{155} + z^{151} + z^{150} + z^{149} + z^{148} + z^{144} + z^{143} + z^{142} + z^{141} + z^{136} + z^{134} + z^{133} + z^{131} + z^{129} + z^{127} + z^{125} + z^{124} + z^{123} + z^{122} + z^{121} + z^{119} + z^{118} + z^{116} + z^{114} + z^{113} + z^{111} + z^{107} + z^{106} + z^{104} + z^{101} + z^{100} + z^{99} + z^{95} + z^{93} + z^{86} + z^{82} + z^{81} + z^{80} + z^{78} + z^{75} + z^{74} + z^{73} + z^{70} + z^{69} + z^{66} + z^{65} + z^{64} + z^{63} + z^{60} + z^{58} + z^{57} + z^{55} + z^{52} + z^{50} + z^{49} + z^{48} + z^{47} + z^{43} + z^{38} + z^{33} + z^{27} + z^{25} + z^{24} + z^{23} + z^{22} + z^{20} + z^{19} + z^{18} + z^{17} + z^{15} + z^{14} + z^{13} + z^{12} + z^{11} + z^{10} + z^{9} + z^{7} + z^{6} + z^{5} + z^{2} + z + 1$$