MuSig2: Simple Two-Round Schnorr Multi-Signatures

Jonas Nick[1], Tim Ruffing[1], and Yannick Seurin[2]

[1] Blockstream
[2] ANSSI, Paris, France

October 11, 2020

Abstract. Multi-signatures enable a group of signers to produce a single signature on a given message. Recently, Drijvers *et al.* (S&P'19) showed that all thus far proposed two-round multi-signature schemes in the DL setting (without pairings) are insecure under concurrent sessions, i.e., if a single signer participates in multiple signing sessions concurrently. While Drijvers *et al.* improve the situation by constructing a secure two-round scheme, saving a round comes with the price of having less compact signatures. In particular, the signatures produced by their scheme are more than twice as large as Schnorr signatures, which arguably are the most natural and compact among all practical DL signatures and are therefore becoming popular in cryptographic applications (e.g., support for Schnorr signature verification has been proposed to be included in Bitcoin). If one needs a multi-signature scheme that can be used as a drop-in replacement for Schnorr signatures, then one is either forced to resort to a three-round scheme such as MuSig (Maxwell *et al.*, DCC 2019) or MSDL-pop (Boneh, Drijvers, and Neven, ASIACRYPT 2018), or to accept that signing sessions are only secure when run sequentially, which may be hard to enforce in practice, e.g., when the same signing key is used by multiple devices.

In this work, we propose MuSig2, a novel and simple two-round multi-signature scheme variant of the MuSig scheme. Our scheme is the first multi-signature scheme that simultaneously *i)* is secure under concurrent signing sessions, *ii)* supports key aggregation, *iii)* outputs ordinary Schnorr signatures, *iv)* needs only two communication rounds, and *v)* has similar signer complexity as regular Schnorr signatures. Furthermore, our scheme is the first multi-signature scheme in the DL setting that supports preprocessing of all but one rounds, effectively enabling a non-interactive signing process, without forgoing security under concurrent sessions. The combination of all these features makes MuSig2 highly practical. We prove the security of MuSig2 under the one-more discrete logarithm (OMDL) assumption in the random oracle model, and the security of a more efficient variant in the combination of the random oracle and algebraic group models.

1 Introduction

1.1 Background on Multi-Signatures

Multi-signature schemes [IN83] enable a group of signers (each possessing an own private/public key pair) to run an interactive protocol to produce a single signature σ on a message m. A recent spark of interest in multi-signatures is motivated by the idea of using them as a drop-in replacement for ordinary (single-signer) signatures in applications such as cryptocurrencies that support signatures already. For example the Bitcoin community, awaiting the adoption of Schnorr signatures [Sch91] as proposed in BIP 340 [WNR20], is seeking for practical multi-signature schemes which are *fully compatible* with Schnorr signatures: multi-signatures produced by a group of signers should just be ordinary Schnorr signatures and should be verifiable like Schnorr signatures, i.e., they can be verified using the ordinary Schnorr verification algorithm given only a single *aggregate public key* that can be computed from the set of public keys of the signers and serves as a compact representation of it.

This provides a number of benefits that reach beyond simple compatibility with an upcoming system: Most importantly, multi-signatures enjoy the efficiency of Schnorr signatures, which are very compact and cheap to store on the blockchain. Moreover, if multi-signatures can be verified as ordinary Schnorr signatures, the additional complexity introduced by multi-signatures remains on the side of the signers and is not exposed to verifiers who need not be concerned with multi-signatures at all and can simply run Schnorr signature verification. Verifiers, who are just given the signature and the aggregate public key, in fact do not even learn whether the signature was created by a single signer or by a group of signers (or equivalently, whether the public key is an aggregation of multiple keys), which is advantageous for the privacy of users.

Multi-signatures Based on Schnorr Signatures. A number of modern and practical proposals [BN06, BCJ08, MWLD10, STV+16, MPSW19, DEF+19, NRSW20] for multi-signature schemes are based on Schnorr signatures. The Schnorr signature scheme [Sch91] relies on a cyclic group $\mathbb{G}$ of prime order p, a generator g of $\mathbb{G}$, and a hash function H. A private/public key pair is a pair $(x, X) \in \{0, \ldots, p-1\} \times \mathbb{G}$ where $X = g^x$. To sign a message m, the signer draws a random integer r in $\mathbb{Z}_p$, computes a nonce $R = g^r$, the challenge $c = H(X, R, m)$, and $s = r + cx$. The signature is the pair (R, s), and its validity can be checked by verifying whether $g^s = RX^c$.

The naive way to design a multi-signature scheme fully compatible with Schnorr signatures would be as follows. Say a group of n signers want to cosign a message m, and let $L = \{X_1 = g^{x_1}, \ldots, X_n = g^{x_n}\}$ be the multiset[3] of all their public keys. Each signer randomly generates and communicates to others a nonce $R_i = g^{r_i}$; then, each of them computes $R = \prod_{i=1}^{n} R_i$, $c = H(\widetilde{X}, R, m)$ where $\widetilde{X} = \prod_{i=1}^{n} X_i$ is the product of individual public keys, and a partial signature $s_i = r_i + cx_i$; partial signatures are then combined into a single signature (R, s) where $s = \sum_{i=1}^{n} s_i \bmod p$. The validity of a signature (R, s) on message m for public keys $\{X_1, \ldots, X_n\}$ is equivalent to $g^s = R\widetilde{X}^c$ where $\widetilde{X} = \prod_{i=1}^{n} X_i$ and $c = H(\widetilde{X}, R, m)$. Note that this is exactly the verification equation for an ordinary key-prefixed Schnorr signature with respect to the aggregate public key $\widetilde{X}$. However, as already pointed out many times [HMP95, Lan96, MH96, MOR01], this simplistic protocol is vulnerable to a rogue-key attack where a corrupted signer sets its public key to $X_1 = g^{x_1}(\prod_{i=2}^{n} X_i)^{-1}$, allowing him to produce signatures for public keys $\{X_1, \ldots, X_n\}$ by himself. One way to generically prevent rogue-key attacks is to require that users prove possession of the secret key, e.g., by attaching a zero-knowledge proof to their public keys [RY07, BDN18]. However, this makes key management cumbersome, complicates implementations, and is not compatible with existing and widely used key serialization formats.

The MuSig Scheme. A different and more direct approach proposed by Bellare and Neven [BN06] is to work in the *plain public-key model*, where the only requirement is that each potential signer has a public key. To date, the only multi-signature scheme that is fully compatible with Schnorr signatures and provably secure without proofs of possession is MuSig by Maxwell *et al.* [MPSW19], independently proven secure by Boneh, Drijvers, and Neven [BDN18].

[3] Since we do not impose any constraint on the key setup, the adversary can choose corrupted public keys arbitrarily, hence the same public key can appear multiple times in L.

In order to overcome rogue-key attacks in the plain public-key model, MuSig computes partial signatures s_i with respect to "signer-dependent" challenges

$$c_i = \mathsf{H}_{\mathrm{agg}}(L, X_i) \cdot \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m),$$

where $\widetilde{X}$ is the *aggregate public key* corresponding to the multiset of public keys $L = \{X_1, \ldots, X_n\}$. It is defined as $\widetilde{X} = \prod_{i=1}^{n} X_i^{a_i}$ where $a_i = \mathsf{H}_{\mathrm{agg}}(L, X_i)$ (note that the a_i's only depend on the public keys of the signers). This way, the verification equation of a signature (R, s) on message m for public keys $L = \{X_1, \ldots, X_n\}$ becomes

$$g^s = R \prod_{i=1}^{n} X_i^{a_i c} = R\widetilde{X}^c,$$

where $c = \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$. This recovers the key aggregation property enjoyed by the naive scheme, albeit with respect to a more complex aggregate key $\widetilde{X} = \prod_{i=1}^{n} X_i^{a_i}$.

In order to be able to simulate an honest signer in a run of the signing protocol via the standard way of programming the random oracle $\mathsf{H}_{\mathrm{sig}}$, MuSig has an initial commitment round (like the scheme by Bellare and Neven [BN06]) where each signer commits to its share R_i before receiving the shares of other signers.

As a result, the signing protocol of MuSig requires three communication rounds, and only the initial commitment round can be preprocessed without knowing the message to be signed.[4]

Two-Round Schemes. Following the scheme by Bellare and Neven [BN06], in which signing requires three rounds of interaction, multiple attempts to reduce this number to two rounds [BN06, BCJ08, STV+16, MPSW19] (without requiring pairings) were foiled by Drijvers *et al.* [DEF+19]. In their pivotal work, they show that all thus far proposed two-round schemes in the DL setting cannot be proven secure and are vulnerable to attacks with subexponential complexity when the adversary is allowed to engage in an arbitrary number of concurrent sessions (*concurrent security*), as required by the standard definition of unforgeability.

If one prefers a scheme in the DL setting with fewer communication rounds, only two options remain, and none of them is fully satisfactory. The first option is the mBCJ scheme by Drijvers *et al.* [DEF+19], a repaired variant of the scheme by Bagherzandi, Cheon, and Jarecki [BCJ08]. While mBCJ needs only two rounds, it does not output ordinary Schnorr signatures and is thus not suitable as a drop-in replacement for Schnorr signatures, e.g., in cryptocurrencies whose validation rules support Schnorr signatures (such as proposed for Bitcoin). The second option is MuSig-DN (MuSig with Deterministic Nonces) [NRSW20], which however relies on heavy zero-knowledge proofs to prove a deterministic derivation of the nonce to all cosigners. This increases the complexity of the implementation significantly and makes MuSig-DN, even though it needs only two rounds, in fact less efficient than three-round MuSig in common setting due to the expensive zero-knowledge proofs. Moreover, in neither of these two-round schemes is it possible to reduce the rounds further by preprocessing the first round without knowledge of the message to be signed.

1.2 Our Contribution

We propose a novel and simple two-round variant of the MuSig scheme that we call MuSig2. In particular, we remove the preliminary commitment phase, so that signers start right away by sending nonces. However, to obtain a scheme secure under concurrent sessions, each signer i sends a list of $\nu \geq 2$ nonces $R_{i,1}, \ldots, R_{i,\nu}$ (instead of a single nonce R_i), and effectively uses a linear combination $\hat{R}_i = \prod_{j=1}^{\nu} R_{i,j}^{b_j}$ of these ν nonces, where $b_1 = 1$ and the coefficients b_j, $2 \leq j \leq \nu$, are derived via a hash function

Except for MuSig-DN [NRSW20], which relies on rather complex and expensive zero-knowledge proofs (proving time $\approx 1\,\mathrm{s}$), our scheme is the first multi-signature scheme that simultaneously

[4] The second move of the protocol is independent of the message to be signed, which makes it tempting to process this move without knowledge of the message. But revealing the second move to the cosigners before the message is fixed renders the scheme insecure [Nic19].

i) is secure under concurrent signing sessions, *ii)* supports key aggregation, *iii)* outputs ordinary Schnorr signatures, and *iv)* needs only two communication rounds. Furthermore, our scheme is the first in the DL setting that supports preprocessing of all but one rounds, effectively enabling non-interactive signing without forgoing security under concurrent sessions.

In comparison to other multi-signature schemes based on Schnorr signatures, we pay for this round efficiency by relying on a stronger assumption: instead of the standard DL assumption, we need the One-More Discrete Logarithm (OMDL) assumption [BP02, BNPS03], which states that it is hard to find the discrete logarithm of $q+1$ group elements by making at most q calls to an oracle solving the discrete logarithm problem.

We give two independent security proofs which reduce the security of MuSig2 to the hardness of the OMDL problem. Our first proof relies on the random oracle model (ROM), and requires $\nu \geq 4$ in general. Our second proof additionally assumes the algebraic group model (AGM) [FKL18], and for this ROM+AGM proof, $\nu = 2$ nonces are sufficient.

Assuming a group element is as large as a (collision-resistant) hash of a group element, the overhead of MuSig2 as compared to normal (three-round) MuSig is broadcasting $\nu - 2$ group elements and a multi-exponentiation of size $\nu - 1$. As a result, MuSig2 is highly practical. In particular for $\nu = 2$ nonces, the overhead is just a single exponentiation.

2 Technical Overview

2.1 The Challenge of Constructing Two-Round Schemes

Already an obsolete preliminary version [MPSW18] of the MuSig paper [MPSW19] proposed a 2-round variant of MuSig where the initial commitment round is omitted and claimed provable security under the OMDL assumption. We will call this scheme InsecureMuSig in the following. However, Drijvers *et al.* [DEF+19] discovered a flaw in the security proof (as well as in the proof of the other 2-round DL-based multi-signature schemes by Bagherzandi *et al.* [BCJ08] and Ma *et al.* [MWLD10]). They show through a meta-reduction that the concurrent security of these schemes cannot be reduced to the DL or OMDL problem using an algebraic black-box reduction (assuming the OMDL problem is hard).[5]

In addition to the meta-reduction, Drijvers *et al.* [DEF+19] also gave a concrete attack of subexponential complexity based on Wagner's algorithm [Wag02] for solving the Generalized Birthday Problem [Wag02], which has led to similar attacks on Schnorr blind signatures [Sch01]. Their attack breaks InsecureMuSig and the other aforementioned multi-signature schemes and inherently exploits the ability to run multiple sessions concurrently. Recently, Benhamouda *et al.* [BLOR20] gave a novel, simple, and very efficient attack of polynomial complexity, which confirm and extend these negative results.

A Concrete Attack. We outline the attack by Drijvers *et al.* [DEF+19] in order to provide an intuition for how we can overcome their negative results. The attack relies on Wagner's algorithm for solving the Generalized Birthday Problem [Wag02], which can be defined as follows for the purpose of this paper: Given a constant value $t \in \mathbb{Z}_p$, an integer $k_{\max}$, and access to random oracle H mapping onto $\mathbb{Z}_p$, find a set $\{q_1, \ldots, q_{k_{\max}}\}$ of $k_{\max}$ queries such that $\sum_{k=1}^{k_{\max}} H(q_k) = t$. While for $k_{\max} \leq 2$, the complexity of this problem is the same as finding a preimage ($k_{\max} = 1$) or a collision ($k_{\max} = 2$) in the random oracle, the problem becomes, maybe surprisingly, easy for large $k_{\max}$. In particular, Wagner [Wag02] gives a subexponential algorithm assuming that $k_{\max}$ is not bounded.

The attack proceeds as follows. The adversary opens $k_{\max}$ concurrent signing sessions, in which it plays the role of the signer with public key $X_2 = g^{x_2}$, and receives $k_{\max}$ nonces $R_1^{(1)}, \ldots, R_1^{(k_{\max})}$ from the honest signer with public key $X_1 = g^{x_1}$. Let $\widetilde{X} = X_1^{a_1} X_2^{a_2}$ be the corresponding aggregate public key. Given a forgery target message m^*, the adversary sets $R^* := \prod_{k=1}^{k_{\max}} R_1^{(k)}$ and uses

[5] We refer the interested reader to Appendix C for a high-level explanation of why the meta-reduction cannot be adapted to work with our scheme.

Wagner's algorithm to find nonces $R_2^{(k)}$ such that

$$\sum_{k=1}^{k_{\max}} \underbrace{\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R_1^{(k)} R_2^{(k)}, m^{(k)})}_{=:\, c^{(k)}} = \underbrace{\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R^*, m^*)}_{=:\, c^*}. \tag{1}$$

Having received $R_2^{(k)}$, the honest signer will reply with partial signatures $s_1^{(k)} = r_1^{(k)} + c^{(k)} \cdot a_1 x_1$. Let $r^* := \sum_{k=1}^{k_{\max}} r_1^{(k)} = \mathsf{DL}(R^*)$. The adversary is able to obtain

$$s_1^* := \sum_{k=1}^{k_{\max}} s_1^{(k)} = \sum_{k=1}^{k_{\max}} r_1^{(k)} + \left(\sum_{k=1}^{k_{\max}} c^{(k)} \right) \cdot a_1 x_1 = r^* + c^* \cdot a_1 x_1,$$

where the last equality follows from Equation (1). The adversary can further complete the value s_1^* to

$$s^* := s_1^* + c^* \cdot a_2 x_2 = r^* + c^* \cdot (a_1 x_1 + a_2 x_2).$$

In other words, (R^*, s^*) is a valid forgery on message m^* with signature hash $c^* = \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R^*, m^*)$. In this example, the forgery is valid for the aggregate public key $\widetilde{X}$, which is the result of aggregating public keys X_1 and X_2. It is however straightforward to adapt the attack to produce a forgery under a different aggregate public key as long as it is the result of aggregating the honest signer's public key X_1 with any multiset of adversarial public keys.

The complexity of this attack is dominated by the complexity of Wagner's algorithm, which is $O(k_{\max}\, 2^{\log_2(p)/(1+\lfloor(\log_2(k_{\max})\rfloor)})$. While this is subexponential (and not polynomial), the attack is practical for common parameters and moderately large numbers $k_{\max}$ of sessions. For example, for a group size of $p \approx 2^{256}$ as common for elliptic curves, a value of $k_{\max} = 128$ brings the complexity of the attack down to approximately 2^{39} operations, which is practical even on off-the-shelf hardware. If the attacker is able to open more sessions concurrently, the improved polynomial-time attack by Benhamouda *et al.* [BLOR20] assumes $k_{\max} > \log_2 p$ sessions, but then has complexity $O(k_{\max} \log_2 p)$ and a negligible running time in practice.

2.2 Our Solution

The attack by Drijvers *et al.* (and similarly the attack by Benhamouda *et al.*) relies on the ability to control the signature hash by controlling the aggregate nonce $R_1^{(k)} R_2^{(k)}$ (on the LHS of Equation (1)) in the first round of each of the concurrent signing sessions. Since all signers must know the aggregate nonce at the end of the first round, it seems hard to prevent the adversary from being able to control the aggregate nonce on the LHS without adding a preliminary commitment round.

Our high-level idea to solve this problem and to foil the attacks is to accept that the adversary can control the LHS of the equation but prevent it from controlling the RHS instead.

The main novelty in our work is to let every signer i send a list of $\nu \geq 2$ nonces $R_{i,1}, \ldots, R_{i,\nu}$ and let it effectively use a random linear combination $\hat{R}_i = \prod_{j=1}^{\nu} R_{i,j}^{b_j}$ of those nonces in lieu of the former single nonce R_i. The linear coefficients b_j are derived via a hash function $\mathsf{H}_{\mathrm{non}}$ (modeled as a random oracle) applied the nonces of all signers, i.e., $b_j = \mathsf{H}_{\mathrm{non}}(j, \widetilde{X}, (\prod_{i=1}^{n} R_{i,1}, \ldots, \prod_{i=1}^{n} R_{i,\nu}), m)$.[6]

As a result, whenever the adversary tries different values for $R_2^{(k)}$, the coefficients $b_1^{(k)}, \ldots, b_\nu^{(k)}$ change, and so does the honest signer's effective nonce $\hat{R}_1^{(k)} = \prod_{j=1}^{\nu} R_{1,j}^{b_j}$ as well as the value $R^* := \prod_{k=1}^{k_{\max}} \hat{R}_1^{(k)}$ on the RHS of Equation (1). This ensures that the RHS is no longer a constant value, which is however an essential prerequisite in the definition of the Generalized Birthday Problem and thus for the applicability of Wagner's algorithm.

[6] Since the values $R_{i,j}$ end up as input to a hash function, one may wonder why we propose to take products $\prod_{i=1}^{n} R_{i,j}$ instead of simply concatenating all $R_{i,j}$. While we believe that concatenation also yields a secure scheme, we note the products $\prod_{i=1}^{n} R_{i,j}$ anyway need to be computed when computing $R = \prod_{i=1}^{n} \hat{R}_i = \prod_{i=1}^{n} \prod_{j=1}^{\nu} R_{i,j}^{b_j} = \prod_{j=1}^{\nu} \left(\prod_{i=1}^{n} R_{i,j}\right)^{b_j}$ with a minimal number of exponentiations.

With this idea in mind, it is tempting to fall back to a single nonce only ($\nu = 1$) but instead rely just on the coefficient b_1. However, the adversary can effectively eliminate this simple tweak by instead considering the equation

$$\sum_{k=1}^{k_{\max}} \frac{\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, (R_1^{(k)})^{b_1^{(k)}}, m^{(k)})}{b_1^{(k)}} = \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R^*, m^*).$$

While this explains why using $\nu \geq 2$ is essential, our security proof will demonstrate that fixing $b_1 = 1$ (i.e., randomizing only the remaining coefficients $b_2, \ldots, b_\nu$) is an optimization that does not hamper security.

2.3 Proving Security

Before we describe how to prove MuSig2 secure, we first take a step back to InsecureMuSig in order to understand the flaw in its purported security proof. Then, we explain how the usage of more than once nonce in MuSig2 enables us to fix that flaw.

The Difficulty of Simulating Signatures. Following the textbook security proof of Schnorr signatures, a natural but necessarily flawed approach to reduce the security of InsecureMuSig[7] to the DL problem in the ROM will be to let the reduction announce the challenge group element X_1 as the public key of the honest signer and fork the execution of the adversary in order to extract the discrete logarithm of X_1 from the two forgeries output by the adversary in its two executions (using the Forking Lemma [BN06, PS00]).

The insurmountable difficulty for the reduction in this approach is to simulate the honest signer's participation in signing sessions without knowledge of the secret key of the honest signer. From the perspective of the reduction, simply omitting the preliminary commitment phase enables the adversary to know the combined nonce R before the reduction learns it, which prevents the reduction from simulating the signing oracle using the standard technique of programming the random oracle on the signature challenge $\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$. In more details, observe that in InsecureMuSig, an adversary (controlling public keys $X_2, \ldots, X_n$) can impose the value of $R = \prod_{i=1}^{n} R_i$ used in signing sessions since it can choose $R_2, \ldots, R_n$ after having received R_1 from the honest signer (controlling the public key $X_1 = g^{x_1}$). This forbids to use the textbook way of simulating the honest signer in the Random Oracle Model (ROM) without knowing x_1 by randomly drawing s_1 and c, computing $R_1 = g^{s_1}(X_1)^{-a_1 c}$, and later programming $\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m) := c$, since the adversary might have made the random oracle query $\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$ *before* starting the corresponding signing session.

The Flawed Security Proof of InsecureMuSig. The hope of Maxwell *et al.* [MPSW18] was to rely on the stronger OMDL assumption instead of the DL assumption in order to solve this problem without a commitment round. The DL oracle in the formulation of the OMDL problem would enable the reduction to obtain s_1 via a DL oracle query for the discrete logarithm of $R_1(X_1)^{a_1 c}$, using a fresh DL challenge as R_1 in each signing session, whose discrete logarithm could be computed once x_1 had been retrieved. Together with the DL challenge X_1 used as public key of the honest signer, this would mean that the reduction computes the DL of $q_s + 1$ challenge elements using only q_s DL oracle calls, where q_s is the number of signing sessions asked for by the adversary, i.e., the reduction would solve the OMDL problem.

