Nested MuSig2

Nadav Kohen

Chaincode Labs

nadav@chaincode.com

Abstract. Bitcoin Improvement Proposal 327 specifies a variant of the MuSig2 multi-signature protocol that is becoming widely adopted in Bitcoin applications. This protocol enables multiple participants to collaboratively compute (BIP 340) Schnorr signatures for a single aggregate public key efficiently, while preventing external parties from distinguishing whether multiple signers were involved. It has been widely proposed that it *should* be secure to allow MuSig2 participant keys to themselves be "nested" MuSig2-aggregated keys. No security argument has previously been presented for this practice, though various applications have been proposed that assume the security of such an operation.
In this work, we propose NestedMuSig2, a recursive variant of MuSig2 that enables a tree of nested cosigners to privately generate aggregate Schnorr signatures while maintaining all of the efficiency and security benefits of MuSig2, including non-interactive public key aggregation. Nested signers in this scheme cannot distinguish between cosigners that are using further nesting and those that are not. In particular, this means that NestedMuSig2 is compatible with all existing protocols that use MuSig2. We reduce the security of NestedMuSig2 to the AOMDL assumption in the random oracle model. Similarly, we reduce the security of a more efficient and compact variant to the AOMDL assumption in the random oracle model used in conjunction with the algebraic group model.

1 Introduction

1.1 Background

Multi-signature schemes with key aggregation allow a group of signing keys to be non-interactively aggregated so that participants with knowledge of the signing keys can then run an interactive protocol to produce a single signature for the aggregate public key. Similarly, threshold signature schemes allow a group of signers to generate an aggregate key for which any sufficiently large subset of signers can collaboratively produce signatures. In recent years, various multi-signature and threshold signature schemes have been proposed whose aggregate signatures are indistinguishable from standard single-signer signatures, making these schemes compatible with applications that already support signatures, such as Bitcoin.

In Bitcoin and other cryptocurrencies, it is often desirable to attach non-trivial constraints to coins that must be satisfied before they can be spent. Such constraints are often referred to as "smart contracts." However, using expressive scripts to enforce such policies is expensive for the network, which must redundantly store and execute these scripts when coins are spent. This cost is then transferred to the user in the form of transaction and/or "gas" fees that must be paid in order to include contracts on-chain. Furthermore, executing contract logic on-chain requires revealing script conditions to the public, which is undesirable because it affords little privacy. Multi-signature and threshold signature spending conditions have long been used as building blocks to construct more complex contracts, especially those that the underlying scripting functionality does not natively support. Even in contexts where more complex smart-contracting primitives are available, multi-signature and threshold signature approaches offer improved privacy and scalability compared to enforcing all contract logic on-chain.

For instance, in multi-party contracts, it is common to construct a tree of unsigned or partially signed transactions whose leaves enumerate all possible contract executions. Each non-leaf transaction in such a tree requires a multi-signature from the set of all involved parties to ensure that only the transactions belonging to the collaboratively constructed tree may be used to spend funds locked in the contract. Examples of protocols that use this design pattern include the Lightning Network

[PD16] (payment channels), Ark [AAC+25] (transaction batching), Discreet Log Contracts [GKK22] (oracle contracts), and BitVM [Lin25] (optimistic verification of arbitrary program executions).

Historically, Bitcoin applications such as these have used the opcode OP_CHECKMULTISIG to enforce multi-signature and threshold conditions, which amounts to the trivial threshold signature where each party concatenates a signature, and no aggregation or compression is employed. With the activation of the Taproot Soft Fork in 2021, the use of BIP 340 [WNR20] Schnorr signatures became available to Bitcoin contracts, as did the efficient Schnorr multi-signature protocol MuSig2 [NRS21], codified in BIP 327 [NRJ22].

The MuSig2 Scheme. MuSig2 is a multi-signature protocol that outputs aggregate signatures that are indistinguishable from single-signer Schnorr signatures (in particular, they have the same verification algorithm). This means that protocols that previously used OP_CHECKMULTISIG to enforce multi-signatures can now use MuSig2 to both compress their on-chain footprint and to preserve privacy by becoming indistinguishable from normal single-signer activity. Protocols such as the Lightning Network have begun migrating to the use of MuSig2-based pre-signed transaction trees [Osu22].

There are applications using or upgrading to use MuSig2 that have expressed interest in optionally replacing some participant signer keys with multi-signature or threshold aggregate keys, recursively. The most common reason for this desire is that it would enable threshold key management across all protocols in which parties participate in MuSig2 aggregation. Threshold key management is essential, especially for securing large amounts of Bitcoin, because it allows users to replace single keys that represent ownership of funds with multiple keys, so that multiple keys must be lost or stolen before funds are lost. Nesting a multi-signature within MuSig2 does not directly accomplish this goal, as most key management protocols require threshold signatures, but it is possible to use replicated secret sharing (RSS) in conjunction with MuSig2 to achieve a threshold signature scheme [Seu24]. The Iceberg scheme [GKP26] instantiates such an RSS-based approach to embedding threshold aggregate keys within MuSig2, and its security is proven by reducing directly to the security results proven in this paper.

One more direct use case for nesting multi-signatures is that some off-chain protocols require linking off-chain multi-party identities to on-chain multi-party identities (such as during channel announcement on the Lightning network [Mou23]). Requiring such linking in an off-chain protocol has at least three benefits: First, the signature acts as a proof of reserves tying actual on-chain funds to off-chain actions, second, this link allows other interested parties to monitor the on-chain funds, which may be moved by satisfying some spending conditions that involve revealing information that is important to the state of other off-chain participants (such as when a Lightning channel force-closes with active hash time locked contracts), third, the link to on-chain funds acts as a DoS-prevention mechanism because requiring messages to include fresh on-chain addresses forces spammers to pay on-chain fees. Linking on-chain keys with off-chain keys can be accomplished by having the off-chain keys sign a message alongside the on-chain keys in a standard multi-signature, but this commits the off-chain protocol to only working with certain kinds of expected on-chain address types that can be deterministically reconstructed from participant keys (such as a 2-of-2 multi-signature address from two public keys). If instead, the on-chain key itself (which is normally a multi-signature, but could also be a threshold signature or something else) participates as a MuSig2 signer alongside the off-chain keys (or their aggregation), then the off-chain network does not need to verify any details about the on-chain address allowing for more extensibility and user flexibility, as well as optimized verification involving fewer keys.

Another use case for nested multi-signatures is when a multi-party contract is constructed using a tree of partially signed transactions as described above, but where one or more parties can be optionally composed of multiple parties. For instance, this is the case if multiple parties wish to execute a multi-party contract within an instance of Ark. In this case, one party is the operator and the other party is the user, both of which may be made up of multiple individuals. Furthermore, it is not desirable for the operator to need to know that the user is actually multiple users, which can be accomplished by aggregating user keys and having users execute a nested multi-signature scheme.

1.2 Our Contribution

We propose a novel two-round nested multi-signature scheme, NestedMuSig2, which is a variant of MuSig2 that, rather than generating Schnorr signatures, recursively generates NestedMuSig2 partial signatures for one less level of nesting. When the nesting depth is set to one, NestedMuSig2 is exactly the same as MuSig2. To the best of our knowledge, this is the first academic work on any kind of nested signature scheme.

NestedMuSig2 not only generates outputs that match MuSig2's interface, but a collection of NestedMuSig2 participants (at nesting depth two) produce aggregate messages that are indistinguishable from those of a single MuSig2 participant. In particular, NestedMuSig2 results in a globally two-round protocol, where the first round can be pre-computed before the message to be signed is determined, and the aggregate result is a first-round MuSig2 message. Just as with MuSig2, this means that NestedMuSig2 also effectively enables non-interactive signing in this more complicated setting.

All of this is accomplished without altering the MuSig2 non-interactive key aggregation algorithm, which we simply use recursively at every level of nesting.

We generalize the security model used for MuSig2, of existential unforgeability under chosen message attack for concurrent signing sessions, such that the adversary has control of all but one key. In the nested setting, this allows the adversary to construct arbitrary signing trees in which it controls all but one leaf, and that leaf's key may be used in concurrent signing sessions.

We prove the security of NestedMuSig2 under the same assumptions that MuSig2 is proved secure. Namely, the algebraic one-more discrete logarithm (AOMDL) problem [NRS21], which is a weaker and falsifiable variant of the one-more discrete logarithm problem. As was done in the MuSig2 paper, we provide two independent security proofs. One security proof in the random oracle model (ROM) applies to NestedMuSig2 using $\nu = 2^{D+1}$ nonces, where D is a bound on the depth of nesting (e.g., $D = 1$ for MuSig2). The second proof additionally assumes the algebraic group model (AGM) [FKL18] and applies to NestedMuSig2 using only $\nu = 2$ nonces, which is likely to be the standard in practice.

In Appendix A and Appendix B we discuss two optimized variants of NestedMuSig2, which we call NestedMuSig2* and MerkleNestedMuSig2*.

2 Technical Overview

2.1 Constructing a Multi-Party MuSig2 Protocol

Our goal is to construct an efficient multi-party computation that enables a set of parties to emulate a single MuSig2 participant, where each party's cooperation is necessary. Ideally, this protocol will have non-interactive key generation and will not increase the global round complexity when composed with MuSig2. We begin by outlining the functionality we need to simulate.

A MuSig2 participant generates two messages, one for each round of MuSig2. Participant i's first-round message is a uniformly random tuple, $(R_{i,1}, \ldots, R_{i,\nu})$, of group elements committing to their contribution to the aggregate nonce. Next, that participant's second-round message, (R, s_i), depends on an input tuple $(R_1, \ldots, R_\nu)$, which is the coordinate-wise product of all parties' first-round values, the input message, m, and the multi-set of cosigner public keys, $\{pk_j\}_{j \neq i}$. The second message has $R := \prod_{j=1}^{\nu} R_j^{b^{j-1}}$, where $\widetilde{X}$ is participant i's public key, X_i, aggregated with the other input public keys and $b := \mathsf{H}_{\mathsf{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$. Furthermore, s_i is uniquely determined by the first round message and input to the second round, more specifically,

$$g^{s_i} = X_i^{a_i c} \prod_{j=1}^{\nu} R_{i,j}^{b^{j-1}},$$

where $c := \mathsf{H}_{\mathsf{sig}}(\widetilde{X}, R, m)$ and a_i is participant i's key aggregation coefficient from the KeyAgg algorithm. In other words, $s_i := (a_i c) x_i + \sum_{j=1}^{\nu} r_{i,j} b^{j-1}$, where x_i is participant i's private key and $r_{i,j}$ is the discrete log of $R_{i,j}$ from the first round. The value s_i is often referred to as a "partial signature" because it yields a valid signature (for the aggregate public key) when summed with all other partial signatures.

Partial signatures look very similar to standard Schnorr signatures, which have the form $s := cx + r$, but where c is multiplied by a_i, and the randomness is a more complicated (yet still uniformly random) value. This suggests that we should be able to use a protocol very similar to MuSig2 to achieve our goal. The only real complication is that we must not only generate a (partial) signature but also a proper first-round message.

As a naïve first attempt, we might try to have each nested party generate their own uniformly random first-round message, $(R_{j,1,1}, \ldots, R_{j,\nu,1})$, and then multiply these messages together coordinate-wise to compute the aggregate first-round message, $(R_{i,1,0}, \ldots, R_{i,\nu,0})$, where $R_{i,k,0} := \prod_{j=1}^{\nu} R_{j,k,1}$. Nested participant j can then compute

$$s_{j,1} := (a_{j,1}a_{i,0}c)x_{j,1} + \sum_{k=1}^{\nu} r_{j,k,1}b^{k-1},$$

where $x_{j,1}$ is nested participant j's private key, $a_{j,1}$ is participant j's key aggregation coefficient at the nested level, and $a_{i,0}$ is the nested group's collective key aggregation coefficient at the top level. As a result, the sum of all nested participant's partial signatures is

$$s_{i,0} = \sum_{j=1}^{n} s_{j,1} = a_{i,0}c\sum_{j=1}^{n} a_{j,1}x_{j,1} + \sum_{k=1}^{\nu}\left(\sum_{j=1}^{n} r_{j,k,1}\right)b^{k-1} = (a_{i,0}c)x_{i,0} + \sum_{k=1}^{\nu} r_{i,k,0}b^{k-1}.$$

This works out syntactically; however, this protocol is insecure. A variant of the concrete attack on an insecure original version of MuSig from [DEF+19] applies here. It is essentially the same attack as applies to MuSig2 (as described in Section 2.1 of [NRS21]) if one uses $b = 1$ instead of a hash. Indeed, we have not introduced a new nonce hash for the lower-level variant of MuSig2, and adding one will fix this problem. We leave the first round unchanged, but alter the first-round aggregation algorithm to first compute $R'_{i,k,0} := \prod_{j=1}^{\nu} R_{j,k,1}$ as before, and then set $R_{i,k,0} := (R'_{i,k,0})^{b_1^{k-1}}$, where $b_1 := \mathsf{H}_{\text{non}}(X_{i,0}, (R'_{i,1,0}, \ldots, R'_{i,\nu,0}))$. Note that we have not included the message, m, in the lower-level nonce hash, as this would require knowing the message to be signed during the first round, which is not required in MuSig2. We will show that this omission does not affect security so long as the top-level nonce hash still includes the message.

During the second round, nested participant j is given all inputs to both nonce hash functions so that they can compute b_1 and b, and then they compute

$$s_{j,1} := (a_{j,1}a_{i,0}c)x_{j,1} + \sum_{k=1}^{\nu} r_{j,k,1}(b_1 b)^{k-1}.$$

Adding together all of these partial signatures yields the aggregate partial signature

$$s_{i,0} = a_{i,0}c\sum_{j=1}^{n} a_{j,1}x_{j,1} + \sum_{k=1}^{\nu}\left(\left(\sum_{j=1}^{n} r_{j,k,1}\right)b_1^{k-1}\right)b^{k-1} = (a_{i,0}c)x_{i,0} + \sum_{k=1}^{\nu} r_{i,k,0}b^{k-1}.$$

We call this scheme NestedMuSig2.

2.2 Generalizing to Higher Depth

The singly-nested scheme we have just described can be easily generalized to deeper nesting. Let participant j at nesting level ℓ be a single signer that is not using further nesting and is located beneath participant i at nesting level $\ell - 1$ in a cosigner tree (see Figure 1 for an example cosigner tree). We specify their first-round message to be the uniformly random tuple $(R_{j,1,\ell}, \ldots, R_{j,\nu,\ell})$, which aggregate to $R'_{i,k,\ell-1} := \prod_{j=1}^{n} R_{j,k,\ell}$ "internally" and $R_{i,k,\ell-1} := (R'_{i,k,\ell-1})^{b_\ell^{k-1}}$ "externally," where $b_\ell := \mathsf{H}_{\text{non}}(X_{i,\ell-1}, (R'_{i,1,\ell-1}, \ldots, R'_{i,\nu,\ell-1}))$. We specify their second-round message given the message to be signed, the internal aggregate first-round message for all $\ell + 1$ levels of nesting they are a part of, and all public keys at each of those levels to be

$$s_{j,\ell} := \check{c}x_{j,\ell} + \sum_{k=1}^{\nu} r_{j,k,\ell}\check{b}^{k-1},$$

where $\check{c}$ is the product of the signature hash, c, with all relevant key aggregation coefficients (there are $\ell + 1$ in total), and $\check{b}$ is the product of all of the nonce hashes at each level. It is straightforward to see that adding all the partial signatures at a fixed depth correctly produces the expected aggregate partial signature for the next level up. We call this general scheme NestedMuSig2, which does not conflict with the naming in the previous section since this scheme is equal to the scheme above when the nesting depth is equal to 1. Furthermore, when the nesting depth is set to 0, then NestedMuSig2 becomes equal to MuSig2.

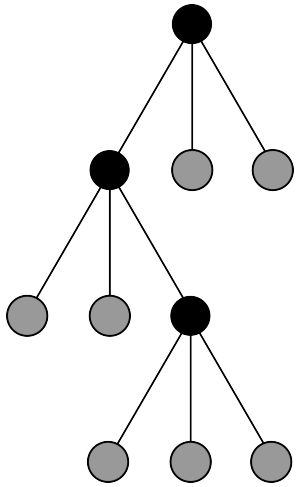

Fig. 1. A visualization of a cosigner tree, with aggregators colored black and nested signers colored grey. This is the point of view from a single bottom-most node. It is possible that any of the other grey nodes are themselves aggregators for further nesting, but this does not concern us (and cannot be distinguished by us).

2.3 Proving Security

Both results proving the security of NestedMuSig2, Theorem 1 and Theorem 2, generalize the corresponding theorems for MuSig2 [NRS21]. In both cases, we simulate an honest signer, whose signing key we do not know, by using the interactive AOMDL assumption's DLOG oracle. The AOMDL assumption [NRS21] requires us to solve more discrete log challenges than we use this oracle for, however, so we simulate the first signing round by using the AOMDL challenge oracle to embed challenges into the signer's nonces. Subsequently, if an adversary successfully generates an aggregate forgery, we extract the honest signer's private key from this adversary. Lastly, we use this private key and knowledge of every DLOG query and response to solve for all of the challenges that were embedded in the first rounds of signing. The primary obstacle lies in extracting the honest signer's private key from an adversary capable of producing aggregate forgeries.

In the random oracle model (ROM), this extraction is achieved by first forking the signature challenge hash in the standard way to extract the aggregate private key, and subsequently forking the key aggregation coefficient hash once at each level of depth. Each H_{agg} forking extracts a key that is one level deeper in the forgery cosigner tree. In the end, this results in the extraction of the honest signer's private key at a leaf of this tree. However, the amount of forking required by this proof method greatly reduces the security bounds, and also forces us to introduce a large number of nonces per signing session so that enough AOMDL challenges are created in order to overcome the number of times we may be forced to query the DLOG_g oracle for a single signing session across all branches of execution. For example, MuSig2 requires $\nu = 4$ nonces while a singly-nested instantiation of NestedMuSig2 requires $\nu = 8$ nonces of all participants (including non-nested cosigners at the top level).

Meanwhile, if we assume not only the ROM but also the algebraic group model (AGM) [FKL18], then it becomes possible to avoid the proof-artifacts of forking by extracting the honest signer's private key directly from the forgery. This results in a security proof for NestedMuSig2 where only $\nu = 2$ nonces are required for any nesting depth.

3 Preliminaries

3.1 Notation and Definitions

NOTATION. The security parameter is denoted λ. Given a sampleable set, S, we denote $s \leftarrow\!\$\, S$ the operation of sampling an element of S uniformly at random and assigning it to s. If A is a randomized algorithm, we let $y := \mathsf{A}(x_1, \ldots; \rho)$ denote the operation of running A on inputs $x_1, \ldots$ and random coins ρ and assigning its output to y, and $y \leftarrow \mathsf{A}(x_1, \ldots)$ when coins ρ are chosen uniformly at random. Given a game, $\mathsf{Game}_{\mathcal{A}}$, parameterized by an adversary $\mathcal{A}$, we define the advantage of $\mathcal{A}$ in $\mathsf{Game}_{\mathcal{A}}$ as $\mathsf{Adv}^{\mathsf{Game}}_{\mathcal{A}}(\lambda) := \Pr[\mathsf{Game}_{\mathcal{A}}(\lambda) = \mathtt{true}]$.

GROUP DESCRIPTION. A *group description* is a triple $(\mathbb{G}, p, g)$ where $\mathbb{G}$ is a cyclic group of order p and g is a generator of $\mathbb{G}$. A (prime-order) *group generation algorithm* is an algorithm GrGen which on input 1^λ returns a group description $(\mathbb{G}, p, g)$ where p is a λ-bit prime. The group $\mathbb{G}$ is denoted multiplicatively, and we conflate group elements and their encoding when given as input to hash functions.

AOMDL PROBLEM. The *algebraic* one-more discrete log problem (AOMDL) was introduced in [NRS21] as a weaker and falsifiable variant of the one-more discrete log problem (OMDL) [BNP+03]. Given a group description, $(\mathbb{G}, p, g)$, an adversary $\mathcal{A}$ wins the AOMDL game if it outputs the discrete logs, $(x_1, \ldots, x_{q+1})$, (with base g) of $q+1$ challenges, $(X_1, \ldots, X_{q+1})$, obtained from the CH oracle after having queried the DLOG_g oracle at most q times. However, every query made to the DLOG_g oracle must provide an algebraic representation of the input as a linear combination of g and all previous challenges.

Game $\mathsf{AOMDL}^{\mathcal{A}}_{\mathsf{GrGen}}(\lambda)$	Oracle $\mathrm{CH}()$	Oracle $\mathrm{DLOG}_g(X, (\alpha, (\beta_i)_{1 \le i \le c}))$
$(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$	$c := c + 1$	$/\!/\ X = g^\alpha \prod_{i=1}^{c} X_i^{\beta_i}$ for $X_i = g^{x_i}$
$c := 0\,;\ q := 0$	$x_c \leftarrow\!\$\, \mathbb{Z}_p$	$q := q + 1$
$\vec{y} \leftarrow \mathcal{A}^{\mathrm{CH}, \mathrm{DLOG}_g}(\mathbb{G}, p, g)$	$X := g^{x_c}$	**return** $\alpha + \sum_{i=1}^{c} \beta_i x_i$
$\vec{x} := (x_1, \ldots, x_c)$	**return** X	
return $(\vec{y} = \vec{x} \ \wedge\ q < c)$		

Fig. 2. The AOMDL problem.

Definition 1. *Let* GrGen *be a group generation algorithm, and let the game* $\mathsf{AOMDL}^{\mathcal{A}}_{\mathsf{GrGen}}$ *be defined as in Figure 2. The algebraic one-more discrete logarithm (AOMDL) problem is hard for* GrGen *if for any p.p.t. algorithm* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{A},\mathsf{GrGen}}(\lambda) := \Pr\left[\mathsf{AOMDL}^{\mathcal{A}}_{\mathsf{GrGen}}(\lambda) = \mathtt{true}\right] = \mathsf{negl}(\lambda).$$

3.2 Syntax and Security Definition of Nested Multi-Signature Schemes

We make the same simplifying assumptions that are made in [NRS21]. We restrict syntax and security to two-round signing algorithms, where first-round algorithms receive no inputs other than the global setup parameters.

SYNTAX. We define a two-round nested multi-signature scheme with key aggregation, Σ, to be a tuple of algorithms $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{KeyAgg}, \mathsf{Sign}, \mathsf{SignAgg}, \mathsf{SignAggExt}, \mathsf{Sign}', \mathsf{SignAgg}', \mathsf{Ver})$. Some schemes may also have a deterministic algorithm, Sign'', to post-process the final output of $\mathsf{SignAgg}'$, but no such post-processing is needed in NestedMuSig2.

The algorithm Setup takes as input the security parameter and generates global parameters, which are implicitly given as input to all other algorithms. The algorithm KeyGen is a randomized algorithm that takes no input and outputs a key pair. The algorithm KeyAgg is deterministic and aggregates its input multiset of public keys into a single public key.

Each signer, i, at nesting depth, d, executes the randomized algorithm Sign with no input to receive their first-round outputs, $out_{i,d}$, and some secret state, $state_{i,d}$. The aggregator for this depth level then runs the deterministic algorithm SignAgg on these outputs to compute the (internal) aggregate first-round message out_d to be broadcast to all signers at this level of depth and their subsigners (should they exist). The aggregator also runs the deterministic algorithm SignAggExt on input out_d to compute the external aggregate first-round message $out_{i^*,d-1}$, where i^* is the index of the aggregator as a signer in the execution of NestedMuSig2 (or MuSig2) at nesting level $d-1$. Once all cosigner keys and (internal) aggregate first-round messages are known to a signer, i, at depth d, then they can run Sign'. This algorithm takes as input the first-round secret state, $state_{i,d}$, all (internal) aggregate first-round messages, $(out^{\ell})_{d \leq \ell \leq 0}$, the secret key, $sk_{i,d}$, of signer i, a message, m, to sign, and a tree of all cosigners, $\{pk_{i,\ell}\}_{2 \leq i \leq n_\ell}^{d \leq \ell \leq 0}$, and returns this signer's second-round state, $state'_{i,d}$, and second-round output, $out'_{i,d}$. Upon receiving each of these second-round outputs and states, the aggregator can run the deterministic algorithm $\mathsf{SignAgg}'$ to compute $(state'_d, out'_d)$, which is the aggregate partial signature, $(state'_{i^*,d-1}, out'_{i^*,d-1})$, unless $d = 0$, in which case $\sigma = (state'_0, out'_0)$ is the top-level aggregate Schnorr signature.

While it is possible for sibling nodes in a cosigner tree (see Figure 1) to generate a partial signature by broadcasting messages to each other (although this leads to a sub-optimal communication complexity), the resulting aggregate messages for each round must be communicated to the next level above them in the tree by a single party, making the use of an aggregator node essentially unavoidable. However, aggregator nodes are not trusted entities in our security model, and so any signer or other entity can act as an aggregator node.