This simulation technique however fails in a subtle way when combined with the Forking Lemma, since the adversary might be forked in the middle of a signing session, when it has received R_1 but has not returned $R_2, \ldots, R_n$ to the reduction yet. Now assume that the adversary sends different values $R_2, \ldots, R_n$ and $R'_2, \ldots, R'_n$ in the two executions of the fork, resulting in different signatures hashes c and c' respectively. This implies that in order to correctly simulate the signing oracle in the forked execution, the reduction needs *two* queries to the DL oracle, both of which are related to the same single challenge R_1. Since the answer of the first DL oracle query will already be enough to compute the discrete logarithm of R_1 later on, the second query does not provide any additional useful information to the reduction (neither about the discrete logarithm of R_1 nor about the

[7] Observe that InsecureMuSig is identical to a purported MuSig2 with a just a single nonce, i.e., $\nu = 1$.

discrete logarithm of another DL challenge) and is thus wasted. As a result, the reduction forgoes any hope to solve the OMDL problem when making the second query. Exactly this issue is exploited in the meta-reduction by Drijvers *et al.* [DEF+19] in order to extract the signing key from the reduction (which was supposed to simulate *without* knowledge of the signing key).

How Multiple Nonces in MuSig2 Help the Reduction. With MuSig2 however, the reduction can handle this situation. Now assume $\nu = 2$, i.e., the reduction will obtain two (instead of one) group elements $R_{1,1}, R_{1,2}$ as challenges from the OMDL challenger during the first round of each signing session. This will allow the reduction to make two DL queries per signing session, and thus be able to simulate signatures even if the adversary forces different signature hashes $c \neq c'$ in the two executions.

Recall that the honest signer simulated by the reduction effectively uses the linear combination $\hat{R}_1 = R_{1,1}R_{1,2}^{b_2}$ as nonce. If we additionally ensure that whenever c' and c differ, then also the coefficients b_2' and b_2 in the two executions differ, and the two DL queries made by reduction will give linear independent equations that can be solved for the discrete logarithms of the challenges $R_{1,1}$ and $R_{1,2}$. Similarly, in the case that $c = c'$, the reduction needs only one DL query to simulate the honest signer in both executions, and thus it can use the free DL query to obtain a second linear independent equation.

Note that for this simulation technique, it is not important how the adversary controls the signature hashes c and c'. So far we only considered the case that the adversary influences c and c' by choosing its nonces depending on the honest signer's nonce. Thus, the reduction works equally for an adversary which controls the signature hash computed as $\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$ not by influencing R but instead by being able to choose the message m or the set of signers L (and thus the aggregate public key $\widetilde{X}$) only in the second round of the signing protocol, i.e., after having seen the honest signer's nonce. This explains why our scheme enables preprocessing and broadcasting the nonces (the first round) without having determined the message and the set of signers. This is in contrast to existing schemes, which are vulnerable to essentially the same attack as explained above if the adversary is given the ability to select the message or the set of signers after having seen the honest signer's nonce [Nic19].

So far we discussed only how the reduction is able to handle two different executions of the adversary (due to a single fork). However, since our reduction needs to fork the adversary twice to support key aggregation, it needs to handle four possible executions of the adversary. As a consequence, it will need four DL queries as well as $\nu \geq 4$ nonces. Moreover, if the number q of $\mathsf{H}_{\mathrm{non}}$ hash queries made by the adversary for $\nu = 4$ nonces is too large, e.g., close to $\sqrt{p}$, we cannot guarantee that the reduction will obtain four linear independent equations with sufficient probability, and thus we may need to resort to $\nu \geq 5$.

2.4 A More Efficient Solution in the Algebraic Group Model

In the algebraic group model (AGM) [FKL18], the adversary is assumed to be algebraic, i.e., whenever it outputs a group element, it is required to output a representation of this group element in the base formed by all group elements it has received so far. While the AGM is idealized, it is a strictly weaker model than the generic group model (GGM) [Sho97], i.e, security proofs in the AGM carry over to the GGM but the AGM imposes fewer restrictions on the adversary. Security proofs in the AGM work via reductions to hard problems (similar to the standard model) because computational problems such as DL and OMDL are not information-theoretically hard in the AGM (as opposed to the GGM). In the AGM, the Schnorr signature scheme (and related schemes such Schnorr blind signatures [CP93]) can be proven secure using a straight-line reduction which does not fork the execution of the adversary [FPS20].

The main technical reason why our ROM proof works only for $\nu \geq 4$ nonces is that our reduction needs to handle four executions of the adversary due to two applications of the Forking Lemma. Since this fundamental reason for requiring $\nu \geq 4$ in the plain ROM simply disappears in the AGM, we are able to prove MuSig2 with $\nu = 2$ nonces secure in AGM+ROM.

2.5 Concurrent Work: FROST

Concurrently to our work, Komlo and Goldberg [KG20] updated the draft of their threshold Schnorr signature scheme FROST to rely on a similar idea of using a linear combination of multiple nonces in order to remove a communication round while achieving security under concurrent sessions. However, the exact method of combining nonces in FROST is different to MuSig2.

A major difference between MuSig2 and FROST is that the idea is employed in different contexts. Komlo and Goldberg [KG20] consider the more general threshold signature setting, which inherently requires an interactive key setup and handling of some possible disruptive signers. We consider multi-signatures only, but this enables us to focus on features unique to multi-signatures, e.g., non-interactive key aggregation.

A second major difference is the cryptographic model. The FROST security proof relies on a non-standard heuristic which models the hash function (a public primitive) used for deriving the coefficients for the linear combination as a one-time VRF (a primitive with a secret key) in the security proof. This treatment requires an additional communication round in FROST preprocessing stage and to disallow concurrent sessions in this stage, resulting in a modified scheme FROST-Interactive. As a consequence, the modified FROST-Interactive scheme that is proven secure differs significantly from the round-efficient FROST scheme that is recommended for deployment. The proof reduces the security of FROST-Interactive to the DL problem. In contrast, our MuSig2 proofs use the well-established ROM (or alternatively, AGM+ROM) to model the hash function as a random oracle and rely on the hardness of the OMDL problem.

3 Preliminaries

3.1 Notation and Definitions

Notation. The security parameter is denoted λ. Given a non-empty 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 $y := A(x_1, \ldots; \rho)$ denote the operation of running A on inputs $x_1, \ldots$ and random coins ρ and assigning its output to y, and $y \leftarrow A(x_1, \ldots)$ when coins ρ are chosen uniformly at random.

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 representation when given as input to hash functions.

Definition 1 (OMDL Problem). *Let $(\mathbb{G}, p, g)$ be group parameters. Let $\mathsf{DL}_g(\cdot)$ be an oracle taking as input an element $X \in \mathbb{G}$ and returning $x \in \{0, \ldots, p-1\}$ such that $g^x = X$. An algorithm $\mathcal{A}$ is said to (q, t, ε)-solve the OMDL problem w.r.t. $(\mathbb{G}, p, g)$ if on input $q+1$ random group elements $X_1, \ldots, X_{q+1}$, it runs in time at most t, makes at most q queries to $\mathsf{DL}_g(\cdot)$, and returns $x_1, \ldots, x_{q+1} \in \{0, \ldots, p-1\}$ such that $X_i = g^{x_i}$ for all $1 \leq i \leq q+1$ with probability at least ε, where the probability is taken over the random draw of $X_1, \ldots, X_q$ and the random coins of $\mathcal{A}$.*

3.2 Syntax and Security Definition of Multi-Signature Schemes

To keep the notation simple, we make a few simplifying assumptions in the remainder of the paper. In particular, we restrict our syntax and security model to two-round signing algorithms, and in order to model that the first round can be preprocessed without having determined a message to be signed or the public keys of all signers, those two inputs are given only to the second round of the signing algorithm. It is straightforward to extend our model to signing algorithms with a different number of rounds or different input handling. Our syntax further assumes that each signer outputs a signature, but most multi-signature schemes (in particular the one presented in this paper) can be easily modified so that a single designated participant computes the final signature.

SYNTAX. A multi-signature scheme Σ consists of algorithms $(\mathsf{Setup}, \mathsf{KeyGen}, (\mathsf{Sign}, \mathsf{Sign}', \mathsf{Sign}''), \mathsf{Ver})$. System-wide parameters par are generated by the setup algorithm Setup taking as input the security parameter. For notational simplicity, we assume that par is given as input to all other algorithms and do not denote it explicitly in the following.

The randomized key generation algorithm takes no input and returns a private/public key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow_{\$} \mathsf{KeyGen}()$. The interactive signature algorithm $(\mathsf{Sign}, \mathsf{Sign}', \mathsf{Sign}'')$ is run by each signer and proceeds in a sequence of three steps (two communication rounds) resulting in the output of a signature σ as follows:

$$\begin{aligned}(msg_1, state_1) &\leftarrow \mathsf{Sign}(\mathsf{sk}_1)\\ (msg_1', state_1') &\leftarrow \mathsf{Sign}'(state_1, m, ((\mathsf{pk}_2, msg_2), \ldots, (\mathsf{pk}_n, msg_n)))\\ \sigma &\leftarrow \mathsf{Sign}''(state_1', (msg_2', \ldots, msg_n')).\end{aligned}$$

The deterministic verification algorithm Ver takes a multiset of public keys $L = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$, a message m, and a signature σ, and returns 1 if the signature is valid for L and m and 0 otherwise.

Correctness requires that for every λ, every message m, every integer n, and every $j \in \{1, \ldots, n\}$,

$$\Pr\left[\begin{array}{l}\mathsf{par} \leftarrow \mathsf{Setup}(1^\lambda)\\ (\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathsf{KeyGen}(),\ i = 1 \ldots n\\ (msg_i, state_i) \leftarrow \mathsf{Sign}(\mathsf{sk}_i),\ i = 1 \ldots n\\ (msg_i', state_i') \leftarrow \mathsf{Sign}'(state_i, m, ((\mathsf{pk}_1, msg_1), \ldots, (\mathsf{pk}_{i-1}, msg_{i-1}),\\ \qquad\qquad (\mathsf{pk}_{i+1}, msg_{i+1}), \ldots, (\mathsf{pk}_n, msg_n))),\ i = 1 \ldots n\\ \sigma \leftarrow \mathsf{Sign}''(state_j', (msg_1', \ldots, msg_{j-1}', msg_{j+1}', \ldots, msg_n'))\\ L := \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}\\ b \leftarrow \mathsf{Ver}(L, m, \sigma)\end{array} : b = 1\right]$$

is equal to 1.

SECURITY. Our security model is the same as the one of Bellare and Neven [BN06] and requires that it is infeasible to forge multi-signatures involving at least one honest signer. As in previous work [MOR01, Bol03, BN06], we assume *wlog* that there is a single honest signer and that the adversary has corrupted all other signers, choosing corrupted public keys arbitrarily (and potentially as a function of the honest signer's public key).

The security game $\mathsf{EUF\text{-}CMA}_\Sigma^{\mathcal{A}}$ is defined as follows (see also Figure 1). A key pair $(\mathsf{sk}_1, \mathsf{pk}_1)$ is generated for the honest signer and the adversary $\mathcal{A}$ gets pk_1. The adversary can engage in any number of (concurrent) signing sessions with the honest signer before returning a forgery attempt. Formally, $\mathcal{A}$ has access to oracles SIGN, SIGN$'$, and SIGN$''$ implementing the three steps of the signing algorithm with the honest signer's secret key. Eventually, the adversary returns a multiset of public keys $L = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$, a message m, and a signature σ. The game returns 1 (representing a win of $\mathcal{A}$) if $\mathsf{pk}_1 \in L$, the forgery is valid, i.e., $\mathsf{Ver}(L, m, \sigma) = 1$, and the adversary never made a SIGN$'$ query for multiset L and message m. In addition, if we work in the random oracle model, the adversary can make arbitrary random oracle queries at any stage of the game.

Security is formally defined as follows.

Definition 2 (EUF-CMA). *Let $\Sigma = (\mathsf{Setup}, \mathsf{KeyGen}, (\mathsf{Sign}, \mathsf{Sign}', \mathsf{Sign}''), \mathsf{Ver})$ be a 2-round multi-signature scheme. Let game $\mathsf{EUF\text{-}CMA}_\Sigma^{\mathcal{A}}$ be as defined in Figure 1. Then Σ is* existentially unforgeable under chosen-message attacks *(EUF-CMA) if for any p.p.t. adversary $\mathcal{A}$,*

$$\mathsf{Adv}_{\Sigma,\mathcal{A}}^{\text{euf-cma}}(\lambda) := \Pr\left[1 \leftarrow \text{EUF-CMA}_\Sigma^{\mathcal{A}}(\lambda)\right] = \mathsf{negl}(\lambda)\,.$$

4 Our New Multi-Signature Scheme

4.1 Description

Our new multi-signature scheme MuSig2 is parameterized by a group generation algorithm GrGen and defined in Figure 2.

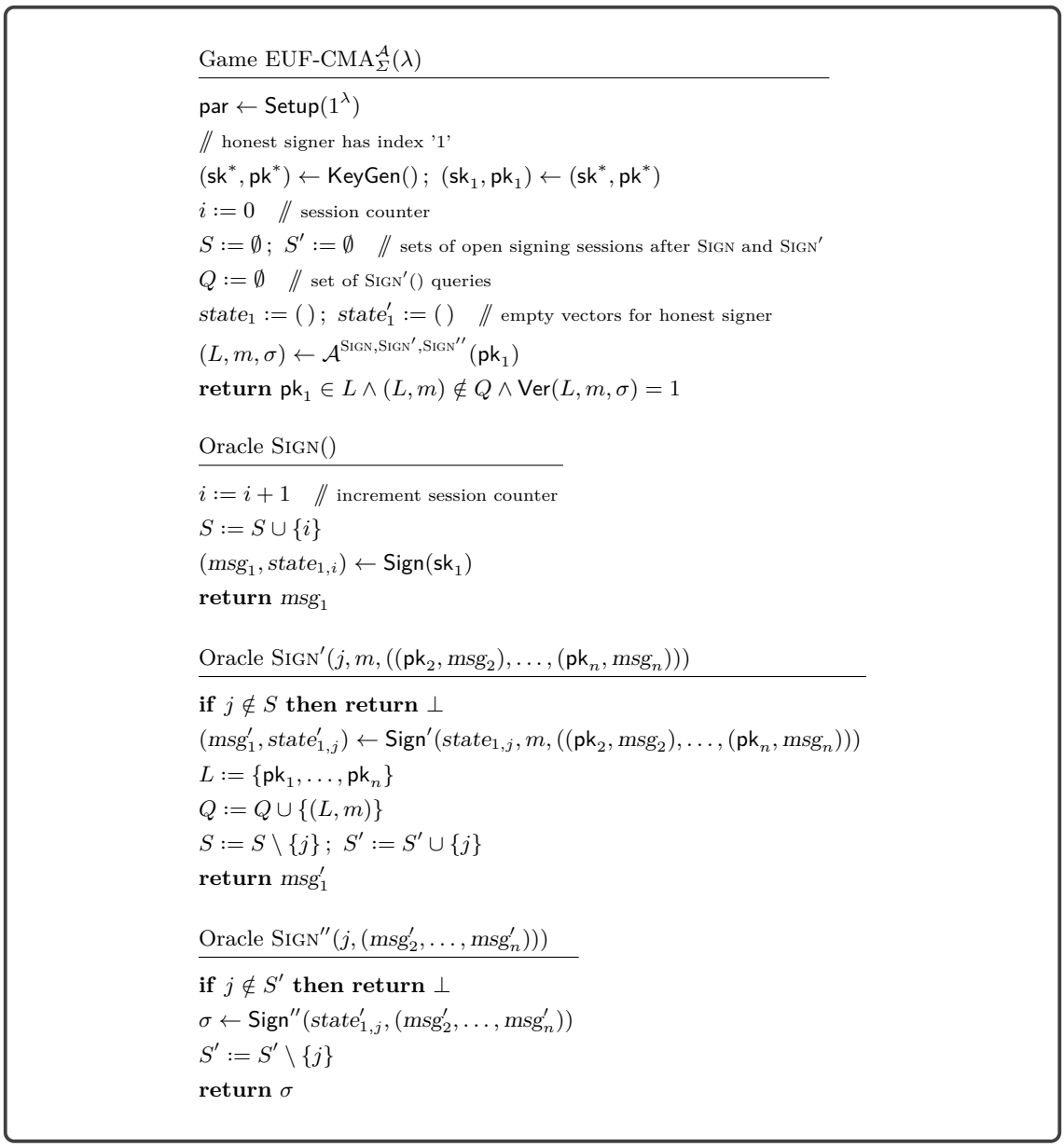

Fig. 1. The EUF-CMA security game for a multi-signature scheme Σ.

Parameters setup. On input 1^λ, the setup algorithm Setup runs $(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$, selects three hash functions[8] $\mathsf{H}_{\mathrm{agg}}$, $\mathsf{H}_{\mathrm{non}}$, and $\mathsf{H}_{\mathrm{sig}}$ from $\{0,1\}^*$ to $\mathbb{Z}_p$, and returns $\mathsf{par} := ((\mathbb{G}, p, g), \mathsf{H}_{\mathrm{agg}}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{sig}})$.

Key generation. Each signer generates a random private key $x \leftarrow_\$ \mathbb{Z}_p$ and computes the corresponding public key $X = g^x$.

First signing step (Sign and communication round). Let X_1 and x_1 be the public and private key of a specific signer. For each $j \in \{1, \ldots, \nu\}$ the signer generates random $r_{1,j} \leftarrow_\$ \mathbb{Z}_p$, computes $R_{1,j} = g^{r_{1,j}}$, and broadcasts $(R_{1,1}, \ldots, R_{1,\nu})$ to all potential cosigners.[9]

Second signing step (Sign' and communication round). Let m be the message to sign, let $X_2, \ldots, X_n$ be the public keys of other cosigners, and let $L = \{X_1, \ldots, X_n\}$ be the multiset of all public keys involved in the signing process.[10] For $i \in \{1, \ldots, n\}$, the signer computes $a_i = \mathsf{H}_{\mathrm{agg}}(L, X_i)$ and then the "aggregate" public key $\widetilde{X} = \prod_{i=1}^n X_i^{a_i}$. Upon reception of $(R_{2,1}, \ldots, R_{2,\nu}), \ldots, (R_{n,1}, \ldots, R_{n,\nu})$ from the other cosigners, the signer computes $R_j = \prod_{i=1}^n R_{i,j}$ for each $j \in \{1, \ldots, \nu\}$, and then the coefficient vector $(b_1, \ldots, b_\nu)$ as $b_1 = 1$ and $b_j = \mathsf{H}_{\mathrm{non}}(j, \widetilde{X}, (R_1, \ldots, R_\nu), m)$ for $j \in \{2, \ldots, \nu\}$. Then it computes

$$
\begin{aligned}
R &= \prod_{j=1}^{\nu} R_j^{b_j}, \\
c &= \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m), \\
s_1 &= c a_1 x_1 + \sum_{j=1}^{\nu} r_{1,j} b_j \bmod p,
\end{aligned}
$$

and sends s_1 to all other cosigners.

Final signing step (Sign''). Finally, upon reception of $s_2, \ldots, s_n$ from the other cosigners, the signer can compute $s = \sum_{i=1}^n s_i \bmod p$. The signature is $\sigma = (R, s)$.

Verification. Given a multiset of public keys $L = \{X_1, \ldots, X_n\}$, a message m, and a signature $\sigma = (R, s)$, the verifier computes $a_i = \mathsf{H}_{\mathrm{agg}}(L, X_i)$ for $i \in \{1, \ldots, n\}$, $\widetilde{X} = \prod_{i=1}^n X_i^{a_i}$, $c = \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$ and accepts the signature if $g^s = R \prod_{i=1}^n X_i^{a_i c} = R \widetilde{X}^c$.

Correctness is straightforward to verify. Note that verification is exactly the same as for standard key-prefixed Schnorr signatures with respect to the "aggregate" public key $\widetilde{X} = \prod_{i=1}^n X_i^{a_i}$.

4.2 Practical Considerations

The Choice of the Number ν of Nonces. Since we provide a security proof in the ROM (Section 5) assuming $\nu \geq 4$ nonces as well as a second independent proof in the ROM+AGM (Section 6) assuming $\nu = 2$ nonces, one may wonder about the choice of ν in practice.

Since there is no evidence that $\mathsf{MuSig2}$ is insecure for $\nu = 2$, this is the obvious and most efficient choice if one is willing to accept the combination of the ROM and the AGM (remember that the AGM is strictly weaker than the GGM, i.e., it puts fewer restrictions on the adversary as compared to the GGM). Moreover, our ROM+AGM proof offers a tighter reduction to the OMDL problem.

If one prefers a ROM-only proof instead, then given a choice of p, the concrete security bound in Theorem 1 guides the choice of ν. In particular, ν must be chosen such that the term $43q^4/(p-1)^{\nu-3}$ is small for a number q of (random oracle plus signing) queries which is appropriate for the desired security level.

[8] Hash function $\mathsf{H}_{\mathrm{agg}}$ is used to compute the aggregate key, $\mathsf{H}_{\mathrm{non}}$ is used to aggregate nonces, and $\mathsf{H}_{\mathrm{sig}}$ to compute the signature. These hash functions can be constructed from a single one using proper domain separation.

[9] The cosigners (represented by their public keys) need not yet be determined in the first round.

[10] As in [BN06], indices $1, \ldots, n$ are local references to signers, defined within the specific signer instance at hand.

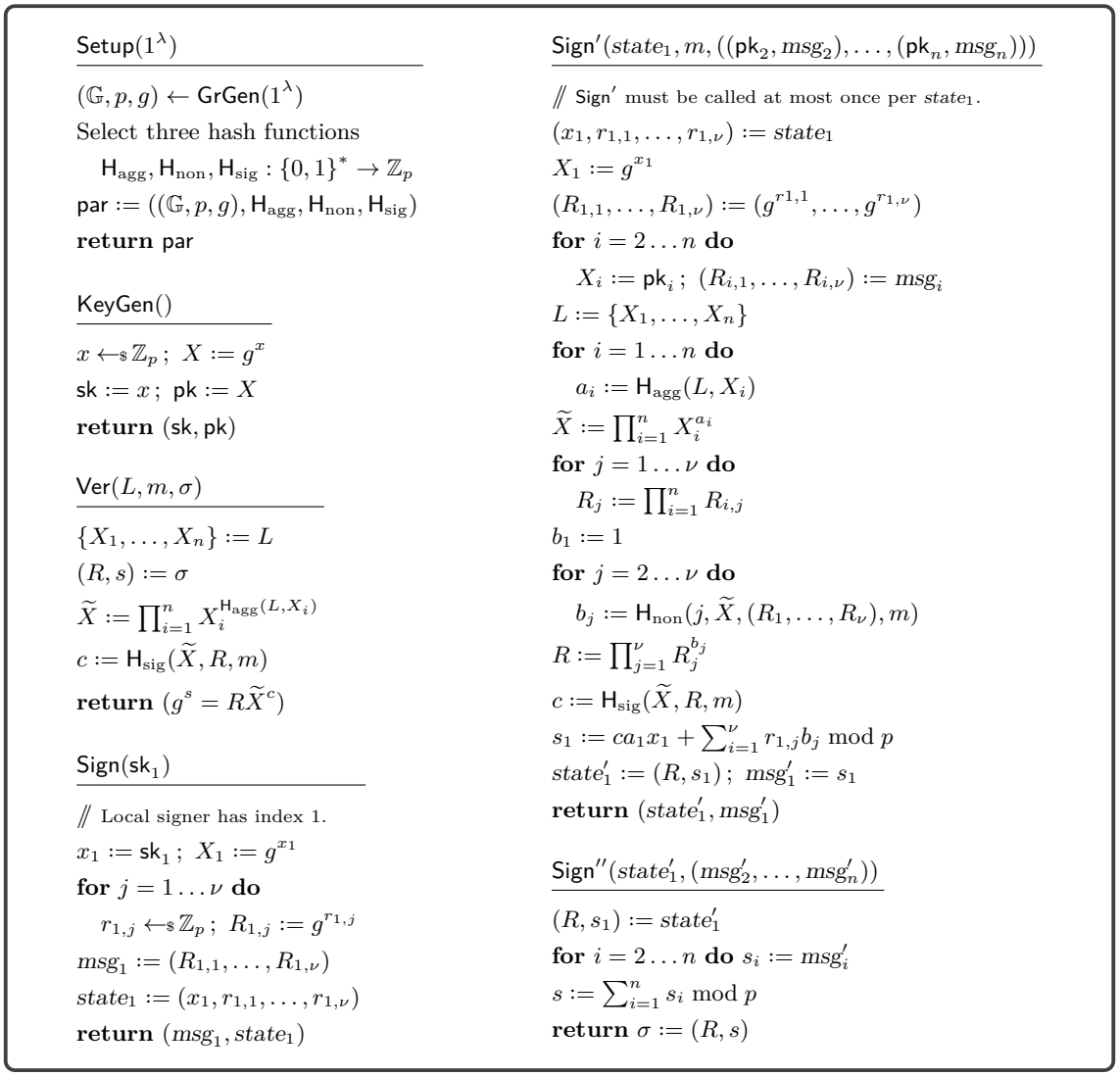

Fig. 2. The multi-signature scheme MuSig2[GrGen]. Public parameters par returned by Setup are implicitly given as input to all other algorithms.

Deterministic Nonces are Insecure. To protect against failures in the randomness generation, it is common in practice to derandomize the signing procedure of DL-based signature schemes by deriving the random values used as exponents for the nonces ($r_{1,j}$ in our case) using a deterministic pseudorandom function of the secret key and the message instead of drawing the nonces uniformly at random. However, this technique is in general insecure when applied to multi-signatures, and Maxwell *et al.* [MPSW19] describe an attack that applies to essentially all Schnorr multi-signature schemes in the literature when derandomized naively, including MuSig2. Therefore our signing protocol requires a secure random number generator for generating the values $r_{1,j}$. The only known way to sidestep this issue is to securely derandomize the signing protocol using expensive zero-knowledge proofs as proposed in the MuSig-DN [NRSW20] scheme.

Statefulness. After executing Sign' with some state the signer must make sure to never run Sign' again with the same state. Otherwise, the signer will reuse the nonce, allowing trivial extraction of the secret key. (Again, similar attacks apply to essentially all Schnorr multi-signature schemes, except the fully deterministic MuSig-DN [NRSW20].) Guaranteeing correct state transitions may be difficult in practice if the state is written to persistent storage. In particular, the state may be reused by accident when restoring a backup or through a deliberate attack on the physical storage.

5 Security of MuSig2 in the ROM

In this section, we establish the security of MuSig2 in the random oracle model.

Theorem 1. *Assume that there exists a $(t, q_s, q_h, N, \varepsilon)$-adversary $\mathcal{A}$ against the multi-signature scheme* MuSig2 *with $\nu \geq 4$, group parameters $(\mathbb{G}, p, g)$ and with hash functions $\mathsf{H}_{\mathrm{agg}}$, $\mathsf{H}_{\mathrm{non}}$, $\mathsf{H}_{\mathrm{sig}} : \{0,1\}^* \to \mathbb{Z}_p$ modeled as random oracles. Then there exists an algorithm $\mathcal{D}$ which $(\nu q_s, t', \varepsilon')$-solves the OMDL problem for $(\mathbb{G}, p, g)$, with*

$$t' = 4(t + q(N + 2\nu - 2))t_{\mathrm{exp}} + O(qN)$$

where $q = (\nu - 1)(q_h + q_s) + 1$ and t_{exp} is the time of an exponentiation in $\mathbb{G}$ and

$$\varepsilon' \geq \frac{\varepsilon^4}{q^3} - \frac{11}{p} - \frac{43q^4}{(p-1)^{\nu-3}}.$$

Before proving the theorem, we start with an informal explanation of the key techniques used in the proof. Let us recall the security game defined in Section 3.2, adapting the notation to our setting. Group parameters $(\mathbb{G}, p, g)$ are fixed and a key pair (x^*, X^*) is generated for the honest signer. The target public key X^* is given as input to the adversary $\mathcal{A}$. Then, the adversary can engage in protocol executions with the honest signer by providing a message m to sign and a multiset L of public keys involved in the signing process where X^* occurs at least once, and simulating all signers except one instance of X^*.

The Double-Forking Technique. This technique is already used by Maxwell *et al.* in the security proof for MuSig [MPSW19]. We are repeating the idea below with slightly modified notation.

The first difficulty is to extract the discrete logarithm x^* of the challenge public key X^*. The standard technique for this would be to "fork" two executions of the adversary in order to obtain two valid forgeries (R, s) and (R', s') for the same multiset of public keys $L = \{X_1, \ldots, X_n\}$ with $X^* \in L$ and the same message m such that $R = R'$, $\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$ was programmed in both executions to some common value h_{sig}, $\mathsf{H}_{\mathrm{agg}}(L, X_i)$ was programmed in both executions to the same value a_i for each i such that $X_i \neq X^*$, and $\mathsf{H}_{\mathrm{agg}}(L, X^*)$ was programmed to two distinct values h_{agg} and h'_{agg} in the two executions, implying that

$$g^s = R(X^*)^{n^* h_{\mathrm{agg}} h_{\mathrm{sig}}} \prod_{\substack{i \in \{1,\ldots,n\} \\ X_i \neq X^*}} X_i^{a_i h_{\mathrm{sig}}}$$

$$g^{s'} = R(X^*)^{n^* h'_{\mathrm{agg}} h_{\mathrm{sig}}} \prod_{\substack{i \in \{1,\ldots,n\} \\ X_i \neq X^*}} X_i^{a_i h_{\mathrm{sig}}},$$

where n^* is the number of times X^* appears in L. This would allow to compute the discrete logarithm of X^* by dividing the two equations above.

However, simply forking the executions with respect to the answer to the query $\mathsf{H}_{\text{agg}}(L, X^*)$ does not work: indeed, at this moment, the relevant query $\mathsf{H}_{\text{sig}}(\widetilde{X}, R, m)$ might not have been made yet by the adversary,[11] and there is no guarantee that the adversary will ever make this same query again in the second execution, let alone return a forgery corresponding to the same H_{sig} query. In order to remedy this situation, we fork the execution of the adversary *twice*: once on the answer to the query $\mathsf{H}_{\text{sig}}(\widetilde{X}, R, m)$, which allows us to retrieve the discrete logarithm of the aggregate public key $\widetilde{X}$ with respect to which the adversary returns a forgery, and then on the answer to $\mathsf{H}_{\text{agg}}(L, X^*)$, which allows us to retrieve the discrete logarithm of X^*.

As in Bellare and Neven [BN06], our technical tool to handle forking of the adversary is a "generalized Forking Lemma" which extends Pointcheval and Stern's Forking Lemma [PS00] and which does not mention signatures nor adversaries and only deals with the outputs of an algorithm $\mathcal{A}$ run twice on related inputs. However, the generalized Forking Lemma of Bellare and Neven [BN06] is not general enough for our setting, and we rely on a slight variant which we state and prove in Appendix A.

Simulating the Honest Signer. For now, consider the scheme with $\nu = 1$. (We will illustrate the problem of this choice further down in this section.) The adversary has access to an interactive signing oracle, which enables it to open sessions with the honest signer. The signing oracle consists of three sub-oracles Sign, Sign′, and Sign″ but note that we can *wlog* ignore Sign″, which computes the final signature $s = \sum_{i=1}^{n} s_i \bmod p$, because it does not depend on the secret key x^* and thus the adversary can simply simulate it locally.

The reduction's strategy for simulating the signing oracle is to use the DL oracle available in the formulation of the OMDL problem as follows. Whenever the adversary starts the k-th signing session by querying Sign, the reduction uses a fresh DL challenge $R_{1,1}$ from the OMDL challenger and returns it as its nonce to the adversary. At any later time the adversary queries Sign′ with session counter k, a message m to sign, and $n-1$ pairs of public keys and group elements $(X_2, R_{2,1}), \ldots, (X_n, R_{n,1})$. The reduction then sets $L = \{X_1 = X^*, X_2, \ldots, X_n\}$, computes $\widetilde{X}$, and uses the DL oracle in the formulation of the OMDL problem to compute s_1 as follows.

$$R = \prod_{i=1}^{n} R_{i,1}, \qquad c = \mathsf{H}_{\text{sig}}(\widetilde{X}, R, m), \qquad s_1 = \mathsf{DL}_g(R_{1,1} + ca_1\widetilde{X}),$$

It then returns s_1 to the adversary. We use a fresh DL challenge as $R_{1,1}$ in each signing query, and the reduction will be able to compute its discrete logarithm $r_{1,1}$ once x^* has been retrieved via $r_{1,1} = ca_1x^* - s_1$.

Leveraging Two or More Nonces. The main obstacle in the proof and the novelty in this work is to handle adversaries whose behavior follows this pattern: The adversary initiates a signing session by querying the oracle Sign to obtain $R_{1,1}$, then makes a query $\mathsf{H}_{\text{sig}}(\widetilde{X}, R, m)$, for which it will output a forgery later, and only then continues the signing session with a query to Sign′ with $m, ((X_2, R_{2,1}), \ldots, (X_n, R_{n,1}))$. Our goal is to fork the execution of the adversary at the H_{sig} query. But then, the adversary may send different values $m, ((X_2, R_{2,1}), \ldots, (X_n, R_{n,1}))$ in the two executions. In that case, this results in different signature hashes and requires the reduction simulating the honest signer to make two DL oracle queries in order to answer the Sign′ query. Consequently, the reduction will lose the OMDL game because it had only requested the single OMDL challenge $R_{1,1}$.

This is exactly where $\nu \geq 2$ nonces will come to the rescue. Now assume $\nu = 2$, i.e., the reduction will obtain two (instead of one) group elements $R_{1,1}, R_{1,2}$ as challenges from the OMDL challenger. This will allow the reduction to make two DL queries. In order to answer Sign′, the reduction

[11] In fact, it is easy to see that the adversary can only guess the value of the aggregate public key $\widetilde{X}$ corresponding to L at random before making the relevant queries $\mathsf{H}_{\text{agg}}(L, X_i)$ for $X_i \in L$, so that the query $\mathsf{H}_{\text{sig}}(\widetilde{X}, R, m)$ can only come after the relevant queries $\mathsf{H}_{\text{agg}}(L, X_i)$ except with negligible probability.

follows the MuSig2 scheme by computing $\widetilde{X}$ from the public keys, and b_2 by hashing $\widetilde{X}$, m and all R values of the signing session with H_{non}. The reduction then aggregates the nonces of the honest signer into its effective nonce $\hat{R}_1 = R_{1,1}R_{1,2}^{b_2}$, queries the signature hash c and replies to the adversary with $s_1 = \mathsf{DL}_g(\hat{R}(X^*)^{a_1 c})$.

Now the reduction is able to make another DL_g query to compute s_1' and answer the SIGN$'$ query in the second execution. Moreover, to ensure that the OMDL challenge responses $r_{1,1}$ and $r_{1,2}$ can be computed after extracting x^*, the reduction programs H_{non} to give different responses in each execution after a fork. Let us assume for now that the signing session was started with a SIGN query after the H_{agg} fork. Then we can distinguish the following two cases depending on when H_{non} is queried with the inputs corresponding to the signing session:

- H_{non} **is queried *after* the** H_{sig} **fork.** Regardless of what values the adversary sends in SIGN$'$, the second execution will use a value b_2' that is different from b_2 in the first execution. In order to answer the signing query, the reduction uses DL_g to compute s_1' resulting in a system of linear equations with unknowns $r_{1,1}$ and $r_{1,2}$:

$$r_{1,1} + b_2 r_{1,2} = s_1 - a_1 c x^* \bmod p$$
$$r_{1,1} + b_2' r_{1,2} = s_1' - a_1' c' x^* \bmod p$$

  Because the system is linearly independent (as $b_2 \neq b_2'$) we can solve for the unknowns and forward them to the OMDL challenger.
- H_{non} **is queried *before* the** H_{sig} **fork.** This implies that b_2 in the first execution is equal to b_2' in the second execution and requires the reduction to ensure that a_1' and c' are identical in both executions. Then the input to the DL_g query is also identical and the reduction can reuse the result of the DL_g query from the first execution. Otherwise, the reduction would need a second DL_g query to compute s_1' but would not have a second, independent equation that allows solving for $r_{1,1}$ and $r_{1,2}$.
  The value a_1 is equal to a_1' because the inputs of H_{non} contain $\widetilde{X}$ which implies that the corresponding H_{agg} happened before H_{non} and therefore before the fork. Similarly, H_{sig} requires the aggregate nonce R of the signing session and therefore H_{non} must be queried before the corrsponding H_{sig}. In order to argue that $c = c'$, observe that from the inputs (and output) of a H_{non} query it is possible to compute the inputs of the H_{sig} query. Therefore, the reduction can make such an "internal" H_{sig} query for every H_{non} query it receives. This H_{sig} query is before the fork point implying $c = c'$ as desired. (The reduction does not need to handle the case that this H_{sig} query *is* the fork point, because then the values L and m of forgery were queried in a signing session and thus the forgery invalid.) Now the reduction has a DL_g query left to compute the discrete logarithm of $R_{1,1}$, which enables to compute the discrete logarithm of $R_{1,2}$ after x^* has been extracted.

More generally, if the signing session can be started before the H_{agg} fork, the reduction may have to provide different signatures in all four executions. To answer the signature queries nonetheless, the reduction requires four DL queries and therefore can not be applied to MuSig2 with $\nu < 4$. Similar to the above, if H_{non} is queried after H_{sig}, the reduction ends up with four equations allowing to compute $r_{1,1}, \ldots, r_{1,4}$ of the signing session if they are linearly independent. Otherwise, signatures will be identical across executions and the remaining DL_g queries are used to set up a linear system to solve for the unknowns.

Since the coefficients of the linear system are responses to H_{non} queries and drawn uniformly at random, we need to analyze whether the system has a unique solution. For example, if q is close to $\sqrt{p}$, then H_{non} can be queried often enough that with high probability the adversary can complete a signing session such that the equations in the linear system are not independent. In this case, one needs to increase ν even further to $\nu \geq 5$. This lets the reduction draw additional OMDL challenges $R_{1,5} = g^{r_{1,5}}, \ldots, R_\nu = g^{r_{1,\nu}}$. For each additional challenge we select a coefficient vector for $r_{1,2}, ..., r_{1,\nu}$ that is linearly independent of the existing coefficient row vectors and use a DL_g query to add an equation to the linear system. As with $\nu = 4$ we use one DL query per OMDL challenge. But because the system now has $\nu \geq 5$ uniformly random coefficients per equation, the probability that there are H_{non} responses that would make the four equations controlled by the adversary linearly dependent is negligible, which we can show using Lemma 7.

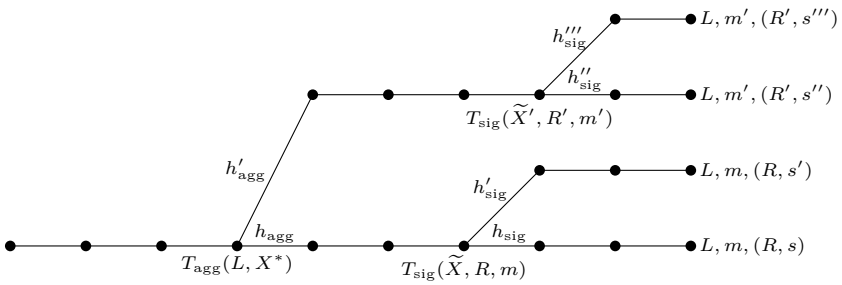

Fig. 3. A possible execution of algorithm $\mathcal{D}$. Each path from the leftmost root to one of the four rightmost leaves represent an execution of the adversary. Each vertex symbolizes an assignment to tables T_{agg} or T_{sig} used to program H_{agg} and H_{sig}, and the edge originating from this vertex symbolizes the value used for the assignment. Leaves symbolize the forgery returned by the adversary. Only vertices and edges that are relevant to the forgery are labeled.

5.1 Security Proof

Proof Overview. We first construct a "wrapping" algorithm $\mathcal{B}$ which essentially runs the adversary and returns a forgery together with some information about the adversary execution, unless some bad events happen.[12] Algorithm $\mathcal{B}$ simulates the random oracles H_{agg}, H_{non}, and H_{sig} uniformly at random and the signing oracle by obtaining ν DL challenges from the OMDL challenger for each Sign query and by making a single query to the DL oracle for each Sign$'$ query. Then, we use $\mathcal{B}$ to construct an algorithm $\mathcal{C}$ which runs the forking algorithm $\mathsf{Fork}^{\mathcal{B}}$ as defined in Section 3 (where the fork is w.r.t. the answer to the H_{sig} query related to the forgery), allowing it to return a multiset of public keys L together with the discrete logarithm of the corresponding aggregate public key. Finally, we use $\mathcal{C}$ to construct an algorithm $\mathcal{D}$ computing the DL of the public key of the honest signer by running $\mathsf{Fork}^{\mathcal{C}}$ (where the fork is now w.r.t. the answer to the H_{agg} query related to the forgery). Throughout the proof, the reader might find helpful to refer to Figure 3 which illustrates the inner working of $\mathcal{D}$.

Using careful programming of random oracles, it is ensured that the ν DL challenges that $\mathcal{B}$ obtains in each Sign query are identical across all executions of $\mathcal{B}$. Since $\mathcal{D}$ (via $\mathcal{C}$ and $\mathcal{B}$) obtains $1 + \nu q_s$ DL challenges (one for the public key of the honest signer and ν for each of the q_s signing sessions) and solved all of these challenges using at most νq_s queries to the DL oracle (one for each of the q_s signing session in at most $4 \leq \nu$ executions due to double-forking), algorithm $\mathcal{D}$ solves the OMDL problem.

Normalizing Assumptions. In all the following, we assume that the adversary never repeats a query, and only makes "well-formed" queries, meaning that $X^* \in L$ and $X \in L$ for any query $\mathsf{H}_{\text{agg}}(L, X)$ and $2 \leq j \leq \nu$ for any query $\mathsf{H}_{\text{non}}(j, \ldots)$. This is without loss of generality, since "ill-formed" queries are irrelevant and could simply be answered uniformly at random in the simulation. Moreover, we assume that the adversary closes every signing session, i.e., for every Sign query it will also make a corresponding Sign$'$ query at some point. This is again without loss of generality because missing Sign$'$ query could be emulated after the adversary has terminated (using a set of public keys and a message m which are different from the forgery in the output but otherwise arbitrary, to make sure not to invalidate a valid forgery). We also assume *wlog* that the adversary makes exactly q_h queries to each random oracle and opens exactly q_s signing sessions.

Lemma 1. *Assume that there exists a $(t, q_s, q_h, N, \varepsilon)$-adversary $\mathcal{A}$ in the random oracle model against the multi-signature scheme* MuSig2 *with group parameters $(\mathbb{G}, p, g)$ and let $q = (\nu - 1)(q_h + q_s) + 1$. Then there exists an algorithm $\mathcal{B}$ that takes as input $\nu q_s + 1$ uniformly random*

[12] In particular, we must exclude the case where the adversary is able to find two distinct multisets of public keys L and L' such that the corresponding aggregate public keys are equal, since when this happens the adversary can make a signing query for (L, m) and return the resulting signature σ as a forgery for (L', m). Jumping ahead, this will correspond to bad event $\mathsf{KeyColl}$ defined in the proof of Lemma 1.

group elements $X^*, U_1, \ldots, U_{\nu q_s}$ *and uniformly random scalars* $h_{\mathrm{agg},1}, \ldots, h_{\mathrm{agg},q}$, $h_{\mathrm{non},1}, \ldots, h_{\mathrm{non},q}$, $h_{\mathrm{sig},1}, \ldots, h_{\mathrm{sig},q} \in \mathbb{Z}_p$,[13] *makes at most* q_s *queries to a discrete logarithm oracle* $\mathsf{DL}_g(\cdot)$*, and, with accepting probability (as defined in Lemma 4) at least*

$$\varepsilon - 2q^2/p,$$

outputs $(i_{\mathrm{agg}}, j_{\mathrm{agg}}, i_{\mathrm{sig}}, j_{\mathrm{sig}}, L, R, s, \mathbf{a})$ *where* $i_{\mathrm{agg}}, i_{\mathrm{sig}} \in \{1, \ldots, q\}$, $j_{\mathrm{agg}}, j_{\mathrm{sig}} \in \{0, \ldots, q\}$, $L = \{X_1, \ldots, X_n\}$ *is a multiset of public keys such that* $X^* \in L$, $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbb{Z}_p^n$ *is a tuple of scalars such that* $a_i = h_{\mathrm{agg},i_{\mathrm{agg}}}$ *for any* i *such that* $X_i = X^*$*, and*

$$g^s = R \prod_{i=1}^{n} X_i^{a_i h_{\mathrm{sig},i_{\mathrm{sig}}}}. \tag{2}$$

Proof. We construct algorithm $\mathcal{B}$ as follows. It initializes three empty sets T_{agg}, T_{non} and T_{sig} for storing key-value pairs (k, v), which we write in assignment form "$T(k) := v$" for a set T. The sets represent tables for storing programmed values for respectively $\mathsf{H}_{\mathrm{agg}}$, $\mathsf{H}_{\mathrm{non}}$ and $\mathsf{H}_{\mathrm{sig}}$. It also initializes and three counters $\mathsf{ctrh}_{\mathrm{agg}}$, $\mathsf{ctrh}_{\mathrm{non}}$, $\mathsf{ctrh}_{\mathrm{sig}}$, and ctrs (initially zero) that will be incremented respectively each time an entry of the form $T_{\mathrm{agg}}(\cdot, X^*)$ is assigned, each time an assignment is made in T_{sig}, and each time the adversary makes a SIGN query. It also initializes two flags $\mathsf{BadOrder}$ and $\mathsf{KeyColl}$ that will help keep track of bad events. In order to keep track of the open signing sessions, $\mathcal{B}$ initializes an empty set $\mathsf{Sessions}$. Then, it picks random coins ρ_F, runs the adversary $\mathcal{A}$ with the public key X^* as input and answers its queries as follows.