For instance, if Alice (signer 1 at depth 1 with key $pk_{1,1}$) and Bob (signer 2 at depth 1 with key $pk_{2,1}$) are collaborating under aggregator Abby (acting as signer 1 at depth 0 with key $pk_{1,0} := \mathsf{KeyAgg}(pk_{1,1}, pk_{2,1})$) and Abby is collaborating with Carol (signer 2 at depth 0 with key $pk_{2,0}$) under aggregator Alberic (a top level-aggregator node with key $\widetilde{pk} := \mathsf{KeyAgg}(pk_{1,0}, pk_{2,0})$), then a full signing run proceeds as follows:

$(out_{1,1}, state_{1,1}) \leftarrow \mathsf{Sign}()$ // Alice
$(out_{2,1}, state_{2,1}) \leftarrow \mathsf{Sign}()$ // Bob
$out^1 := \mathsf{SignAgg}(out_{1,1}, out_{2,1})$ // Abby
$out_{1,0} := \mathsf{SignAggExt}(out^1, pk_{1,0})$ // Abby
$(out_{2,0}, state_{2,0}) \leftarrow \mathsf{Sign}()$ // Carol
$out^0 := \mathsf{SignAgg}(out_{1,0}, out_{2,0})$ // Alberic
$(out'_{1,1}, state'_{1,1}) \leftarrow \mathsf{Sign}'(state_{1,1}, (out^1, out^0), sk_{1,1}, m, \{pk_{2,1}, pk_{2,0}\})$ // Alice
$(out'_{2,1}, state'_{2,1}) \leftarrow \mathsf{Sign}'(state_{2,1}, (out^1, out^0), sk_{2,1}, m, \{pk_{1,1}, pk_{2,0}\})$ // Bob
$(out'_{1,0}, state'_{1,0}) := \mathsf{SignAgg}'((out'_{1,1}, out'_{2,1}), state'_{1,1})$ // Abby
$(out'_{2,0}, state'_{2,0}) \leftarrow \mathsf{Sign}'(state_{2,0}, (out^0), sk_{2,0}, m, \{pk_{1,0}\})$ // Carol
$\sigma := \mathsf{SignAgg}'((out'_{1,0}, out'_{2,0}), state'_{1,0})$ // Alberic

Notice that Carol could actually be an aggregator node for nested signers beneath her, but Alice and Bob need not be made aware if this is the case, and cannot distinguish whether or not such nesting is occurring.

The deterministic verification algorithm Ver takes an aggregate public key, $\widetilde{pk}$, a message, m, and a signature, σ, as input and returns `true` if the signature is valid for $\widetilde{pk}$ and m and `false` otherwise.

There are four differences between a two-round *nested* multi-signature scheme with key aggregation compared to the syntax of the non-nested version (e.g., as in Section 3.2 of [NRS21]). First, there is a new algorithm, $\mathsf{SignAggExt}$, which produces a first-round aggregate output. Second, the algorithm Sign' is made aware of all higher levels it is nested beneath by requiring the extra input of the higher-level $\mathsf{SignAgg}$ outputs and public keys. In particular, Sign' is implicitly parameterized by the nesting depth where it is used. Third, the algorithm $\mathsf{SignAgg}'$ does not produce Schnorr signatures that can be tested with Ver, but rather produces second-round aggregate outputs, that is, aggregate partial signatures. Lastly, we now use KeyAgg recursively; however, we do not change its syntax because we do not require the nesting depth to be added as input, but other nested multi-signature schemes may wish to add the depth level to the KeyAgg algorithm's input.

Correctness requires that for every λ, every message m, every tree $CTree$, and every $j \in CTree$,

$$\Pr\left[\mathsf{CORRECT}_{\Sigma,m,CTree,j}(\lambda) = \mathtt{true}\right] = 1,$$

where $\mathsf{CORRECT}_{\Sigma,m,CTree,j}$ is defined in Figure 3.

Privacy. We define a two-round nested multi-signature scheme, Σ, to be private if it is not possible to distinguish between honestly generated aggregate outputs and individual outputs at one level higher. More precisely, Σ is private if for any p.p.t. adversary $\mathcal{A}$, any number of nested signers n, every message m, every tree of cosigners $\{pk_{i,d}\}_{2\le i\le n_d}^{0\le d<\Lambda}$ (with $\Lambda \ge 1$), every index $x \in \{1,\ldots,n\}$, and every p.p.t. algorithm $OutGen$ that outputs some Λ-tuple in the range of $\mathsf{SignAgg}$,

$$\left|\Pr\left[\mathsf{Private}^{\mathcal{A},OutGen}_{n,m,\{pk_{i,d}\}_{2\le i\le n_d}^{0\le d<\Lambda},x}(\lambda) = \mathtt{true}\right] - \frac{1}{2}\right| = \mathsf{negl}(\lambda),$$

where the game $\mathsf{Private}$ is defined in Figure 4.

Security. If we were unconcerned with the privacy property of $\mathsf{NestedMuSig2}$, that counterparties cannot distinguish between cosigners who are individual keys and cosigners who are utilizing nesting and only appear to be individual keys in aggregate, then we could view $\mathsf{NestedMuSig2}$ as a convoluted multi-signature scheme such that the multiset of public keys is given by a cosigner tree. That is to say, all of the keys at the leaves of a cosigner tree must cooperate in generating partial signatures in order for the aggregate multi-signature to be computed, just as is the case for a non-nested multi-signature scheme. Thus, the unforgeability game we introduce for nested multi-signature schemes is derived directly from the unforgeability game for security under concurrent signing sessions used in previous works on multi-signatures with key aggregation [BDN18; DEF+19; MPS+19; NRS21]. The privacy properties distinguishing nested multi-signatures from non-nested multi-signatures are orthogonal to the analysis of unforgeability because the adversary knows and controls all cosigner trees.

Namely, our security game has a single honest signer whose key pair is randomly generated, and the adversary is given oracle access to all of the signing functions of the honest party, where the adversary can choose inputs to these signing functions however they like. In the end, the adversary's goal is to produce a forgery for a cosigner-tree-and-message pair such that the honest signer's key is present in the cosigner tree and such that the pair has not previously been given as input in signing queries to the honest signer. See Figure 5 for the full details of the security game, $\mathsf{EUF\text{-}CMA}^{\mathcal{A}}_{\Sigma}$.

Definition 2. *Given a two-round nested multi-signature scheme with key aggregation,*

$$\Sigma = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{KeyAgg}, \mathsf{Sign}, \mathsf{SignAgg}, \mathsf{SignAggExt}, \mathsf{Sign}', \mathsf{SignAgg}', \mathsf{Ver}),$$

let $\mathsf{EUF\text{-}CMA}^{\mathcal{A}}_{\Sigma}$ *be defined as in Figure 5. Then* Σ *is existentially unforgeable under chosen-message attack (EUF-CMA) if for any p.p.t. adversary* $\mathcal{A}$*,*

$$\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\Sigma}(\lambda) := \Pr\left[\mathsf{EUF\text{-}CMA}^{\mathcal{A}}_{\Sigma}(\lambda) = \mathtt{true}\right] = \mathsf{negl}(\lambda).$$

We do not provide oracle access to the deterministic functions $\mathsf{SignAgg}, \mathsf{SignAggExt}$, or $\mathsf{SignAgg}'$ as the adversary controls all aggregator nodes (because it controls the inputs to the honest signer's signing functions) and can run these functions locally since they take no secret inputs. As is the

Game $\mathsf{CORRECT}^{\mathcal{A}}_{\Sigma,m,CTree,j}(\lambda)$

$par \leftarrow \mathsf{Setup}(1^\lambda)$
$v_0 := Root(CTree)$
$\mathsf{KeyGen}(v_0)$
$\mathsf{Round}_1(v_0)$
$\mathsf{Round}_2(v_0, m, \emptyset, \emptyset)$
$\sigma := (state'_j, out'_{v_0})$
return $\mathsf{Ver}(pk_{v_0}, m, \sigma)$

$\mathsf{KeyGen}(v)$

if $IsLeaf(v)$ **then**
  $(sk_v, pk_v) := \mathsf{KeyGen}()$
else
  for $w \in Children(v)$ **do**
    $\mathsf{KeyGen}(w)$
  $pk_v := \mathsf{KeyAgg}((pk_w)_{w \in Children(v)})$

$\mathsf{Round}_1(v)$

if $IsLeaf(v)$ **then**
  $(out_v, state_v) := \mathsf{Sign}()$
else
  for $w \in Children(v)$ **do**
    $\mathsf{Round}_1(w)$
  $out\_internal_v := \mathsf{SignAgg}((out_w)_{w \in Children(v)})$
  $out_v := \mathsf{SignAggExt}(out\_internal_v, pk_v)$

$\mathsf{Round}_2(v, m, (out^d)_{0 \le d < \Lambda}, \{pk_{i,d}\}_{2 \le i \le n_d}^{0 \le d < \Lambda})$

if $IsLeaf(v)$ **then**
  $(out'_v, state'_v) := \mathsf{Sign}'(state_v, (out^d)_{0 \le d < \Lambda}, sk_v, m, \{pk_{i,d}\}_{2 \le i \le n_d}^{0 \le d < \Lambda}))$
else
  $out^\Lambda := out\_internal_v$
  for $w \in Children(v)$ **do**
    $\{pk_{i,\Lambda}\}_{2 \le i \le |Children(v)|} := \{pk_u\}_{u \in Children(v) \setminus \{w\}}$ // Any indexing will work
    $\mathsf{Round}_2(w, m, (out^d)_{0 \le d < \Lambda+1}, \{pk_{i,d}\}_{2 \le i \le n_d}^{0 \le d < \Lambda+1})$
  $w_0 := Children(v)[0]$ // Any choice of $w_0 \in Children(v)$ will work for the state
  $(out'_v, state'_v) := \mathsf{SignAgg}'((out'_w)_{w \in Children(v)}, state'_{w_0})$

Fig. 3. The correctness game for a nested multi-signature scheme Σ. The parameter $CTree$ is a rooted tree given as a linked data structure, where the function $Root$ returns the root node, each node has a list of $Children$, and $IsLeaf(v)$ is true if and only if $Children(v) = \emptyset$.

Game $\mathsf{Private}^{\mathcal{A},OutGen}_{n,m,\{pk_{i,d}\}_{2\le i\le n_d}^{0\le d<\Lambda},x}(\lambda)$

$par \leftarrow \mathsf{Setup}(1^\lambda)$

$b \leftarrow\!\!\$\ \{0,1\}$

if $b = 0$ **then**

  $(sk, pk) \leftarrow \mathsf{KeyGen}()$

  $(out, state) \leftarrow \mathsf{Sign}()$

  $(out', state') \leftarrow \mathsf{Sign}'(state, OutGen(out), sk, m, \{pk_{i,d}\}_{2\le i\le n_d}^{0\le d<\Lambda})$

else

  for $j \in \{1, \ldots, n\}$ **do**

    $(sk_i, pk_{i,\Lambda}) \leftarrow \mathsf{KeyGen}()$

  $pk := \mathsf{KeyAgg}(\{pk_{1,\Lambda}, \ldots, pk_{n,\Lambda}\})$

  for $j \in \{1, \ldots, n\}$ **do**

    $(out_j, state_j) \leftarrow \mathsf{Sign}()$

  $out_{int} := \mathsf{SignAgg}(out_1, \ldots, out_n)$

  $out := \mathsf{SignAggExt}(out_{int}, pk)$

  for $j \in \{1, \ldots, n\}$ **do**

    $(out'_j, state'_j) \leftarrow \mathsf{Sign}'(state_j, (out_{int}, OutGen(out)), sk_j, m, \{pk_{i,\Lambda}\}_{i\neq j} \cup \{pk_{i,d}\}_{2\le i\le n_d}^{0\le d<\Lambda})$

  $(out', state') := \mathsf{SignAgg}'(out'_1, \ldots, out'_n, state'_x)$

$\hat{b} \leftarrow \mathcal{A}(pk, out, out', state')$

return $\hat{b} = b$

Fig. 4. The privacy game for a nested multi-signature scheme Σ.

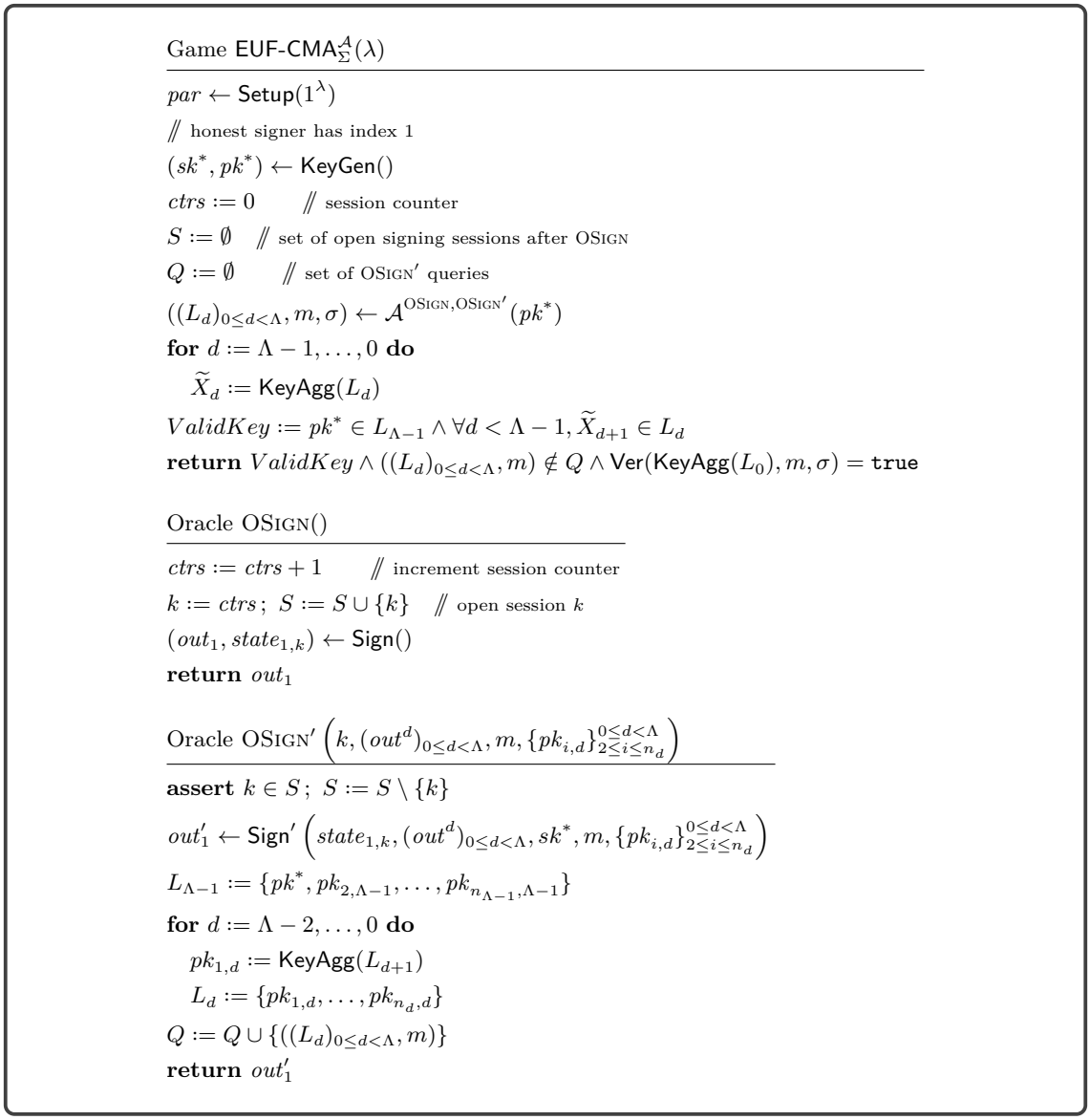

Fig. 5. The EUF-CMA security game for a nested multi-signature scheme Σ.

case with non-nested multi-signature unforgeability, the adversary can use cosigner trees containing the honest signer's public key in multiple places (including as siblings), and can query using this tree multiple times using the same message to simulate a signing session in which the honest signer participates multiple times.

4 The Nested Multi-Signature Scheme NestedMuSig2

4.1 Description

The scheme NestedMuSig2 aims to instantiate a variant of MuSig2 that itself generates MuSig2 or NestedMuSig2 partial signatures after partial signature aggregation instead of generating valid Schnorr signatures. Thus, each level of NestedMuSig2 signers aggregate their partial signatures into a single partial signature at the level above them, until a normal MuSig2 aggregation occurs, generating a valid Schnorr signature in the end. Notably, NestedMuSig2 is identical to MuSig2 when the nesting level, Λ, is set to 1. As with MuSig2, NestedMuSig2 is parameterized by a group generation algorithm, GrGen, and an integer number of nonces, ν. The scheme is defined as follows (also see Figure 6).

Parameters setup (Setup). Given 1^λ, a group, $(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$, is generated and then four hash functions, $\mathsf{H}_{\mathrm{agg}}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\overline{\mathrm{non}}}$, and $\mathsf{H}_{\mathrm{sig}}$, are selected from $\{0,1\}^*$ to $\mathbb{Z}_p$. Lastly, the values $((\mathbb{G}, p, g), \mathsf{H}_{\mathrm{agg}}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\overline{\mathrm{non}}}, \mathsf{H}_{\mathrm{sig}})$ are returned.

Key generation (KeyGen). Each signer generates a random private key $x \leftarrow\!\!\$\ \mathbb{Z}_p$ and returns the public key $X := g^x$.

Key aggregation (KeyAgg). As in MuSig2, given a multiset $L = \{X_1, \ldots, X_n\}$, the aggregate key is computed as $\widetilde{X} := \prod_{i=1}^{n} X_i^{a_i}$, where a_i is set to $\mathsf{KeyAggCoef}(L, X_i) := \mathsf{H}_{\mathrm{agg}}(L, X_i)$.

First signing round (Sign, SignAgg, and SignAggExt) As in MuSig2, each signer can perform Sign without knowledge of any cosigners (at any level of nesting) and without knowing the message to be signed.

Sign : Each signer generates ν values $r_j \leftarrow\!\!\$\ \mathbb{Z}_p$ and outputs the corresponding ν nonces, $(R_1 := g^{r_1}, \ldots, R_\nu := g^{r_\nu})$.

SignAgg : Upon receiving all first-round outputs from all signers,

$$(R_{1,1}, \ldots, R_{1,\nu}), \ldots, (R_{n,1}, \ldots, R_{n,\nu}),$$

the aggregator aggregates them by computing $R_j := \prod_{i=1}^{n} R_{i,j}$ for each $j \in \{1, \ldots, \nu\}$ and returns $(R_1, \ldots, R_\nu)$.

SignAggExt : After computing the aggregate first-round message, $(R'_1, \ldots, R'_\nu)$, using SignAgg, the aggregator computes $b := \mathsf{H}_{\mathrm{non}}(\widetilde{X}, (R'_1, \ldots, R'_\nu))$, where $\widetilde{X}$ is the aggregate key of its signers, and then the aggregator computes $R_j := (R'_j)^{b^{j-1}}$ for each $j \in \{1, \ldots, \nu\}$ and returns $(R_1, \ldots, R_\nu)$ as its external first-round output.

Second signing round (Sign′ and SignAgg′) Let $(x_{1,\Lambda-1}, X_{1,\Lambda-1})$ be a specific signer's key pair at depth Λ.

Sign′: Given the aggregate (internal) first-round outputs,

$$(R_{1,\Lambda-1}, \ldots, R_{\nu,\Lambda-1}), \ldots, (R_{1,0}, \ldots, R_{\nu,0}),$$

the message to be signed, m, and a tree of co-signers, $\{X_{i,d}\}_{2 \le i \le n_d}^{0 \le d < \Lambda}$, the signer computes $a_{1,d} := \mathsf{KeyAggCoef}(L_d, X_{1,d})$ and $b_d := \mathsf{H}_{\mathrm{non}}(X_{1,d-1}, (R_{1,d}, \ldots, R_{\nu,d}))$ for every $d \in \{1, \ldots, \Lambda - 1\}$, where $L_d := \{X_{1,d}, X_{2,d}, \ldots, X_{n_d,d}\}$ and $X_{1,d-1} := \mathsf{KeyAgg}(L_d)$. Then, letting $(r_{1,1}, \ldots, r_{1,\nu})$ be the signer's state from the first signing round, they compute

$$\begin{aligned}
\widetilde{X} &:= \mathsf{KeyAgg}(L_0) \\
a_{1,0} &:= \mathsf{KeyAggCoef}(L_0, X_{1,0}) \\
b_0 &:= \mathsf{H}_{\overline{\mathrm{non}}}(\widetilde{X}, (R_{1,0}, \ldots, R_{\nu,0}), m) \\
R &:= \textstyle\prod_{j=1}^{\nu} R_{j,0}^{b_0^{j-1}}
\end{aligned}$$

$$\check{b} := \prod_{\ell=0}^{\Lambda-1} b_\ell \bmod p$$
$$c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$$
$$\check{c} := c \prod_{\ell=0}^{\Lambda-1} a_{1,\ell} \bmod p$$
$$s_1 := \check{c} x_{1,\Lambda-1} + \sum_{j=1}^{\nu} r_{1,j} \check{b}^{j-1} \bmod p$$

and output s_1.

$\mathsf{SignAgg}'$: The aggregator receives the partial signatures of all signers, $(s_1, \ldots, s_n)$, and aggregates them by computing and returning the sum $s := \sum_{i=1}^{n} s_i \bmod p$.

Verification Given a top-level aggregate key, $\widetilde{X}$, a message, m, and a top-level aggregate signature, $\sigma = (R, s)$, the verifier accepts the signature if $g^s = R\widetilde{X}^c$, where $c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$. This is the standard verification algorithm for a Schnorr signature.

Correctness can be verified by induction on the depth of nesting, where the base case is MuSig2.

Information-theoretical privacy is straightforward to verify. The joint distribution on the outputs of KeyGen and Sign is uniformly random, and the subsequent output of Sign' is determined by these values and the distribution of *OutGen*. Similarly, it is easy to see that the same holds for the outputs of KeyAgg, SignAggExt, and $\mathsf{SignAgg}'$ in Figure 4, with the same dependence on the distribution of *OutGen* (which is given a uniformly random input in both cases).

We follow MuSig2's design decision to aggregate (and not require signers to verify) first-round messages, as this significantly reduces the communication and computation required during second-round signing without affecting the unforgeability of aggregate signatures. The most significant aspect of NestedMuSig2 not dictated by MuSig2 is that lower-level nonce hashes are computed without reference to the message to be signed. This is important because if this were not the case, then nested signers would require knowledge of the message to be signed during the first round in order to compute SignAggExt. This would pose a serious issue to the feasibility of NestedMuSig2 because the fact that MuSig2 does not have this restriction means that many applications will use/are already using MuSig2 with the first round being computed before the message is determined.

4.2 Practical Considerations

The Choice of the Number ν of Nonces. In practice, the number of nonces used by nested signers must match the number of nonces used at the non-nested top-level of MuSig2 being run. Therefore, if protocols using MuSig2 wish to allow participants to use NestedMuSig2 without relying on the AGM, more than $\nu = 4$ nonces may be required. Applications using MuSig2 today likely follow the BIP 327 specification [NRJ22] which specifies $\nu = 2$. Thus, we expect that in practice this will be the most common choice for the number of nonces, and this emphasizes the importance of Theorem 2, which proves the security of NestedMuSig2$[\nu = 2]$ in the ROM+AGM.

Deterministic Nonces are Insecure. As with MuSig2 [NRS21], it is insecure to naïvely derandomize NestedMuSig2 signing, and a secure random number generator is required for generating the nonce values.

Statefulness. As with MuSig2 [NRS21], it is crucial that Sign' is never executed twice with the same state. This would result in effective nonce reuse and would allow any party that sees both partial signatures to trivially extract the signer's private key.

Optimized Key Aggregation. As with MuSig2 [NRS21], it is possible to save one exponentiation in the key aggregation algorithm by setting the exponent of the second distinct input public key (if there are at least two distinct keys) and the exponents of all copies of this key to the constant 1. This is discussed further in Appendix B.

Optimized Nonce Aggregation. If D is small (e.g., $D = 2$), then the multiplicative aggregation of nonce commitments into the value $\check{b}$ of NestedMuSig2 is perfectly fine. But if deeper nesting is desirable or if easier extensibility with other subprotocols (such as nested threshold keys) is preferred, then it is better to compute $\check{b}$ using a hash tree rather than multiplying hashes. This is discussed further in Appendix A.

$\mathsf{Setup}(1^\lambda)$

$(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$
Select four hash functions
$\mathsf{H}_{\mathrm{agg}}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\overline{\mathrm{non}}}, \mathsf{H}_{\mathrm{sig}} : \{0,1\}^* \to \mathbb{Z}_p$
$par :=$
$\quad ((\mathbb{G}, p, g), \mathsf{H}_{\mathrm{agg}}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\overline{\mathrm{non}}}, \mathsf{H}_{\mathrm{sig}})$
return par

$\mathsf{KeyGen}()$

$x \leftarrow\!\!\$\ \mathbb{Z}_p\ ;\ X := g^x$
$sk := x\ ;\ pk := X$
return (sk, pk)

$\mathsf{KeyAggCoef}(L, X_i)$

return $\mathsf{H}_{\mathrm{agg}}(L, X_i)$

$\mathsf{KeyAgg}(L)$

$\{X_1, \ldots, X_n\} := L$
for $i := 1 \ldots n$ **do**
$\quad a_i := \mathsf{KeyAggCoef}(L, X_i)$
return $\widetilde{X} := \prod_{i=1}^n X_i^{a_i}$

$\mathsf{Ver}(\widetilde{X}, m, \sigma)$

$(R, s) := \sigma$
$c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$
return $(g^s = R\widetilde{X}^c)$

$\mathsf{Sign}()$

// Local signer has index 1.
for $j := 1 \ldots \nu$ **do**
$\quad r_{1,j} \leftarrow\!\!\$\ \mathbb{Z}_p\ ;\ R_{1,j} := g^{r_{1,j}}$
$out_1 := (R_{1,1}, \ldots, R_{1,\nu})$
$state_1 := (r_{1,1}, \ldots, r_{1,\nu})$
return $(out_1, state_1)$

$\mathsf{SignAgg}(out_1, \ldots, out_n)$

for $i := 1 \ldots n$ **do**
$\quad (R_{i,1}, \ldots, R_{i,\nu}) := out_i$
for $j := 1 \ldots \nu$ **do**
$\quad R_j := \prod_{i=1}^n R_{i,j}$
return $(R_1, \ldots, R_\nu)$

$\mathsf{SignAggExt}(out, \widetilde{X})$

$(R'_1, \ldots, R'_\nu) := out$
$b := \mathsf{H}_{\mathrm{non}}(\widetilde{X}, (R'_1, \ldots, R'_\nu))$
for $j := 1 \ldots \nu$ **do**
$\quad R_j := (R'_j)^{b^{j-1}}$
return $(R_1, \ldots, R_\nu)$

$\mathsf{Sign}'(state_1, (out^d)_{0 \le d < \Lambda}, sk_1, m, \{pk_{i,d}\}_{2 \le i \le n_d}^{0 \le d < \Lambda})$

// Sign' must be called at most once per $state_1$.
$(r_{1,1}, \ldots, r_{1,\nu}) := state_1$
$pk_{1,\Lambda-1} := g^{sk_1}$
$L_{\Lambda-1} := \{pk_{1,\Lambda-1}, \ldots, pk_{n_{\Lambda-1},\Lambda-1}\}$
$a_{1,\Lambda-1} := \mathsf{KeyAggCoef}(L_{\Lambda-1}, pk_{1,\Lambda-1})$
$pk_{1,\Lambda-2} := \mathsf{KeyAgg}(L_{\Lambda-1})$
for $d := \Lambda - 2, \ldots, 0$ **do**
$\quad b_{d+1} := \mathsf{H}_{\mathrm{non}}(pk_{1,d}, out^{d+1})$
$\quad L_d := \{pk_{1,d}, \ldots, pk_{n_d,d}\}$
$\quad a_{1,d} := \mathsf{KeyAggCoef}(L_d, pk_{1,d})$
$\quad pk_{1,d-1} := \mathsf{KeyAgg}(L_d)$
$\widetilde{X} := pk_{1,-1}$
$(R_{1,0}, \ldots, R_{\nu,0}) := out^0$
$b_0 := \mathsf{H}_{\overline{\mathrm{non}}}(\widetilde{X}, (R_{1,0}, \ldots, R_{\nu,0}), m)$
$R := \prod_{j=1}^\nu R_{j,0}^{b_0^{j-1}}$
$\check{b} := \prod_{\ell=0}^{\Lambda-1} b_\ell \bmod p$
$c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$
$\check{c} := c \prod_{\ell=0}^{\Lambda-1} a_{1,\ell} \bmod p$
$s_1 := \check{c} sk_1 + \sum_{j=1}^\nu r_{1,j} \check{b}^{j-1} \bmod p$
$state'_1 := R\ ;\ out'_1 := s_1$
return $(state'_1, out'_1)$

$\mathsf{SignAgg}'(out'_1, \ldots, out'_n, state'_1)$

$R := state'_1$
$(s_1, \ldots, s_n) := (out'_1, \ldots, out'_n)$
$s := \sum_{i=1}^n s_i \bmod p$
return $\sigma := (R, s)$

Fig. 6. The nested multi-signature scheme $\mathsf{NestedMuSig2}[\mathsf{GrGen}, \nu]$. Public parameters par returned by Setup are implicitly given as input to all other algorithms. We use a helper algorithm $\mathsf{KeyAggCoef}$ as a wrapper for $\mathsf{H}_{\mathrm{agg}}$ to make the description of the scheme more modular, which will help us describe the variants $\mathsf{NestedMuSig2}^*$ and $\mathsf{MerkleNestedMuSig2}^*$ in Appendix B.

Aggregator Node Responsibilities. While it is possible for a collection of cosigners to generate their aggregate NestedMuSig2 messages without an aggregator node, this approach requires a quadratic amount of communication instead of a linear amount of communication. Furthermore, the goal of NestedMuSig2 is to allow many parties to work together while remaining indistinguishable from a single participant at the next level, and this goal essentially requires the use of an (untrusted) aggregator node to play the role of the group, in aggregate, for purposes of communicating with participants at that next level. As such, we assume the use of an untrusted aggregator node, which may be one of the participants.

In the first round of signing, which may occur before the message is determined, the aggregator runs SignAgg and SignAggExt upon receiving all public outputs of Sign from each participant. The aggregator stores the output of SignAgg and communicates the output of SignAggExt as its first-round output (in place of invoking Sign itself). In the second round, the aggregator node concatenates the first-round output of SignAgg to the Sign' input arguments and forwards these inputs to all participants. Upon receiving partial signatures in response, the aggregator runs $\mathsf{SignAgg}'$ and outputs the result (in place of invoking Sign' itself).

Partial Verification and Accountability. The nonces $out_i = (R_{i,1}, \ldots, R_{i,\nu})$ and the partial signature $out'_i = s_i$ output by each signer during a signing session can be verified by checking the equation

$$g^{s_i} = \hat{R}_i X_i^{\check{c}},$$

where the effective nonce, $\hat{R}_i = \prod_{j=1}^{\nu} R_{i,j}^{\check{b}^{j-1}}$, with $\check{b}$ and $\check{c}$ being computed as in Sign' of Figure 6. If this *partial verification* passes for all signers at all levels, then the top-level aggregate signature will pass standard Schnorr signature verification. Note that it is possible to forge MuSig2 partial signatures [Gib23], so the only purpose of partial verification is to identify invalid partial signatures without providing any proof-of-knowledge guarantees.

Partial verification enables honest aggregator nodes to identify disruptive signers who generate invalid partial signatures.

5 Security of NestedMuSig2 in the ROM

In this section, we establish the security of NestedMuSig2 with $\nu = 2^{D+1}$ nonces in the random oracle model, where D is the nesting depth. Please note that for most applications, $D \leq 2$, so that what follows is quite practically applicable.

Theorem 1. *Let* GrGen *be a group generation algorithm for which the AOMDL problem is hard. Then the nested multi-signature scheme* $\mathsf{NestedMuSig2}[\mathsf{GrGen}, \nu = 2^{D+1}]$ *is EUF-CMA in the random oracle model for* $\mathsf{H}_{\mathrm{agg}}$, $\mathsf{H}_{\mathrm{non}}$, $\mathsf{H}_{\overline{\mathrm{non}}}$, $\mathsf{H}_{\mathrm{sig}} : \{0,1\}^* \to \mathbb{Z}_p$.

Precisely, for any adversary $\mathcal{A}$ *against* $\mathsf{NestedMuSig2}[\mathsf{GrGen}, \nu = 2^{D+1}]$ *running in time at most* t*, making at most* q_s OSign *queries and at most* q_h *queries to each random oracle, and such that the size of* L_d *in any signing session and in the forgery is at most* N*, and the maximum depth of any signing session is at most* D*, there exists an algorithm* $\mathcal{E}$ *taking as input group parameters* $(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$*, running in time at most*

$$t' = 2^{D+1}(t + q(3N + 3\nu - 3))t_{\exp} + O(qN)$$

where $q = (N+4)q_h + (DN + D + 1)q_s + DN + 1$ *and* $t_{\exp}$ *is the time of an exponentiation in* $\mathbb{G}$*, making at most* $2^{D+1}q_s$ DLog_g *queries, and solving the AOMDL problem with an advantage*

$$\mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{E},\mathsf{GrGen}}(\lambda) \geq \frac{(\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu=2^{D+1}]}(\lambda))^{2^{D+1}}}{2^{2^{D+1}-D-2}q^{2^{D+1}-1}} - \frac{2^{D+2}q + 2\binom{2^{D+1}q+D}{D}^2 + 12}{2^\lambda}.$$

Before proving the theorem, we begin with a high-level description of our strategy.

Generalizing the Double-Forking Technique. Informally, the security proof for non-nested MuSig2 [NRS21] in the ROM proceeds with two applications of the generalized forking lemma [BN06]. A successful forgery algorithm is first rewound to the first time it queries $\mathsf{H}_{\mathrm{sig}}$ to obtain the challenge hash, c, for its forgery. This results in two forgeries, s and s', such that $g^s = R\widetilde{X}^c$ and $g^{s'} = R\widetilde{X}^{c'}$. From these two equations we can see that $\widetilde{X} = g^{\frac{s-s'}{c-c'}}$. This is the standard technique for extracting a private key from a Schnorr signature. Now that we have built an algorithm for extracting the aggregate private key, $\tilde{x}$, constructed by aggregating the target honest signer's key with some other keys, we take advantage of the key aggregation equation,

$$\widetilde{X} = (X^*)^{n^* a^*} \prod_{X_i \neq X^*} X_i^{a_i},$$

where $a^* = \mathsf{H}_{\mathrm{agg}}(L, X^*)$ and n^* is the number of times X^* appears in the multiset of signers L, has the same structure as the Schnorr verification equation. Specifically, we rewind our $\tilde{x}$-solving algorithm to when a^* is assigned so that we get a second discrete log, $\tilde{x}'$, for the equation

$$\widetilde{X}' = (X^*)^{n^*(a^*)'} \prod_{X_i \neq X^*} X_i^{a_i}$$

from which we can conclude that $X^* = g^{\frac{\tilde{x}-\tilde{x}'}{n^*(a^*-(a^*)')}}$. Of course, we must take special care in both steps to ensure that the factors we wish to cancel (R in the first case and $\prod_{X_i \neq X^*} X_i^{a_i}$ in the second) are not altered by rewinding.

We generalize this method by forking at most $D+1$ times, where D is a bound on the depth of all cosigner trees; we fork $\mathsf{H}_{\mathrm{sig}}$ to extract from a forgery algorithm an aggregate private key constructed using the honest signer's key, and then we fork $\mathsf{H}_{\mathrm{agg}}$ once at each level of depth to extract a sub-aggregate private key that is one step closer to the honest signer's private key in the cosigner tree in exactly the same manner as is described in the previous paragraph for extracting $\mathrm{DLog}_g(X^*)$ from $\mathrm{DLog}_g(\widetilde{X})$.

Because we fork $\mathsf{H}_{\mathrm{agg}}$ many times but require that all of the other invocations of $\mathsf{H}_{\mathrm{agg}}$ do not change, we employ Bellare, Dai, and Li's "*local* forking lemma" [BDL19], which is a variant of Bellare and Neven's "generalized forking lemma" [BN06] (which is used in the MuSig2 security proof). It is possible to use the generalized forking lemma by programming $\mathsf{H}_{\mathrm{agg}}$ to assign its values for each fresh multiset in a random order and arguing that there is a non-negligible chance that the value we wish to fork on is assigned last, but this is unnecessarily complex and results in a degraded security bound. Meanwhile, the local forking lemma accomplishes exactly what we want by ensuring that all relevant unforked $\mathsf{H}_{\mathrm{agg}}$ values are not rerandomized.

Lemma 1 (Local Forking Lemma [BDL19]). *Let $q \geq 1$ be an integer. Let $\mathcal{A}$ be a randomized algorithm which takes a main input inp generated by some probabilistic algorithm* $\mathsf{InpGen}()$*, takes input random coins from some sampleable set R, has access to an oracle, H, and returns either a distinguished failure symbol, $\perp$, or a tuple (z, out), where z is one of the queries $\mathcal{A}$ made to H and out is some side output. The accepting probability of $\mathcal{A}$, denoted $acc(\mathcal{A})$, is defined as the probability, over inp $\leftarrow \mathsf{InpGen}()$, the random coins of $\mathcal{A}$, and the randomness used in $H[T]$ in Figure 7, that $\mathcal{A}^{H[T]}$ returns a non-$\perp$ output. Consider the algorithm $\mathsf{Fork}^{\mathcal{A}}$ described in Figure 7. Let frk be the probability (over inp $\leftarrow \mathsf{InpGen}()$, $H[T]$'s randomness, and the random coins of $\mathsf{Fork}^{\mathcal{A}}$) that $\mathsf{Fork}^{\mathcal{A}}$ returns a non-$\perp$ output. Then if q is an upper bound on the number of queries $\mathcal{A}$ can make to H,*

$$frk \geq \frac{acc(\mathcal{A})^2}{q}.$$

Simulating the Honest Signer. In both the ROM proof of this section and the AGM + ROM proof of the following section, the reduction's strategy for simulating the signing oracle available to the adversary in the EUF-CMA game (see Figure 5) is the standard strategy for two-round multi-party Schnorr signature schemes [MPS+19; NRS21; CGR+23; BGC+25]. In response to first-round signing queries, we generate nonces by using the AOMDL Ch oracle. In response to second-round signing queries, we cannot honestly generate signatures since we do not know the

Oracle $H[T](z)$	$\mathsf{Fork}^{\mathcal{A}}(inp)$
if $T(z) = \bot$ **then**	$\rho \leftarrow\!\!\$\ R$ // pick random coins for $\mathcal{A}$
$T(z) \leftarrow\!\!\$\ S$	$\alpha := \mathcal{A}^{H[T]}(inp; \rho)$
return $T(z)$	**if** $\alpha = \bot$ **then return** $\bot$
	$(z, out) := \alpha$
	$T' := T$ // copy T into T'
	$T'(z) \leftarrow\!\!\$\ S$ // and refresh $T'(z)$
	$\alpha' := \mathcal{A}^{H[T']}(inp; \rho)$
	if $\alpha' = \bot$ **then return** $\bot$
	$(z', out') := \alpha'$
	if $z \neq z' \vee h_i = h'_i$ **then return** $\bot$
	return (z, out, out')

Fig. 7. The "forking" algorithm $\mathsf{Fork}^{\mathcal{A}}$ built from $\mathcal{A}$.

private key or the nonce scalars from the first round, so instead we query the AOMDL DLOG_g oracle with the right-hand side of the partial signature verification equation (which can be computed along with an algebraic representation from the input to the signing query). This method is necessary as it is not possible to simulate signing only by programming the random oracle, as is done with single-party Schnorr signatures, because the nonce values must be committed to in the first round before the message or cosigner tree is known (hence it is not possible first to choose s and e and then to compute R and to program H_{sig}).

The construction of the aggregate nonce being computed from a product of $\nu \geq 2$ nonces involving the coefficient $b = \mathsf{H}_{\overline{\text{non}}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$ ensures that if the reduction programs $\mathsf{H}_{\overline{\text{non}}}$ to call H_{sig} preemptively, then the adversary cannot change the values $\widetilde{X}$, $(R_1, \ldots, R_\nu)$, and m in different executions while we fork on H_{sig}. This is why at least 2 nonces are necessary, even in the ROM + AGM proof. In the ROM proof of this section, we use even more nonces in order to deal with the fact that the adversary can query the signing oracle (which in turn queries the AOMDL DLOG_g oracle) many times within a single signing session due to our forking strategy.

5.1 Security Proof

Proof Overview. We first construct an algorithm $\mathcal{B}$ that simulates the EUF-CMA environment to the adversary $\mathcal{A}$ and also maintains bookkeeping information, programs random oracles, and simulates the honest signing oracles as discussed above. Then we construct an algorithm $\mathcal{C}$ that runs $\mathsf{Fork}^{\mathcal{B}}$, forking on the signature hash to extract the aggregate private key. Next, we construct a sequence of algorithms, $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{\Lambda-1}$, where $\mathcal{D}_1$ runs $\mathsf{Fork}^{\mathcal{C}}$, forking on the aggregation coefficient hash, to extract the aggregate private key involving the honest signer one level deeper than the aggregate private key extracted by $\mathcal{C}$, and likewise each $\mathcal{D}_\ell$ runs $\mathsf{Fork}^{\mathcal{D}_{\ell-1}}$, forking on the aggregation coefficient hash, each to extract the next private key until $\mathcal{D}_{\Lambda-1}$ extracts the honest signers private key. Finally, we construct an algorithm $\mathcal{E}$, which uses the AOMDL oracles and programmed random oracles to simulate the environment to $\mathcal{D}_{\Lambda-1}$ and to compute an AOMDL solution from the extracted private key by the subsequent solving of a system of equations that is generated during signing oracle simulation.

Normalizing Assumptions and Conventions. Let a (t, q_s, q_h, N, D)-adversary be an adversary running in time at most t, making at most q_s OSign queries, at most q_h queries to each random oracles, such that the depth, Λ, of any cosigner tree, $(L_\ell)_{0 \leq \ell < \Lambda}$, is at most D and such that the branching factor of any cosigner tree, $\max_{0 \leq \ell < \Lambda} |L_\ell|$, is at most N.

We assume that the adversary only makes "well-formed" random oracle queries, that is, queries match the expected interfaces. This is without loss of generality, as "ill-formed" queries could be

answered uniformly at random. We also assume that the adversary makes exactly q_h queries to each random oracle, exactly q_s queries to the OSIGN oracle, and makes exactly one OSIGN$'$ query corresponding to each OSIGN query. This is without loss of generality because any missing queries can be made arbitrarily after an adversary has terminated.

Lemma 2. *Given some integer ν, let $\mathcal{A}$ be a (t, q_s, q_h, N, D)-adversary in the random oracle model against the nested multi-signature scheme $\mathsf{NestedMuSig2}[\mathsf{GrGen}, \nu]$ that produces a nested forgery, and let $q = (N+4)q_h+(DN+D+1)q_s+DN+1$. Then there exists an algorithm $\mathcal{B}$ that takes as input group parameters $(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$, uniformly random group elements $X^*, U_1, \ldots, U_{\nu q_s} \in \mathbb{G}$, and has access to the random oracles $\mathsf{H}'_{\mathrm{sig}}, \mathsf{H}'_{\mathrm{non}}, \mathsf{H}'_{\overline{\mathrm{non}}}$, and $\mathsf{H}'_{\mathrm{agg}}$, makes at most q_s queries to a discrete logarithm oracle DLOG_g, and with accepting probability (as defined in Lemma 1)*

$$acc(\mathcal{B}) \geq \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu]}(\lambda) - \frac{6q^2}{2^\lambda}$$

outputs a tuple $(z_{\mathrm{sig}}, c, (z^d_{\mathrm{agg}})_{0\leq d<\Lambda}, (L_d)_{0\leq d<\Lambda}, R, s, (\vec{a}_d)_{0\leq d<\Lambda})$ where $L_d = \{X_{1,d}, \ldots, X_{n_d,d}\}$ is a multiset of public keys such that $X^* \in L_{\Lambda-1}$ and $\widetilde{X}_d := \mathsf{KeyAgg}(L_d) \in L_{d-1}$ (where KeyAgg is run using $\mathsf{H}'_{\mathrm{agg}}$) and $z^d_{\mathrm{agg}} = (L_d, \widetilde{X}_{d+1})$ for all $0 < d < \Lambda$, $\vec{a}_d = (a_{1,d}, \ldots, a_{n_d,d}) \in \mathbb{Z}_p^n$ is a tuple of scalars such that $a_{i,d} = \mathsf{H}'_{\mathrm{agg}}(L_d, X_{i,d})$, $c = \mathsf{H}'_{\mathrm{sig}}(z_{\mathrm{sig}})$, and

$$g^s = R\prod_{i=1}^{n_0} X_{i,0}^{a_{i,0}c}. \tag{1}$$

Proof. Algorithm $\mathcal{B}$ is given in Figures 8 and 9. Essentially, $\mathcal{B}$ runs $\mathcal{A}$ a single time by simulating all of its random oracles and simulating each nested signing query by using a single call to a discrete log oracle. The simulated random oracles also provide $\mathcal{B}$ with additional data that it forwards in its output. These simulated random oracles are also programmed to detect bad events (which will happen with negligible probability) as well as to trigger $\mathsf{H}_{\mathrm{sig}}$ queries corresponding to $\mathsf{H}_{\overline{\mathrm{non}}}$ queries so that $\mathcal{A}$ cannot separate when values in the $T_{\overline{\mathrm{non}}}$ and T_{sig} tables are initialized.

The output of $\mathcal{B}$ is correct when it does not return $\perp$ by construction. We bound the number of times *ctrh* is incremented. The value *ctrh* is incremented within $\mathsf{H}_{\mathrm{non}}$ at most once per hash query and at most $D-1$ times per signing query (running total: $q_h + (D-1)q_s$), and similarly *ctrh* is incremented within $\mathsf{H}_{\overline{\mathrm{non}}}$ at most once per hash query and at most once per signing query (running total: $2q_h + Dq_s$). The counter is incremented within $\mathsf{H}_{\mathrm{sig}}$ at most once per direct hash query, once per $\mathsf{H}_{\overline{\mathrm{non}}}$ query, and once during $\mathcal{B}$'s Ver invocation (running total: $2q_h + Dq_s + q_h + (q_h + q_s) + 1 = 4q_h + (D+1)q_s + 1$). Lastly, *ctrh* is incremented at most N times within each $\mathsf{H}_{\mathrm{agg}}$ query and there are at most q_h direct queries from $\mathcal{A}$, there are at most Dq_s possible-*ctrh*-incrementing invocations resulting from OSIGN$'$ queries (one per fresh KeyAgg), and at most D possible-*ctrh*-incrementing invocations during the verification of the forgery (again from KeyAgg calls). Thus, *ctrh* is incremented within $\mathsf{H}_{\mathrm{agg}}$ at most $N(q_h + Dq_s + D)$ times so that, in total, it is incremented at most $q = (N+4)q_h + (DN+D+1)q_s + DN + 1$ times.