- *Hash query* $\mathsf{H}_{\mathrm{agg}}(L, X)$*:* (Recall that by assumption, $X^* \in L$ and $X \in L$.) If $T_{\mathrm{agg}}(L, X)$ is undefined, then $\mathcal{B}$ increments $\mathsf{ctrh}_{\mathrm{agg}}$, randomly assigns $T_{\mathrm{agg}}(L, X') \leftarrow\!\!\$\ \mathbb{Z}_p$ for all $X' \in L \setminus \{X^*\}$, and assigns $T_{\mathrm{agg}}(L, X^*) := h_{\mathrm{agg},\mathsf{ctrh}_{\mathrm{agg}}}$. Then, it returns $T_{\mathrm{agg}}(L, X)$.
- *Hash query* $\mathsf{H}_{\mathrm{non}}(j, \widetilde{X}, (R_1, \ldots, R_\nu), m)$*:* (Recall that by assumption, $2 \le j \le \nu$.) If the entry $T_{\mathrm{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$ is undefined, then algorithm $\mathcal{B}$ assigns $T_{\mathrm{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m) := (1, h_{\mathrm{non},\mathsf{ctrh}_{\mathrm{non}}+1}, \ldots, h_{\mathrm{non},\mathsf{ctrh}_{\mathrm{non}}+(\nu-1)})$ and increments $\mathsf{ctrh}_{\mathrm{non}}$ by $\nu - 1$. $\mathcal{B}$ parses $(b_1, \ldots, b_\nu) := T_{\mathrm{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$ and computes $R := \prod_{i=j}^{\nu} R_j^{b_j}$, If $T_{\mathrm{sig}}(\widetilde{X}, R, m)$ is undefined, then $\mathcal{B}$ makes an "internal" query to $\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$. Finally it returns b_j.
- *Hash query* $\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$*:* If $T_{\mathrm{sig}}(\widetilde{X}, R, m)$ is undefined, then $\mathcal{B}$ increments $\mathsf{ctrh}_{\mathrm{sig}}$ and assigns $T_{\mathrm{sig}}(\widetilde{X}, R, m) := h_{\mathrm{sig},\mathsf{ctrh}_{\mathrm{sig}}}$. Then, it returns $T_{\mathrm{sig}}(\widetilde{X}, R, m)$.
- *Signing query* SIGN()*:* $\mathcal{B}$ increments ctrs, adds ctrs to $\mathsf{Sessions}$, lets $k' := \nu(\mathsf{ctrs} - 1) + 1$ and sends $(R_{1,1} := U_{k'}, \ldots, R_{1,\nu} := U_{k'+\nu-1})$ to the adversary.
- *Signing query* $\mathrm{SIGN}'(k, m, ((\mathsf{pk}_2, msg_2), \ldots, (\mathsf{pk}_n, msg_n)))$*:* If $k \notin \mathsf{Sessions}$ then the signing query is answered with $\perp$. Otherwise, $\mathcal{B}$ removes k from $\mathsf{Sessions}$. Let $k' := \nu(k-1) + 1$ and $R_{1,1} := U_{k'}, \ldots, R_{1,\nu} := U_{k'+\nu-1}$. Let $X_i := \mathsf{pk}_i$ for each $i \in \{2, \ldots, n\}$ and let $L := \{X_1 = X^*, X_2, \ldots, X_n\}$. If $T_{\mathrm{agg}}(L, X^*)$ is undefined,[14] $\mathcal{B}$ makes an "internal" query to $\mathsf{H}_{\mathrm{agg}}(L, X^*)$ which ensures that $T_{\mathrm{agg}}(L, X_i)$ is defined for each $i \in \{1, \ldots, n\}$. It sets $a_i := T_{\mathrm{agg}}(L, X_i)$ and computes $\widetilde{X} := \prod_{i=1}^{n} X_i^{a_i}$. Then $\mathcal{B}$ sets $(R_{i,1}, \ldots, R_{i,\nu}) := msg_i$ for each $i \in \{2, \ldots, n\}$ and computes $R_j := \prod_{i=1}^{n} R_{i,j}$ for each $j \in \{1, \ldots, \nu\}$. If $T_{\mathrm{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$ is undefined, then $\mathcal{B}$ makes an "internal" query to $\mathsf{H}_{\mathrm{non}}(2, \widetilde{X}, (R_1, \ldots, R_\nu), m)$. It sets $b_1 := 1$, parses $(b_2, \ldots, b_\nu) := T_{\mathrm{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$, aggregates the nonces as $R := \prod_{j=1}^{\nu} R_j^{b_j}$, and sets $c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$. (Note that $T_{\mathrm{sig}}(\widetilde{X}, R, m)$ is defined due to the internal $\mathsf{H}_{\mathrm{sig}}$ query when handling $\mathsf{H}_{\mathrm{non}}$ queries, which also implies that $\mathcal{B}$ can save the exponentiations necessary to compute R using caching.) Finally, $\mathcal{B}$ computes the honest signer's effective nonce $\hat{R}_1 := \prod_{j=1}^{\nu} R_{1,j}^{b_j}$, sets $s_1 := \mathsf{DL}_g(\hat{R}_1 (X^*)^{a_1 c})$ by querying the DL oracle, and returns s_1.

[13] Strings $h_{\mathrm{agg},i}$, $h_{\mathrm{non},i}$, $h_{\mathrm{sig},i}$ will be used to answers queries to $\mathsf{H}_{\mathrm{agg}}$, $\mathsf{H}_{\mathrm{non}}$, $\mathsf{H}_{\mathrm{sig}}$, respectively. $\mathcal{B}$ needs at most $q = (\nu - 1)(q_h + q_s) + 1$ answers per hash oracle because each hash function query uses up at most $\nu - 1$ elements, each signing query requires updating the tables corresponding to the hash functions and thereby uses up at most $\nu - 1$ elements and final verification of the validity of the forgery uses up at most one element.

[14] This is true *iff* L never appeared in a previous $\mathsf{H}_{\mathrm{agg}}$ or signing query.

If $\mathcal{A}$ returns $\perp$, then $\mathcal{B}$ outputs $\perp$ as well. Otherwise, the adversary returns a purported forgery (R, s) for a public key multiset L such that $X^* \in L$ and a message m. Then, $\mathcal{B}$ parses L as $\{X_1 = X^*, \ldots, X_n\}$ and checks the validity of the forgery as follows. If $T_{\mathrm{agg}}(L, X^*)$ is undefined, it makes an internal query to $\mathsf{H}_{\mathrm{agg}}(L, X^*)$ which ensures that $T_{\mathrm{agg}}(L, X_i)$ is defined for each $i \in \{1, \ldots, n\}$, sets $a_i := T_{\mathrm{agg}}(L, X_i)$, and computes $\widetilde{X} := \prod_{i=1}^{n} X_i^{a_i}$. If $T_{\mathrm{sig}}(\widetilde{X}, R, m)$ is undefined, it makes an internal query to $\mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$ and lets $c := T_{\mathrm{sig}}(\widetilde{X}, R, m)$.[15] If $g^s \neq R\widetilde{X}^c$, i.e., the forgery is not a valid signature, or if the forgery is invalid because the adversary made a $\textsc{Sign}'$ query for L and m, $\mathcal{B}$ outputs $\perp$. Otherwise, it takes the following additional steps. Let

- i_{agg} be the index such that $T_{\mathrm{agg}}(L, X^*) = h_{\mathrm{agg},i_{\mathrm{agg}}}$,
- j_{agg} be the value of $\mathsf{ctrh}_{\mathrm{non}}$ at the moment $T_{\mathrm{agg}}(L, X^*)$ is assigned.
- i_{sig} be the index such that $T_{\mathrm{sig}}(\widetilde{X}, R, m) = h_{\mathrm{sig},i_{\mathrm{sig}}}$,
- j_{sig} be the value of $\mathsf{ctrh}_{\mathrm{non}}$ at the moment $T_{\mathrm{sig}}(\widetilde{X}, R, m)$ is assigned.

$\mathcal{B}$ sets $\mathsf{BadOrder} := \mathtt{true}$ and returns $\perp$ if the assignment $T_{\mathrm{agg}}(L, X^*) := h_{\mathrm{agg},i_{\mathrm{agg}}}$ occurred *after* the assignment $T_{\mathrm{sig}}(\widetilde{X}, R, m) := h_{\mathrm{sig},i_{\mathrm{sig}}}$. $\mathcal{B}$ sets $\mathsf{KeyColl} := \mathtt{true}$ and returns $\perp$ if if there exists another multiset of public keys L' such that, at the end of the execution, $T_{\mathrm{agg}}(L', X')$ is defined for each $X' \in L'$ and the aggregate keys corresponding to L and L' are equal. Otherwise, it returns $(i_{\mathrm{agg}}, i_{\mathrm{sig}}, L, R, s, \mathbf{a})$, where $\mathbf{a} = (a_1, \ldots, a_n)$. By construction, $a_i = h_{\mathrm{agg},i_{\mathrm{agg}}}$ for each i such that $X_i = X^*$, and the validity of the forgery implies Equation (2).

We now lower bound the accepting probability of $\mathcal{B}$. Since $h_{\mathrm{agg},1}, \ldots, h_{\mathrm{agg},q}$, $h_{\mathrm{non},1}, \ldots, h_{\mathrm{non},q}$ and $h_{\mathrm{sig},1}, \ldots, h_{\mathrm{sig},q}$ are uniformly random, $\mathcal{B}$ perfectly simulates the security experiment to the adversary. Moreover, when the adversary eventually returns a forgery, $\mathcal{B}$ returns a non-$\perp$ output unless $\mathsf{BadOrder}$ or $\mathsf{KeyColl}$ is set to $\mathtt{true}$. Hence, $\mathsf{acc}(\mathcal{B}) \geq \varepsilon - \Pr[\mathsf{BadOrder}] - \Pr[\mathsf{KeyColl}]$ by the union bound.

It remains to upper bound $\Pr[\mathsf{BadOrder}]$ and $\Pr[\mathsf{KeyColl}]$, for which it will be convenient to introduce the following wording. Note that by construction of $\mathcal{B}$, for any multiset L' appearing at some point in the queries of the adversary or in its forgery, assignments $T_{\mathrm{agg}}(L', X')$ for all $X' \in L'$ are concomitant and occur the first time L' appears either in a query to $\mathsf{H}_{\mathrm{agg}}$, or in a signing query, or in the forgery. We will refer to the set of assignments $\{T_{\mathrm{agg}}(L', X') := a', X' \in L'\}$ as the *set of* T_{agg} *assignments related to* L'. Note that there are at most q sets of T_{agg} assignments and that each of them contains a unique assignment $T_{\mathrm{agg}}(L', X^*) := h_{\mathrm{agg},i}$ for some $i \in \{1, \ldots, q\}$.

In order to upper bound the probability that $\mathsf{BadOrder}$ is set to $\mathtt{true}$, we upper bound the probability that some set of T_{agg} assignments related to some multiset L' (not necessarily the one returned in the forgery) results in the aggregate key $\widetilde{X}'$ corresponding to L' being equal to the first argument of a defined entry in table T_{sig} (which is clearly a necessary condition for $\mathsf{BadOrder}$ to be set to $\mathtt{true}$). Considering the i-th set of T_{agg} assignments, one has

$$\widetilde{X}' = (X^*)^{n^* h_{\mathrm{agg},i}} \cdot Z$$

where $n^* \geq 1$ is the number of times X^* appears in L' and $h_{\mathrm{agg},i}$ is uniformly random in $\mathbb{Z}_p$ and independent of Z which accounts for public keys different from X^* in L'. Hence, $\widetilde{X}'$ is uniformly random in a set of at least p group elements. Since there are always at most q defined entries in T_{sig} and at most q sets of T_{agg} assignments, $\mathsf{BadOrder}$ is set to $\mathtt{true}$ with probability at most q^2/p.

In order to upper bound the probability that $\mathsf{KeyColl}$ is set to $\mathtt{true}$, we upper bound the probability that some set of T_{agg} assignments related to some multiset L' (not necessarily the one returned in the forgery) results in the aggregate key $\widetilde{X}'$ corresponding to L' being equal to the aggregate key $\widetilde{X}''$ corresponding to some previous set of T_{agg} assignments related to some other multiset L'' (again, neither L' nor L'' need be the multiset returned in the forgery). Since each aggregate key is uniform in a set of at least p group elements and independent of other aggregate keys, this happens with probability at most q^2/p.

Combining all of the above, we obtain

$$\mathsf{acc}(\mathcal{B}) \geq \varepsilon - 2q^2/p. \qquad \square$$

[15] In general, we cannot assume that the adversary has made the random oracle queries corresponding to its forgery attempt, even though the forgery is valid only with negligible probability in this case.

Using $\mathcal{B}$, we now construct an algorithm $\mathcal{C}$ which returns a multiset of public keys L together with the discrete logarithm of the corresponding aggregate key.

Lemma 2. *Assume that there exists a $(t, q_s, q_h, N, \varepsilon)$-adversary $\mathcal{A}$ in the random oracle model against the multi-signature scheme* MuSig2*, group parameters $(\mathbb{G}, p, g)$ and let $q = (\nu-1)(q_h+q_s)+1$. Then there exists an algorithm $\mathcal{C}$ that takes as input $\nu q_s + 1$ uniformly random group elements $X^*, U_1, \ldots, U_{\nu q_s}$ and uniformly random scalars $h_{\mathrm{agg},1}, \ldots, h_{\mathrm{agg},q}$, $h_{\mathrm{non},1}, h'_{\mathrm{non},1}, \ldots, h_{\mathrm{non},q}, h'_{\mathrm{non},q} \in \mathbb{Z}_p$, makes at most $2q_s$ queries to a discrete logarithm oracle $\mathsf{DL}_g(\cdot)$, and, with accepting probability (as defined in Lemma 4) at least*

$$\frac{\varepsilon^2}{q} - \frac{4q+1}{p},$$

outputs a tuple $(i_{\mathrm{agg}}, j_{\mathrm{agg}}, L, \mathbf{a}, \tilde{x})$ where $i_{\mathrm{agg}} \in \{1, \ldots, q\}$, $j_{\mathrm{agg}} \in \{0, \ldots, q\}$, $L = \{X_1, \ldots, X_n\}$ is a multiset of public keys such that $X^* \in L$, $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbb{Z}_p^n$ is a tuple of scalars such that $a_i = h_{\mathrm{agg},i_{\mathrm{agg}}}$ for any i such that $X_i = X^*$, and $\tilde{x}$ is the discrete logarithm of $\widetilde{X} = \prod_{i=1}^n X_i^{a_i}$ in base g.

Proof. Algorithm $\mathcal{C}$ runs $\mathsf{Fork}^{\mathcal{B}}$ with $\mathcal{B}$ as defined in Lemma 1 and takes additional steps as described below. The mapping with notation of our Forking Lemma (Lemma 4 in Appendix A) is as follows:

- X^*, $h_{\mathrm{agg},1}, \ldots, h_{\mathrm{agg},q}$, and $U_1, \ldots, U_{\nu q_s}$ play the role of inp,
- $h_{\mathrm{sig},1}, \ldots, h_{\mathrm{sig},q}$ play the role of $h_1, \ldots, h_q$,
- $h_{\mathrm{non},1}, h'_{\mathrm{non},1} \ldots, h_{\mathrm{non},q}, h'_{\mathrm{non},q}$ play the role of $v_1, v'_1, \ldots, v_m, v'_m$,
- $(i_{\mathrm{sig}}, j_{\mathrm{sig}})$ play the role of (i, j),
- $(i_{\mathrm{agg}}, j_{\mathrm{agg}}, L, R, s, \mathbf{a})$ play the role of out.

In more details, $\mathcal{C}$ picks random coins ρ_A and runs algorithm $\mathcal{B}$ on coins ρ_A, group elements $X^*, U_1, \ldots, U_{\nu q_s}$, and scalars $h_{\mathrm{agg},1}, \ldots, h_{\mathrm{agg},q}$, $h_{\mathrm{non},1}, \ldots, h_{\mathrm{non},q}$, $h_{\mathrm{sig},1}, \ldots, h_{\mathrm{sig},q} \in \mathbb{Z}_p$, where $h_{\mathrm{sig},1}, \ldots, h_{\mathrm{sig},q}$ are drawn uniformly at random by $\mathcal{C}$. Recall that the values $h_{\mathrm{agg},1}, \ldots, h_{\mathrm{agg},q}$ and $h_{\mathrm{non},1}, \ldots, h_{\mathrm{non},q}$ are part of the *input* of $\mathcal{C}$ and the former will be the same in both runs of $\mathcal{B}$. All DL oracle queries made by $\mathcal{B}$ are relayed by $\mathcal{C}$ to its own DL oracle. If $\mathcal{B}$ returns $\perp$, $\mathcal{C}$ returns $\perp$ as well. Otherwise, if $\mathcal{B}$ returns a tuple $(i_{\mathrm{agg}}, j_{\mathrm{agg}}, i_{\mathrm{sig}}, j_{\mathrm{sig}}, L, R, s, \mathbf{a})$, where $L = \{X_1, \ldots, X_n\}$ and $\mathbf{a} = (a_1, \ldots, a_n)$, $\mathcal{C}$ runs $\mathcal{B}$ again with the same random coins on input

$$\begin{aligned}
&X^*, U_1, \ldots, U_{\nu q_s},\\
&h_{\mathrm{agg},1}, \ldots, h_{\mathrm{agg},q},\\
&h_{\mathrm{non},1}, \ldots, h_{\mathrm{non},j_{\mathrm{sig}}}, h'_{\mathrm{non},j_{\mathrm{sig}}+1}, \ldots, h'_{\mathrm{non},q},\\
&h_{\mathrm{sig},1}, \ldots, h_{\mathrm{sig},i_{\mathrm{sig}}-1}, h'_{\mathrm{sig},i_{\mathrm{sig}}}, \ldots, h'_{\mathrm{sig},q},
\end{aligned}$$

where $h'_{\mathrm{sig},i_{\mathrm{sig}}}, \ldots, h'_{\mathrm{sig},q}$ are uniformly random. If $\mathcal{B}$ returns $\perp$ in this second run, $\mathcal{C}$ returns $\perp$ as well. If $\mathcal{B}$ returns another tuple $(i'_{\mathrm{agg}}, j_{\mathrm{agg}}, i'_{\mathrm{sig}}, j'_{\mathrm{sig}}, L', R', s', \mathbf{a}')$, where $L' = \{X'_1, \ldots, X'_{n'}\}$ and $\mathbf{a}' = (a'_1, \ldots, a'_{n'})$, $\mathcal{C}$ proceeds as follows. Let $\widetilde{X} = \prod_{i=1}^n X_i^{a_i}$ and $\widetilde{X}' = \prod_{i=1}^{n'} (X'_i)^{a'_i}$ denote the aggregate public keys corresponding to the two forgeries. If $i_{\mathrm{sig}} \neq i'_{\mathrm{sig}}$, or $i_{\mathrm{sig}} = i'_{\mathrm{sig}}$ and $h_{\mathrm{sig},i_{\mathrm{sig}}} = h'_{\mathrm{sig},i_{\mathrm{sig}}}$, then $\mathcal{C}$ returns $\perp$. Otherwise, if $i_{\mathrm{sig}} = i'_{\mathrm{sig}}$ and $h_{\mathrm{sig},i_{\mathrm{sig}}} \neq h'_{\mathrm{sig},i_{\mathrm{sig}}}$, we will prove shortly that necessarily

$$i_{\mathrm{agg}} = i'_{\mathrm{agg}},\ j_{\mathrm{agg}} = j_{\mathrm{agg}},\ L = L',\ R = R',\ \text{and } \mathbf{a} = \mathbf{a}', \tag{3}$$

which implies in particular that $\widetilde{X} = \widetilde{X}'$. By Lemma 1, the two outputs returned by $\mathcal{B}$ are such that

$$g^s = R\widetilde{X}^{h_{\mathrm{sig},i_{\mathrm{sig}}}} \quad \text{and} \quad g^{s'} = R'(\widetilde{X}')^{h'_{\mathrm{sig},i_{\mathrm{sig}}}} = R\widetilde{X}^{h'_{\mathrm{sig},i_{\mathrm{sig}}}},$$

which allows $\mathcal{C}$ to compute the discrete logarithm of $\widetilde{X}$ as

$$\tilde{x} = (s - s')(h_{\mathrm{sig},i_{\mathrm{sig}}} - h'_{\mathrm{sig},i_{\mathrm{sig}}})^{-1} \bmod p.$$

Then $\mathcal{C}$ returns $(i_{\mathrm{agg}}, j_{\mathrm{agg}}, L, \mathbf{a}, \tilde{x})$.