Thus, it only remains to show that the bound on $acc(\mathcal{B})$ holds. If $\mathcal{B}$ never sets any of its Boolean flags to `true`, then $acc(\mathcal{B})$ is bounded by the advantage of $\mathcal{A}$ because $\mathcal{B}$ asserts that the forgery is valid. Hence, by the union bound we have that $acc(\mathcal{B}) \geq \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu]}(\lambda) - \Pr[\mathsf{BadOrder}] - \Pr[\mathsf{BadOrder}'] - \Pr[\mathsf{KeyColl}]$. Each of these flags can only be set to `true` within $\mathsf{H}_{\mathrm{agg}}$ in the case that $T_{\mathrm{agg}}(L, X)$ is being set for the first time, in which case each $T_{\mathrm{agg}}(L, X')$ is being set for the first time for each $X' \in L$. Because each of these values is distributed uniformly over $\mathbb{Z}_p$, we have that $\widetilde{X} := \mathsf{KeyAgg}(L)$ is also distributed uniformly over $\mathbb{G}$, which has size $p \geq 2^{\lambda-1}$. Because each of T_{sig}, K, and T_{agg} has at most q entries and there are fewer than q total calls to $\mathsf{H}_{\mathrm{agg}}$, the probability of a collision resulting in one of these flags being set to `true` is at most $3\left(\frac{q^2}{2^{\lambda-1}}\right) = \frac{6q^2}{2^\lambda}$. Therefore,

$$acc(\mathcal{B}) \geq \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu]}(\lambda) - \frac{6q^2}{2^\lambda}.$$

□

$\mathcal{B}^{\mathsf{H}'_{\mathrm{sig}},\mathsf{H}'_{\mathrm{non}},\mathsf{H}'_{\overline{\mathrm{non}}},\mathsf{H}'_{\mathrm{agg}}}((\mathbb{G},p,g),(X^*,U_1,\ldots,U_{\nu q_s}))$

// tables for storing random oracle answers
$T_{\mathrm{agg}} := \emptyset$; $T_{\mathrm{non}} := \emptyset$; $T_{\overline{\mathrm{non}}} := \emptyset$; $T_{\mathrm{sig}} := \emptyset$
$ctrh := 0$ // counter for random oracle queries
$ctrs := 0$ // counter for signing session
$S := \emptyset$ // open sessions
$Q := \emptyset$ // completed sessions
$K := \emptyset$ // aggregate public keys
// flags for bad events
$\mathsf{BadOrder} := \mathtt{false}$; $\mathsf{BadOrder}' := \mathtt{false}$
$\mathsf{KeyColl} := \mathtt{false}$
$\rho_A \leftarrow\!\$\ R$ // randomness for $\mathcal{A}$
$\mathsf{H}s := (\mathsf{H}_{\mathrm{agg}},\mathsf{H}_{\mathrm{non}},\mathsf{H}_{\overline{\mathrm{non}}},\mathsf{H}_{\mathrm{sig}})$
$frg := \mathcal{A}^{\mathsf{H}s,\textsc{OSign},\textsc{OSign}'}((\mathbb{G},p,g),X^*;\rho_A)$
$((L_d)_{0\le d<\Lambda},m,(R,s)) := frg$
assert $X^* \in L_{\Lambda-1}$
for $d := \Lambda-1,\ldots,1$ **do**
  $\widetilde{X}_d := \mathsf{KeyAgg}(L_d)$
  assert $\widetilde{X}_d \in L_{d-1}$
$\widetilde{X}_0 := \mathsf{KeyAgg}(L_0)$
assert $((L_d)_{0\le d<\Lambda},m) \notin Q \wedge \mathsf{Ver}(\widetilde{X}_0,m,(R,s))$
assert $\neg\mathsf{BadOrder} \wedge \neg\mathsf{BadOrder}' \wedge \neg\mathsf{KeyColl}$
$\widetilde{X}_\Lambda := X^*$
for $d := \Lambda-1,\ldots 0$ **do**
  $\{\widetilde{X}_{d+1},X_{2,d},\ldots,X_{n_d,d}\} := L_d$
  // T_{agg} entries are defined due to KeyAgg calls
  $z^d_{\mathrm{agg}} := (L_d,\widetilde{X}_{d+1})$
  $a_{1,d} := T_{\mathrm{agg}}(z^d_{\mathrm{agg}})$
  for $i := 2,\ldots,n_d$ **do**
    $a_{i,d} := T_{\mathrm{agg}}(L_d,X_{i,d})$
  $\vec{a}_d = (a_{1,d},\ldots,a_{n_d,d})$
$z_{\mathrm{sig}} := (\widetilde{X}_0,R,m)$
$c := \mathsf{H}_{\mathrm{sig}}(z_{\mathrm{sig}})$
return $(z_{\mathrm{sig}},c,(z^d_{\mathrm{agg}})_{0\le d<\Lambda},(L_d)_{0\le d<\Lambda},R,s,(\vec{a}_d)_{0\le d<\Lambda})$

$\textsc{OSign}()$

$ctrs := ctrs+1$
$S := S \cup \{ctrs\}$
$\hat{k} := \nu(ctrs-1)+1$
$(R_{1,1}\ldots,R_{1,\nu}) := (U_{\hat{k}},\ldots,U_{\hat{k}+\nu-1})$
return $(R_{1,1}\ldots,R_{1,\nu})$

$\textsc{OSign}'(k,(out^d)_{0\le d<\Lambda},m,\{X_{i,d}\}^{0\le d<\Lambda}_{2\le i\le n_d})$

assert $k \in S$; $S := S \setminus \{k\}$
$k' := \nu(k-1)+1$
$Q := Q \cup \{((L_d)_{0\le d<\Lambda},m)\}$
$(R_{1,1},\ldots,R_{1,\nu}) := (U_{k'},\ldots,U_{k'+\nu-1})$
$X_{1,\Lambda-1} := X^*$
for $d := \Lambda-1,\ldots,0$ **do**
  $L_d := \{X_{1,d},\ldots,X_{n_d,d}\}$
  $a_{1,d} := \mathsf{KeyAggCoef}(L_d,X_{1,d})$
  $X_{1,d-1} := \mathsf{KeyAgg}(L_d)$
  $b_d := \mathsf{H}_{\mathrm{non}}(X_{1,d-1},out^d)$
$\widetilde{X} := X_{1,-1}$
$(R_1,\ldots,R_\nu) := out^0$
$b_0 := \mathsf{H}_{\overline{\mathrm{non}}}(\widetilde{X},out^0,m)$
// R can be cached from $\mathsf{H}_{\overline{\mathrm{non}}}$ in previous line
$R := \prod_{j=1}^{\nu} R_j^{b_0^{j-1}}$
$c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X},R,m)$
$\check{c} := c\prod_{d=0}^{\Lambda-1} a_{1,d}$
$\check{b} := \prod_{d=0}^{\Lambda-1} b_d$
$\hat{R}_1 := \prod_{j=1}^{\nu} R_{1,j}^{\check{b}^{j-1}}$
$\hat{k} := \nu(ctrs-1)+1$
$\alpha := 0$; $(\beta_i)_{1\le i\le\hat{k}} := (0,\ldots,0)$
$\beta_1 := \check{c}$
$(\beta_{k'},\ldots,\beta_{k'+\nu-1}) := (\check{b}^0,\ldots,\check{b}^{\nu-1})$
$s_1 := \mathrm{DLog}_g(\hat{R}_1(X^*)^{\check{c}},(\alpha,(\beta_i)_{1\le i\le\hat{k}}))$
return s_1

Fig. 8. Algorithm $\mathcal{B}$ from the proof of Lemma 2.

$\mathsf{H}_{\mathrm{agg}}(L, X)$

if $T_{\mathrm{agg}}(L, X) = \perp$ **then**
  for $X' \in L$ **do**
    $ctrh := ctrh + 1$
    $T_{\mathrm{agg}}(L, X') := \mathsf{H}'_{\mathrm{agg}}(L, X')$
  $\widetilde{X} := \mathsf{KeyAgg}(L)$ // does not increment $ctrh$
  if $\exists R, m : T_{\mathrm{sig}}(\widetilde{X}, \overline{R}, m) \neq \perp$ **then**
    $\mathsf{BadOrder} := \mathtt{true}$
  if $\exists L' : T_{\mathrm{agg}}(L', \widetilde{X}) \neq \perp$ **then**
    $\mathsf{BadOrder}' := \mathtt{true}$
  if $\widetilde{X} \in K$ **then**
    $\mathsf{KeyColl} := \mathtt{true}$
  $K := K \cup \{\widetilde{X}\}$
return $T_{\mathrm{agg}}(L, X)$

$\mathsf{H}_{\overline{\mathrm{non}}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$

if $T_{\overline{\mathrm{non}}}(\widetilde{X}, (R_1, \ldots, R_\nu), m) = \perp$ **then**
  $ctrh := ctrh + 1$
  $T_{\overline{\mathrm{non}}}(\widetilde{X}, (R_1, \ldots, R_\nu), m) := \mathsf{H}'_{\overline{\mathrm{non}}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$
  $b_0 := T_{\overline{\mathrm{non}}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$
  $R := \prod_{j=1}^{\nu} R_j^{b_0^{j-1}}$
  $\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$ // preemptive $\mathsf{H}_{\mathrm{sig}}$ query
return $T_{\overline{\mathrm{non}}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$

$\mathsf{H}_{\mathrm{non}}(\widetilde{X}, (R_1, \ldots, R_\nu))$

if $T_{\mathrm{non}}(\widetilde{X}, (R_1, \ldots, R_\nu)) = \perp$ **then**
  $ctrh := ctrh + 1$
  $T_{\mathrm{non}}(\widetilde{X}, (R_1, \ldots, R_\nu)) := \mathsf{H}'_{\mathrm{non}}(\widetilde{X}, (R_1, \ldots, R_\nu))$
return $T_{\mathrm{non}}(\widetilde{X}, (R_1, \ldots, R_\nu))$

$\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$

if $T_{\mathrm{sig}}(\widetilde{X}, R, m) = \perp$ **then**
  $ctrh := ctrh + 1$
  $T_{\mathrm{sig}}(\widetilde{X}, R, m) := \mathsf{H}'_{\mathrm{sig}}(\widetilde{X}, R, m)$
return $T_{\mathrm{sig}}(\widetilde{X}, R, m)$

Fig. 9. Simulated random oracles for algorithm $\mathcal{B}$.

Lemma 3. *Given some integer ν, let $\mathcal{A}$ be a (t, q_s, q_h, N, D)-adversary in the random oracle model against the nested multi-signature scheme* $\mathsf{NestedMuSig2}[\mathsf{GrGen}, \nu]$ *that produces a nested forgery and let $q = (N+4)q_h + (DN+D+1)q_s + DN + 1$. Then there exists an algorithm $\mathcal{C}$ that takes as input group parameters $(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$, uniformly random group elements $X^*, U_1, \ldots, U_{\nu q_s} \in \mathbb{G}$, and has access to the random oracles $\mathsf{H}'_{\mathrm{sig}}, \mathsf{H}'_{\mathrm{non}}, \mathsf{H}'_{\overline{\mathrm{non}}}$, and $\mathsf{H}'_{\mathrm{agg}}$, makes at most $2q_s$ queries to a discrete logarithm oracle DLOG_g, and with accepting probability (as defined in Lemma 1)*

$$acc(\mathcal{C}) \geq \frac{(\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu]}(\lambda))^2}{q} - \frac{12q}{2^\lambda}$$

outputs a tuple $((z^d_{\mathrm{agg}})_{0\leq d<\Lambda}, (L_d)_{0\leq d<\Lambda}, (\vec{a}_d)_{0\leq d<\Lambda}, \tilde{x})$ where $L_d = \{X_{1,d}, \ldots, X_{n_d,d}\}$ is a multiset of public keys such that $X^* \in L_{\Lambda-1}$ and $\widetilde{X}_d := \mathsf{KeyAgg}(L_d) \in L_{d-1}$ (where KeyAgg *is run using* $\mathsf{H}'_{\mathrm{agg}}$*) and $z^d_{\mathrm{agg}} = (L_d, \widetilde{X}_{d+1})$ for all $0 < d < \Lambda$, $\vec{a}_d = (a_{1,d}, \ldots, a_{n_d,d}) \in \mathbb{Z}_p^n$ is a tuple of scalars such that $a_{i,d} = \mathsf{H}'_{\mathrm{agg}}(L_d, X_{i,d})$, and $\tilde{x}$ is the discrete logarithm of $\widetilde{X} = \prod_{i=1}^{n_0} X_{i,0}^{a_{i,0}}$ in base g.*

Proof. We define the algorithm $\mathcal{C}$ to run $\mathsf{Fork}^{\mathcal{B}}$ with $inp = ((\mathbb{G}, p, g), X^*, U_1, \ldots, U_{\nu q_s})$, $z = z_{\mathrm{sig}}$, and $out = ((z^d_{\mathrm{agg}})_{0\leq d<\Lambda}, (L_d)_{0\leq d<\Lambda}, R, s, (\vec{a}_d)_{0\leq d<\Lambda})$. The algorithm receives

$$\begin{aligned}&(z_{\mathrm{sig}},\\ &(c, (z^d_{\mathrm{agg}})_{0\leq d<\Lambda}, (L_d)_{0\leq d<\Lambda}, R, s, (\vec{a}_d)_{0\leq d<\Lambda}) := out,\\ &(c', ((z^d_{\mathrm{agg}})')_{0\leq d<\Lambda}, (L'_d)_{0\leq d<\Lambda}, R', s', (\vec{a}'_d)_{0\leq d<\Lambda}) := out')\end{aligned}$$

from which it computes

$$\tilde{x} = (s - s')(\mathsf{H}_{\mathrm{sig}}(z_{\mathrm{sig}}) - \mathsf{H}_{\mathrm{sig}}(z'_{\mathrm{sig}}))^{-1} \bmod p$$

and returns $((z^d_{\mathrm{agg}})_{0\leq d<\Lambda}, (L_d)_{0\leq d<\Lambda}, (\vec{a}_d)_{0\leq d<\Lambda}, \tilde{x})$. In the next paragraph, we argue that the only components of *out* that differ from *out*′ are $c \neq c'$ and $s \neq s'$; in this case,

$$g^s = R\widetilde{X}^c \text{ and } g^{s'} = R\widetilde{X}^{c'} \Rightarrow g^{s-s'} = \widetilde{X}^{c-c'} \Rightarrow g^{\tilde{x}} = \widetilde{X}.$$

Because both executions of $\mathcal{B}$ are identical within $\mathcal{C}$ up until the assignment of $\mathsf{H}_{\mathrm{sig}}(z_{\mathrm{sig}}) = T_{\mathrm{sig}}(\widetilde{X}, R, m)$ and $\mathsf{H}_{\mathrm{sig}}(z'_{\mathrm{sig}}) = T'_{\mathrm{sig}}(\widetilde{X}', R', m')$, it follows that $\widetilde{X} = \widetilde{X}'$ and $R = R'$ because these arguments must have been assigned before the forking point. Because $\mathcal{B}$'s assertions have all succeeded, we have that $\widetilde{X} = \mathsf{KeyAgg}(L_0) = \mathsf{KeyAgg}(L'_0)$ and $\neg\mathsf{KeyColl}$ is `true` in both executions so that $L_0 = L'_0$, and in particular $X_{1,0} = X'_{1,0}$ since this key is in L_0 as constructed during $\mathcal{B}$'s forgery verification. Because $\neg\mathsf{BadOrder}$ is `true`, we have that $T_{\mathrm{agg}}(L_0, X_{i,0})$ must have been assigned before the forking point for every $X_{i,0} \in L_0$, and thus $z^0_{\mathrm{agg}} = (z^0_{\mathrm{agg}})'$, and $\vec{a}_0 = \vec{a}'_0$. Similarly, by the same argument as above, because there have been no key collisions and $\mathcal{B}$'s assertions have succeeded, we can conclude from $X_{1,d} = X'_{1,d}$ that $\mathsf{KeyAgg}(L_{d+1}) = \mathsf{KeyAgg}(L'_{d+1})$ so that $L_{d+1} = L'_{d+1}$, and because $\neg\mathsf{BadOrder}$ is true, $T_{\mathrm{agg}}(L_{d+1}, X_{i,d+1})$ must have been assigned before the forking point for every $X_{i,d+1} \in L_{d+1}$ and therefore $z^d_{\mathrm{agg}} = (z^d_{\mathrm{agg}})'$ and $\vec{a}_d = \vec{a}'_d$ for all d.

It remains to prove the bound on the accepting probability of $\mathcal{C}$, which follows directly from Lemma 1 and Lemma 2. Letting $\epsilon = \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu]}(\lambda)$,

$$\begin{aligned}acc(\mathcal{C}) = acc(\mathsf{Fork}^{\mathcal{B}}) &\geq \frac{acc(\mathcal{B})^2}{q}\\ &\geq \frac{\left(\epsilon - \frac{6q^2}{2^\lambda}\right)^2}{q}\\ &\geq \frac{\epsilon^2}{q} - \frac{12q}{2^\lambda}.\end{aligned}$$

□

Lemma 4. *Given some integer ν, let $\mathcal{A}$ be a (t, q_s, q_h, N, D)-adversary in the random oracle model against the nested multi-signature scheme $\mathsf{NestedMuSig2}[\mathsf{GrGen}, \nu]$ that produces a nested forgery and let $q = (N+4)q_h + (DN+D+1)q_s + DN + 1$. Then for each $\ell \in \{1, \ldots, D-1\}$ there exists an algorithm $\mathcal{D}_\ell$ that takes as input group parameters $(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$, uniformly random group elements $X^*, U_1, \ldots, U_{\nu q_s} \in \mathbb{G}$, and has access to the random oracles $\mathsf{H}'_{\mathrm{sig}}, \mathsf{H}'_{\mathrm{non}}, \mathsf{H}'_{\overline{\mathrm{non}}}$, and $\mathsf{H}'_{\mathrm{agg}}$, that makes at most $2^{\ell+1} q_s$ queries to a discrete logarithm oracle DLOG_g, and with accepting probability (as defined in Lemma 1)*

$$acc(\mathcal{D}_\ell) \geq \frac{(\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu]}(\lambda))^{2^{\ell+1}}}{2^{2^{\ell+1}-\ell-2} q^{2^{\ell+1}-1}} - \frac{12}{2^\lambda}$$

outputs a tuple $((z^d_{\mathrm{agg}})_{\ell \leq d < \Lambda}, (L_d)_{\ell \leq d < \Lambda}, (\vec{a}_d)_{\ell \leq d < \Lambda}, \tilde{x}_\ell)$ where $L_d = \{X_{1,d}, \ldots, X_{n_d,d}\}$ is a multiset of public keys such that $X^* \in L_{\Lambda-1}$ and $\widetilde{X}_d := \mathsf{KeyAgg}(L_d) \in L_{d-1}$ (where KeyAgg is run using $\mathsf{H}'_{\mathrm{agg}}$) and $z^d_{\mathrm{agg}} = (L_d, \widetilde{X}_{d+1})$ for all $\ell < d < \Lambda$, $\vec{a}_d = (a_{1,d}, \ldots, a_{n_d,d}) \in \mathbb{Z}_p^n$ is a tuple of scalars such that $a_{i,d} = \mathsf{H}'_{\mathrm{agg}}(L_d, X_{i,d})$, and $\tilde{x}_\ell$ is the discrete logarithm of $\widetilde{X}_\ell = \prod_{i=1}^n X_{i,\ell}^{a_{i,\ell}}$ in base g.

Proof. Letting $\mathcal{D}_0 := \mathcal{C}$, we define the algorithm $\mathcal{D}_\ell$ to run first $\mathsf{Fork}^{\mathcal{D}_{\ell-1}}$ with the values $inp = ((\mathbb{G}, p, g), X^*, U_1, \ldots, U_{\nu q_s})$, $z = z^{\ell-1}_{\mathrm{agg}}$, and $out = ((z^d_{\mathrm{agg}})_{\ell \leq d < \Lambda}, (L_d)_{\ell-1 \leq d < \Lambda}, (\vec{a}_d)_{\ell-1 \leq d < \Lambda}, \tilde{x}_{\ell-1})$. The algorithm receives

$$\begin{aligned}
&(z^{\ell-1}_{\mathrm{agg}},\\
&((z^d_{\mathrm{agg}})_{\ell \leq d < \Lambda}, (L_d)_{\ell-1 \leq d < \Lambda}, (\vec{a}_d)_{\ell-1 \leq d < \Lambda}, \tilde{x}_{\ell-1}) := out,\\
&(((z^d_{\mathrm{agg}})')_{\ell \leq d < \Lambda}, (L'_d)_{\ell-1 \leq d < \Lambda}, (\vec{a}'_d)_{\ell-1 \leq d < \Lambda}, \tilde{x}'_{\ell-1}) := out')
\end{aligned}$$

and then $\mathcal{D}_\ell$ computes

$$\tilde{x}_\ell = (\tilde{x}_{\ell-1} - \tilde{x}'_{\ell-1})(\tilde{n})^{-1}(a_{i^*,\ell-1} - a'_{i^*,\ell-1})^{-1} \bmod p,$$

where $\tilde{n}$ is the number of times $\widetilde{X}_\ell := \mathsf{KeyAgg}(L_\ell)$ appears in $L_{\ell-1}$ and i^* is the index of $\widetilde{X}_\ell$ in $L_{\ell-1}$, and returns the tuple $((z^d_{\mathrm{agg}})_{\ell \leq d < \Lambda}, (L_d)_{\ell \leq d < \Lambda}, (\vec{a}_d)_{\ell \leq d < \Lambda}, \tilde{x}_\ell)$. In the next paragraph, we argue that the only components of out that differ from out' are $\tilde{x}_{\ell-1} \neq \tilde{x}'_{\ell-1}$ and $a_{i,\ell-1} \neq a'_{i,\ell-1}$ (in $\vec{a}_{\ell-1}$ and $\vec{a}'_{\ell-1}$, respectively) for all i such that $X_{i,\ell-1} = \widetilde{X}_\ell$; in this case, we have

$$\begin{aligned}
g^{\tilde{x}_{\ell-1}} &= \prod_{i=1}^{n_{\ell-1}} (X_{i,\ell-1})^{a_{i,\ell-1}} = (\widetilde{X}_\ell)^{\tilde{n} a_{i^*,\ell-1}} \prod_{X_{i,\ell-1} \neq \widetilde{X}_\ell} X_{i,\ell-1}^{a_{i,\ell-1}}\\
g^{\tilde{x}'_{\ell-1}} &= \prod_{i=1}^{n_{\ell-1}} (X'_{i,\ell-1})^{a'_{i,\ell-1}} = (\widetilde{X}_\ell)^{\tilde{n} a'_{i^*,\ell-1}} \prod_{X_{i,\ell-1} \neq \widetilde{X}_\ell} X_{i,\ell-1}^{a_{i,\ell-1}}
\end{aligned}$$

from which we can conclude that $g^{\tilde{x}_\ell} = \widetilde{X}_\ell$.

Because both executions of $\mathcal{D}_{\ell-1}$ are identical within $\mathcal{D}_\ell$ up until the assignment of $\mathsf{H}_{\mathrm{agg}}(z^{\ell-1}_{\mathrm{agg}}) = T_{\mathrm{agg}}(L_{\ell-1}, \widetilde{X}_\ell)$ and $\mathsf{H}_{\mathrm{agg}}((z^{\ell-1}_{\mathrm{agg}})') = T'_{\mathrm{agg}}(L'_{\ell-1}, \widetilde{X}'_\ell)$, it follows that $L_{\ell-1} = L'_{\ell-1}$ and $\widetilde{X}_\ell = \widetilde{X}'_\ell$ because these arguments must have been assigned before the forking point. From the latter equality, we have that $\mathsf{KeyAgg}(L_\ell) = \mathsf{KeyAgg}(L'_\ell)$ and since $\neg\mathsf{KeyColl}$ is $\mathtt{true}$ in all executions, we must have that $L_\ell = L'_\ell$, and that by a similar argument $L_d = L'_d$ for all $d \geq \ell$. From $\widetilde{X}_\ell = \widetilde{X}'_\ell$ we also see that because $\neg\mathsf{BadOrder}'$ is $\mathtt{true}$ that $T_{\mathrm{agg}}(L_\ell, X_{i,\ell})$ must have been assigned before the forking point for all $X_{i,\ell} \in L_\ell$ so that $z^\ell_{\mathrm{agg}} = (z^\ell_{\mathrm{agg}})'$ and $\vec{a}_\ell = \vec{a}'_\ell$, and similarly for all $d \geq \ell$. Lastly, because we fork at the assignment of $\mathsf{H}_{\mathrm{agg}}(z^{\ell-1}_{\mathrm{agg}})$ which is assigned alongside all of the coefficients involved in $\mathsf{KeyAgg}(L_{\ell-1})$ and we are using the local forking lemma, which ensures that none of the other coefficients involved are changed, we can conclude that $\overline{a}_i = \overline{a}'_i$ for all i such that $X_{i,\ell-1} \neq \widetilde{X}_\ell$.

It remains to prove the bound on the accepting probability of $\mathcal{D}_\ell$. If we let ϵ be the advantage of $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu]}(\lambda)$, noting that $\mathcal{D}_{\ell-1}$ may trigger at most $2^\ell q$ random oracle assignments, then by induction on $\ell \geq 2$ we get

$$acc(\mathcal{D}_\ell) \geq \frac{acc(\mathcal{D}_{\ell-1})^2}{2^\ell q}$$

$$
\begin{aligned}
&\geq \frac{\left(\frac{\epsilon^{2^\ell}}{2^{2^\ell-\ell-1}q^{2^\ell-1}} - \frac{12}{2^\lambda}\right)^2}{2^\ell q} \\
&\geq \frac{\epsilon^{2^{\ell+1}}}{2^{2^{\ell+1}-\ell-2}q^{2^{\ell+1}-1}} - \frac{2 \cdot 12\epsilon^{2^\ell}}{(2^\ell q)(2^{2^\ell-\ell-1}q^{2^\ell-1})2^\lambda} \\
&\geq \frac{\epsilon^{2^{\ell+1}}}{2^{2^{\ell+1}-\ell-2}q^{2^{\ell+1}-1}} - \frac{12}{2^\lambda}.
\end{aligned}
$$

We check this inequality directly for $\ell = 1$ to establish the base case. We begin with a numerator of $12q$ in place of 12 when $\ell = 0$ from the previous lemma, so that by the first three of the above inequalities (with $\ell = 1$),

$$
acc(\mathcal{D}_1) \geq \frac{\epsilon^4}{2q^3} - \epsilon^2 \frac{2 \cdot 12q}{2q^2 2^\lambda} \geq \frac{\epsilon^4}{2q^3} - \frac{12}{2^\lambda}.
$$

□

We are now ready to prove Theorem 1, which we restate below for convenience, by constructing from $\mathcal{D}_{\Lambda-1}$ an algorithm $\mathcal{E}$ solving the AOMDL problem.