It is easy to see that $\mathcal{C}$ returns a non-$\perp$ output *iff* $\mathsf{Fork}^{\mathcal{B}}$ does, so that by Lemma 4 and Lemma 1, $\mathcal{C}$'s accepting probability is at least

$$\begin{aligned}\mathsf{acc}(\mathcal{B})\left(\frac{\mathsf{acc}(\mathcal{B})}{q}-\frac{1}{p}\right) &\geq \frac{(\varepsilon-2q^2/p)^2}{q}-\frac{\varepsilon-2q^2/p}{p}\\ &\geq \frac{\varepsilon^2}{q}-\frac{\varepsilon(4q+1)}{p}+\frac{2q^2(2q+1)}{p^2}\\ &\geq \frac{\varepsilon^2}{q}-\frac{4q+1}{p}\end{aligned}$$

It remains to prove the equalities of Equation (3). In $\mathcal{B}$'s first execution, $h_{\mathrm{sig},i_{\mathrm{sig}}}$ is assigned to $T_{\mathrm{sig}}(\widetilde{X},R,m)$, while is $\mathcal{B}$'s second execution, $h'_{\mathrm{sig},i_{\mathrm{sig}}}$ is assigned to $T_{\mathrm{sig}}(\widetilde{X}',R',m')$. Note that these two assignments can happen either because of a direct query to $\mathsf{H}_{\mathrm{sig}}$ by the adversary, during a query to $\mathsf{H}_{\mathrm{non}}$, during a $\textsc{Sign}'$ query, or during the final verification of the validity of the forgery. Up to these two assignments, the two executions are identical since $\mathcal{B}$ runs $\mathcal{A}$ on the same random coins and input, uses the same values $h_{\mathrm{agg},1},\ldots,h_{\mathrm{agg},q}$ for $T_{\mathrm{agg}}(\cdot,X^*)$ assignments, the same values $h_{\mathrm{sig},1},\ldots,h_{\mathrm{sig},i_{\mathrm{sig}}-1}$ for T_{sig} assignments, and the same values $h_{\mathrm{non},1},\ldots,h_{\mathrm{non},j_{\mathrm{sig}}}$ for T_{non} assignments, $T_{\mathrm{agg}}(\cdot,X\neq X^*)$ assignments, and DL oracle outputs s_1 in $\textsc{Sign}'$ queries. Since both executions are identical up to the two assignments $T_{\mathrm{sig}}(\widetilde{X},R,m):=h_{\mathrm{sig},i_{\mathrm{sig}}}$ and $T_{\mathrm{sig}}(\widetilde{X}',R',m'):=h'_{\mathrm{sig},i_{\mathrm{sig}}}$, the arguments of the two assignments must be the same, which in particular implies that $R=R'$ and $\widetilde{X}=\widetilde{X}'$. Assume that $L\neq L'$. Then, since $\widetilde{X}=\widetilde{X}'$, this would mean that $\mathsf{KeyColl}$ is set to $\mathtt{true}$ in both executions, a contradiction since $\mathcal{B}$ returns a non-$\perp$ output in both executions. Hence, $L=L'$. Since in both executions of $\mathcal{B}$, $\mathsf{BadOrder}$ is not set to $\mathtt{true}$, assignments $T_{\mathrm{agg}}(L,X^*):=h_{\mathrm{agg},i_{\mathrm{agg}}}$ and $T_{\mathrm{agg}}(L',X^*):=h_{\mathrm{agg},i'_{\mathrm{agg}}}$ necessarily happened *before* the fork. This implies that $i_{\mathrm{agg}}=i'_{\mathrm{agg}}$ and $\mathbf{a}=\mathbf{a}'$. □

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

Theorem 1. *Assume that there exists a $(t,q_s,q_h,N,\varepsilon)$-adversary $\mathcal{A}$ against the multi-signature scheme $\mathsf{MuSig2}$ with $\nu\geq 4$, group parameters $(\mathbb{G},p,g)$ and with hash functions $\mathsf{H}_{\mathrm{agg}}$, $\mathsf{H}_{\mathrm{non}}$, $\mathsf{H}_{\mathrm{sig}}:\{0,1\}^*\to\mathbb{Z}_p$ modeled as random oracles. Then there exists an algorithm $\mathcal{D}$ which $(\nu q_s,t',\varepsilon')$-solves the OMDL problem for $(\mathbb{G},p,g)$, with*

$$t'=4(t+q(N+2\nu-2))t_{\exp}+O(qN)$$

where $q=(\nu-1)(q_h+q_s)+1$ and $t_{\exp}$ is the time of an exponentiation in $\mathbb{G}$ and

$$\varepsilon'\geq\frac{\varepsilon^4}{q^3}-\frac{11}{p}-\frac{43q^4}{(p-1)^{\nu-3}}.$$

Proof. Algorithm $\mathcal{D}$ runs $\mathsf{Fork}^{\mathcal{C}}$ with $\mathcal{C}$ as defined in Lemma 2 and takes additional steps as described below. The mapping with the notation in our Forking Lemma (Lemma 4 in Appendix A) is as follows:

- $X^*,U_1,\ldots,U_{\nu q_s}$ play the role of inp,
- $h_{\mathrm{agg},1},\ldots,h_{\mathrm{agg},q}$ play the role of $h_1,\ldots,h_q$,
- $(h_{\mathrm{non},1},h'_{\mathrm{non},1}),(h''_{\mathrm{non},1},h'''_{\mathrm{non},1}),\ldots,(h_{\mathrm{non},q},h'_{\mathrm{non},q}),(h''_{\mathrm{non},q},h'''_{\mathrm{non},q})$ play the role of $v_1,v'_1,\ldots,v_m,v'_m$,
- $(i_{\mathrm{agg}},j_{\mathrm{agg}})$ play the role of (i,j),
- $(L,\mathbf{a},\tilde{x})$ play the role of out.

In more details, $\mathcal{D}$ picks random coins ρ_B and runs algorithm $\mathcal{C}$ on coins ρ_B, group elements $X^*,U_1,\ldots,U_{\nu q_s}$, and uniformly random scalars $h_{\mathrm{agg},1},\ldots,h_{\mathrm{agg},q}$, $h_{\mathrm{non},1},h'_{\mathrm{non},1},\ldots,h_{\mathrm{non},q},h'_{\mathrm{non},q}\in\mathbb{Z}_p$. It relays all DL oracle queries made by $\mathcal{C}$ to its own DL oracle, caching the DL oracle replies to avoid making multiple identical queries. If $\mathcal{C}$ returns $\perp$, $\mathcal{D}$ returns $\perp$ as well. Otherwise, if $\mathcal{C}$ returns a tuple $(i_{\mathrm{agg}},j_{\mathrm{agg}},L,\mathbf{a},\tilde{x})$, $\mathcal{D}$ runs $\mathcal{C}$ again with the same random coins ρ_B on input $X^*,U_1,\ldots,U_{\nu q_s}$,

$$h_{\mathrm{agg},1},\ldots,h_{\mathrm{agg},i_{\mathrm{agg}}-1},h'_{\mathrm{agg},i_{\mathrm{agg}}},\ldots,h'_{\mathrm{agg},q},\text{ and}$$

$$h_{\mathrm{non},1}, h'_{\mathrm{non},1} \ldots, h_{\mathrm{non},j_{\mathrm{agg}}}, h'_{\mathrm{non},j_{\mathrm{agg}}}, h''_{\mathrm{non},j_{\mathrm{agg}}+1}, h'''_{\mathrm{non},j_{\mathrm{agg}}+1}, \ldots, h''_{\mathrm{non},q}, h'''_{\mathrm{non},q},$$

where $h'_{\mathrm{agg},i_{\mathrm{agg}}}, \ldots, h'_{\mathrm{agg},q}$ and $h''_{\mathrm{non},j_{\mathrm{agg}}+1}, h'''_{\mathrm{non},j_{\mathrm{agg}}+1}, \ldots, h''_{\mathrm{non},q}, h'''_{\mathrm{non},q}$ are uniformly random. If $\mathcal{C}$ returns $\perp$ in this second run, $\mathcal{D}$ returns $\perp$ as well. If $\mathcal{C}$ returns another tuple $(i'_{\mathrm{agg}}, j_{\mathrm{agg}}, L', \mathbf{a}', \tilde{x}')$, $\mathcal{D}$ proceeds as follows. Let $L = \{X_1, \ldots, X_n\}$, $\mathbf{a} = (a_1, \ldots, a_n)$, $L' = \{X'_1, \ldots, X'_{n'}\}$, and $\mathbf{a}' = (a'_1, \ldots, a'_n)$. Let n^* be the number of times X^* appears in L. If $i_{\mathrm{agg}} \neq i'_{\mathrm{agg}}$, or $i_{\mathrm{agg}} = i'_{\mathrm{agg}}$ and $h_{\mathrm{agg},i_{\mathrm{agg}}} = h'_{\mathrm{agg},i_{\mathrm{agg}}}$, $\mathcal{D}$ returns $\perp$. Otherwise, if $i_{\mathrm{agg}} = i'_{\mathrm{agg}}$ and $h_{\mathrm{agg},i_{\mathrm{agg}}} \neq h'_{\mathrm{agg},i_{\mathrm{agg}}}$, then

$$L = L' \text{ and } a_i = a'_i \text{ for each } i \text{ such that } X_i \neq X^*. \tag{4}$$

By Lemma 2, we have that

$$g^{\tilde{x}} = \prod_{i=1}^{n} X_i^{a_i} = (X^*)^{n^* h_{\mathrm{agg},i_{\mathrm{agg}}}} \prod_{\substack{i \in \{1,\ldots,n\} \\ X_i \neq X^*}} X_i^{a_i},$$

$$g^{\tilde{x}'} = \prod_{i=1}^{n} X_i^{a'_i} = (X^*)^{n^* h'_{\mathrm{agg},i_{\mathrm{agg}}}} \prod_{\substack{i \in \{1,\ldots,n\} \\ X_i \neq X^*}} X_i^{a_i}.$$

Thus, $\mathcal{D}$ can compute the discrete logarithm of X^* as

$$x^* = (\tilde{x} - \tilde{x}')(n^*)^{-1}(h_{\mathrm{agg},i_{\mathrm{agg}}} - h'_{\mathrm{agg},i_{\mathrm{agg}}})^{-1} \bmod p.$$

We will now prove the equalities in Equation (4). In the two executions of $\mathcal{B}$ run within the first execution of $\mathcal{C}$, $h_{\mathrm{agg},i_{\mathrm{agg}}}$ is assigned to $T_{\mathrm{agg}}(L, X^*)$, while in the two executions of $\mathcal{B}$ run within the second execution of $\mathcal{C}$, $h'_{\mathrm{agg},i_{\mathrm{agg}}}$ is assigned to $T_{\mathrm{agg}}(L', X^*)$. Note that these two assignments can happen either because of a direct query $\mathsf{H}_{\mathrm{agg}}(L, X)$ made by the adversary for some key $X \in L$ (not necessarily X^*), during a signing query, or during the final verification of the validity of the forgery. Up to these two assignments, the four executions of $\mathcal{A}$ are identical since $\mathcal{B}$ runs $\mathcal{A}$ on the same random coins and the same input, uses the same values $h_{\mathrm{agg},1}, \ldots, h_{\mathrm{agg},i_{\mathrm{agg}}-1}$ for $T_{\mathrm{agg}}(\cdot, X^*)$ assignments, the same values $h_{\mathrm{sig},1}, \ldots, h_{\mathrm{sig},q}$ for T_{sig} assignments, the same values $h_{\mathrm{non},1}, \ldots, h_{\mathrm{non},j_{\mathrm{agg}}}$ for T_{non} assignments, $T_{\mathrm{agg}}(\cdot, X \neq X^*)$ assignments, and the DL oracle outputs s_1 in $\textsc{Sign}'$ queries (note that this relies on the fact that in the four executions of $\mathcal{B}$, $\mathsf{BadOrder}$ is not set to `true`). Since the four executions of $\mathcal{B}$ are identical up to the assignments $T_{\mathrm{agg}}(L, X^*) := h_{\mathrm{agg},i_{\mathrm{agg}}}$ and $T_{\mathrm{agg}}(L', X^*) := h'_{\mathrm{agg},i_{\mathrm{agg}}}$, the arguments of these two assignments must be the same, which implies that $L = L'$. Besides, all values $T_{\mathrm{agg}}(L, X)$ for $X \in L \setminus \{X^*\}$ are chosen uniformly at random by $\mathcal{B}$ using the same coins in the four executions, which implies that $a_i = a'_i$ for each i such that $X_i \neq X^*$. This shows the equalities in Equation (4).

Recall that $\mathcal{D}$ internally ran four executions of $\mathcal{B}$ (throughout forking in $\mathsf{Fork}^{\mathcal{B}}$ as well as $\mathsf{Fork}^{\mathcal{C}}$). Consider a $\textsc{Sign}$ query handled by $\mathcal{B}$, and let i be the index such that the group elements $U_i, \ldots, U_{i+\nu-1}$ as drawn by $\mathcal{D}$ were assigned to $R_{1,1}, \ldots R_{1,\nu}$ by $\mathcal{B}$ when handling this query. In the corresponding $\textsc{Sign}'$ query, algorithm $\mathcal{B}$ has computed b_j for each $j \in \{1, \ldots, \nu\}$, a_i and c has queried the DL oracle with

$$s_1 := \mathsf{DL}_g\left(\left(\prod_{j=1}^{\nu} R_{1,j}^{b_j}\right)(X^*)^{a_1 c}\right). \tag{5}$$

Note that all four executions of $\mathcal{B}$ have been passed same group elements $U_i, \ldots, U_{i+\nu-1}$ as input to be used in $\textsc{Sign}$ signing queries. However, when handling the corresponding $\textsc{Sign}'$ queries, $\mathcal{B}$ may have made different queries to the DL oracle in the four executions.[16] Note however that a DL query has been made in every execution due to the normalizing assumption that the adversary $\mathcal{A}$ closes every opened signing session.

Algorithm $\mathcal{D}$ initializes a flag LinDep representing a bad event and attempts to deduce the discrete logarithm of all challenges which were used in each $\textsc{Sign}$ query in all four executions of $\mathcal{B}$ as follows. For each $\textsc{Sign}$ query, $\mathcal{D}$ proceeds to build a system of ν linear equations in the unknowns

[16] For example, the adversary may have replied with different L, m or R values in different executions, or algorithm $\mathcal{B}$ may have received different "h_{non}" values.

$r_1, \ldots, r_\nu$. Let P be the partition of the four executions of $\mathcal{B}$ where two executions are in the same component if their $b_2, \ldots, b_\nu$ variables in the SIGN′ query handler were assigned to the same "h_{non}" values drawn by $\mathcal{D}$.[17] For every $k \in \{1, \ldots, |P|\}$ let $b_j^{(k)}, s_1^{(k)}, a_1^{(k)}, c^{(k)}$ be the b_j, s_1, a_1, c value in the SIGN′ signing queries in the k-th component; we will show below that algorithm $\mathcal{B}$ ensures that these values are the same within all executions in a single component. As a consequence, also the group elements passed to its DL oracle (see Equation (5)) are identical within every component. Thus, due to the caching of DL oracle replies in $\mathcal{D}$, algorithm $\mathcal{D}$ needed only $|P|$ DL queries to its own DL oracle to answer the DL oracle queries originating by all four executions of $\mathcal{B}$. Then $\mathcal{D}$ has the following linear equations:

$$\sum_{j=1}^{\nu} b_i^{(k)} r_i = s_1^{(k)} - a_1^{(k)} c^{(k)} x^*, \quad k \in \{1, \ldots, |P|\} \tag{6}$$

Let $\mathbf{b}^{(k)} := (b_1^{(k)}, \ldots, b_\nu^{(k)})$. If the vectors $\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(|P|)}$ are linearly dependent, then $\mathcal{D}$ sets $\mathsf{LinDep} := \mathtt{true}$ and returns $\perp$. Otherwise, it continues to complete the linear system by using $\nu - |P|$ remaining DL queries as follows. For each $k \in \{|P|+1, \ldots, \nu\}$, it picks coefficients $\mathbf{b}^{(k)}$ such that $\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(\nu)}$ are linearly independent and obtains the following equations:

$$\sum_{j=1}^{\nu} b_i^{(k)} r_i = \mathsf{DL}_g\left(\sum_{j=1}^{\nu} R_{1,j}^{b_j^{(k)}}\right), \quad l \in \{|P|+1, \ldots, \nu\} \tag{7}$$

At this stage, $\mathcal{D}$ has a system of ν linear independent equations with ν unknowns. Because the system is consistent by construction, it has a unique solution $r_1, \ldots, r_\nu$, which is computed by $\mathcal{D}$ and provided as output.

It remains to show that if for some given SIGN query, two executions of $\mathcal{B}$ are in the same component of P, then $\mathcal{B}$ passed the same group element to the DL oracle (see Equation (5)) in the corresponding SIGN′ query in both executions. To this end, we show

$$b_j = b'_j \text{ for each } j \in \{1, \ldots, \nu\}, \quad a_1 = a'_1, \quad \text{and} \quad c = c', \tag{8}$$

where the non-primed and primed terms are the values used in the SIGN′ query in the respective execution. The equality $b_j = b'_j$, $j \in \{1, \ldots, \nu\}$ follows immediately from the fact that the two executions are in the same component. Moreover, since b_j and b'_j were assigned to the same h_{non} value, both executions of $\mathcal{B}$ are identical up to that point.

To prove $a_1 = a'_1$ we first note that the aggregate key $\widetilde{X}$ is an input in the assignment $b_j = \mathsf{H}_{\text{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$. Because both executions are identical when b_j and b'_j are assigned, we have $\widetilde{X} = \widetilde{X}'$. $\mathcal{B}$ has not set $\mathsf{KeyColl} := \mathtt{true}$ in either of the executions, which implies that $a_1 = a'_1$.

To see that $c = c'$, recall that assignments "$b_j = \mathsf{H}_{\text{non}}(j, \widetilde{X}, (R_1, \ldots, R_\nu), m)$" result in setting the table entry $T_{\text{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$. If entry $T_{\text{sig}}(\widetilde{X}, R, m)$ had already been set when $T_{\text{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$ was set, then $c = c'$ due to the executions being identical. Otherwise, if the value $T_{\text{sig}}(\tilde{x}, R, m)$ had not already been set when $T_{\text{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$ was set, then the internal H_{sig} query in the H_{non} query handler set $T_{\text{sig}}(\widetilde{X}, R, m)$ exactly when the first query $\mathsf{H}_{\text{non}}(j, \widetilde{X}, (R_1, \ldots, R_\nu), m)$ for some j was handled. Since $\mathcal{B}$ did not receive a forgery which invalid due to the values m and L from the forgery having been queried in a SIGN′ query, the internal H_{sig} query was not the H_{sig} fork point. Therefore, both executions are still identical when $T_{\text{sig}}(\widetilde{X}, R, m)$ is set which implies that $c = c'$. This shows the equalities in Equation (8).

Altogether, $\mathcal{D}$ makes $|P|$ DL queries initiated by $\mathcal{B}$ (as in Eq. (6)) and $\nu - |P|$ additional DL queries (as in Eq. (7)) per initiated signing session. Thus, the total number of DL queries is then exactly νq_s.

Neglecting the time needed to compute discrete logarithms and solve linear equation systems, the running time t' of $\mathcal{D}$ is twice the running time of $\mathcal{C}$, which itself is twice the running time

[17] For example, all four executions (as visualized in Figure 3) are in the same component if the corresponding T_{non} value was set before the H_{agg} fork point, and two executions in the same branch of the H_{agg} fork are in the same component if the T_{non} value was set before the H_{sig} fork point.

of $\mathcal{B}$. The running time of $\mathcal{B}$ is the running time t of $\mathcal{A}$ plus the time needed to maintain tables T_{agg}, T_{non}, and T_{sig} (we assume each assignment takes unit time) and answer signing and hash queries. The sizes of T_{agg}, T_{non}, and T_{sig} are at most qN, q, and q respectively. Answering signing queries is dominated by the time needed to compute the aggregate key as well as the honest signer's effective nonce, which is at most Nt_{exp} and $(\nu-1)t_{\text{exp}}$ respectively. Answering hash queries is dominated by the time to compute the aggregate nonce which is at most $(\nu-1)t_{\text{exp}}$. Therefore, $t' = 4(t + q(N + 2\nu - 2))t_{\text{exp}} + O(qN)$.

Clearly, $\mathcal{D}$ is successful *iff* $\mathsf{Fork}^{\mathcal{C}}$ returns a non-$\perp$ answer and LinDep is not set to `true`. Let T'_{non} be the union of the four tables T_{non} from the four executions of $\mathcal{B}$.[18] The probability that LinDep is set to `true`, i.e., that the vectors $\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(|P|)}$ are linearly dependent in some signing session, is clearly bounded by the probability that a family of four or fewer linearly dependent vectors can be found in T'_{non}. By construction of $\mathcal{B}$, we have $|T'_{\text{non}}| \leq 4q$, and thus by Lemma 7 (Appendix B),

$$\Pr[\mathsf{LinDep}] \leq \frac{(4q)^4}{6(p-1)^{\nu-3}} \leq \frac{43q^4}{(p-1)^{\nu-3}}.$$

By Lemma 4 and Lemma 2, the success probability of $\mathsf{Fork}^{\mathcal{C}}$ is at least

$$\begin{aligned}
\mathsf{acc}(\mathsf{Fork}^{\mathcal{C}}) = \mathsf{acc}(\mathcal{C})\left(\frac{\mathsf{acc}(\mathcal{C})}{q} - \frac{1}{p}\right) &\geq \frac{(\varepsilon^2/q - (4q+1)/p)^2}{q} - \frac{\varepsilon^2/q - (4q+1)/p}{p} \\
&\geq \frac{\varepsilon^4}{q^3} - \frac{(8+2/q)}{q\cdot p} - \frac{1}{q\cdot p} \\
&\geq \frac{\varepsilon^4}{q^3} - \frac{11}{p}.
\end{aligned}$$

Altogether, the success probability of $\mathcal{D}$ is at least

$$\mathsf{acc}(\mathcal{D}) \geq \mathsf{acc}(\mathsf{Fork}^{\mathcal{C}}) - \Pr[\mathsf{LinDep}] \geq \frac{\varepsilon^4}{q^3} - \frac{11}{p} - \frac{43q^4}{(p-1)^{\nu-3}}.$$ □

6 Security of MuSig2 in the ROM + AGM

By assuming the ROM and AGM, we can show the security of $\mathsf{MuSig2}$ with only $\nu = 2$ nonces.