Theorem 1. *Let* GrGen *be a group generation algorithm for which the AOMDL problem is hard. Then the nested multi-signature scheme* $\mathsf{NestedMuSig2}[\mathsf{GrGen}, \nu = 2^{D+1}]$ *is EUF-CMA in the random oracle model for* $\mathsf{H}_{\mathrm{agg}}$, $\mathsf{H}_{\mathrm{non}}$, $\mathsf{H}_{\overline{\mathrm{non}}}$, $\mathsf{H}_{\mathrm{sig}} : \{0,1\}^* \to \mathbb{Z}_p$.

Precisely, for any adversary $\mathcal{A}$ *against* $\mathsf{NestedMuSig2}[\mathsf{GrGen}, \nu = 2^{D+1}]$ *running in time at most* t*, making at most* q_s OSign *queries and at most* q_h *queries to each random oracle, and such that the size of* L_d *in any signing session and in the forgery is at most* N*, and the maximum depth of any signing session is at most* D*, there exists an algorithm* $\mathcal{E}$ *taking as input group parameters* $(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$*, running in time at most*

$$
t' = 2^{D+1}(t + q(3N + 3\nu - 3))t_{\exp} + O(qN)
$$

where $q = (N+4)q_h + (DN + D + 1)q_s + DN + 1$ *and* $t_{\exp}$ *is the time of an exponentiation in* $\mathbb{G}$*, making at most* $2^{D+1}q_s$ DLog_g *queries, and solving the AOMDL problem with an advantage*

$$
\mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{E},\mathsf{GrGen}}(\lambda) \geq \frac{(\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu=2^{D+1}]}(\lambda))^{2^{D+1}}}{2^{2^{D+1}-D-2}q^{2^{D+1}-1}} - \frac{2^{D+2}q + 2\binom{2^{D+1}q+D}{D}^2 + 12}{2^\lambda}.
$$

Proof. Note that in all executions of $\mathcal{A}$ during forking in all of the lemmas above, $\mathcal{A}$ always returns a forgery at a fixed level of depth, $\Lambda \leq D$. We define the algorithm $\mathcal{E}$ to play the AOMDL game as follows. First, $\mathcal{E}$ makes $\nu q_s + 1$ queries to the challenge oracle $X^*, U_1, \ldots, U_{\nu q_s} \leftarrow \textsc{Ch}()$ and initializes a Boolean flag $\mathsf{LinDep} := \mathtt{false}$. Then, $\mathcal{E}$ runs $\mathsf{Fork}^{\mathcal{D}_{\Lambda-1}}$ with $inp = ((\mathbb{G}, p, g), X^*, U_1, \ldots, U_{\nu q_s})$, $z = z^{\Lambda-1}_{\mathrm{agg}}$, and $out = (L^{\Lambda-1}, \vec{a}_{\Lambda-1}, \tilde{x}_{\Lambda-1})$, but simulates the discrete log oracle to $\mathcal{D}_{\Lambda-1}$ by saving queries to a table before forwarding them to $\mathcal{E}$'s discrete log oracle and saving the responses to the table as well; if a query has been made before, $\mathcal{E}$ immediately returns the previous response without querying its own discrete log oracle. Additionally, note that discrete log queries made by $\mathsf{Fork}^{\mathcal{D}_{\Lambda-1}}$ will all be of the form $(Z, (0, \beta_1, 0, \ldots, 0, 1, \beta_{k'+1}, \ldots, \beta^{\nu-1}_{k'+1}, 0, \ldots, 0))$ (where $\beta_{k'+1}$ corresponds to a product of b_d values for some number of ds, which we denote $\check{b}$), and $\mathcal{E}$ is programmed to set $\mathsf{LinDep} := \mathtt{true}$ if it receives two non-equal discrete log queries such that $\beta_{k'+1}$ is equal between these two queries. We receive

$$
(z^{\Lambda-1}_{\mathrm{agg}}, (L_{\Lambda-1}, \vec{a}_{\Lambda-1}, \tilde{x}_{\Lambda-1}) := out, (L'_{\Lambda-1}, \vec{a}'_{\Lambda-1}, \tilde{x}'_{\Lambda-1}) := out')
$$

and then $\mathcal{E}$ computes

$$
x^* = (\tilde{x}_{\Lambda-1} - \tilde{x}'_{\Lambda-1})(n^*)^{-1}(a_{i^*,\Lambda-1} - a'_{i^*,\Lambda-1})^{-1} \bmod p,
$$

where n^* is the number of times X^* appears in $L_{\Lambda-1}$ and i^* is the index of X^* in $L_{\Lambda-1}$. Next, $\mathcal{E}$ asserts that LinDep is $\mathtt{false}$ and returns $\perp$ if this assertion fails. Finally, for each $k \in \{1, \ldots, q_s\}$,

$\mathcal{E}$ constructs a system of linear equations for the unknowns $r_j = \text{DLOG}_g(U_{\nu(k-1)+j})$ where each equation is of the form $\sum_{j=1}^{\nu} \check{b}^{j-1} r_j = s_1 - \check{c}x^*$ (where $\mathcal{E}$ knows all of the scalar coefficients since they were given as input in discrete log queries). If two discrete log queries made in executions of signing session k are ever identical, then this may result in fewer than ν equations. If we have fewer than ν equations for a value of k, $\mathcal{E}$ chooses a value of $\beta_{\nu(k-1)+2}$ that is not equal to any other value of $\beta_{k'+1} = \check{b}$ and computes $Z := \prod_{j=1}^{\nu} R_{1,j}^{\beta_{\nu(k-1)+2}^{j-1}}$ and makes a query for this point using the discrete log oracle, appends this new equation to its system and repeats until there are ν equations. This system of equations is represented by a matrix of the form

$$B = \begin{pmatrix} 1 & (\check{b}^{(1)})^1 & \cdots & (\check{b}^{(1)})^{\nu-1} \\ 1 & (\check{b}^{(2)})^1 & \cdots & (\check{b}^{(2)})^{\nu-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & (\check{b}^{(\nu)})^1 & \cdots & (\check{b}^{(\nu)})^{\nu-1} \end{pmatrix}.$$

This is a square Vandermonde matrix, where the value of $\check{b}$ in each row is distinct, and thus it is guaranteed to have full rank (ν). The algorithm $\mathcal{E}$ solves each of these systems of equations to obtain the discrete logs for $U_1, \ldots, U_{\nu q_s}$ and returns these appended to x^* to solve the AOMDL problem.

To see that we have made exactly νq_s discrete log queries, consider signing query k in two executions that have run identically at all times before the later of the assignments of the $b_d = T_{\text{non}}(\widetilde{X}_d, (R'_{1,d}, \ldots, R'_{\nu,d}))$ and $b_0 = T_{\overline{\text{non}}}(\widetilde{X}, (R_{1,0}, \ldots, R_{\nu,0}), m)$ that are factors in the product defining $\check{b}^{(k)}$, i.e., these two executions have not forked from each other before all of these values being assigned. We will notate the variables in one of these runs normally, and the variables in the other run with a "prime". Clearly, since the assignment of a value to T_{non} or $T_{\overline{\text{non}}}$ is never used as a forking point, we have that $\check{b} = \check{b}'$. Additionally, the input keys to these tables must be identically assigned before any forking so that $\widetilde{X} = \widetilde{X}'$ and $R_{j,0} = R'_{j,0}$ for all j, which in turn implies that $R = R'$. Since the assignment of $c := \mathsf{H}_{\text{sig}}(\widetilde{X}, R, m)$ cannot correspond to a forking point because valid forgeries cannot use keysets and messages previously queried together, we can conclude that $c = c'$ as well. Lastly, since $\widetilde{X}$ and all intermediate aggregate keys $\widetilde{X}_d$ have been assigned before any forking point, and in all executions we have that $\neg\mathsf{BadOrder} \wedge \neg\mathsf{BadOrder}' \wedge \neg\mathsf{KeyColl}$, we can conclude that $\vec{a}_d = \vec{a}'_d$ for all d because all of these values must have been assigned before $\widetilde{X}$ was assigned. Hence, these two executions are guaranteed to yield identical queries to the discrete log oracle during the execution of signing query k. Therefore, because LinDep is `false`, the value of $\check{b}$ is a unique identifier in each signing query so that either a new value is used resulting in a discrete log query, or an old one is used in which case a cached discrete log value is used so that a total of ν discrete log queries are issued by $\mathcal{E}$ for each of the q_s signing sessions.

Because both executions of $\mathcal{D}_{\Lambda-1}$ are identical within $\mathcal{E}$ up until the assignment of $\mathsf{H}_{\text{agg}}(z_{\text{agg}}^{\Lambda-1}) = T_{\text{agg}}(L_{\Lambda-1}, X^*)$ and $\mathsf{H}_{\text{agg}}((z_{\text{agg}}^{\Lambda-1})') = T'_{\text{agg}}(L'_{\Lambda-1}, X^*)$, it follows that $L_{\Lambda-1} = L'_{\Lambda-1}$ because this argument must have been assigned before the forking point. We also know that all of the other coefficients $a_{i,\Lambda-1} = \mathsf{H}_{\text{agg}}(L_{\Lambda-1}, X_{i,\Lambda-1})$ for all i such that $X_i \neq X^*$ are the same between the two executions since we are using the local forking lemma. This gives us that

$$g^{\tilde{x}_{\Lambda-1}} = \prod_{i=1}^{n_{\Lambda-1}} (X_{i,\Lambda-1})^{a_{i,\Lambda-1}} = (X^*)^{n^* a_{i^*,\Lambda-1}} \prod_{X_{i,\Lambda-1} \neq X^*} X_{i,\Lambda-1}^{a_{i,\Lambda-1}}$$

$$g^{\tilde{x}'_{\Lambda-1}} = \prod_{i=1}^{n_{\Lambda-1}} (X'_{i,\Lambda-1})^{a'_{i,\Lambda-1}} = (X^*)^{n^* a'_{i^*,\Lambda-1}} \prod_{X_{i,\Lambda-1} \neq X^*} X_{i,\Lambda-1}^{a_{i,\Lambda-1}}$$

from which we can conclude that $g^{x^*} = X^*$, that is, x^* is the discrete log of X^* in base g.

Ignoring the time taken to compute discrete logarithms and to solve systems of linear equations, the running time, t', of $\mathcal{E}$ is 2^{D+1} times the runtime, t, of $\mathcal{A}$ in addition to the time taken to maintain tables and answer hash and signing queries. Each of the 7 tables has size at most qN. The runtime of signing queries is dominated by the two calls to KeyAgg and product computations for R and $\hat{R}_1$, which require at most N and $(\nu - 1)$ exponentiations, respectively. Similarly, the runtime of

hash queries is also dominated by calls to KeyAgg (in the case of H_{agg}) or the product computation of R (in the case of $\mathsf{H}_{\overline{\text{non}}}$), so that there are at most $(N+\nu-1)$ exponentiations executed per hash query. Thus, the runtime of $\mathcal{E}$ is bounded by $t' = 2^{D+1}(t + q(3N+3\nu-3))t_{\text{exp}} + O(qN)$.

We now bound the accepting probability of $\mathcal{E}$. If we let $\epsilon = \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu]}(\lambda)$, then the logic from the previous lemma and that $\Lambda \leq D$ gives us that

$$acc(\mathsf{Fork}^{\mathcal{D}_{\Lambda-1}}) \geq \frac{\epsilon^{2^{D+1}}}{2^{2^{D+1}-D-2}q^{2^{D+1}-1}} - \frac{12}{2^\lambda}.$$

By the union bound, $\mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{E}}(\lambda) \geq acc(\mathsf{Fork}^{\mathcal{D}_{\Lambda-1}}) - \Pr[\mathsf{LinDep}]$. To bound $\Pr[\mathsf{LinDep}]$, we break up into two cases depending on whether any of the $2^{D+1}q$ possible values of T_{non} or $T_{\overline{\text{non}}}$ is equal to 0. That is to say, we use the fact that

$$\begin{aligned}\Pr[\mathsf{LinDep}] &= \Pr[\mathsf{LinDep} \mid \exists 0] \cdot \Pr[\exists 0] + \Pr[\mathsf{LinDep} \mid \neg\exists 0] \cdot \Pr[\neg\exists 0] \\ &< \Pr[\exists 0] + \Pr[\mathsf{LinDep} \mid \neg\exists 0].\end{aligned}$$

Since the $2^{D+1}q$ values are chosen independently and uniformly at random from a set of size $p \geq 2^{\lambda-1}$, we have that $\Pr[\exists 0] \leq \frac{2^{D+1}q}{p} \leq \frac{2^{D+2}q}{2^\lambda}$. Now, if we have that all of the values of T_{non} and $T_{\overline{\text{non}}}$ are assigned uniformly at random from $\mathbb{Z}_p^*$ (i.e., $\mathbb{Z}_p$ without 0), then $\check{b}$ now has a uniform distribution (since $\mathbb{Z}_p^*$ is a cyclic group) so that $\Pr[\mathsf{LinDep} \mid \neg\exists 0]$ is bounded by the probability of a duplicate existing in a list corresponding to all possible products of all sizes bounded by D of non-zero bs. Note that after at most $2^{D+1}q$ queries there are at most $\binom{2^{D+1}q}{j}$ possible products of j hash values so that the total number of possible $\check{b}$ values is $\sum_{j=1}^{D}\binom{2^{D+1}q}{j} \leq \binom{2^{D+1}q+D}{D}$ so that the probability of a collision occuring between $\check{b}$ values is bounded by $\frac{\binom{2^{D+1}q+D}{D}^2}{2} \cdot \frac{1}{p-1}$. Thus,

$$\Pr[\mathsf{LinDep} \mid \neg\exists 0] \leq \frac{\binom{2^{D+1}q+D}{D}^2}{2(2^{\lambda-1}-1)} < \frac{2\binom{2^{D+1}q+D}{D}^2}{2^\lambda}.$$

Therefore,

$$\mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{E}}(\lambda) \geq \frac{\left(\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu=2^{D+1}]}(\lambda)\right)^{2^{D+1}}}{2^{2^{D+1}-D-2}q^{2^{D+1}-1}} - \frac{2^{D+2}q + 2\binom{2^{D+1}q+D}{D}^2 + 12}{2^\lambda}.$$

□

6 Security of NestedMuSig2 in the ROM + AGM

6.1 Security Proof

In the algebraic group model (AGM) [FKL18], adversaries are assumed to be algebraic. An algorithm, $\mathcal{A}$, is algebraic with respect to some group description, $(\mathbb{G}, p, g)$, if for all group elements, Z, that it outputs (including as input to random oracle or signing queries), it also provides a representation of Z relative to all group elements previously received from its environment. We write $[Z]$ for a group element, Z, augmented with its representation (which is a list of exponents). When the adversary returns a multiset of group elements, L, we write $[L]$ for the multiset of augmented group elements. The result of multiplying and exponentiating augmented group elements is also an augmented group element. Furthermore, all non-index and non-counter integer (i.e., scalar) operations are performed modulo p. Otherwise, we use the same normalizing assumptions and conventions as in the previous section.

By assuming the ROM and the AGM, we can show the tight security of $\mathsf{NestedMuSig2}$ with only $\nu = 2$ nonces.

PROOF OVERVIEW. Unlike the security proof of Schnorr signatures in the AGM, our reduction for NestedMuSig2 does not know the discrete logs of all group elements it provides to the adversary. Namely, it does not know the discrete logs of the values provided by the AOMDL challenge oracle. Thus, it is not possible to simply extract the honest signer's private key from the forgery, and we must resort to the following approach.

We simulate the honest signer in this proof just as we did in the non-AGM proof of the previous section. Namely, we respond to OSIGN queries by calling the AOMDL challenge oracle, and we respond to OSIGN$'$ queries by calling the AOMDL DLOG_g oracle. Because we do not need to fork, only $\nu = 2$ nonces are required. Instead of extracting by forking, we keep track of all group element representations given by the adversary, and construct a system of $q_s + 1$ equations. One equation is derived from every DLOG_g query during each signing session (of which there are q_s in total), and one equation is derived from the verification equation of the forgery paired with the adversary's representation of the nonce used in the forgery. Finally, we argue that this system of equations can be solved with overwhelming probability using our remaining DLOG_g queries. This argument shows that if our random oracle assignments avoid certain values (determined before the assignments are made), then the system of equations just described is non-degenerate. We make this formal by introducing assertions into the random oracle programming that only fail with negligible probability.

Theorem 2. *Let* GrGen *be a group generation algorithm for which the AOMDL problem is hard. Then the multi-signature scheme* $\mathsf{NestedMuSig2}[\mathsf{GrGen}, \nu = 2]$ *is EUF-CMA in the algebraic group model for* GrGen*, and the random oracle model for* $\mathsf{H}_{\mathrm{agg}}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\overline{\mathrm{non}}}, \mathsf{H}_{\mathrm{sig}} : \{0,1\}^* \to \mathbb{Z}_p$.

Precisely, for any algebraic adversary $\mathcal{A}$ *against* $\mathsf{NestedMuSig2}[\mathsf{GrGen}, \nu = 2]$ *running in time at most* t*, making at most* q_s OSIGN *queries, at most* q_h *queries to each random oracle, such that the size of* L *in any signing session and in the forgery is at most* N*, and such that the depth,* Λ*, of any signing session or forgery is at most* D*, there exists an algorithm* $\mathcal{B}$ *running in time at most*

$$t' = t + O(qDN) \cdot t_{\exp} + O(q^2),$$

where $q = \binom{2q_h + Dq_s + D}{D} + DN(q_s + 1) + Nq_h$ *and* $t_{\exp}$ *is the time of an exponentiation in* $\mathbb{G}$ *and making at most* $2q_s$ DLOG_g *queries such that*

$$\mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{B},\mathsf{GrGen}}(\lambda) \geq \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu=2]}(\lambda) - \frac{30q^4}{2^\lambda}.$$

Proof. Consider $\mathcal{B}$ given in Figure 10, which reduces a $\mathsf{NestedMuSig2}[\mathsf{GrGen}, \nu = 2]$ forger to an AOMDL attacker, simulating the OSIGN and OSIGN$'$ oracles to $\mathcal{A}$ in the usual way.

It remains to describe how the subroutine $\mathsf{Solve}^{\mathrm{DLOG}}$ works, and bound the probability that it returns $\perp$. First, for every $X_{i,d} \in L_d$ we retrieve $T_{aggrep}(L_d, i)$ giving us

$$X_{i,d} = g^{\alpha_{i,d}} (X^*)^{\beta_{i,d}} \prod_{k=1}^{q_s} (R^{(k)}_{1,1,\Lambda^{(k)}})^{\gamma_{i,1,k,d}} (R^{(k)}_{1,2,\Lambda^{(k)}})^{\gamma_{i,2,k,d}}.$$

For purposes of reasoning about which values are independent of which others (generally due to the order of a value's assignment), it is important that we retrieve the representation of $X_{i,d}$ from its first call to $\mathsf{H}_{\mathrm{agg}}$ rather than what might be returned by the adversary in the forgery. We have $\widetilde{X}_\Lambda = X^*$ and then for each $0 \leq d < \Lambda$ we compute a representation,

$$\widetilde{X}_d = g^{\tilde{\alpha}_d} (X^*)^{\tilde{\beta}_d} \prod_{k=1}^{q_s} (R^{(k)}_{1,1,\Lambda^{(k)}})^{\tilde{\gamma}_{1,k,d}} (R^{(k)}_{1,2,\Lambda^{(k)}})^{\tilde{\gamma}_{2,k,d}},$$

recursively as follows:

$$\begin{aligned}
\widetilde{X}_d &= X_{1,d-1} \\
&= (X_{1,d}^{a_{1,d}}) \prod_{i=2}^{n} X_{i,d}^{a_{i,d}} \\
&= (\widetilde{X}_{d+1}^{a_{1,d}}) g^{\alpha_d} (X^*)^{\beta_d} \prod_{k=1}^{q_s} (R^{(k)}_{1,1,\Lambda^{(k)}})^{\gamma_{1,k,d}} (R^{(k)}_{1,2,\Lambda^{(k)}})^{\gamma_{2,k,d}} \\
&= g^{\alpha_d + a_{1,d}\tilde{\alpha}_{d+1}} (X^*)^{\beta_d + a_{1,d}\tilde{\beta}_{d+1}} \prod_{k=1}^{q_s} (R^{(k)}_{1,1,\Lambda^{(k)}})^{\gamma_{1,k,d} + a_{1,d}\tilde{\gamma}_{1,k,d+1}} (R^{(k)}_{1,2,\Lambda^{(k)}})^{\gamma_{2,k,d} + a_{1,d}\tilde{\gamma}_{2,k,d+1}},
\end{aligned}$$

$\mathcal{B}^{\mathcal{A},\text{CH},\text{DLOG}}(\lambda,(\mathbb{G},p,g))$

$X_1 := X^* \leftarrow \text{CH}()$
$ctrs := 0\,;\ S := \emptyset\,;\ Q := \emptyset\,;\ T_{eq} := \emptyset$
$T_{\text{agg}} := \emptyset\,;\ T_{\text{non}} := \emptyset\,;\ T_{\overline{\text{non}}} := \emptyset\,;\ T_{\text{sig}} := \emptyset$
$T_{\text{akey}} := \emptyset\,;\ T_{\text{sigrep}} := \emptyset\,;\ T_{aggrep} := \emptyset$
$((L_d)_{0\le d<\Lambda}, m, ([R], s))$
$\quad \leftarrow \mathcal{A}^{\text{OSIGN},\text{OSIGN}',\mathsf{H}_{\text{agg}},\mathsf{H}_{\text{non}},\mathsf{H}_{\overline{\text{non}}},\mathsf{H}_{\text{sig}}}(X_1)$
assert $X^* \in L_{\Lambda-1}$
$[\widetilde{X}_{\Lambda-1}] := \mathsf{KeyAgg}([L_{\Lambda-1}])$
for $d := \Lambda-2,\dots,0$ **do**
  assert $\widetilde{X}_{d+1} \in L_d\,;\ [\widetilde{X}_d] := \mathsf{KeyAgg}([L_d])$
$c := \mathsf{H}_{\text{sig}}([\widetilde{X}_0],[R],m)$
assert $((L_d)_{0\le d<\Lambda}, m) \notin Q \wedge g^s = R\widetilde{X}_0^c$
$(r_1^{(1)},\dots,r_2^{(q_s)},x^*) \leftarrow \mathsf{Solve}^{\text{DLOG}}((L_d)_{0\le d<\Lambda},R,s)$
return $(x^*, r_1^{(1)},\dots,r_2^{(q_s)})$

$\text{OSIGN}(\Lambda^{(k)})$

$ctrs := ctrs+1\,;\ k := ctrs\,;\ S := S \cup \{(k,\Lambda^{(k)})\}$
$R^{(k)}_{1,1,\Lambda^{(k)}} \leftarrow \text{CH}()\,;\ R^{(k)}_{1,2,\Lambda^{(k)}} \leftarrow \text{CH}()$
return $(R^{(k)}_{1,1,\Lambda^{(k)}}, R^{(k)}_{1,2,\Lambda^{(k)}})$

$\text{OSIGN}'(k,([R^{(k)}_{1,d}],[R^{(k)}_{2,d}])_{0\le d<\Lambda}, m^{(k)}, ([X^{(k)}_{i,d}])^{0\le d<\Lambda}_{2\le i\le n_d})$

assert $(k,\Lambda) \in S\,;\ S := S \setminus \{(k,\Lambda)\}$
$[X^{(k)}_{1,\Lambda-1}] := (X^*, 0, 1, (0,0)_{1\le k\le q_s})$ // trivial rep.
for $d := \Lambda-1,\dots,0$ **do**
  $[L^{(k)}_d] := \{[X^{(k)}_{1,d}],[X^{(k)}_{2,d}],\dots,[X^{(k)}_{n_d,d}]\}$
  $a^{(k)}_{1,d} := \mathsf{H}_{\text{agg}}([L^{(k)}_d],[X^{(k)}_{1,d}])$
  $[X_{1,d-1}] := \mathsf{KeyAgg}([L^{(k)}_d])$
$[\widetilde{X}^{(k)}] := [X^{(k)}_{1,-1}]$
for $d := 1,\dots,\Lambda-1$
  $b^{(k)}_d := \mathsf{H}_{\text{non}}([X^{(k)}_{1,d}],([R^{(k)}_{1,d}],[R^{(k)}_{2,d}]))$
$b^{(k)}_0 := \mathsf{H}_{\overline{\text{non}}}([\widetilde{X}^{(k)}],([R^{(k)}_{1,0}],[R^{(k)}_{2,0}]),m^{(k)})$
$\check{b}^{(k)} := \prod_{d=0}^{\Lambda} b^{(k)}_d\,;\ [R^{(k)}] := [R^{(k)}_{1,0}][R^{(k)}_{2,0}]^{\check{b}^{(k)}}$
$c^{(k)} := \mathsf{H}_{\text{sig}}([\widetilde{X}^{(k)}],[R^{(k)}],m^{(k)})$
$\check{c}^{(k)} := c^{(k)} \prod_{d=0}^{\Lambda-1} a^{(k)}_{1,d}$
$s^{(k)}_1 := \text{DLOG}_g([R^{(k)}_{1,1,\Lambda}][R^{(k)}_{1,2,\Lambda}]^{\check{b}^{(k)}}[X^{(k)}_{1,\Lambda}]^{\check{c}^{(k)}})$
$T_{eq}(k) := (s^{(k)}_1, \check{b}^{(k)}, \check{c}^{(k)})$
$Q := Q \cup \{((L^{(k)}_d)_{0\le d\le\Lambda}, m^{(k)})\}$
return $s^{(k)}_1$