Theorem 2. *Assume that there exists an algebraic $(t, q_s, q_h, N, \varepsilon)$-adversary $\mathcal{A}$ against the multi-signature scheme $\mathsf{MuSig2}$ with $\nu = 2$, group parameters $(\mathbb{G}, p, g)$ and hash functions H_{agg}, H_{sig}, $\mathsf{H}_{\text{non}} : \{0,1\}^* \to \mathbb{Z}_p$ modeled as random oracles. Then there exists an algorithm $\mathcal{B}$ which $(2q_s, t', \varepsilon')$-solves the OMDL problem for $(\mathbb{G}, p, g)$, with*

$$t' = t + O(qN)\cdot t_{\text{exp}} + O(q^3)$$

where t_{exp} is the time of an exponentiation in $\mathbb{G}$ and

$$\varepsilon' \geq \varepsilon - 14q^3/p.$$

Proof. Let $\mathcal{A}$ be an algebraic $(t, q_s, q_h, N, \varepsilon)$-adversary in the EUF-CMA$_{\mathsf{MuSig2}}$ security game. With respect to the queries made by $\mathcal{A}$, we make the same normalizing assumptions as introduced in Section 5.1: We assume that the $\mathcal{A}$ never repeats a query, and only makes "well-formed" queries, meaning that $X^* \in L$ and $X \in L$ for any query $\mathsf{H}_{\text{agg}}(L, X)$ and $j = 2$ for any query $\mathsf{H}_{\text{non}}(j, \ldots)$. Moreover, we assume that the forger closes every signing session, makes exactly q_h queries to each random oracle and opens exactly q_s signing sessions.

Let $q = q_h + (N+2)(q_s+1)$. Consider Figure 4 which specifies $\mathsf{Game}_0 = \text{EUF-CMA}^{\mathcal{A}}_{\mathsf{MuSig2}}(\lambda)$. Since the adversary is algebraic, for each group element Z it outputs (including those contained in queries to random oracles) it returns a representation $(\alpha, \beta, (\gamma_{1,k}, \gamma_{2,k})_{1\leq k\leq q_s})$ in basis $(g, X^*, (R^{(k)}_{1,1}, R^{(k)}_{1,2})_{1\leq k\leq q_s})$ such that

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

[18] For taking unions, recall that we model tables T as sets of pairs (k, v) representing assignments $T(k) := v$.

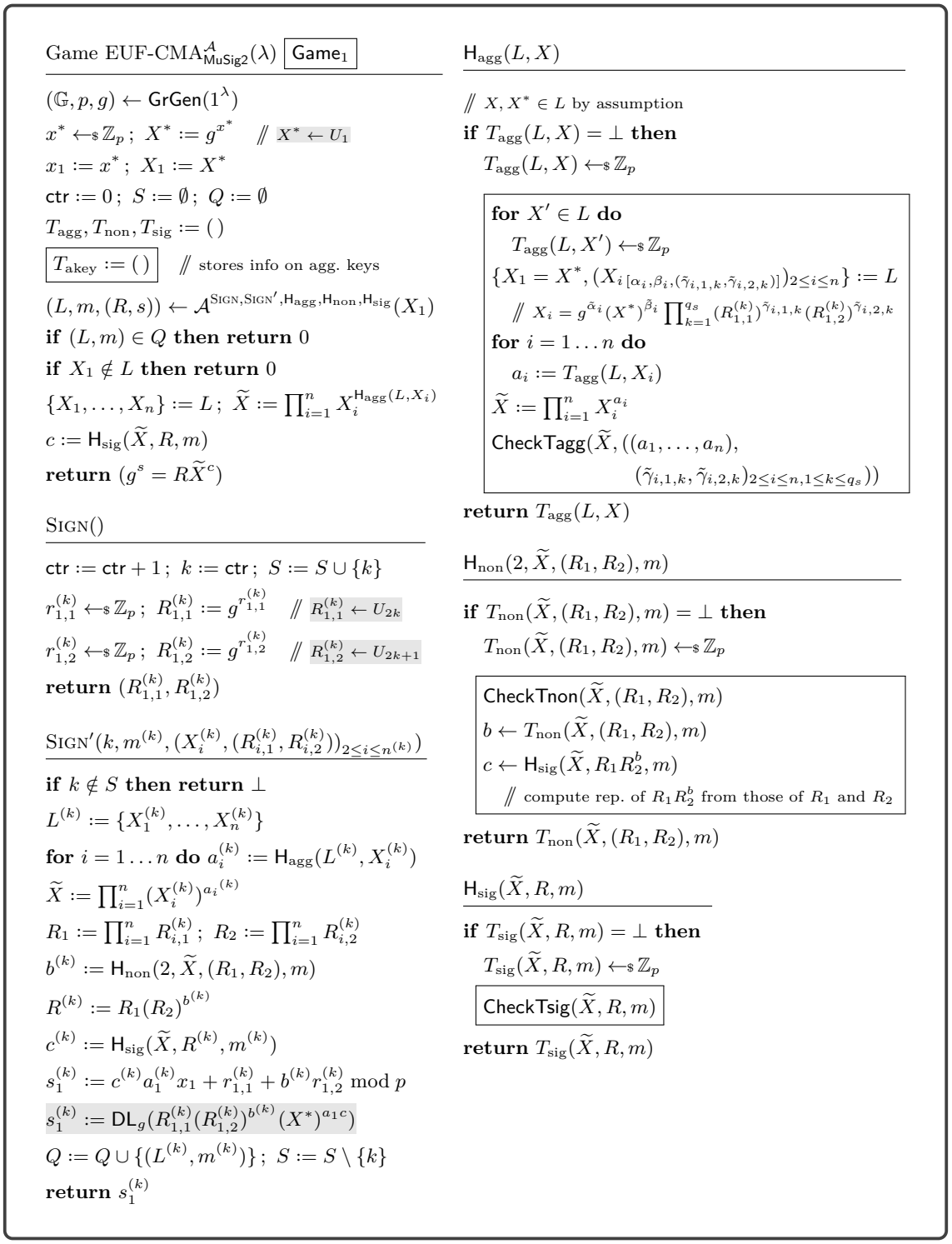

Fig. 4. Games used in the proof of Theorem 2. Comments in gray show how reduction $\mathcal{B}$ simulates Game_1. Procedures $\mathsf{CheckTagg}$, $\mathsf{CheckTnon}$, and $\mathsf{CheckTsig}$ are defined in Figures 5–7.

Procedure $\mathsf{CheckTagg}(\widetilde{X}, ((a_1, \ldots, a_n), (\tilde{\gamma}_{i,1,k}, \tilde{\gamma}_{i,2,k})_{2 \le i \le n, 1 \le k \le q_s}))$

1 : **if** $T_{\text{akey}}(\widetilde{X}) \neq \perp$
2 : **abort game and return 0**
3 : $T_{\text{akey}}(\widetilde{X}) := ((a_1, \ldots, a_n), (\tilde{\gamma}_{i,1,k}, \tilde{\gamma}_{i,2,k})_{2 \le i \le n, 1 \le k \le q_s})$
4 : **if** $\exists (R, m) : T_{\text{sig}}(\widetilde{X}, R, m) \neq \perp$ **then**
5 : **abort game and return 0**
6 : **if** $\exists (R_1, R_2, m) : T_{\text{non}}(\widetilde{X}, R_1, R_2, m) \neq \perp$ **then**
7 : **abort game and return 0**
8 : **for** $k = 1 \ldots q_s$ **do**
9 : **if** $\exists i \in \{2, \ldots, n\} : \tilde{\gamma}_{i,1,k} \neq 0$ **then**
10 : **if** $\tilde{\gamma}_{1,k} := \sum_{i=2}^{n} a_i \tilde{\gamma}_{i,1,k} = 0$ **then**
11 : **abort game and return 0**
12 : $K_1 := \{k \in \{1, \ldots, q_s\} : \exists i \in \{2, \ldots, n\}, \tilde{\gamma}_{i,1,k} \neq 0\}$
13 : **for** $\left((\widetilde{X}', R_1', R_2', m'), b'\right) \in T_{\text{non}}$ **do**
14 : **if** $\exists k \in K_1, i \in \{2, \ldots, n\} : \tilde{\gamma}_{i,2,k} - b' \tilde{\gamma}_{i,1,k} \neq 0$ **then**
15 : **if** $\theta_k := \sum_{i=2}^{n} a_i (\tilde{\gamma}_{i,2,k} - b' \tilde{\gamma}_{i,1,k}) = 0$ **then**
16 : **abort game and return 0**
17 : **for** $k \in K_1$ **do**
18 : // check whether there exists a T_{non} entry such that $\theta_k = 0$ holds independently of $(a_1, \ldots, a_n)$
19 : **if** $\exists \big((\widetilde{X}', R_1', R_2', m'), b'\big) \in T_{\text{non}} : (T_{\text{akey}}(\widetilde{X}') \neq \perp)$
20 : $\wedge \big(\forall i \in \{2, \ldots, n\}, \tilde{\gamma}_{i,2,k} - b' \tilde{\gamma}_{i,1,k} = 0\big)$ **then**
21 : // such a T_{non} entry is necessarily unique
22 : $\hat{a}_1^{(k)} := T_{\text{akey}}(\widetilde{X}')[1]$ // first component of $T_{\text{akey}}(\widetilde{X}')$
23 : $\hat{c}^{(k)} := T_{\text{sig}}(\widetilde{X}', R_1' (R_2')^{b'}, m')$
24 : **if** $\forall k \in K_1, (\hat{a}_1^{(k)}, \hat{c}^{(k)}) \neq \perp$ **then**
25 : **if** $\theta_0 := a_1 + \sum_{i=2}^{n} a_i \tilde{\beta}_i - \sum_{k \in K_1} \hat{a}_1^{(k)} \hat{c}^{(k)} \sum_{i=2}^{n} a_i \tilde{\gamma}_{i,1,k} = 0$ **then**
26 : // a_1 is a different random sample than $\hat{a}^{(k)}$, $k \in K_1$, because we did not abort in line 5.
27 : **abort game and return 0**

Fig. 5. Procedures used to check bad events for a T_{agg} assignment in Game_1.

Procedure $\mathsf{CheckTnon}(\widetilde{X}', (R_1', R_2'), m')$

1 : $b' := T_{\text{non}}(\widetilde{X}', (R_1', R_2'), m')$
2 : **if** $\exists(\widetilde{X}'', (R_1'', R_2''), m'') \neq (\widetilde{X}', (R_1', R_2'), m') : b' = T_{\text{non}}(\widetilde{X}'', (R_1'', R_2''), m'')$ **then**
3 : **abort game and return 0**
4 : **for** $\widetilde{X} \in T_{\text{akey}}$ **do**
5 : $((a_1, \ldots, a_n), (\tilde{\gamma}_{i,1,k}, \tilde{\gamma}_{i,2,k})_{2 \leq i \leq n, 1 \leq k \leq q_s}) := T_{\text{akey}}(\widetilde{X})$
6 : **for** $k = 1 \ldots q_s$ **do**
7 : **if** $\left(\tilde{\gamma}_{1,k} := \sum_{i=2}^{n} a_i \tilde{\gamma}_{i,1,k} \neq 0\right) \wedge \left(\sum_{i=2}^{n} a_i \tilde{\gamma}_{i,2,k} - b' \tilde{\gamma}_{1,k} = 0\right)$ **then**
8 : **abort game and return 0**
9 : **for** $(\widetilde{X}, R_{[\alpha,\beta,(\gamma_{1,k},\gamma_{2,k})]}, m) \in T_{\text{sig}}$ **do**
10 : **if** $T_{\text{akey}}(\widetilde{X}) \neq \perp$ **then**
11 : $c := T_{\text{sig}}(\widetilde{X}, R, m)$
12 : $((a_1, \ldots, a_n), (\tilde{\gamma}_{i,1,k}, \tilde{\gamma}_{i,2,k})_{2 \leq i \leq n, 1 \leq k \leq q_s}) := T_{\text{akey}}(\widetilde{X})$
13 : **for** $k = 1 \ldots q_s$ **do**
14 : **if** $\forall i = 2 \ldots n : \tilde{\gamma}_{i,1,k} = 0) \wedge (\gamma_{1,k} \neq 0$ **then**
15 : **if** $c \sum_{i=2}^{n} a_i \tilde{\gamma}_{i,2,k} + \gamma_{2,k} - b' \gamma_{1,k} = 0$ **then**
16 : **abort game and return 0**
17 : **if** $\exists i \in \{2, \ldots, n\} : \tilde{\gamma}_{i,1,k} \neq 0$ **then**
18 : **if** $\left(\delta_{1,k} := \gamma_{1,k} + c \sum_{i=2}^{n} a_i \tilde{\gamma}_{i,1,k} \neq 0\right) \wedge \left(c \sum_{i=2}^{n} a_i \tilde{\gamma}_{i,2,k} + \gamma_{2,k} - b' \delta_{1,k} = 0\right)$ **then**
19 : **abort game and return 0**

Fig. 6. Procedures used to check bad events for a T_{non} assignment in Game_1.

Procedure $\mathsf{CheckTsig}(\widetilde{X}, R, m)$

1 : $c \coloneqq T_{\text{sig}}(\widetilde{X}, R, m)$
2 : **if** $T_{\text{akey}}(\widetilde{X}) \neq \perp$ **then**
3 : $((a_1, \ldots, a_n), (\tilde{\gamma}_{i,1,k}, \tilde{\gamma}_{i,2,k})_{2 \leq i \leq n, 1 \leq k \leq q_s}) \coloneqq T_{\text{akey}}(\widetilde{X})$
4 : **for** $k = 1 \ldots q_s$ **do**
5 : **if** $\exists i \in \{2, \ldots, n\} : \tilde{\gamma}_{i,1,k} \neq 0$ **then**
6 : **if** $(\tilde{\gamma}_{1,k} \neq 0) \wedge (\delta_{1,k} \coloneqq \gamma_{1,k} + c\tilde{\gamma}_{1,k} \neq 0)$ **then**
7 : **abort game and return 0**
8 : **if** $(\forall i = 2 \ldots n : \tilde{\gamma}_{i,1,k} = 0) \wedge (\gamma_{1,k} = 0)$ **then**
9 : **if** $\theta_k \coloneqq \sum_{i=2}^{n} a_i \tilde{\gamma}_{i,2,k} \neq 0 \wedge c\theta_k + \gamma_{2,k} = 0$ **then**
10 : **abort game and return 0**
11 : **for** $(\widetilde{X}', (R_1', R_2'), m') \in T_{\text{non}}$ **do**
12 : $b' \coloneqq T_{\text{non}}(\widetilde{X}', (R_1', R_2'), m')$
13 : **if** $\left(\theta_k \coloneqq \sum_{i=2}^{n} a_i \left(\tilde{\gamma}_{i,2,k} - b'\tilde{\gamma}_{i,1,k}\right) \neq 0\right) \wedge \left(c\theta_k + \gamma_{2,k} - b'\gamma_{1,k} = 0\right)$ **then**
14 : **abort game and return 0**
15 : $K_1 \coloneqq \{k \in \{1, \ldots, q_s\} : \exists i \in \{2, \ldots, n\}, \tilde{\gamma}_{i,1,k} \neq 0\}$
16 : $K_1' \coloneqq \{k \in \{1, \ldots, q_s\} : \forall i \in \{2, \ldots, n\}, \tilde{\gamma}_{i,1,k} = 0 \ \wedge \gamma_{1,k} \neq 0\}$
17 : **for** $k \in K_1 \cup K_1'$ **do**
18 : **if** $\exists \left((\widetilde{X}', R_1', R_2', m'), b'\right) \in T_{\text{non}} :$
19 : $(T_{\text{akey}}(\widetilde{X}') \neq \perp)$
20 : $\wedge\, (\widetilde{X}', R_1'(R_2')^{b'}, m') \neq (\widetilde{X}, R, m)$
21 : $\wedge \left[\left(k \in K_1 \wedge \forall i \in \{2, \ldots, n\}, \tilde{\gamma}_{i,2,k} - b'\tilde{\gamma}_{i,1,k} = 0\right)\right.$
22 : $\left.\vee\ (k \in K_1' \wedge \gamma_{2,k} - b'\gamma_{1,k} = 0)\right]$ **then**
23 : // such a T_{non} entry is necessarily unique
24 : $\hat{a}_1^{(k)} \coloneqq T_{\text{akey}}(\widetilde{X}')[1]$ // first component of $T_{\text{akey}}(\widetilde{X}')$
25 : $\hat{c}^{(k)} \coloneqq T_{\text{sig}}(\widetilde{X}', R_1'(R_2')^{b'}, m')$
26 : **if** $\forall k \in K_1 \cup K_1', (\hat{a}_1^{(k)}, \hat{c}^{(k)}) \neq \perp$ **then**
27 : **if** $\theta_0 \coloneqq a_1 + \sum_{i=2}^{n} a_i \tilde{\beta}_i - \sum_{k \in K_1} \hat{a}_1^{(k)} \hat{c}^{(k)} \sum_{i=2}^{n} a_i \tilde{\gamma}_{i,1,k} \neq 0$ **then**
28 : **if** $c\theta_0 + \beta - \sum_{k \in K_1} \hat{a}_1^{(k)} \hat{c}^{(k)} \gamma_{1,k} - \sum_{k \in K_1'} \hat{a}_1^{(k)} \hat{c}^{(k)} \gamma_{1,k} = 0$ **then**
29 : // c is a different random sample than $\hat{c}^{(k)}$ due to line 20.
30 : **abort game and return 0**

Fig. 7. Procedures used to check bad events for a T_{sig} assignment in Game_1.

where $(R_{1,1}^{(k)}, R_{1,2}^{(k)})$ are the nonces returned by the k-th query to SIGN(). We denote $Z_{[\alpha,\beta,(\gamma_{1,k},\gamma_{2,k})]}$ such an augmented output and sometimes omit the representation when it is not used in the proof. By definition,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{game}_0}(\lambda) = \mathsf{Adv}_{\mathsf{MuSig2},\mathcal{A}}^{\text{euf-cma}}(\lambda). \tag{9}$$

In order to bound the probability of certain bad events in Game_0, we define a game Game_1 which differs from Game_0 by a number of additional steps and checks as specified in Figures 4–7. We introduce the following short notation for the events that one of these checks fail and makes Game_1 abort and return 0:

$$\mathsf{Bad}(P, C, I) := \mathsf{Game}_1 \text{ aborts in the } I\text{-th invocation of procedure } P \text{ in line } C,$$

and combining all the invocations and code lines

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

Then, since in CheckTagg, the freshly computed value $\widetilde{X}$, is random and independent of the other table keys in T_{akey}, the table keys in T_{sig} and the table keys in T_{non}, we have by code inspection for each invocation $I \in \{1, \ldots, q_h\}$ of CheckTagg that

$$\begin{aligned}
\Pr\left[\mathsf{Bad}(\mathsf{CheckTagg}, 2, I)\right] &\leq |T_{\text{akey}}|/p \leq q/p, \\
\Pr\left[\mathsf{Bad}(\mathsf{CheckTagg}, 5, I)\right] &\leq |T_{\text{sig}}|/p \leq q/p, \\
\Pr\left[\mathsf{Bad}(\mathsf{CheckTagg}, 7, I)\right] &\leq |T_{\text{non}}|/p \leq q/p.
\end{aligned}$$

Furthermore, since in CheckTagg, the freshly computed value a_n $(n \neq 1)$ is random and independent from any state, we have by code inspection for each invocation $I \in \{1, \ldots, q_h\}$ of CheckTagg that

$$\begin{aligned}
\Pr\left[\mathsf{Bad}(\mathsf{CheckTagg}, 11, I)\right] &\leq q_s/p \leq q/p, \\
\Pr\left[\mathsf{Bad}(\mathsf{CheckTagg}, 16, I)\right] &\leq |T_{\text{non}}|\, q_s/p \leq q^2/p.
\end{aligned}$$

and similarly, since the freshly computed value a_1 is random and independent from any state,

$$\Pr\left[\mathsf{Bad}(\mathsf{CheckTagg}, 27, I)\right] \leq 1/p.$$

CheckTagg is invoked at most $q_h + N(q_s + 1) \leq q$ times because H_{agg} is invoked at most q_h times by the adversary, at most N times when Game_1 verifies the forgery, and at most Nq_s times in SIGN′. Thus, by the union bound

$$\Pr\left[\mathsf{Bad}(\mathsf{CheckTagg})\right] \leq q_h(5q^2 + 1) \leq 6q^3. \tag{10}$$

Since in CheckTnon, the freshly assigned T_{non} value b' is random and independent of the other T_{non} values, we have by code inspection for each invocation $I \in \{1, \ldots, q_h\}$ of CheckTnon that

$$\begin{aligned}
\Pr\left[\mathsf{Bad}(\mathsf{CheckTnon},\ 3, I)\right] &\leq |T_{\text{non}}|/p \leq q/p, \\
\Pr\left[\mathsf{Bad}(\mathsf{CheckTnon},\ 8, I)\right] &\leq |T_{\text{akey}}|\, q_s/p \leq q^2/p, \\
\Pr\left[\mathsf{Bad}(\mathsf{CheckTnon}, 16, I)\right] &\leq |T_{\text{sig}}|\, q_s/p \leq q^2/p, \\
\Pr\left[\mathsf{Bad}(\mathsf{CheckTnon}, 19, I)\right] &\leq |T_{\text{sig}}|\, q_s/p \leq q^2/p.
\end{aligned}$$

CheckTnon is invoked at most $q_h + q_s \leq q$ times because H_{non} is invoked at most q_h times by the adversary, and at most q_s times in SIGN′. Thus, by the union bound

$$\Pr\left[\mathsf{Bad}(\mathsf{CheckTnon})\right] \leq (q_h + q_s)(4q^2)/p \leq 4q^3/p. \tag{11}$$

Since in CheckTsig, the freshly assigned T_{sig} value c is random and independent of the other T_{sig} values, we have by code inspection for each invocation $I \in \{1, \ldots, q_h\}$ of CheckTsig that

$$\Pr\left[\mathsf{Bad}(\mathsf{CheckTsig},\ 7, I)\right] \leq q_s/p \leq q/p,$$

$$\Pr[\mathsf{Bad}(\mathsf{CheckTsig}, 10, I)] \le q_s/p \le q/p,$$
$$\Pr[\mathsf{Bad}(\mathsf{CheckTsig}, 14, I)] \le q_s\,|T_{\text{non}}|/p \le q^2/p,$$
$$\Pr[\mathsf{Bad}(\mathsf{CheckTsig}, 30, I)] \le 1/p.$$

$\mathsf{CheckTsig}$ is invoked at most $q_h + q_s + 1 \le q$ times because H_{sig} is invoked at most q_h times by the adversary, at most once for every invocation of $\textsc{Sign}'$, and once when Game_1 verifies the forgery. Thus, by the union bound

$$\Pr[\mathsf{Bad}(\mathsf{CheckTsig})] \le (q_h + q_s + 1)(3q^2 + 1)/p \le 4q^3/p. \tag{12}$$

Let

$$\mathsf{Bad} = \mathsf{Bad}(\mathsf{CheckTagg}) \cup \mathsf{Bad}(\mathsf{CheckTnon}) \cup \mathsf{Bad}(\mathsf{CheckTsig}).$$

Combining Equations (10), (11), and (12), we have by the union bound

$$\Pr[\mathsf{Bad}] \le 14q^3/p$$

and by the fundamental lemma of game hopping

$$\begin{aligned} \mathsf{Adv}^{\mathsf{game}_1}_{\mathcal{A}}(\lambda) &\ge \mathsf{Adv}^{\mathsf{game}_0}_{\mathcal{A}}(\lambda) - \Pr[\mathsf{Bad}] \\ &\ge \mathsf{Adv}^{\mathsf{game}_0}_{\mathcal{A}}(\lambda) - 14q^3/p. \end{aligned} \tag{13}$$

Now we define an algorithm $\mathcal{B}$ which solves the OMDL problem whenever Game_1 returns 1. On input $((\mathbb{G}, p, g), (U_1, \dots U_{2q_s+1}))$, where $U_1, \dots, U_{2q_s+1}$ are the OMDL challenge elements, it sets $X^* = U_1$ and runs $\mathcal{A}$ on input X^*.