$\mathsf{H}_{\text{agg}}([L],[X])$

// $X \in L$ by assumption
if $T_{\text{agg}}(L,X) = \perp$ **then**
  for $X' \in L$ **do**
    $T_{\text{agg}}(L,X') \leftarrow\!\$\ \mathbb{Z}_p$
  $\{[X_1],\dots,[X_n]\} := [L]$
  for $i := 1\dots n$ **do**
    $(X_i,\alpha_i,\beta_i,(\gamma_{i,1,k},\gamma_{i,2,k})_{1\le k\le q_s}) := [X_i]$
    $T_{aggrep}(L,i) :=$
      $(\alpha_i,\beta_i,(\gamma_{i,1,k},\gamma_{i,2,k})_{1\le k\le q_s})$
  for $i := 1\dots n$ **do**
    $a_i := T_{\text{agg}}(L,X_i)$
  $\widetilde{X} := \prod_{i=1}^n X_i^{a_i}$
return $T_{\text{agg}}(L,X)$

$\mathsf{H}_{\text{sig}}([\widetilde{X}],[R],m)$

if $T_{\text{sig}}(\widetilde{X},R,m) = \perp$ **then**
  $T_{\text{sig}}(\widetilde{X},R,m) \leftarrow\!\$\ \mathbb{Z}_p$
  $(R,\alpha,\beta,(\gamma_{1,k},\gamma_{2,k})_{1\le k\le q_s}) := [R]$
  $T_{\text{sigrep}}(\widetilde{X},R,m) := (\alpha,\beta,(\gamma_{1,k},\gamma_{2,k})_{1\le k\le q_s})$
return $T_{\text{sig}}(\widetilde{X},R,m)$

$\mathsf{H}_{\text{non}}([\widetilde{X}],([R_1],[R_2]))$

if $T_{\text{non}}(\widetilde{X},(R_1,R_2)) = \perp$ **then**
  $T_{\text{non}}(\widetilde{X},(R_1,R_2)) \leftarrow\!\$\ \mathbb{Z}_p$
return $T_{\text{non}}(\widetilde{X},(R_1,R_2))$

$\mathsf{H}_{\overline{\text{non}}}([\widetilde{X}],([R_1],[R_2]),m)$

if $T_{\overline{\text{non}}}(\widetilde{X},(R_1,R_2),m) = \perp$ **then**
  $T_{\overline{\text{non}}}(\widetilde{X},(R_1,R_2),m) \leftarrow\!\$\ \mathbb{Z}_p$
  $b := T_{\overline{\text{non}}}(\widetilde{X},(R_1,R_2),m)$
  $[R] := [R_1][R_2]^b$
  $c \leftarrow \mathsf{H}_{\text{sig}}([\widetilde{X}],[R],m)$
return $T_{\overline{\text{non}}}(\widetilde{X},(R_1,R_2),m)$

Fig. 10. Game used in the proof of Theorem 2.

where

$$\begin{aligned}
\alpha_d &:= \sum_{i=2}^{n} a_{i,d}\alpha_{i,d} \bmod p,\\
\beta_d &:= \sum_{i=2}^{n} a_{i,d}\beta_{i,d} \bmod p,\\
\gamma_{1,k,d} &:= \sum_{i=2}^{n} a_{i,d}\gamma_{i,1,k,d} \bmod p \text{ for all } k \in \{1,\dots,q_s\},\\
\gamma_{2,k,d} &:= \sum_{i=2}^{n} a_{i,d}\gamma_{i,2,k,d} \bmod p \text{ for all } k \in \{1,\dots,q_s\}.
\end{aligned}$$

This implies that we have for all $k \in \{1,\dots,q_s\}$ that $\tilde{\alpha}_\Lambda = \tilde{\gamma}_{1,k,\Lambda} = \tilde{\gamma}_{2,k,\Lambda} = 0$, $\tilde{\beta}_\Lambda = 1$, and for $0 \le d < \Lambda$ the recursions

$$\begin{aligned}
\tilde{\alpha}_d &= \alpha_d + a_{1,d}\tilde{\alpha}_{d+1},\\
\tilde{\beta}_d &= \beta_d + a_{1,d}\tilde{\beta}_{d+1},\\
\tilde{\gamma}_{1,k,d} &= \gamma_{1,k,d} + a_{1,d}\tilde{\gamma}_{1,k,d+1},\\
\tilde{\gamma}_{2,k,d} &= \gamma_{2,k,d} + a_{1,d}\tilde{\gamma}_{2,k,d+1}.
\end{aligned}$$

A simple induction shows that, using the convention that the empty product is equal to 1,

$$\begin{aligned}
\tilde{\alpha}_d &= \sum_{j=d}^{\Lambda} \alpha_j \prod_{\ell=d}^{j-1} a_{1,\ell},\\
\tilde{\beta}_d &= \sum_{j=d}^{\Lambda} \beta_j \prod_{\ell=d}^{j-1} a_{1,\ell},\\
\tilde{\gamma}_{1,k,d} &= \sum_{j=d}^{\Lambda} \gamma_{1,k,j} \prod_{\ell=d}^{j-1} a_{1,\ell},\\
\tilde{\gamma}_{2,k,d} &= \sum_{j=d}^{\Lambda} \gamma_{2,k,j} \prod_{\ell=d}^{j-1} a_{1,\ell}.
\end{aligned}$$

In particular, we have the representation

$$\widetilde{X} = \widetilde{X}_0 = g^{\tilde{\alpha}_0}(X^*)^{\tilde{\beta}_0} \textstyle\prod_{k=1}^{q_s} (R^{(k)}_{1,1,\Lambda^{(k)}})^{\tilde{\gamma}_{1,k,0}} (R^{(k)}_{1,2,\Lambda^{(k)}})^{\tilde{\gamma}_{2,k,0}}.$$

We retrieve the representation of R given during the first call to $\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$ by looking up $(\alpha, \beta, (\gamma_{1,k}, \gamma_{2,k})_{1\le k\le q_s}) := T_{\mathrm{sigrep}}(\widetilde{X}, R, m)$. Thus we have

$$R = g^{\alpha}(X^*)^{\beta} \prod_{k=1}^{q_s} (R^{(k)}_{1,1})^{\gamma_{1,k}} (R^{(k)}_{1,2})^{\gamma_{2,k}}.$$

Combining our representations for R and $\widetilde{X}$, along with the verification equation $g^s = R\widetilde{X}^c$, this gives us that

$$g^s = g^{\alpha + c\tilde{\alpha}_0}(X^*)^{\beta + c\tilde{\beta}_0} \textstyle\prod_{k=1}^{q_s} (R^{(k)}_{1,1,\Lambda^{(k)}})^{\gamma_{1,k} + c\tilde{\gamma}_{1,k,0}} (R^{(k)}_{1,2,\Lambda^{(k)}})^{\gamma_{2,k} + c\tilde{\gamma}_{2,k,0}}.$$

Additionally, each of the q_s entries of T_{eq} give us an equation of the form

$$g^{s_1^{(k)}} = R^{(k)}_{1,1,\Lambda^{(k)}} (R^{(k)}_{1,2,\Lambda^{(k)}})^{\check{b}^{(k)}} (X^*)^{\check{c}^{(k)}}.$$

Altogether, we have obtained a system of $q_s + 1$ equations in the unknowns x^* and $r_j^{(k)} = \text{DLog}_g(R_{1,j,\Lambda^{(k)}}^{(k)})$. Ordering these variables $r_1^{(1)}, \ldots, r_1^{(q_s)}, r_2^{(1)}, \ldots, r_2^{(q_s)}, x^*$, this system of equations has the coefficient matrix:

$$M = \begin{pmatrix} 1 & \cdots & 0 & \check{b}^{(1)} & \cdots & 0 & \check{c}^{(1)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & \check{b}^{(q_s)} & \check{c}^{(q_s)} \\ \delta_{1,1} & \cdots & \delta_{1,q_s} & \delta_{2,1} & \cdots & \delta_{2,q_s} & \delta_0 \end{pmatrix}$$

where for $k \in \{1, \ldots, q_s\}$,

$$\begin{aligned} \delta_{1,k} &= \gamma_{1,k} + c\tilde{\gamma}_{1,k,0}, \\ \delta_{2,k} &= \gamma_{2,k} + c\tilde{\gamma}_{2,k,0}, \text{ and} \\ \delta_0 &= \beta + c\tilde{\beta}_0. \end{aligned}$$

Subtracting $\delta_{1,k}$ times the k-th row from the last row, for $k = 1, \ldots, q_s$, yields

$$M' = \begin{pmatrix} 1 & \cdots & 0 & \check{b}^{(1)} & \cdots & 0 & \check{c}^{(1)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & \check{b}^{(q_s)} & \check{c}^{(q_s)} \\ 0 & \cdots & 0 & \eta_1 & \cdots & \eta_{q_s} & \eta_0 \end{pmatrix}$$

where

$$\begin{aligned} \eta_k &= \delta_{2,k} - \check{b}^{(k)}\delta_{1,k} \quad \text{for } k \in \{1, \ldots, q_s\}, \\ \eta_0 &= \delta_0 - \sum_{k=1}^{q_s} \check{c}^{(k)}\delta_{1,k}. \end{aligned}$$

If all of the values $\eta_0, \eta_1, \ldots, \eta_{q_s}$ are congruent to 0, then Solve returns $\perp$ (the probability of this occurrence is what we must bound). Otherwise, let k^* be the first positive index such that $\eta_{k^*} \neq 0$ or let $k^* = 0$ if η_0 is the only non-zero value. Then we can easily solve our system of equations by invoking $\text{DLog}_g([R_{1,2,\Lambda^{(k)}}^{(k)}])$ for all $k \neq k^*$ in $\{1, \ldots, q_s\}$, and in the case that $k^* \neq 0$, also calling $\text{DLog}_g([X^*])$. This is a total of q_s additional queries to the discrete log oracle (bringing $\mathcal{B}$'s total to $2q_s$). We can then use these values to eliminate all but the entry η_{k^*} of the last row of M', which allows us to solve for $\text{DLog}_g(R_{1,2,\Lambda^{(k^*)}}^{(k^*)})$ (or $\text{DLog}_g(X^*)$ if $k^* = 0$) without making an oracle query. After all of these values are known, it is easy to eliminate all non-diagonal entries of M', resulting in a solution for the values $R_{1,1,\Lambda^{(k)}}^{(k)}$ for all $k \in \{1, \ldots, q_s\}$.

All that remains is to bound the probability that $\eta_k = 0$ for all $k \in \{0, 1, \ldots, q_s\}$. We accomplish this by discovering a set of "collisions" involving hash values such that we can assure Solve does not return $\perp$ if none of these "collisions" occur.

At a very high proof-sketch-level, in the "normal" situation where c (the signature hash for the forgery) is instantiated last (i.e., subsequently to $\check{b}^{(k)}$), and where the $a_{1,d}$, which are key aggregation coefficients in the forgery, are also instantiated after every $\check{b}^{(k)}$, then η_0 has negligible chance of being 0 since c is chosen uniformly at random and independent of all of the other values appearing in η_0. We must also argue that the net effect of c in η_0 is not multiplied by 0 except with negligible probability; this is where we use the assumption that the $a_{1,d}$ are assigned after every $\check{b}^{(k)}$. Next, in the "strange" case where there is some k such that $\check{b}^{(k)}$ is assigned in between $a_{1,d}$ and c, then we argue that $\eta_k = c \cdot \theta_k + \theta'_k$, where θ_k has a negligible chance of being 0 and c is independent of both θ_k and θ'_k making the probability that $\eta_k = 0$ negligible. Finally, in the other "strange" cases where $\check{b}^{(k)}$ is assigned after c, then we argue that $\eta_k = \delta_{2,k} - \check{b}^{(k)} \cdot \delta_{1,k}$ has a negligible chance of being 0 since we will assert that all nonce hash values are non-zero so that $\check{b}^{(k)}$ is assigned to be uniformly at random and independent of all other values (again, arguing that $\delta_{1,k}$ is equal to 0 with only negligible probability).

The motivation for breaking into cases in this way is that we are generalizing the proof that works in the case that the adversary does not make any signing queries (i.e., $q_s = 0$), in which case our only option is to argue that $\eta_0 \neq 0$ except with negligible probability. This is a sense in which the $a_{1,d}$ followed by c being the last relevant hash values assigned is the generic case.

To this end, we begin by expanding the definition of η_0 so as to isolate c, and then to isolate the $a_{1,d}$ sequentially within the coefficient of c,

$$\begin{aligned}
\eta_0 &= \beta + c\tilde{\beta}_0 - \sum_{k=1}^{q_s} \check{c}^{(k)}(\gamma_{1,k} + c\tilde{\gamma}_{1,k,0}) \\
&= c\theta_0 + \beta - \sum_{k=1}^{q_s} \check{c}^{(k)}\gamma_{1,k}, \text{ where} \\
\theta_0 &= \tilde{\beta}_0 - \sum_{k=1}^{q_s} \check{c}^{(k)}\tilde{\gamma}_{1,k,0} \\
&= \sum_{j=0}^{\Lambda} \beta_j \prod_{d=0}^{j-1} a_{1,d} - \sum_{k=1}^{q_s} \check{c}^{(k)} \sum_{j=0}^{\Lambda} \gamma_{1,k,j} \prod_{d=0}^{j-1} a_{1,d}.
\end{aligned}$$

Sequentially isolating $a_{1,d}$ terms within θ_0 yields the recursive definition $\theta_{0,\Lambda} = 1$, and for all $0 \leq d < \Lambda$,

$$\theta_{0,d} = a_{1,d}\theta_{0,d+1} + \beta_d - \sum_{k=1}^{q_s} \check{c}^{(k)}\gamma_{1,k,d},$$

where $\theta_0 = \theta_{0,0}$. If we let

$$\begin{aligned}
K_{-1} &:= \{k \in \{1, \ldots, q_s\} \mid \gamma_{1,k} \neq 0\}, \text{ and} \\
K_d &:= \{k \in \{1, \ldots, q_s\} \mid \exists i \in \{2, \ldots, n_d\}, \gamma_{i,1,k,d} \neq 0\} \text{ for all } d \in \{0, \ldots, \Lambda\},
\end{aligned}$$

then we can rewrite η_0 as

$$\begin{aligned}
\eta_0 &= c\theta_0 + \beta - \sum_{k \in K_{-1}} \check{c}^{(k)}\gamma_{1,k}, \text{ where} \\
\theta_0 &= \theta_{0,0}, \text{ where} \\
\theta_{0,d} &= a_{1,d}\theta_{0,d+1} + \beta_d - \sum_{k \in K_d} \check{c}^{(k)}\gamma_{1,k,d} \text{ and } \theta_{0,\Lambda} = 1.
\end{aligned}$$

In the reasoning that follows, we collect a list of assertions that we can make, which fail with negligible probability, such that if all assertions succeed, then Solve will always succeed in Figure 10. Note that unlike in non-nested MuSig2, because H_{agg} is not exclusively used at a single level we cannot assume that $X^* \in L$ in calls to H_{agg}, and thus we cannot assume that the relevant key is at index 1. To combat this, in all assertions made below, we loop through all possible choices of assigning index 1 to each key in L. Additionally, unlike non-nested MuSig2, we do not know which depth level, Λ, the forgery will eventually be for, so we must assert for all of the following cases for all possible values of $\Lambda \in \{0, \ldots, D\}$. We now break into cases depending on the order of hash value assignments (noting that we will assert that freshly aggregated keys have not been used in queries to other oracles, which will guarantee that $a_{1,d+1}$ is generated before $a_{1,d}$, which is generated before c):

1. The case that $\check{b}^{(k)}$ is assigned before $a_{1,d}$ for all $d \in \{0, \ldots, \Lambda - 1\}$ and all $k \in \bigcup_{\ell=d}^{\Lambda-1} K_\ell$, and $\check{b}^{(k)}$ is assigned before c for all $k \in \bigcup_{\ell=-1}^{\Lambda-1} K_\ell$.
   We will deal with this case last. The reason this case ends up being so difficult is that to make an assertion like "if $\theta_0 \neq 0$ then $\eta_0 \neq 0$," we need to have the ability to look up the values $\check{c}^{(k)}$ for many values of k because they appear in the definitions of θ_0 and η_0. However, our case premises do not help us to know what these values may be, since all hash values may be initialized by the adversary well before relevant calls are made to OSign′. Furthermore, we

cannot just try to find every collection of q_s b-value entries of T_{non} because then there is no guarantee that $\binom{q_h}{q_s}$ is poly-bounded (for example, this value grows exponentially in q_s when $q_h = 2q_s$). Thus, we must do extra case-work in which we will relate $\check{b}^{(k)}$ to the known γ values by some equations, and we accomplish this by reducing all of the cases where these equations do not hold for some k to arguments that $\eta_k \neq 0$ for those k.

2. The case that there exists some $k \in \bigcup_{\ell=-1}^{\Lambda-1} K_\ell$ such that c is assigned before $\check{b}^{(k)}$.
Here, we want $\delta_{1,k} = \gamma_{1,k} + c\tilde{\gamma}_{1,k,0} \neq 0$, in which case

$$\eta_k = \delta_{2,k} - \check{b}^{(k)}\delta_{1,k} = 0 \Leftrightarrow \check{b}^{(k)} = \frac{\delta_{2,k}}{\delta_{1,k}} \tag{2}$$

(we will assert this does not occur within H_{non} and $\mathsf{H}_{\overline{\text{non}}}$), where $\check{b}^{(k)}$ is assigned uniformly at random after both $\delta_{1,k}$ and $\delta_{2,k}$ have been defined, and thus independently of the right-hand side.
If $k \in K_{-1} \setminus \bigcup_{\ell=0}^{\Lambda-1} K_\ell$, then $\delta_{1,k} = \gamma_{1,k} \neq 0$. Otherwise, if $k \in \bigcup_{\ell=0}^{\Lambda-1} K_\ell$, then we can assert within H_{agg} that

$$\tilde{\gamma}_{1,k,d} = \gamma_{1,k,d} \neq 0 \Leftrightarrow a_{i,d} \neq \frac{-\sum_{j\neq i} a_{j,d}\gamma_{j,1,k,d}}{\gamma_{i,1,k,d}}, \tag{3}$$

where d is the largest value such that $k \in K_d$ so that $\tilde{\gamma}_{1,k,d} = \gamma_{1,k,d}$, and i is an index such that $\gamma_{i,1,k,d} \neq 0$. Next, for each $\ell < d$ we can assert within H_{agg} that if $\tilde{\gamma}_{1,k,\ell+1} \neq 0$ then

$$\tilde{\gamma}_{1,k,\ell} \neq 0 \Leftrightarrow a_{1,\ell} \neq \frac{-\gamma_{1,k,\ell}}{\tilde{\gamma}_{1,k,\ell+1}}. \tag{4}$$

Thus,

$$\delta_{1,k} \neq 0 \Leftrightarrow c \neq \frac{-\gamma_{1,k}}{\tilde{\gamma}_{1,k,0}} \tag{5}$$

(we will assert this does not occur when the denominator is non-zero within H_{sig}), where c is chosen uniformly at random independently of the right-hand side.

3. The case that there exists some $0 \leq d < \Lambda$ and some $k \in \bigcup_{\ell=d}^{\Lambda-1} K_\ell$ such that $a_{1,d}$ is assigned before $\check{b}^{(k)}$, but $\check{b}^{(k)}$ is still assigned before c.
Here, we isolate c within the definition of η_k and get that

$$\begin{aligned}
\eta_k &= c\theta_k + \gamma_{2,k} - \check{b}^{(k)}\gamma_{1,k}, \text{with} \\
\theta_k &= \tilde{\gamma}_{2,k,0} - \check{b}^{(k)}\tilde{\gamma}_{1,k,0} \\
&= \sum_{j=0}^{\Lambda} \gamma_{2,k,j} \prod_{\ell=0}^{j-1} a_{1,\ell} - \check{b}^{(k)} \sum_{j=0}^{\Lambda} \gamma_{1,k,j} \prod_{\ell=0}^{j-1} a_{1,\ell}.
\end{aligned}$$

More generally, if we let

$$\theta_{k,d} = \tilde{\gamma}_{2,k,d} - \check{b}^{(k)}\tilde{\gamma}_{1,k,d},$$

then it is easy to verify that $\theta_{k,0} = \theta_k$ and

$$\theta_{k,d} = a_{1,d}\theta_{k,d+1} + \gamma_{2,k,d} - \check{b}^{(k)}\gamma_{1,k,d},$$

which gives us a factoring of θ_k in sequential order of $a_{1,d}$. Since $k \in \bigcup_{\ell=d}^{\Lambda-1} K_\ell$, we have by Assertions 3 and 4 above that $\tilde{\gamma}_{1,k,d} \neq 0$, so that within H_{non} and $\mathsf{H}_{\overline{\text{non}}}$ we can assert that

$$\theta_{k,d} \neq 0 \Leftrightarrow \check{b}^{(k)} \neq \frac{\tilde{\gamma}_{2,k,d}}{\tilde{\gamma}_{1,k,d}}. \tag{6}$$

Next, we can assert within H_{agg} that if $\theta_{k,d} \neq 0$, then for all $\ell < d$ we have

$$\theta_{k,\ell} \neq 0 \Leftrightarrow a_{1,\ell} \neq \frac{\check{b}^{(k)}\gamma_{1,k,\ell} - \gamma_{2,k,\ell}}{\theta_{k,\ell+1}}. \tag{7}$$

In particular, this implies that $\theta_k = \theta_{k,0} \neq 0$ so that within H_{sig} we can assert that

$$\eta_k \neq 0 \Leftrightarrow c \neq \frac{\check{b}^{(k)}\gamma_{1,k} - \gamma_{2,k}}{\theta_k}, \tag{8}$$

where c is chosen uniformly at random, independent of the right-hand side.

As mentioned above, our goal for Case 1 is to relate $\check{b}^{(k)}$ to the γ values by some equations. We begin by including assertions in H_{non} and $\mathsf{H}_{\overline{\text{non}}}$ that products involving outputs of T_{non} and $T_{\overline{\text{non}}}$ of size at most D contain no collisions, so that potential $\check{b}^{(k)}$ values are unique, and our equations will uniquely determine k. We choose the equations $\gamma_{2,k,d} - \check{b}^{(k)}\gamma_{1,k,d} = 0$ for $k \in K_d \setminus \bigcup_{\ell=d+1}^{\Lambda-1} K_\ell$, and $\gamma_{2,k} - \check{b}^{(k)}\gamma_{1,k} = 0$ for $k \in K_{-1} \setminus \bigcup_{\ell=0}^{\Lambda-1} K_\ell$. We need all of these equations because if $k \notin K_d$, then $\gamma_{1,k,d} = 0$ so that the corresponding equation does not depend on $\check{b}$. These equations are chosen because, when they do not hold for some k, we have easy ways to show that $\eta_k \neq 0$. Consider the following cases:

1(a). The subcase that there exists some d such that there exists a $k \in K_d \setminus \bigcup_{\ell=d+1}^{\Lambda-1} K_\ell$ with $\gamma_{2,k,d} - \check{b}^{(k)}\gamma_{1,k,d} \neq 0$ or there exists a $k \in K_{d-1} \setminus \bigcup_{\ell=d}^{\Lambda-1} K_\ell$ with $\tilde{\gamma}_{2,k,d} \neq 0$. We take d to be the largest such value so that we may assume that for all $d' > d$ and all $k \in \bigcup_{\ell=d'}^{\Lambda-1} K_\ell$, $\gamma_{2,k,d} - \check{b}^{(k)}\gamma_{1,k,d} = 0$ and for all $k \in \bigcup_{\ell=d'-1} K_\ell$, $\tilde{\gamma}_{2,k,d'} = 0$.
Here, suppose that we have a $k \in K_d \setminus \bigcup_{\ell=d+1}^{\Lambda-1} K_\ell$ with $\gamma_{2,k,d} - \check{b}^{(k)}\gamma_{1,k,d} \neq 0$, then since $k \notin \bigcup_{\ell=d+1}^{\Lambda-1} K_\ell$ we have that $\tilde{\gamma}_{1,k,d+1} = 0$ (by definition), and by our premise, $\tilde{\gamma}_{2,k,d+1} = 0$ so that $\theta_{k,d+1} = 0$. Therefore, $\theta_{k,d} = \gamma_{2,k,d} - \check{b}^{(k)}\gamma_{1,k,d} \neq 0$ and so we can conclude that $\eta_k \neq 0$ by Assertions 7 and 8 above. On the other hand, if we have a $k \in K_{d-1} \setminus \bigcup_{\ell=d}^{\Lambda-1} K_\ell$ with $\tilde{\gamma}_{2,k,d} \neq 0$, then since $k \notin \bigcup_{\ell=d}^{\Lambda-1} K_\ell$, we have that $\tilde{\gamma}_{1,k,d} = 0$ and thus $\theta_{k,d} = \tilde{\gamma}_{2,k,d} \neq 0$, so that we can again conclude that $\eta_k \neq 0$ by Assertions 7 and 8 above.