Algorithm $\mathcal{B}$ simulates Game_0[19] without knowledge of the discrete logarithm x^* of X^* as follows (see comments in Figure 4): when $\mathcal{A}$ makes the k-th $\textsc{Sign}$ query, $\mathcal{B}$ sets $R_{1,1} = U_{2k}$ and $R_{1,2} = U_{2k+1}$; it then simulates $\textsc{Sign}'$ without knowledge of x^* by querying its DL oracle available from the OMDL problem (analogous to algorithm $\mathcal{B}$ from Section 5).

When $\mathcal{A}$ returns its forgery $(L, M, (R, s))$, algorithm $\mathcal{B}$ proceeds as follows. Since $\mathcal{A}$ is algebraic, whenever it queries oracle H_{agg} then for every key $X_i \in L$ it provides a representation $(\tilde{\alpha}_i, \tilde{\beta}_i, \tilde{\gamma}_{i,1,1}, \tilde{\gamma}_{i,2,1}, \dots, \tilde{\gamma}_{i,1,q_s}, \tilde{\gamma}_{i,2,q_s}) \in \mathbb{Z}_p^{2+2q_s}$ in basis $(g, X^*, R^{(1)}_{1,1}, R^{(1)}_{2,1}, \dots, R^{(q_s)}_{1,1}, R^{(q_s)}_{2,1})$ such that

$$X_i = g^{\tilde{\alpha}_i}(X^*)^{\tilde{\beta}_i} \prod_{k=1}^{q_s} (R^{(k)}_{1,1})^{\tilde{\gamma}_{i,1,k}} (R^{(k)}_{1,2})^{\tilde{\gamma}_{i,2,k}}.$$

Using the representations of X_i, $\mathcal{B}$ computes a representation of $\widetilde{X} = (X^*)^{a_1} \cdot \prod_{i=2}^n X_i^{a_i}$:

$$\widetilde{X} = g^{\tilde{\alpha}}(X^*)^{\tilde{\beta}} \prod_{k=1}^{q_s} (R^{(k)}_{1,1})^{\tilde{\gamma}_{1,k}} (R^{(k)}_{1,2})^{\tilde{\gamma}_{2,k}} \tag{14}$$

where

$$\tilde{\alpha} = \sum_{i=2}^n a_i \tilde{\alpha}_i \bmod p, \tag{15}$$

$$\tilde{\beta} = a_1 + \sum_{i=2}^n a_i \tilde{\beta}_i \bmod p, \tag{16}$$

$$\tilde{\gamma}_{1,k} = \sum_{i=2}^n a_i \tilde{\gamma}_{i,1,k} \bmod p \text{ for all } k \in \{1, \dots, q_s\}, \tag{17}$$

$$\tilde{\gamma}_{2,k} = \sum_{i=2}^n a_i \tilde{\gamma}_{i,2,k} \bmod p \text{ for all } k \in \{1, \dots, q_s\}. \tag{18}$$

[19] We use Game_1 only to bound $\Pr[\mathsf{Bad}]$. Algorithm $\mathcal{B}$ simulates Game_0 and does not (need to) perform the additional checks introduced in Game_1.

Similarly, $\mathcal{A}$ gives a representation $(\alpha, \beta, \gamma_{1,1}, \gamma_{2,1} \ldots, \gamma_{1,q_s}, \gamma_{2,q_s}) \in \mathbb{Z}_p^{2+2q_s}$ of R when querying H_{sig}:

$$R = g^{\alpha}(X^*)^{\beta} \prod_{k=1}^{q_s} (R_{1,1}^{(k)})^{\gamma_{1,k}} (R_{1,2}^{(k)})^{\gamma_{2,k}}. \tag{19}$$

Validity of the forgery implies that $g^s = R\widetilde{X}^c$, which together with (14) and (19) yields

$$g^s = g^{\alpha+c\tilde{\alpha}}(X^*)^{\beta+c\tilde{\beta}} \prod_{k=1}^{q_s} (R_{1,1}^{(k)})^{\gamma_{1,k}+c\tilde{\gamma}_{1,k}} (R_{1,2}^{(k)})^{\gamma_{2,k}+c\tilde{\gamma}_{2,k}}. \tag{20}$$

Similarly, validity of the partial signature $s_1^{(k)}$ returned in the k-th signing session yields

$$g^{s_1^{(k)}} = R_{1,1}^{(k)} (R_{1,2}^{(k)})^{b^{(k)}} (X^*)^{a_1^{(k)} c^{(k)}} \text{ for all } k \in \{1, \ldots, q_s\}. \tag{21}$$

Combining (20) and (21), $\mathcal{A}$ obtains a linear system with $q_s + 1$ equations and $2q_s + 1$ unknowns $x^*, r_{1,1}^{(1)}, r_{1,2}^{(1)}, \ldots, r_{1,1}^{(q_s)}, r_{1,2}^{(q_s)}$ which are the discrete logarithms of $X^*, R_{1,1}^{(1)}, R_{1,2}^{(1)}, \ldots, R_{1,1}^{(q_s)}, R_{1,2}^{(q_s)}$ respectively:

$$\begin{aligned}
s - \alpha - c\tilde{\alpha} &= (\beta + c\tilde{\beta})x^* + \sum_{k=1}^{q_s} (\gamma_{1,k} + c\tilde{\gamma}_{1,k}) r_{1,1}^{(k)} + (\gamma_{2,k} + c\tilde{\gamma}_{2,k}) r_{1,2}^{(k)} \\
s_1^{(1)} &= a_1^{(1)} c^{(1)} x^* + r_{1,1}^{(1)} + b^{(1)} r_{1,2}^{(1)} \\
&\vdots \\
s_1^{(q_s)} &= a_1^{(q_s)} c^{(q_s)} x^* + r_{1,1}^{(q_s)} + b^{(q_s)} r_{1,2}^{(q_s)}.
\end{aligned}$$

The coefficient matrix M of this linear system is

$$M = \begin{pmatrix}
\delta_{1,1} & \delta_{2,1} & \cdots & \delta_{1,q_s} & \delta_{2,q_s} & \beta + c\tilde{\beta} \\
1 & b^{(1)} & 0 & \cdots & 0 & a_1^{(1)} c^{(1)} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & \cdots & 0 & 1 & b^{(q_s)} & a_1^{(q_s)} c^{(q_s)}
\end{pmatrix} \tag{22}$$

where

$$\delta_{1,k} := \gamma_{1,k} + c\tilde{\gamma}_{1,k} \text{ and } \delta_{2,k} := \gamma_{2,k} + c\tilde{\gamma}_{2,k}, \quad k \in \{1, \ldots, q_s\}. \tag{23}$$

We will prove in Lemma 3 below that when Game_1 returns 1, then necessarily this matrix has rank $q_s + 1$. Hence, $\mathcal{B}$ is able to compute $x^*, r_{1,1}^{(1)}, r_{1,2}^{(1)}, \ldots, r_{1,1}^{(q_s)}, r_{1,2}^{(q_s)}$ by solving the linear system after querying the DL oracle q_s times to add q_s equations to the system such that the system has a unique solution. This is possible because the system is consistent by construction and the existing coefficient vectors are linearly independent. More specifically, for every $j \in \{1, \ldots, q_s\}$, $\mathcal{B}$ chooses a coefficient vector $\mathbf{v}_j = (v_{j,0}, v_{j,1}^{(1)}, v_{j,2}^{(1)}, \ldots, v_{j,1}^{(q_s)}, v_{j,2}^{(q_s)}) \in \mathbb{Z}_p^{2q_s+1}$ such that the vector is linearly independent of the rows of coefficient matrix and adds the following equation to the linear system:

$$\mathsf{DL}_g \left((X^*)^{v_{j,0}} \prod_{k=1}^{q_s} (R_{1,1}^{(k)})^{v_{j,1}^{(k)}} (R_{1,2}^{(k)})^{v_{j,2}^{(k)}} \right) = \mathbf{v}_j \cdot \begin{pmatrix} x^* \\ r_{1,1}^{(1)} \\ \vdots \\ r_{1,2}^{(q_s)} \end{pmatrix}.$$

This allows $\mathcal{B}$ to solve the linear system and output the solution $x^*, r_{1,1}^{(1)}, r_{1,2}^{(1)}, \ldots, r_{1,1}^{(q_s)}, r_{1,2}^{(q_s)}$ to the OMDL problem. In total, $\mathcal{B}$ queried the DL oracle q_s times to answer signing queries and q_s times to complete the linear system, which totals $2q_s$ DL oracle queries.

Algorithm $\mathcal{B}$ succeeds whenever Game_0 returns 1 and the matrix M defined by Equation (22) has rank $q_s + 1$. By Lemma 3, these two conditions are necessarily fulfilled whenever Game_1 returns 1. (Recall that Game_0 and Game_1 are identical unless Game_1 aborts and returns 0 early). By Equations (9) and (13), the success probability of $\mathcal{B}$ is

$$\varepsilon' \geq \mathsf{Adv}^{\mathsf{game}_1}_{\mathcal{A}}(\lambda) \geq \mathsf{Adv}^{\mathsf{game}_0}_{\mathcal{A}}(\lambda) - 14q^3/p = \varepsilon - 14q^3/p.$$

The running time t' of $\mathcal{B}$ is the running time t of $\mathcal{A}$ plus the time needed to maintain tables T_{agg}, T_{non}, and T_{sig} (we assume each assignment takes unit time), answer signing and hash queries, and solve the linear equation system. All tables have size in $O(qN)$, and $\mathcal{B}$ needs $O(q_s N)t_{\text{exp}} = O(qN)t_{\text{exp}}$ 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 time in $O(q_s^3)$. Thus

$$t' = t + O(qN) \cdot t_{\text{exp}} + O(q^3).$$

This concludes the proof. □

Lemma 3. *Consider an execution of* Game_1 *which returns 1. Then matrix* M *defined by Equation (22) has rank* $q_s + 1$.

Proof. Subtracting $b^{(k)}$ times the $2k-1$-th column from the $2k$-th column, $k = 1, \ldots, q_s$, and $a_i^{(k)} c^{(k)}$ times the $2k-1$-th column from the last column, $k = 1, \ldots, q_s$, yields

$$\begin{pmatrix} \delta_{1,1} & \zeta_1 & \cdots & \delta_{1,q_s} & \zeta_{q_s} & \eta \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & 0 & 1 & 0 & 0 \end{pmatrix}$$

where $\zeta_k := \delta_{2,k} - b^{(k)}\delta_{1,k}$ and

$$\eta := \beta + c\tilde{\beta} - \sum_{k=1}^{q_s} a_1^{(k)} c^{(k)} \delta_{1,k}.$$

By reordering the columns, we obtain

$$\begin{pmatrix} \eta & \zeta_1 & \cdots & \zeta_{q_s} & \delta_{1,1} & \delta_{1,2} & \cdots & \delta_{1,q_s} \\ 0 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 1 \end{pmatrix}.$$

Being in row echelon form, we see that the system has rank $< q_s + 1$ if and only if the left $q_s + 1$ coefficients of the first row are all zero, i.e.,

$$0 = \beta + c\tilde{\beta} - \sum_{k=1}^{q_s} a_1^{(k)} c^{(k)} \delta_{1,k} \tag{24}$$

$$0 = \delta_{2,k} - b^{(k)}\delta_{1,k} \quad \text{for } k \in \{1, \ldots, q_s\}. \tag{25}$$

By (16), (17), (18), and (23), these equations are equivalent to

$$0 = c\underbrace{\left(a_1 + \sum_{i=2}^{n} a_i \tilde{\beta}_i - \sum_{k=1}^{q_s} a_1^{(k)} c^{(k)} \sum_{i=2}^{n} a_i \tilde{\gamma}_{i,1,k} \right)}_{:=\theta_0} + \beta - \sum_{k=1}^{q_s} a_1^{(k)} c^{(k)} \gamma_{1,k} \tag{26}$$

$$0 = c\underbrace{\left(\sum_{i=2}^{n} a_i \left(\tilde{\gamma}_{i,2,k} - b^{(k)} \tilde{\gamma}_{i,1,k} \right) \right)}_{:=\theta_k} + \gamma_{2,k} - b^{(k)} \gamma_{1,k} \quad \text{for } k \in \{1, \ldots, q_s\}. \tag{27}$$

Assume that (26) and (27) hold at the end of the execution. We show that necessarily, one of the bad events leading to Game_1 returning 0 must have happened, a contradiction. Consider all T_{agg}, T_{non}, and T_{sig} table entries that appear in (26) and (27). These are:

– $a_i = T_{\mathrm{agg}}(L, X_i)$, $i \in \{1, \ldots, n\}$
– $c = T_{\mathrm{sig}}(\widetilde{X}, R, m)$
– $a_1^{(k)} = T_{\mathrm{agg}}(L^{(k)}, X_1)$, $k \in \{1, \ldots, q_s\}$
– $b^{(k)} = T_{\mathrm{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$, $k \in \{1, \ldots, q_s\}$
– $c^{(k)} = T_{\mathrm{sig}}(\widetilde{X}^{(k)}, R^{(k)}, m^{(k)})$, $k \in \{1, \ldots, q_s\}$.

We will show that some bad event necessarily happened during the random assignment of one of these table values.

Claim. For an execution of Game_1 which returns 1, the following properties holds:

(P1) when the assignment of $T_{\mathrm{sig}}(\widetilde{X}, R, m)$ occurs, $T_{\mathrm{akey}}(\widetilde{X}) \neq \perp$;
(P2) for every $k \in \{1, \ldots, q_s\}$, one has $(\widetilde{X}^{(k)}, R^{(k)}, m^{(k)}) \neq (\widetilde{X}, R, m)$
(P3) for every $k \in \{1, \ldots, q_s\}$, one has $(\exists i \in \{2, \ldots, n\} : \tilde{\gamma}_{i,1,k} \neq 0) \Rightarrow (\tilde{\gamma}_{1,k} \neq 0)$;
(P4) for every $k \in \{1, \ldots, q_s\}$, one has $(\exists i \in \{2, \ldots, n\} : \tilde{\gamma}_{i,1,k} \neq 0) \Rightarrow (\delta_{1,k} \neq 0)$;
(P5) for every $k \in \{1, \ldots, q_s\}$, when the assignment of $T_{\mathrm{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$ occurs, $T_{\mathrm{akey}}(\widetilde{X}^{(k)}) \neq \perp$.

Proof. We prove each item in turn.

(P1) Assume that $T_{\mathrm{akey}}(\widetilde{X}) = \perp$ when the assignment of $T_{\mathrm{sig}}(\widetilde{X}, R, m)$ occurs. Since for the forgery, $T_{\mathrm{akey}}(\widetilde{X}) \neq \perp$ at the end of the execution, Game_1 would necessarily have returned 0 at line 5 of $\mathsf{CheckTagg}(\widetilde{X}, \ldots)$, which would necessarily have been called after the assignment of $T_{\mathrm{sig}}(\widetilde{X}, R, m)$.
(P2) Assume that $\exists k \in \{1, \ldots, q_s\} : (\widetilde{X}^{(k)}, R^{(k)}, m^{(k)}) = (\widetilde{X}, R, m)$. Since the forgery is valid, we have $(L, m) \notin Q$, where $Q = \{(L^{(k)}, m^{(k)}) : k \in \{1, \ldots, q_s\}\})$. Since $m^{(k)} = m$, we have $L^{(k)} \neq L$. Since additionally $\widetilde{X}^{(k)} = \widetilde{X}$, Game_1 would have necessarily have returned 0 at line 2 of the later of the two invocations $\mathsf{CheckTagg}(\widetilde{X}, \ldots)$ and $\mathsf{CheckTagg}(\widetilde{X}^{(k)}, \ldots)$.
(P3) Assume that $(\exists i \in \{2, \ldots, n\} : \tilde{\gamma}_{i,1,k} \neq 0)$ and $\tilde{\gamma}_{1,k} = 0$. Then Game_1 would necessarily have returned 0 at line 11 of $\mathsf{CheckTagg}(\widetilde{X}, \ldots)$.
(P4) Assume that $(\exists i \in \{2, \ldots, n\} : \tilde{\gamma}_{i,1,k} \neq 0)$ and $\delta_{1,k} = 0$. By (P3), we have that $\tilde{\gamma}_{1,k} \neq 0$. Hence, Game_1 would necessarily have returned 0 at line 7 of $\mathsf{CheckTsig}(\widetilde{X}, R, m)$.
(P5) Assume that $T_{\mathrm{akey}}(\widetilde{X}^{(k)}) = \perp$ when the assignment of $T_{\mathrm{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$ occurs. Since for the k-th signing session, $T_{\mathrm{akey}}(\widetilde{X}^{(k)}) \neq \perp$ at the end of the execution, Game_1 would necessarily have returned 0 at line 7 of $\mathsf{CheckTagg}(\widetilde{X}^{(k)}, \ldots)$, which would necessarily have been called after the assignment of $T_{\mathrm{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$.

This proves the claim. ■

In all the following, we let

$$K_0 := \{k \in \{1, \ldots, q_s\} : \forall i \in \{2, \ldots, n\}, \tilde{\gamma}_{i,1,k} = 0\}$$
$$K_1 := \{k \in \{1, \ldots, q_s\} : \exists i \in \{2, \ldots, n\}, \tilde{\gamma}_{i,1,k} \neq 0\}.$$

Note that $\{1, \ldots, q_s\}$ is the disjoint union of K_0 and K_1 and that

$$\theta_0 = a_1 + \sum_{i=2}^{n} a_i \tilde{\beta}_i - \sum_{k \in K_1} a_1^{(k)} c^{(k)} \sum_{i=2}^{n} a_i \tilde{\gamma}_{i,1,k}.$$

We distinguish two cases depending on the values of θ_k, $k \in \{1, \ldots, q_s\}$ at the end of the execution:

1. $\theta_k = 0$ for all $k \in \{1, \ldots, q_s\}$; we distinguish the following sub-cases:
   (a) there exists $k \in K_1$ such that $T_{\mathrm{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$ was randomly assigned after $T_{\mathrm{akey}}(\widetilde{X})$: then by (P3), $\tilde{\gamma}_{1,k} \neq 0$ so that $\mathsf{CheckTnon}$ would have aborted and returned 0 at line 8;

(b) for all $k \in K_1$, $T_{\text{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$ was randomly assigned before $T_{\text{akey}}(\widetilde{X})$: then we distinguish the following sub-cases (below we let $b^{(k)} = T_{\text{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$):

i. there exists $k \in K_1$ and $i \in \{2, \ldots, n\}$ such that $\tilde{\gamma}_{i,2,k} - b^{(k)}\tilde{\gamma}_{i,1,k} \neq 0$; then $\mathsf{CheckTagg}$ would have aborted and returned 0 at line 16.

ii. for all $k \in K_1$ and all $i \in \{2, \ldots, n\}$, $\tilde{\gamma}_{i,2,k} - b^{(k)}\tilde{\gamma}_{i,1,k} = 0$; (then note that $\theta_k = 0$ trivially holds for $k \in K_1$); again we distinguish:

A. $\theta_0 = 0$; then, by (P5), we have $T_{\text{akey}}(\widetilde{X}^{(k)}) \neq \perp$ when the entry $T_{\text{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$ is assigned, so that Game_1 would necessarily have returned 0 at line 27 of $\mathsf{CheckTagg}(\widetilde{X}, \ldots)$;

B. $\theta_0 \neq 0$; let

$$K_1' := \{k \in K_0 : \gamma_{1,k} \neq 0\}.$$

Then (26) is equivalent to

$$0 = c\theta_0 + \beta - \sum_{k \in K_1} a_1^{(k)} c^{(k)} \gamma_{1,k} - \sum_{k \in K_1'} a_1^{(k)} c^{(k)} \gamma_{1,k},$$

while for $k \in K_1'$, (27) is equivalent to

$$0 = c \sum_{i=2}^{n} a_i \tilde{\gamma}_{i,2,k} + \gamma_{2,k} - b^{(k)} \gamma_{1,k}.$$

We distinguish:

– there exists $k \in K_1'$ such that $T_{\text{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$ was randomly assigned after $T_{\text{sig}}(\widetilde{X}, R, m)$; Then by (P5), Game_1 would necessarily have returned 0 at line 16 in the invocation of $\mathsf{CheckTnon}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$.

– for all $k \in K_1'$, $T_{\text{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$ was randomly assigned before $T_{\text{sig}}(\widetilde{X}, R, m)$; Then by (P1) and (P2), Game_1 would necessarily have returned 0 at line 30 of $\mathsf{CheckTsig}(\widetilde{X}, R, m)$.