1(b). The subcase that there exists $k \in K_{-1} \setminus \bigcup_{\ell=0}^{\Lambda-1} K_\ell$ such that $\gamma_{2,k} - \check{b}^{(k)}\gamma_{1,k} \neq 0$.
Here, since $k \notin \bigcup_{\ell=0}^{\Lambda-1} K_\ell$, we have that $\tilde{\gamma}_{1,k,0} = 0$ by the assertions in Case 2, and we are not in Case 1(a) so that we can assume that $\tilde{\gamma}_{2,k,0} = 0$ as well implying that $\theta_k = 0$ so that $\eta_k = \gamma_{2,k} - \check{b}^{(k)}\gamma_{1,k} \neq 0$ by our premise.

1(c). The subcase negating all previous subcases so that for all $k \in K_d \setminus \bigcup_{\ell=d+1}^{\Lambda-1} K_\ell$, $\gamma_{2,k,d} - \check{b}^{(k)}\gamma_{1,k,d} = 0$, and for all $k \in K_{-1} \setminus \bigcup_{\ell=0}^{\Lambda-1} K_\ell$, $\gamma_{2,k} - \check{b}^{(k)}\gamma_{1,k} = 0$.
In this subcase, these equations, along with the assertion that there can be no $\check{b}$ collisions, allows us to look up the values $a_{1,d}^{(k)}, b_d^{(k)}$, and $c^{(k)}$ within H_{agg} and H_{sig} by searching for the T_{non} and $T_{\overline{\text{non}}}$ entries whose outputs' products, satisfy the relevant equation, which gives us access to the relevant $\check{b}^{(k)}$ and $\check{c}^{(k)}$ values. Within H_{agg}, this allows us to assert for each $d \in \{0, \ldots, \Lambda - 1\}$ that if $\theta_{0,d+1} \neq 0$, then

$$\theta_{0,d} = a_{1,d}\theta_{0,d+1} + \beta_d - \sum_{k \in K_d} \check{c}^{(k)}\gamma_{1,k,d} \neq 0 \Leftrightarrow a_{1,d} \neq \frac{-\beta_d + \sum_{k \in K_d} \check{c}^{(k)}\gamma_{1,k,d}}{\theta_{0,d+1}}, \tag{9}$$

where $a_{1,d}$ was chosen uniformly at random independently of the right-hand side. With these assertions in place, because $\theta_{0,\Lambda} = 1 \neq 0$, we can conclude that $\theta_0 = \theta_{0,0} \neq 0$. Finally, this allows us to assert within H_{sig} that if $\theta_0 \neq 0$, then

$$\eta_0 = c\theta_0 + \beta - \sum_{k \in K_{-1}} \check{c}^{(k)}\gamma_{1,k} \neq 0 \Leftrightarrow c \neq \frac{-\beta + \sum_{k \in K_{-1}} \check{c}^{(k)}\gamma_{1,k}}{\theta_0}, \tag{10}$$

where c is chosen uniformly at random independently of the right-hand side.

Altogether, we construct a new reduction, $\mathcal{C}$, which behaves like $\mathcal{B}$ except that, immediately before returning in a simulated random oracle, $\mathcal{C}$ calls the corresponding AssertNoCollisions function. Additionally, $\mathcal{C}$ has a table T_{prod} which begins empty, and tracks all potential $(\check{b}, \check{c})$ pairs by updating this table at every invocation of H_{non} and of $\mathsf{H}_{\overline{\text{non}}}$. Before each insertion into T_{prod}, $\mathcal{C}$ asserts that

there is no other entry already in T_{prod} with the same value. The entries of T_{prod} have the products $\check{b}$ as their keys, and the values stored there are the values $\check{c}$. Since the $a_{1,d}$ values needed to compute $\check{c}$ are guaranteed to exist for all relevant $\check{b}$ values, we do not need to add entries to T_{prod} if these a-values do not exist, that is, when T_{akey} does not contain an entry for the key in b's table entry, then we do not add an entry to T_{prod} for $\check{b}$ involving b. We also have assertions at the top of AssertNoCollisionsTagg making sure that each new aggregate key has never been used, so we do not need to worry about some $\check{b}$ value becoming potentially relevant after its instantiation. These new functions are depicted in Figures 12, 13, and 14.

By the reasoning above, if $\mathcal{C}$ does not fail on any of its assertions, then Solve is guaranteed to succeed within $\mathcal{B}$. In other words,

$$\mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{B},\mathsf{GrGen}}(\lambda) \geq \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2[GrGen},\nu=2]}(\lambda) - \Pr\left[\mathcal{C} \text{ fails an assertion} \mid \mathcal{A} \text{ succeeds}\right].$$

AssertWithDepth$(\Lambda, \widetilde{X}, vars, d, assertion_function)$

1 : **if** $T_{\text{akey}}(\widetilde{X}) = \bot$ **then return**
2 : $(L, (a_{1,d}, \ldots, a_{n_d,d}), (\beta_{i,d}, \gamma_{i,1,k,d}, \gamma_{i,2,k,d})_{1\leq i\leq n_d, 1\leq k\leq q_s}) := T_{\text{akey}}(\widetilde{X})$
3 : $(X_1, \ldots, X_{n_d}) := L$
4 : **for** $m := 1, \ldots n_d$ **do**
5 : Swap$((X_1, a_{1,d}, \beta_{1,d}, \gamma_{1,1,k,d}, \gamma_{1,2,k,d})_{1\leq k\leq q_s}, (X_m, a_{m,d}, \beta_{m,d}, \gamma_{m,1,k,d}, \gamma_{m,2,k,d})_{1\leq k\leq q_s})$
6 : $K_d := \{k \in \{1, \ldots, q_s\} \mid \exists i > 1, \gamma_{i,1,k,d} \neq 0\}$
7 : $\beta_d := \sum_{i=2}^{n_d} a_{i,d}\beta_{i,d}$
8 : **for** $k := 1, \ldots, q_s$ **do**
9 : $\gamma_{1,k,d} := \sum_{i=2}^{n_d} a_{i,d}\gamma_{i,1,k,d}$; $\gamma_{2,k,d} := \sum_{i=2}^{n_d} a_{i,d}\gamma_{i,2,k,d}$
10 : $vars :=$ Insert$(vars, (K_d, a_{i,d}, \beta_d, \beta_{i,d}, \gamma_{1,k,d}, \gamma_{2,k,d}, \gamma_{i,1,k,d}, \gamma_{i,2,k,d})_{1\leq i\leq n_d, 1\leq k\leq q_s}))$
11 : **if** $\Lambda = 1$ **then**
12 : **for** $\gamma_{1,k,\ell} \in vars$ **do**
13 : $\tilde{\gamma}_{1,k,\ell} := \sum_{j=\ell}^{d+1} \gamma_{1,k,j} \prod_{\ell'=\ell}^{j-1} a_{1,\ell'}$; $\tilde{\gamma}_{2,k,\ell} := \sum_{j=\ell}^{d+1} \gamma_{2,k,j} \prod_{\ell'=\ell}^{j-1} a_{1,\ell'}$
14 : $vars :=$ Insert$(vars, (\tilde{\gamma}_{1,k,\ell}, \tilde{\gamma}_{2,k,\ell}))$
15 : $assertion\_function(vars)$
16 : **else**
17 : AssertWithDepth$(\Lambda - 1, X_1, vars, d + 1, assertion)$

Fig. 11. Recursive subroutine that executes assertion_function after instantiating a, β and γ variables as if $\widetilde{X}$ is $\Lambda - 1$ levels above the bottom of the key tree. Note that this function can be used to instantiate a key subtree, in which case d-indices will begin at 0.

Let $\mathsf{Bad}(P, C, I)$ be the event that $\mathcal{C}$ fails an assertion in the I-th invocation of procedure P in pseudocode line C. Combining all invocations and lines, let

$$\mathsf{Bad}(P) = \bigcup_{C,I} \mathsf{Bad}(P, C, I).$$

Let $q = \binom{2q_h + Dq_s + D}{D} + DN(q_s + 1) + Nq_h$. First, note that AssertNoCollisionsTagg is invoked at most $q_h + DN(q_s + 1) \leq q$ times because H_{agg} is invoked at most q_h times by the adversary, at most DNq_s times in OSign′, and at most DN times when $\mathcal{B}$ verifies the forgery. AssertNoCollisionsTnon is invoked at most $q_h + (D-1)q_s \leq q$ times because H_{non} is invoked at most q_h times by the adversary and at most $(D-1)q_s$ times in OSign′. AssertNoCollisionsT$\overline{\text{non}}$ is invoked at most $q_h + q_s \leq q$ times because $\mathsf{H}_{\overline{\text{non}}}$ is invoked at most q_h times by the adversary and at most q_s times in OSign′.

AssertNoCollisionsTagg$(L, \widetilde{X}, (a_1, \ldots, a_n), (\beta_i, \gamma_{i,1,k}, \gamma_{i,2,k})_{1\le i\le n, 1\le k\le q_s})$

1 : // First ensure that $\widetilde{X}$ is a fresh key with no collisions
2 : **assert** $T_{\text{akey}}(\widetilde{X}) = \perp$
3 : **assert** $\neg\left(\exists(R,m)\colon T_{\text{sig}}(\widetilde{X}, R, m) \neq \perp\right)$
4 : **assert** $\neg\left(\exists(R_1,R_2)\colon T_{\text{non}}(\widetilde{X}, (R_1,R_2)) \neq \perp\right)$
5 : **assert** $\neg\left(\exists(R_1,R_2,m)\colon T_{\overline{\text{non}}}(\widetilde{X}, (R_1,R_2), m) \neq \perp\right)$
6 : $T_{\text{akey}}(\widetilde{X}) := (L, (a_1, \ldots, a_n), (\beta_i, \gamma_{i,1,k}, \gamma_{i,2,k})_{1\le i\le n, 1\le k\le q_s})$
7 : AssertWithDepth$(1, \widetilde{X}, \{\}, 0,$ Assertion2$)$
8 : **for** $\Lambda := 1, \ldots, D$ **do**
9 : AssertWithDepth$(\Lambda, \widetilde{X}, \{\}, 0,$ Assertion3$)$
10 : AssertWithDepth$(\Lambda, \widetilde{X}, \{\}, 0,$ Assertion6$)$
11 : AssertWithDepth$(\Lambda, \widetilde{X}, \{\}, 0,$ Assertion8$)$

Assertion2$((a_{i,0}, \gamma_{i,1,k,0}, \gamma_{1,k,0})_{1\le i\le n_0, 1\le k\le q_s})$

1 : **for** $k := 1, \ldots, q_s$ **do**
2 : **if** $\exists i > 1, \gamma_{i,1,k,0} \neq 0$ **then**
3 : **assert** $\gamma_{1,k,0} \neq 0$ // $\Leftrightarrow a_{i,0} \neq (-\sum_{j\neq i} a_{j,0}\gamma_{j,1,k,0})/\gamma_{i,1,k,0}$

Assertion3$((\tilde{\gamma}_{1,k,0}, \tilde{\gamma}_{1,k,1})_{1\le k\le q_s})$

1 : **for** $k := 1, \ldots, q_s$ **do**
2 : **if** $\tilde{\gamma}_{1,k,1} \neq 0$ **then**
3 : **assert** $\tilde{\gamma}_{1,k,0} \neq 0$ // $\Leftrightarrow a_{1,0} \neq -\gamma_{1,k,0}/\tilde{\gamma}_{1,k,1}$

Assertion6$((\tilde{\gamma}_{1,k,0}, \tilde{\gamma}_{2,k,0}, \tilde{\gamma}_{1,k,1}, \tilde{\gamma}_{2,k,1})_{1\le k\le q_s})$

1 : **for** $k := 1, \ldots, q_s$ **do**
2 : **for** $\check{b}^{(k)} \in T_{prod}$ **do**
3 : $\theta_{k,0} := \tilde{\gamma}_{2,k,0} - \check{b}^{(k)}\tilde{\gamma}_{1,k,0}$; $\theta_{k,1} := \tilde{\gamma}_{2,k,1} - \check{b}^{(k)}\tilde{\gamma}_{1,k,1}$
4 : **if** $\theta_{k,1} \neq 0$ **then**
5 : **assert** $\theta_{k,0} \neq 0$ // $\Leftrightarrow a_{1,0} \neq (\check{b}^{(k)}\gamma_{1,k,0} - \gamma_{2,k,0})/\theta_{k,1}$

Assertion8$((K_d, \beta_d, a_{1,d}, \gamma_{1,k,d}, \gamma_{2,k,d})_{0\le d<\Lambda, 1\le k\le q_s})$

1 : **for** $k \in \cup_{d=0}^{\Lambda-1} K_d$ **do**
2 : **if** $\exists \check{b}^{(k)} \in T_{prod} : \bigvee_{\ell=\Lambda-1}^{0} (k \in K_\ell \setminus \cup_{\ell'>\ell} K_{\ell'} \wedge \gamma_{2,k,\ell} - \check{b}^{(k)}\gamma_{1,k,\ell} = 0)$ **then**
3 : $\check{c}^{(k)} := T_{prod}(\check{b}^{(k)})$
4 : **else** $\check{c}^{(k)} := \perp$
5 : **if** $\forall k \in \cup K_d : \check{c}^{(k)} \neq \perp$ **then**
6 : **for** $d := \Lambda - 1, \ldots, 0$ **do**
7 : $\theta_{0,d} := a_{1,d}\theta_{0,d+1} + \beta_d - \sum_{k\in K_d} \check{c}^{(k)}\gamma_{1,k,d}$ // $\theta_{0,\Lambda} = 1$
8 : **if** $\theta_{0,1} \neq 0$ **then**
9 : **assert** $\theta_{0,0} \neq 0$ // $\Leftrightarrow a_{1,0} \neq (-\beta_0 + \sum_{k\in K_0} \check{c}^{(k)}\gamma_{1,k,0})/\theta_{0,1}$

Fig. 12. Procedure used to check bad events during a T_{agg} assignment in $\mathcal{B}$.

AssertNoCollisionsTnon$(\widetilde{X}, (R_1, R_2))$

1 : $b \coloneqq T_{\mathrm{non}}(\widetilde{X}, (R_1, R_2))$
2 : **if** $T_{\mathrm{akey}}(\widetilde{X}) \neq \perp$ **then**
3 : $\quad (L, (a_1, \ldots, a_n), (\tilde{\beta}_i, \tilde{\gamma}_{i,1,k}, \tilde{\gamma}_{i,2,k})_{1 \leq i \leq n, 1 \leq k \leq q_s}) \coloneqq T_{\mathrm{akey}}(\widetilde{X})$
4 : $\quad T_{subprod}(b) \coloneqq a_1$
5 : $\quad$ **for** $(\check{b}, \check{a}) \in T_{subprod}$ **do**
6 : $\quad\quad$ **if** $|\check{b}|_{factors} < D$ **then**
7 : $\quad\quad\quad T_{subprod}(b\check{b}) \coloneqq a_1\check{a}$
8 : $\quad\quad$ **for** $(\check{b}, \check{c}) \in T_{prod}$ **do**
9 : $\quad\quad\quad$ AssertNoCollisionsTprod$(b\check{b}, a_1\check{c})$

AssertNoCollisionsT$\overline{\mathsf{non}}(\widetilde{X}, (R_1, R_2), m)$

1 : $\overline{b} \coloneqq T_{\overline{\mathrm{non}}}(\widetilde{X}, (R_1, R_2), m)$
2 : $c \coloneqq T_{\mathrm{sig}}(\widetilde{X}, R_1 R_2^{\overline{b}}, m)$
3 : **if** $T_{\mathrm{akey}}(\widetilde{X}) \neq \perp$ **then**
4 : $\quad (L, (a_1, \ldots, a_n), (\beta_i, \gamma_{i,1,k}, \gamma_{i,2,k})_{1 \leq i \leq n, 1 \leq k \leq q_s}) \coloneqq T_{\mathrm{akey}}(\widetilde{X})$
5 : $\quad$ AssertNoCollisionsTprod$(\overline{b}, a_1 c)$
6 : $\quad$ **for** $(\check{b}, \check{a}) \in T_{subprod}$ **do**
7 : $\quad\quad$ AssertNoCollisionsTprod$(\overline{b}\check{b}, a_1\check{a}c)$

AssertNoCollisionsTprod$(\check{b}, \check{c})$

1 : **assert** $\check{b} \neq 0$
2 : **assert** $T_{prod}(\check{b}) = \perp$
3 : $T_{prod}(\check{b}) = \check{c}$
4 : **for** $((\widetilde{X}, R, m), c) \in T_{\mathrm{sig}}$ **do**
5 : $\quad (\alpha, \beta, (\gamma_{1,k}, \gamma_{2,k})_{1 \leq k \leq q_s}) \coloneqq T_{sigrep}(\widetilde{X}, R, m)$
6 : $\quad$ **for** $\Lambda \coloneqq 1, \ldots, D$ **do**
7 : $\quad\quad$ AssertWithDepth$(\Lambda, \widetilde{X}, \{\check{b}, c, (\gamma_{1,k}, \gamma_{2,k})_{1 \leq k \leq q_s}\}, 0,$ Assertion1$)$
8 : $\quad\quad$ AssertWithDepth$(\Lambda, \widetilde{X}, \{c, \beta, (\gamma_{1,k})_{1 \leq k \leq q_s}\}, 0,$ Assertion5$)$

Assertion1$(\check{b}, c, (\gamma_{1,k}, \gamma_{2,k}, \tilde{\gamma}_{1,k,0}, \tilde{\gamma}_{2,k,0})_{1 \leq k \leq q_s})$

1 : **for** $k \coloneqq 1, \ldots, q_s$ **do**
2 : $\quad \delta_{2,k} \coloneqq \gamma_{2,k} + c\tilde{\gamma}_{2,k,0}$
3 : $\quad$ **if** $\delta_{1,k} \coloneqq \gamma_{1,k} + c\tilde{\gamma}_{1,k,0} \neq 0$ **then**
4 : $\quad\quad$ **assert** $\eta_k \coloneqq \delta_{2,k} - \check{b}\delta_{1,k} \neq 0 \quad /\!\!/ \Leftrightarrow \check{b} \neq \delta_{2,k}/\delta_{1,k}$

Assertion5$(\check{b}, (\tilde{\gamma}_{1,k,0}, \tilde{\gamma}_{2,k,0})_{1 \leq k \leq q_s})$

1 : **for** $k \coloneqq 1, \ldots, q_s$ **do**
2 : $\quad$ **if** $\tilde{\gamma}_{1,k,0} \neq 0$ **then**
3 : $\quad\quad$ **assert** $\theta_{k,0} \coloneqq \tilde{\gamma}_{2,k,0} - \check{b}\tilde{\gamma}_{1,k,0} \neq 0 \quad /\!\!/ \Leftrightarrow \check{b} \neq \tilde{\gamma}_{2,k,0}/\tilde{\gamma}_{1,k,0}$

Fig. 13. Procedures used to check bad events during T_{non} and $T_{\overline{\mathrm{non}}}$ assignments in $\mathcal{B}$.

AssertNoCollisionsTsig($\widetilde{X}, R, m, (\alpha, \beta, (\gamma_{1,k}, \gamma_{2,k})_{1\le k\le q_s})$)

1 : $c \coloneqq T_{\mathrm{sig}}(\widetilde{X}, R, m)$
2 : **for** $\Lambda \coloneqq 1, \ldots, D$ **do**
3 : AssertWithDepth($\Lambda, \widetilde{X}, \{c, (\gamma_{1,k})_{1\le k\le q_s}\}, 0,$ Assertion4)
4 : AssertWithDepth($\Lambda, \widetilde{X}, \{c, (\gamma_{1,k}, \gamma_{2,k})_{1\le k\le q_s}\}, 0,$ Assertion7)
5 : AssertWithDepth($\Lambda, \widetilde{X}, \{c, \beta, (\gamma_{1,k})_{1\le k\le q_s}\}, 0,$ Assertion9)

Assertion4($c, (\gamma_{1,k}, \tilde{\gamma}_{1,k,0})_{1\le k\le q_s}$)

1 : **for** $k \coloneqq 1, \ldots, q_s$ **do**
2 : **if** $\tilde{\gamma}_{1,k,0} \neq 0$ **then**
3 : **assert** $\delta_{1,k} \coloneqq \gamma_{1,k} + c\tilde{\gamma}_{1,k,0} \neq 0$ // $\Leftrightarrow c \neq -\gamma_{1,k}/\tilde{\gamma}_{1,k,0}$

Assertion7($c, (\gamma_{1,k}, \gamma_{2,k}, \tilde{\gamma}_{1,k,0}, \tilde{\gamma}_{2,k,0})_{1\le k\le q_s}$)

1 : **for** $k \coloneqq 1, \ldots, q_s$ **do**
2 : **for** $\check{b}^{(k)} \in T_{prod}$ **do**
3 : **if** $\theta_k \coloneqq \tilde{\gamma}_{2,k,0} - \check{b}^{(k)}\tilde{\gamma}_{1,k,0} \neq 0$ **then**
4 : **assert** $\eta_k \coloneqq c\theta_k + \gamma_{2,k} - \check{b}^{(k)}\gamma_{1,k} \neq 0$ // $\Leftrightarrow c \neq (\check{b}^{(k)}\gamma_{1,k} - \gamma_{2,k})/\theta_k$

Assertion9($c, \beta, (K_d, a_{1,d}, \beta_d, \gamma_{1,k}, \tilde{\gamma}_{1,k,0}, \gamma_{1,k,d}, \gamma_{2,k,d})_{1\le k\le q_s, 1\le d<\Lambda}$)

1 : **for** $k \in \cup_{d=0}^{\Lambda-1} K_d$ **do**
2 : **if** $\exists \check{b}^{(k)} \in T_{prod} : \bigvee_{\ell=\Lambda-1}^{0} (k \in K_\ell \setminus \cup_{\ell'>\ell} K_{\ell'} \wedge \gamma_{2,k,\ell} - \check{b}^{(k)}\gamma_{1,k,\ell} = 0)$ **then**
3 : $\check{c}^{(k)} \coloneqq T_{prod}(\check{b}^{(k)})$
4 : **else** $\check{c}^{(k)} \coloneqq \perp$
5 : **if** $\forall k \in \cup K_d : \check{c}^{(k)} \neq \perp$ **then**
6 : **if** $\theta_0 \coloneqq \sum_{j=0}^{\Lambda} \beta_j \prod_{d=0}^{j-1} a_{1,d} - \sum_{k=1}^{q_s} \check{c}^{(k)}\tilde{\gamma}_{1,k,0} \neq 0$ **then**
7 : **assert** $\eta_0 \coloneqq c\theta_0 + \beta - \sum_{k=1}^{q_s} \check{c}^{(k)}\gamma_{1,k} \neq 0$ // $\Leftrightarrow c \neq (-\beta + \sum_{k=1}^{q_s} \check{c}^{(k)}\gamma_{1,k})/\theta_0$

Fig. 14. Procedure used to check bad events during a T_{sig} assignment in $\mathcal{B}$.

AssertNoCollisionsTprod is invoked at most

$$\binom{2q_h + Dq_s}{1} + \binom{2q_h + Dq_s}{2} + \cdots + \binom{2q_h + Dq_s}{D} < \binom{2q_h + Dq_s + D}{D}$$

times since there are at most $2q_h + Dq_s$ values from which we consider all products of size 1 to D. Thus, the function is invoked at most $\binom{2q_h+Dq_s+D}{D} \leq q$ times. AssertNoCollisionsTsig is invoked at most $2(q_h + q_s) + 1 \leq q$ times because H_{sig} is invoked at most q_h times by the adversary, at most once for every invocation of OSign′, at most once for every invocation of $\mathsf{H}_{\overline{\text{non}}}$, and once when $\mathcal{B}$ verifies the forgery. Hence, at any point of the execution of $\mathcal{C}$, $|T_{\text{akey}}| \leq q$, $|T_{\text{non}}| \leq q$, $|T_{\overline{\text{non}}}| \leq q$, $|T_{prod}| \leq q$, and $|T_{\text{sig}}| \leq q$.

Every assertion passed into AssertWithDepth$(\Lambda, \ldots)$ within a loop going through all possible depths is invoked a total of at most $ND(q_s + 1) + Nq_h < q$ times as it traverses a key-tree where all relevant H_{agg} values must be defined and each query to the signing oracle instantiates at most ND H_{agg} values (and the same when $\mathcal{A}$ returns an output to its environment), while each query directly to H_{agg} instantiates at most N values.

Since in AssertNoCollisionsTagg, the freshly drawn value $\widetilde{X}$, as well as the freshly drawn independent values $(a_1, \ldots, a_n)$ are random and independent of the other table entries, we can bound for each invocation, I, of AssertNoCollisionsTagg that

$$\begin{aligned}
\Pr\left[\mathsf{Bad}(\mathsf{AssertNoCollisionsTagg}, 2, I)\right] &\leq \frac{q}{p}, \\
\Pr\left[\mathsf{Bad}(\mathsf{AssertNoCollisionsTagg}, 3, I)\right] &\leq \frac{q}{p}, \\
\Pr\left[\mathsf{Bad}(\mathsf{AssertNoCollisionsTagg}, 4, I)\right] &\leq \frac{q}{p}, \\
\Pr\left[\mathsf{Bad}(\mathsf{AssertNoCollisionsTagg}, 5, I)\right] &\leq \frac{q}{p}, \\
\Pr\left[\mathsf{Bad}(\mathsf{Assertion2}, 3, I)\right] &\leq \frac{q}{p}, \\
\Pr\left[\mathsf{Bad}(\mathsf{Assertion3}, 3, I)\right] &\leq \frac{q}{p}, \\
\Pr\left[\mathsf{Bad}(\mathsf{Assertion6}, 5, I)\right] &\leq \frac{q^2}{p}, \\
\Pr\left[\mathsf{Bad}(\mathsf{Assertion8}, 9, I)\right] &\leq \frac{1}{p}.
\end{aligned}$$

Within AssertNoCollisionsTagg, each of Assertion2, Assertion3, Assertion6 and Assertion8 is invoked at most q times. Thus, by the union bound

$$\Pr\left[\mathsf{Bad}(\mathsf{AssertNoCollisionsTagg})\right] \leq \frac{q(4q + q(2q + q^2 + 1))}{p} \leq \frac{8q^4}{p}.$$

Since in AssertNoCollisionsTprod, the freshly drawn value $\breve{b}$ is either equal to 0 (in which case the most recent random value is 0), or uniformly random in $\mathbb{Z}_p^*$ and independent of all other table entries, we have by code inspection for each invocation I of AssertNoCollisionsTprod that

$$\begin{aligned}
\Pr\left[\mathsf{Bad}(\mathsf{AssertNoCollisionsTprod}, 1, I)\right] &\leq \frac{1}{p}, \\
\Pr\left[\mathsf{Bad}(\mathsf{AssertNoCollisionsTprod}, 2, I)\right] &\leq \frac{q}{p}, \\
\Pr\left[\mathsf{Bad}(\mathsf{Assertion1}, 4, I)\right] &\leq \frac{q}{p}, \\
\Pr\left[\mathsf{Bad}(\mathsf{Assertion5}, 3, I)\right] &\leq \frac{q}{p}.
\end{aligned}$$

Within an invocation of AssertNoCollisionsTprod, Assertion1 and Assertion5 are each invoked at most $q \cdot q$ times. Thus, by the union bound

$$\Pr\left[\mathsf{Bad}(\mathsf{AssertNoCollisionsTprod})\right] \leq \frac{q(1 + q + q^2(2q))}{p} \leq \frac{4q^4}{p}.$$

Since in AssertNoCollisionsTsig, the freshly drawn T_{sig} value c is random and independent of all other table entries, we have by code inspection for each invocation I of AssertNoCollisionsTsig that

$$\Pr\left[\mathsf{Bad}(\mathsf{Assertion4}, 3, I)\right] \leq \frac{q}{p},$$
$$\Pr\left[\mathsf{Bad}(\mathsf{Assertion7}, 4, I)\right] \leq \frac{q^2}{p},$$
$$\Pr\left[\mathsf{Bad}(\mathsf{Assertion9}, 7, I)\right] \leq \frac{1}{p}.$$

Within an invocation of AssertNoCollisionsTsig, each of these three assertion functions is invoked at most q times. Thus, by the union bound

$$\Pr\left[\mathsf{Bad}(\mathsf{AssertNoCollisionsTsig})\right] \leq \frac{q(q+q^2+1)}{p} \leq \frac{3q^4}{p}.$$

Combining these bounds, we have that

$$\Pr\left[\mathcal{C} \text{ fails an assertion}\right] \leq \frac{8q^4}{p} + \frac{4q^4}{p} + \frac{3q^4}{p} = \frac{15q^4}{p} \leq \frac{30q^4}{2^\lambda}.$$

Therefore, we can conclude that

$$\mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{B},\mathsf{GrGen}}(\lambda) \geq \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{MuSig2}[\mathsf{GrGen},\nu=2]}(\lambda) - \frac{30q^4}{2^\lambda}.$$

The running time, t', of $\mathcal{B}$ is the running time, t, of $\mathcal{A}$ plus the time needed to maintain the tables, answering signing and hash queries, and running Solve. All tables have size $O(qN)$, and $\mathcal{B}$ needs $O(qDN)t_{\text{exp}}$ time to compute the aggregate key and the effective nonces when simulating signatures and answering H_{agg} queries, where t_{exp} is the time to compute an exponentiation in $\mathbb{G}$. Solving the linear equation system takes $O(q^2)$ time. Therefore,

$$t' = t + O(qDN) \cdot t_{\text{exp}} + O(q^2).$$

This concludes the proof. □

Acknowledgments

We thank Yannick Seurin for his generous mentorship, Paul Gerhart for his helpful comments, and Pieter Wuille, Jesse Posner, Tim Ruffing and Jonas Nick for their helpful discussions and suggestions. We also thank Clara Shikhelman, Carla Kirk-Cohen, Elle Mouton, Beulah Evanjalin, siv2r, and theStack for their participation in motivating and defining the problem of nesting MuSig2.

References

[AAC+25] M. Argentieri, Z. Avarikioti, A. Camilleri, P. Keer, M. Maffei. *Ark: A UTXO-based Transaction Batching Protocol*. https://docs.arklabs.xyz/ark.pdf. 2025.

[BDL19] M. Bellare, W. Dai, L. Li. "The Local Forking Lemma and Its Application to Deterministic Encryption". In: *ASIACRYPT 2019, Part III*. DOI: 10.1007/978-3-030-34618-8_21.

[BDN18] D. Boneh, M. Drijvers, G. Neven. "Compact Multi-signatures for Smaller Blockchains". In: *ASIACRYPT 2018, Part II*. DOI: 10.1007/978-3-030-03329-3_15.

[BGC+25] R. Baecker, P. Gerhart, D. L. Calsi, L. Russo, D. Schröder, A. Yerukhimovich. *Fully Adaptive FROST in the Algebraic Group Model From Falsifiable Assumptions*. Cryptology ePrint Archive, Paper 2025/1950. Available at https://eprint.iacr.org/2025/1950. 2025.

[BN06] M. Bellare, G. Neven. "Multi-signatures in the plain public-Key model and a general forking lemma". In: *ACM CCS 2006*. DOI: 10.1145/1180405.1180453.

[BNP+03] M. Bellare, C. Namprempre, D. Pointcheval, M. Semanko. "The One-More-RSA-Inversion Problems and the Security of Chaum's Blind Signature Scheme". In: *Journal of Cryptology* 16.3 (2003). DOI: 10.1007/s00145-002-0120-1.

[CGR+23] H. Chu, P. Gerhart, T. Ruffing, D. Schröder. "Practical Schnorr Threshold Signatures Without the Algebraic Group Model". In: *Advances in Cryptology - CRYPTO 2023*. Available at https://eprint.iacr.org/2023/899.pdf.

[DEF+19] M. Drijvers, K. Edalatnejad, B. Ford, E. Kiltz, J. Loss, G. Neven, I. Stepanovs. "On the Security of Two-Round Multi-Signatures". In: *2019 IEEE Symposium on Security and Privacy*. DOI: 10.1109/SP.2019.00050.

[FKL18] G. Fuchsbauer, E. Kiltz, J. Loss. "The Algebraic Group Model and its Applications". In: *CRYPTO 2018, Part II*. DOI: 10.1007/978-3-319-96881-0_2.

[Gib23] A. Gibson. *Forgery with a fake key in MuSig2*. https://gist.github.com/AdamISZ/ca974ed67889cedc738c4a1f65ff620b. 2023.

[GKK22] T. L. Guilly, N. Kohen, I. Kuwahara. "Bitcoin Oracle Contracts: Discreet Log Contracts in Practice". In: *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. DOI: 10.1109/ICBC54727.2022.9805512.

[GKP26] P. Gerhart, N. Kohen, J. Posner. "Iceberg: Signing for Thresholdized Lightning Nodes". In preparation. 2026.

[Lin25] R. Linus. *BitVM 3s – Garbled Circuits for Efficient Computation on Bitcoin*. https://bitvm.org/bitvm3.pdf. 2025.

[Mou23] E. Mouton. *extension-bolt: taproot gossip*. https://github.com/lightning/bolts/pull/1059. 2023.

[MPS+19] G. Maxwell, A. Poelstra, Y. Seurin, P. Wuille. "Simple Schnorr multi-signatures with applications to Bitcoin". In: *Des. Codes Cryptogr.* 87.9 (2019). https://eprint.iacr.org/2018/068.

[NRJ22] J. Nick, T. Ruffing, E. Jin. *MuSig2 for BIP340-compatible Multi-Signatures*. Bitcoin Improvement Proposal 327. https://github.com/bitcoin/bips/blob/master/bip-0327.mediawiki. 2022.

[NRS21] J. Nick, T. Ruffing, Y. Seurin. "MuSig2: Simple Two-Round Schnorr Multi-signatures". In: DOI: 10.1007/978-3-030-84242-0_8.

[Osu22] O. Osuntokun. *extension-bolt: simple taproot channels*. https://github.com/lightning/bolts/pull/995. 2022.

[PD16] J. Poon, T. Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. https://lightning.network/lightning-network-paper.pdf. 2016.

[Seu24] Y. Seurin. *Threshold MuSig from VPSS*. https://gist.github.com/yannickseurin/c5a76b7180102219c77d4b63fe387445. 2024.

[Wag02] D. Wagner. "A Generalized Birthday Problem". In: *CRYPTO 2002*. DOI: 10.1007/3-540-45708-9_19.

[WNR20] P. Wuille, J. Nick, T. Ruffing. *Schnorr Signatures for secp256k1*. Bitcoin Improvement Proposal 340. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki. 2020.

A Optimizing the Nonce Aggregation Algorithm

As described in Section 2.1 of [NRS21], it is imperative that each signer's effective nonce must vary with each of their cosigner's first-round messages. This is the purpose of $\check{b}$'s role in the nonce computation. If no such dependence is introduced, then there is a concrete forgery attack described by Drijvers et. al. in [DEF+19] based on Wagner's algorithm [Wag02] for solving the Generalized Birthday Problem. This helps frame the fact that in both security proofs (of Theorem 1 and Theorem 2), the key property of $\check{b}$ is that there is no $\check{b}$-collision between distinct signing queries. In other words, if $\check{b}$ is fixed between two signing queries, then all inputs to those signing queries must be equal. This property is attained in NestedMuSig2 because the set of values $\{b_i\}_{0 \le i < \Lambda}$ collectively commits to all three of $(out^d)_{0 \le d < \Lambda}$, m, and $\{pk_{i,d}\}_{2 \le i \le n_d}^{0 \le d < \Lambda}$. The design decision to multiply the b values together has the benefit of simplicity and, more importantly, it does not require any changes to the top-level MuSig2 protocol.

However, there are alternative methods to aggregate the commitments $\{b_i\}_{0\le i<\Lambda}$ into the single value $\check{b}$ that offer different trade-offs. We propose an alternative, MerkleNestedMuSig2, in Figure 15 where a hash tree is used to accumulate commitments, and $\check{b}$ is set to be the root of this tree.

The scheme MerkleNestedMuSig2 has three primary benefits when compared to NestedMuSig2:

1. We save on group element exponentiation in SignAggExt.
2. The protocol becomes more extensible for other use cases (e.g., nesting threshold-controlled signing keys) because nested signers are now able to commit to arbitrary data by modifying ϕ_1 in Sign() to be computed using a cryptographic hash function.
3. The security proofs become simpler and have tighter security bounds. In particular, the asymptotic dependence on the maximum depth, D, is no longer exponential assuming the algebraic group model, so that it becomes practical to implement deep nesting safely.

To expand on this last point, observe that the proof of Theorem 1 follows through for MerkleNestedMuSig2 unchanged except that $\Pr[\mathsf{LinDep}]$ becomes bounded by $\frac{(2^{D+1}q)^2}{p} \le \frac{2^{2D+3}q^2}{2^\lambda}$ because the event LinDep simply becomes that there is a collision found for $\mathsf{H}_{\overline{\text{non}}}$. Thus, we can conclude the following.

Theorem 3. *Let* GrGen *be a group generation algorithm for which the AOMDL problem is hard. Then the nested multi-signature scheme* $\mathsf{MerkleNestedMuSig2}[\mathsf{GrGen}, \nu = 2^{D+1}]$ *is EUF-CMA in the random oracle model for* H_{agg}, H_{non}, $\mathsf{H}_{\overline{\text{non}}}$, $\mathsf{H}_{\text{sig}} : \{0,1\}^* \to \mathbb{Z}_p$.

Precisely, for any adversary $\mathcal{A}$ *against* $\mathsf{MerkleNestedMuSig2}[\mathsf{GrGen}, \nu = 2^{D+1}]$ *running in time at most* t*, making at most* q_s OSign *queries and at most* q_h *queries to each random oracle, and such that the size of* L_d *in any signing session and in the forgery is at most* N*, and the maximum depth of any signing session is at most* D*, there exists an algorithm* $\mathcal{E}$ *taking as input group parameters* $(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$*, running in time at most*

$$t' = 2^{D+1}(t + q(3N + 3\nu - 3))t_{\text{exp}} + O(qN)$$

where $q = (N+4)q_h + (DN + D + 1)q_s + DN + 1$ *and* t_{exp} *is the time of an exponentiation in* $\mathbb{G}$*, making at most* $2^{D+1}q_s$ DLog_g *queries, and solving the AOMDL problem with an advantage*

$$\mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{E},\mathsf{GrGen}}(\lambda) \ge \frac{(\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu=2^{D+1}]}(\lambda))^{2^{D+1}}}{2^{2^{D+1}-D-2}q^{2^{D+1}-1}} - \frac{2^{D+2}q + 2^{2D+3}q^2 + 12}{2^\lambda}.$$

Similarly, the proof of Theorem 2 follows through for MerkleNestedMuSig2 unchanged except that we can now set $q = DN(q_s + 1) + Nq_h$ without the summand of $\binom{2q_h + Dq_s + D}{D}$ because we can move all T_{prod} assertions to become $T_{\overline{\text{non}}}$ assertions (we discard T_{prod} entirely). Thus, we can conclude the following.

Theorem 4. *Let* GrGen *be a group generation algorithm for which the AOMDL problem is hard. Then the multi-signature scheme* $\mathsf{MerkleNestedMuSig2}[\mathsf{GrGen}, \nu = 2]$ *is EUF-CMA in the algebraic group model for* GrGen*, and the random oracle model for* H_{agg}, H_{non}, $\mathsf{H}_{\overline{\text{non}}}$, $\mathsf{H}_{\text{sig}} : \{0,1\}^* \to \mathbb{Z}_p$.

Precisely, for any algebraic adversary $\mathcal{A}$ *against* $\mathsf{MerkleNestedMuSig2}[\mathsf{GrGen}, \nu = 2]$ *running in time at most* t*, making at most* q_s OSign *queries, at most* q_h *queries to each random oracle, such that the size of* L *in any signing session and in the forgery is at most* N*, and such that the depth,* Λ*, of any signing session or forgery is at most* D*, there exists an algorithm* $\mathcal{B}$ *running in time at most*

$$t' = t + O(qDN) \cdot t_{\text{exp}} + O(q^2),$$

where $q = DN(q_s + 1) + Nq_h$ *and* t_{exp} *is the time of an exponentiation in* $\mathbb{G}$ *and making at most* $2q_s$ DLog_g *queries such that*

$$\mathsf{Adv}^{\mathsf{AOMDL}}_{\mathcal{B},\mathsf{GrGen}}(\lambda) \ge \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{NestedMuSig2}[\mathsf{GrGen},\nu=2]}(\lambda) - \frac{30q^4}{2^\lambda}.$$

However, there are two main drawbacks to MerkleNestedMuSig2. First, first-round messages sent to aggregator nodes contain an extra λ-bit string, and messages received from aggregator nodes in the second round of signing contain up to N additional λ-bit strings. Second, this protocol is

$\mathsf{Setup}(1^\lambda)$

$(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$
Select four hash functions
$\mathsf{H}_{\mathrm{agg}}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\overline{\mathrm{non}}}, \mathsf{H}_{\mathrm{sig}} : \{0,1\}^* \to \mathbb{Z}_p$
$par :=$
$\quad((\mathbb{G}, p, g), \mathsf{H}_{\mathrm{agg}}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\overline{\mathrm{non}}}, \mathsf{H}_{\mathrm{sig}})$
return par

$\mathsf{KeyGen}()$

$x \leftarrow\!\!\$\ \mathbb{Z}_p\,;\ X := g^x$
$sk := x\,;\ pk := X$
return (sk, pk)

$\mathsf{KeyAggCoef}(L, X_i)$

return $\mathsf{H}_{\mathrm{agg}}(L, X_i)$

$\mathsf{KeyAgg}(L)$

$\{X_1, \ldots, X_n\} := L$
for $i := 1 \ldots n$ **do**
$\quad a_i := \mathsf{KeyAggCoef}(L, X_i)$
return $\widetilde{X} := \prod_{i=1}^n X_i^{a_i}$

$\mathsf{Ver}(\widetilde{X}, m, \sigma)$

$(R, s) := \sigma$
$c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$
return $(g^s = R\widetilde{X}^c)$

$\mathsf{Sign}()$

// Local signer has index 1.
for $j := 1 \ldots \nu$ **do**
$\quad r_{1,j} \leftarrow\!\!\$\ \mathbb{Z}_p\,;\ R_{1,j} := g^{r_{1,j}}$
$\phi_1 \leftarrow\!\!\$\ \{0,1\}^\lambda$
$out_1 := (R_{1,1}, \ldots, R_{1,\nu}, \phi_1)$
$state_1 := (r_{1,1}, \ldots, r_{1,\nu})$
return $(out_1, state_1)$

$\mathsf{SignAgg}(out_1, \ldots, out_n)$

for $i := 1 \ldots n$ **do**
$\quad (R_{i,1}, \ldots, R_{i,\nu}, \phi_i) := out_i$
for $j := 1 \ldots \nu$ **do**
$\quad R_j := \prod_{i=1}^n R_{i,j}$
return $(R_1, \ldots, R_\nu, \{\phi_i\}_{1 \le i \le n})$

$\mathsf{SignAggExt}(out, \widetilde{X})$

$(R'_1, \ldots, R'_\nu, \{\phi_i\}_{1 \le i \le n}) := out$
$\phi := \mathsf{H}_{\mathrm{non}}(\widetilde{X}, out)$
return $(R_1, \ldots, R_\nu, \phi)$

$\mathsf{Sign}'(state_1, (out^d)_{0 \le d < \Lambda}, sk_1, m, \{pk_{i,d}\}_{2 \le i \le n_d}^{0 \le d < \Lambda})$

// Sign' must be called at most once per $state_1$.
$(r_{1,1}, \ldots, r_{1,\nu}) := state_1$
$pk_{1,\Lambda-1} := g^{sk_1}$
$L_{\Lambda-1} := \{pk_{1,\Lambda-1}, \ldots, pk_{n_{\Lambda-1},\Lambda-1}\}$
$a_{1,\Lambda-1} := \mathsf{KeyAggCoef}(L_{\Lambda-1}, pk_{1,\Lambda-1})$
$pk_{1,\Lambda-2} := \mathsf{KeyAgg}(L_{\Lambda-1})$
for $d := \Lambda - 2, \ldots, 0$ **do**
$\quad \phi_{1,d} := \mathsf{H}_{\mathrm{non}}(pk_{1,d}, out^{d+1})$
$\quad (R_{1,d}, \ldots, R_{\nu,d}, \{\phi'_{i,d}\}_{1 \le i \le n_d}) := out^d$
$\quad$ **assert** $\phi_{1,d} = \phi'_{1,d}$ // Verify Merkle root.
$\quad L_d := \{pk_{1,d}, \ldots, pk_{n_d,d}\}$
$\quad a_{1,d} := \mathsf{KeyAggCoef}(L_d, pk_{1,d})$
$\quad pk_{1,d-1} := \mathsf{KeyAgg}(L_d)$
$\widetilde{X} := pk_{1,-1}$
$(R_{1,0}, \ldots, R_{\nu,0}, \{\phi_{i,0}\}_{1 \le i \le n_0}) := out^0$
$\check{b} := \mathsf{H}_{\overline{\mathrm{non}}}(\widetilde{X}, (R_{1,0}, \ldots, R_{\nu,0}), m, \{\phi_{i,0}\}_{1 \le i \le n_0})$
$R := \prod_{j=1}^\nu R_{j,0}^{\check{b}^{j-1}}$
$c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$
$\check{c} := c \prod_{\ell=0}^{\Lambda-1} a_{1,\ell} \bmod p$
$s_1 := \check{c} sk_1 + \sum_{j=1}^\nu r_{1,j} \check{b}^{j-1} \bmod p$
$state'_1 := R\,;\ out'_1 := s_1$
return $(state'_1, out'_1)$

$\mathsf{SignAgg}'(out'_1, \ldots, out'_n, state'_1)$

$R := state'_1$
$(s_1, \ldots, s_n) := (out'_1, \ldots, out'_n)$
$s := \sum_{i=1}^n s_i \bmod p$
return $\sigma := (R, s)$

Fig. 15. The nested multi-signature scheme $\mathsf{MerkleNestedMuSig2}[\mathsf{GrGen}, \nu]$. Public parameters par returned by Setup are implicitly given as input to all other algorithms. Changes from $\mathsf{NestedMuSig2}$ are *highlighted*.

not equivalent to MuSig2 when the nesting depth is set to one. Namely, in the first round, each party is expected to send an additional λ-bit string, and signers append the concatenation of all parties' strings to the nonce hash input. This is not a large change, but it does break compatibility with BIP 327 [NRJ22], which is a drawback as this is the currently adopted MuSig2 specification. That being said, when performing deep nesting while subject to a MuSig2 top-level, it is possible to use a hybrid aggregation approach where the multiplicative approach is used in the top two levels, but the second depth level computes its committment, b_1, as the merkle root of all deeper levels (as in MerkleNestedMuSig2) and $\check{b} := b_1 \cdot b_0$ (as in NestedMuSig2). This hybrid approach is more complicated, but it achieves backwards-compatibility with MuSig2 while avoiding an exponential dependence on D in the AGM.

B Optimizing the Key Aggregation Algorithm

Appendix A of [NRS21] describes the scheme MuSig2*, which is identical to MuSig2 except that the definition of KeyAggCoef is changed to the following:

$$\mathsf{KeyAggCoef}^*(L, X) = \begin{cases} 1 & \text{if } \mathsf{IsSecond}(L, X) \\ \mathsf{H}_{\mathrm{agg}}(L, X) & \text{otherwise.} \end{cases}$$

In other words, if there are at least two distinct keys in L, then the second key does not need to be exponentiated by a hash value during public key aggregation and that signer's private key does not need to be multiplied by a hash during signing. We define the protocols NestedMuSig2* and MerkleNestedMuSig2* to be equivalent to the protocols NestedMuSig2 and MerkleNestedMuSig2 respectively, except that KeyAggCoef* is used in place of KeyAggCoef.

The EUF-CMA security of MuSig2* is proven in Appendix A of [NRS21] by a simulation argument tightly reducing its security to the EUF-CMA security of MuSig2. The exact same argument (except that the programming applied to $\mathsf{H}_{\overline{\mathrm{non}}}$ must also be applied to $\mathsf{H}_{\mathrm{non}}$) works to reduce the EUF-CMA security of NestedMuSig2* and MerkleNestedMuSig2* to the EUF-CMA security of NestedMuSig2 and MerkleNestedMuSig2, respectively.