2. $\theta_k \neq 0$ for some $k \in \{1, \ldots, q_s\}$; we distinguish the following sub-cases:

(a) $T_{\text{sig}}(\widetilde{X}, R, m)$ was randomly assigned after $T_{\text{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$; then, by (P1), $T_{\text{akey}}(\widetilde{X}) \neq \perp$ when $T_{\text{sig}}(\widetilde{X}, R, m) \leftarrow_{\$} \mathbb{Z}_p$ occurs, so that Game_1 would have returned 0 at line 14 of $\mathsf{CheckTsig}$;

(b) $T_{\text{non}}(\widetilde{X}^{(k)}, (R_1^{(k)}, R_2^{(k)}), m^{(k)})$ was randomly assigned after $T_{\text{sig}}(\widetilde{X}, R, m)$; again we distinguish:

i. if $\tilde{\gamma}_{i,1,k} = 0$ for all $i \in \{2, \ldots, n\}$ and $\gamma_{1,k} = 0$, then Game_1 would have returned 0 at line 10 of $\mathsf{CheckTsig}$;

ii. if $\tilde{\gamma}_{i,1,k} = 0$ for all $i \in \{2, \ldots, n\}$ and $\gamma_{1,k} \neq 0$, then Game_1 would have returned 0 at line 16 of $\mathsf{CheckTnon}$;

iii. if $(\exists i \in \{2, \ldots, n\} : \tilde{\gamma}_{i,1,k} \neq 0)$, then by (P4) one has $\delta_{1,k} \neq 0$ so that Game_1 would have returned 0 at line 19 of $\mathsf{CheckTnon}$.

Hence, in all cases we obtain a contradiction, which concludes the proof. □

Acknowledgments

We thank Julian Loss, Greg Maxwell, and Pieter Wuille for their helpful comments and suggestions. We also thank Chelsea Komlo and Ian Goldberg for insightful discussions about multi-signatures and threshold signatures and for helping us understand the relation of our work to theirs [KG20].

References

[BCJ08] Ali Bagherzandi, Jung Hee Cheon, and Stanislaw Jarecki. Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008*, pages 449–458. ACM Press, October 2008.

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 435–464. Springer, Heidelberg, December 2018.

[BLOR20] Fabrice Benhamouda, Tancrède Lepoint, Michele Orrù, and Mariana Raykova. On the (in)security of ROS. Cryptology ePrint Archive, Report 2020/945, 2020. Available at http://eprint.iacr.org/2020/945.pdf.

[BN06] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 390–399. ACM Press, October / November 2006.

[BNPS03] Mihir Bellare, Chanathip Namprempre, David Pointcheval, and Michael Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, June 2003.

[Bol03] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, Heidelberg, January 2003.

[BP02] Mihir Bellare and Adriana Palacio. GQ and Schnorr identification schemes: Proofs of security against impersonation under active and concurrent attacks. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 162–177. Springer, Heidelberg, August 2002.

[CP93] David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 89–105. Springer, Heidelberg, August 1993.

[DEF+19] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the security of two-round multi-signatures. In *2019 IEEE Symposium on Security and Privacy*, pages 1084–1101. IEEE Computer Society Press, May 2019.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018.

[FPS20] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. Blind schnorr signatures and signed ElGamal encryption in the algebraic group model. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 63–95. Springer, Heidelberg, May 2020.

[HMP95] Patrick Horster, Markus Michels, and Holger Petersen. Meta-multisignature schemes based on the discrete logarithm problem. In *IFIP/Sec '95*, IFIP Advances in Information and Communication Technology, pages 128–142. Springer, 1995.

[IN83] K. Itakura and K. Nakamura. A public-key cryptosystem suitable for digital multisignatures. *NEC Research and Development*, 71:1–8, 1983.

[KG20] Chelsea Komlo and Ian Goldberg. FROST: Flexible Round-Optimized Schnorr Threshold Signatures. IACR Cryptology ePrint Archive, Report 2020/852, 2020. Available at https://eprint.iacr.org/2020/852.

[Lan96] Susan K. Langford. Weakness in some threshold cryptosystems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 74–82. Springer, Heidelberg, August 1996.

[MH96] Markus Michels and Patrick Horster. On the risk of disruption in several multiparty signature schemes. In Kwangjo Kim and Tsutomu Matsumoto, editors, *ASIACRYPT'96*, volume 1163 of *LNCS*, pages 334–345. Springer, Heidelberg, November 1996.

[MOR01] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures: Extended abstract. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001*, pages 245–254. ACM Press, November 2001.

[MPSW18] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple Schnorr multi-signatures with applications to Bitcoin. 2018. Preliminary obsolete version of [MPSW19] claiming the security of an insecure two-round scheme. Preserved at https://eprint.iacr.org/2018/068/20180118:124757.

[MPSW19] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple Schnorr multi-signatures with applications to Bitcoin. *Des. Codes Cryptogr.*, 87(9):2139–2164, 2019. Available at https://eprint.iacr.org/2018/068.pdf.

[MWLD10] Changshe Ma, Jian Weng, Yingjiu Li, and Robert H. Deng. Efficient discrete logarithm based multi-signature scheme in the plain public key model. *Des. Codes Cryptogr.*, 54(2):121–133, 2010.

[Nic19] Jonas Nick. Insecure shortcuts in musig, 2019. Available at https://medium.com/blockstream/insecure-shortcuts-in-musig-2ad0d38a97da.

[NRSW20] Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. MuSig-DN: Two-round Schnorr multi-signatures with verifably deterministic nonces. In *ACM Conference on Computer and Communications Security - CCS 2020*, 2020. Available at https://eprint.iacr.org/2020/1057.pdf.

[PS00] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, June 2000.

[RY07] Thomas Ristenpart and Scott Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 228–245. Springer, Heidelberg, May 2007.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991.

[Sch01] Claus-Peter Schnorr. Security of blind discrete log signatures against interactive attacks. In Sihan Qing, Tatsuaki Okamoto, and Jianying Zhou, editors, *ICICS 01*, volume 2229 of *LNCS*, pages 1–12. Springer, Heidelberg, November 2001.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997.

[STV+16] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping authorities "honest or bust" with decentralized witness cosigning. In *2016 IEEE Symposium on Security and Privacy*, pages 526–545. IEEE Computer Society Press, May 2016.

[Wag02] David Wagner. A generalized birthday problem. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 288–303. Springer, Heidelberg, August 2002.

[WNR20] Pieter Wuille, Jonas Nick, and Tim Ruffing. Schnorr signatures for secp256k1. Bitcoin Improvement Proposal 340, 2020. See https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki.

A A Generalized Forking Lemma

As in Bellare and Neven [BN06], our security proof relies on a "generalized Forking Lemma" extending Pointcheval and Stern's Forking Lemma [PS00] and which does not mention signatures nor forgers and only deals with the outputs of an algorithm $\mathcal{A}$ run twice on related inputs. However, the generalized Forking Lemma of Bellare and Neven [BN06] is not general enough for our setting. In short, in BN's lemma, only the values $h_i, \ldots, h_q$ (i.e., the post-fork responses used by the reduction for the random oracle $\mathsf{H}_{\mathrm{sig}}$) are refreshed in the second execution of $\mathcal{A}$, whereas we need to refresh others part of the input of $\mathcal{A}$. In particular, our reduction must obtain fresh OMDL challenges, used as nonces $R_{1,1}, \ldots R_{1,\nu}$ sent by the honest signer in signing sessions as well as fresh bitstrings $b_1, \ldots, b_\nu$ used as responses for the random oracle $\mathsf{H}_{\mathrm{non}}$, after the two executions of $\mathcal{A}$ have forked.

Lemma 4. *Fix integers q and m. Let $\mathcal{A}$ be a randomized algorithm which takes as input some main input inp following some unspecified distribution, elements $h_1, \ldots, h_q$ from some finite non-empty set H, and elements $v_1, \ldots, v_m$ from some finite non-empty set V, and returns either a distinguished failure symbol $\perp$, or a tuple (i, j, out), where $i \in \{1, \ldots, q\}$, $j \in \{0, \ldots, m\}$, and out is some side output. The accepting probability of $\mathcal{A}$, denoted $\mathsf{acc}(\mathcal{A})$, is defined as the probability, over the random draws of inp, $h_1, \ldots, h_q \leftarrow\!\!\$\, H$ as well as $v_1, \ldots, v_m \leftarrow\!\!\$\, V$, and the random coins of $\mathcal{A}$, that $\mathcal{A}$ returns a non-$\perp$ output. Consider algorithm $\mathsf{Fork}^{\mathcal{A}}$, taking as input inp and $v_1, v'_1, \ldots, v_m, v'_m \in V$, described on Figure 8. Let frk be the probability (over the draw of inp, $v_1, v'_1, \ldots, v_m, v'_m \leftarrow\!\!\$\, V$, and the random coins of $\mathsf{Fork}^{\mathcal{A}}$) that $\mathsf{Fork}^{\mathcal{A}}$ returns a non-$\perp$ output. Then*

$$\mathsf{frk} \geq \mathsf{acc}(\mathcal{A}) \left(\frac{\mathsf{acc}(\mathcal{A})}{q} - \frac{1}{|H|} \right).$$

The proof of the lemma is very similar to the one of [BN06, Lemma 1]. As in [BN06], we will need the following two lemmas which are consequences of Jensen's inequality.

Lemma 5. *Let Y be a real-valued random variable. Then $\mathbf{E}[Y^2] \geq \mathbf{E}[Y]^2$.*

Lemma 6. *Let $q \geq 1$ be an integer and $y_1, \ldots, y_q \geq 0$ be real numbers. Then*

$$\sum_{i=1}^{q} y_i^2 \geq \frac{1}{q} \left(\sum_{i=1}^{q} y_i \right)^2.$$

Proof of Lemma 4. Let $\mathsf{acc}(\mathsf{inp})$ be the probability (over the draw of $h_1, \ldots, h_q$, $v_1, \ldots, v_m$, and the random coins of $\mathcal{A}$) that $\mathcal{A}$ returns a non-$\perp$ output when run with inp as first input. Let also $\mathsf{frk}(\mathsf{inp})$ be the probability (over the draw of $v_1, v'_1, \ldots, v_m, v'_m$ and the random coins of $\mathsf{Fork}^{\mathcal{A}}$) that $\mathsf{Fork}^{\mathcal{A}}$ returns a non-$\perp$ output when run with inp as first input. We will show shortly that for all inp,

$$\mathsf{frk}(\mathsf{inp}) \geq \mathsf{acc}(\mathsf{inp}) \left(\frac{\mathsf{acc}(\mathsf{inp})}{q} - \frac{1}{|H|} \right). \tag{28}$$

Then, exactly as in [BN06], taking the expectation over inp we have

$$\begin{aligned}
\mathsf{frk} = \mathbf{E}[\mathsf{frk}(\mathsf{inp})] &\geq \mathbf{E}\left[\mathsf{acc}(\mathsf{inp}) \left(\frac{\mathsf{acc}(\mathsf{inp})}{q} - \frac{1}{|H|} \right) \right] && (29)\\
&= \frac{\mathbf{E}[\mathsf{acc}(\mathsf{inp})^2]}{q} - \frac{\mathbf{E}[\mathsf{acc}(\mathsf{inp})]}{|H|} && (30)\\
&\geq \frac{\mathbf{E}[\mathsf{acc}(\mathsf{inp})]^2}{q} - \frac{\mathbf{E}[\mathsf{acc}(\mathsf{inp})]}{|H|} && (31)\\
&= \mathsf{acc}(\mathcal{A}) \left(\frac{\mathsf{acc}(\mathcal{A})}{q} - \frac{1}{|H|} \right), && (32)
\end{aligned}$$

where we used Lemma 5 for Equation (31). It remains to prove Equation (28).

We fix inp and consider the random experiment of running $\mathsf{Fork}^{\mathcal{A}}(\mathsf{inp}, v_1, v'_1, \ldots, v_m, v'_m)$ with $v_1, v'_1, \ldots, v_m, v'_m \leftarrow\!\!\$\, V$ and random coins. We regard the first two elements of the outputs (i, j, out) and (i', j', out') returned by $\mathcal{A}$ in each of its two executions as random variables denoted I, J, I', and J', with the convention that $I = 0$, resp. $I' = 0$ if $\mathcal{A}$ returns $\perp$ in its first, resp. second execution.

Again, exactly as in the proof of [BN06, Lemma 1], we have

$$\begin{aligned}
\mathsf{frk}(\mathsf{inp}) &= \Pr\left[(I > 0) \wedge (I = I') \wedge (h_I \neq h'_I)\right] \\
&\geq \Pr\left[(I > 0) \wedge (I = I')\right] - \Pr\left[(I > 0) \wedge (h_I = h_{I'})\right] \\
&= \Pr\left[(I > 0) \wedge (I = I')\right] - \frac{\Pr\left[I > 0\right]}{|H|} \\
&= \underbrace{\Pr\left[(I > 0) \wedge (I = I')\right]}_{=:\,\mathsf{pr}} - \frac{\mathsf{acc}(\mathsf{inp})}{|H|}.
\end{aligned}$$

```
1  algorithm Fork^B(inp, v_1, v'_1, ..., v_m, v'_m)
2    pick random coins ρ for B
3    h_1, ..., h_q ←$ H
4    α ← B(inp, (h_1, ..., h_q), (v_1, ..., v_m); ρ)
5    if α = ⊥ then return ⊥
6    else parse α as (i, j, out)
7    h'_i, ..., h'_q ←$ H
8    α' ← B(inp, (h_1, ..., h_{i-1}, h'_i, ..., h'_q), (v_1, ..., v_j, v'_{j+1}, ..., v'_m); ρ)
9    if α' = ⊥ then return ⊥
10   else parse α' as (i', j', out')
11   if (i = i' and h_i ≠ h'_i) then return (i, out, out')
12   else return ⊥
```

Fig. 8. The "forking" algorithm $\mathsf{Fork}^{\mathcal{B}}$ built from $\mathcal{B}$.

It remains to lower bound the term pr. Let R denote the set of random coins for $\mathcal{A}$. For each $i \in \{1, \ldots, q\}$ and $j \in \{0, \ldots, m\}$, we define the function $Y_{i,j} : R \times H^{i-1} \times V^j \to [0, 1]$ as

$$Y_{i,j}(\rho, \mathbf{h}, \mathbf{V}) := \Pr\left[\left\{\begin{array}{l} h_i, \ldots, h_q \leftarrow_{\$} H, \\ v_{j+1}, \ldots, v_m \leftarrow_{\$} V, \\ (I, J, \mathsf{out}) \leftarrow \mathcal{A}(\mathsf{inp}, (h_1, \ldots, h_q), (v_1, \ldots, v_m); \rho) \end{array}\right\} : I = i\right]$$

for all $\rho \in R$, $\mathbf{h} = (h_1, \ldots, h_{i-1}) \in H^{i-1}$, and $\mathbf{V} = (v_1, \ldots, v_j) \in V^j$.

Then

$$\begin{aligned}
\mathsf{pr} &= \sum_{j=0}^{m} \sum_{i=1}^{q} \Pr\left[(I = i) \wedge (J = j) \wedge (I' = i)\right] && (33)\\
&= \sum_{j=0}^{m} \sum_{i=1}^{q} \Pr\left[J = j\right] \cdot \Pr\left[I = i | J = j\right] \cdot \Pr\left[I' = i | (I = i) \wedge (J = j)\right] && (34)\\
&= \sum_{j=0}^{m} \sum_{i=1}^{q} \Pr\left[J = j\right] \sum_{\substack{\rho \in R \\ \mathbf{h} \in H^{i-1} \\ \mathbf{V} \in V^j}} \frac{Y_{i,j}(\rho, \mathbf{h}, \mathbf{V})^2}{|R| \cdot |H|^{i-1} \cdot |V|^j} && (35)\\
&= \sum_{j=0}^{m} \Pr\left[J = j\right] \sum_{i=1}^{q} \mathbf{E}[Y_{i,j}^2] && (36)\\
&\geq \sum_{j=0}^{m} \Pr\left[J = j\right] \sum_{i=1}^{q} \mathbf{E}[Y_{i,j}]^2 && (37)\\
&\geq \frac{1}{q} \sum_{j=0}^{m} \Pr\left[J = j\right] \left(\sum_{i=1}^{q} \mathbf{E}[Y_{i,j}]\right)^2 && (38)\\
&\geq \frac{1}{q} \left(\sum_{j=0}^{m} \Pr\left[J = j\right] \sum_{i=1}^{q} \mathbf{E}[Y_{i,j}]\right)^2 && (39)\\
&= \frac{\mathsf{acc}(\mathsf{inp})^2}{q}. && (40)
\end{aligned}$$

Above we used Lemma 5 to derive Equation (37) and Equation (39) and Lemma 6 for each j with $y_i = \mathbf{E}[Y_{i,j}]$ to derive Equation (38).

□

B Linear Dependence of Random Vectors

Our security proof in the ROM (Appendix 5) relies on the following lemma in order to bound the probability that the adversary manages to obtain linearly dependent vectors across different executions.

Lemma 7. *Let $\mathbb{Z}_p$ be the finite field with p elements, and let $1 \leq f \leq \nu$ be integers. Let $\{\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(q)}\}$ be a family of q vectors in $(\mathbb{Z}_p)^\nu$ such that for each vector $\mathbf{b}^{(k)} = (b_1^{(k)}, b_2^{(k)}, \ldots, b_\nu^{(k)})$, $k \in \{1, \ldots, q\}$, we have $b_1^{(k)} = 1$ and $b_2^{(k)}, \ldots, b_\nu^{(k)}$ are chosen uniformly at random from $\mathbb{Z}_p$. Then the probability that there is a linearly dependent subfamily $\{\mathbf{b}^{(k_1)}, \ldots, \mathbf{b}^{(k_{f'})}\}$ of size $f' < f$ is at most*

$$\frac{q^f}{(f-1)!\,(p-1)^{\nu-f+1}}.$$

Proof. For a fixed f, we can assume *wlog* that $q \geq f$ and that $f' = f$ because these assumptions make the probability that there is a linearly dependent ("ld") subfamily $\{\mathbf{b}^{(k_1)}, \ldots, \mathbf{b}^{(k_{f'})}\}$ only larger. Let $\{\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(f)}\}$ be any subfamily of size f, *wlog* indexed by $\{1, \ldots, f\}$ to keep the

notation simple. Given scalars $u^{(1)}, \ldots, u^{(q)}$ chosen uniformly random from $\mathbb{Z}_p \setminus \{0\}$, consider the vectors $\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(f)}\}$ defined as

$$\mathbf{b}'^{(k)} := u^{(k)}\mathbf{b}^{(k)} = (u^{(k)}, u^{(k)}b_2^{(k)}, \ldots, u^{(k)}b_\nu^{(k)}).$$

It is well-known that $\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(f)}\}$ are linearly independent ("li") if and only if $\{\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(f)}\}$ are linearly independent. Thus

$$\Pr\left[\{\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(f)}\}\,\mathrm{li}\right] = \Pr\left[\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(f)}\}\,\mathrm{li}\right]$$

and we can work with $\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(f)}\}$ instead of $\{\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(f)}\}$ if all we care about is linear independence.

For any k, there are $(p-1)p^{\nu-1}$ choices of $\mathbf{b}'^{(k)}$ and p^{k-1} of those fall into the span of $\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k-1)}\}$ if this family is linearly independent. Thus

$$\Pr\left[\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k)}\}\,\mathrm{li} \,\middle|\, \{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k-1)}\}\,\mathrm{li}\right] = 1 - \frac{p^{k-1}}{(p-1)p^{\nu-1}} = 1 - \frac{p^{k-\nu}}{p-1}.$$

By the law of total probability, we have

$$\begin{aligned}
&\Pr\left[\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k)}\}\,\mathrm{li}\right] \\
&\quad= \Pr\left[\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k)}\}\,\mathrm{li} \,\middle|\, \{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k-1)}\}\ \mathrm{li}\right] \cdot \Pr\left[\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k-1)}\}\mathrm{li}\right] \\
&\quad\quad+ \Pr\left[\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k)}\}\,\mathrm{li} \,\middle|\, \{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k-1)}\}\ \mathrm{ld}\right] \cdot \Pr\left[\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k-1)}\}\,\mathrm{ld}\right] \\
&\quad\quad\quad= \Pr\left[\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k)}\}\,\mathrm{li} \,\middle|\, \{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k-1)}\}\ \mathrm{li}\right] \cdot \Pr\left[\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(k-1)}\}\,\mathrm{li}\right].
\end{aligned}$$

By induction with base case $\Pr[\{\}\,\mathrm{li}] = 1$, we obtain

$$\Pr\left[\{\mathbf{b}'^{(1)}, \ldots, \mathbf{b}'^{(f)}\}\,\mathrm{li}\right] = \prod_{k=1}^{f}\left(1 - \frac{p^{k-\nu}}{p-1}\right) \geq \left(1 - \frac{p^{f-\nu}}{p-1}\right)^f.$$

Switching back to $\mathbf{b}$ and considering the complementary event, we further obtain

$$\Pr\left[\{\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(f)}\}\,\mathrm{ld}\right] \leq 1 - \left(1 - \frac{p^{f-\nu}}{p-1}\right)^f \leq \frac{fp^{f-\nu}}{p-1} \leq \frac{f}{(p-1)^{\nu-f+1}},$$

where the penultimate inequality follows from Bernoulli's inequality and the last inequality holds because $f \leq \nu$.

There are $\binom{q}{f}$ subfamilies of $\{\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(q)}\}$, ignoring their internal order. By the union bound,

$$\begin{aligned}
\Pr\left[\exists\,\{k_1, \ldots, k_f\} \subseteq \{1, \ldots, q\},\, \{\mathbf{b}^{(k_1)}, \ldots, \mathbf{b}^{(k_f)}\}\,\mathrm{ld}\right] &\leq \binom{q}{f}\frac{f}{(p-1)^{\nu-f+1}} \\
&\leq \frac{q^f}{(f-1)!\,(p-1)^{\nu-f+1}}. \qquad \square
\end{aligned}$$

C Inapplicability of the Meta-reduction by Drijvers et al.

Drijvers *et al.* [DEF+19] show that there is no algebraic reduction from the OMDL problem to InsecureMuSig if the OMDL problem is hard. They describe a meta-reduction that solves the OMDL problem by executing a given reduction and simulating the OMDL challenger and the forger. To illustrate the idea, let us consider a reduction similar to the one in the flawed security proof of InsecureMuSig. It runs the forger twice with identical inputs up to the $\mathsf{H}_{\mathrm{sig}}$ query from the forgery and responds to it with a different value in the second execution.

The main idea of the meta-reduction is that it opens a signing session before the reduction forks the simulated forger and then continues the session with a different nonce (or different message)

in each fork, allowing extraction of the secret key. So, after opening the session by requesting the reduction's R_1 value, the meta-reduction obtains an OMDL challenge R that will be the nonce of the forgery and queries H_{sig} with R. Now the forger sends some nonce to continue the signing session and receives s_1 from the reduction. Then the forger queries the DL oracle to produce a forged signature (R, s).

The reduction runs the simulated forger again with exactly the same responses up to the H_{sig} query relevant to the forgery. In this execution the forger continues the signing session by sending a different nonce and receives s_1' from the reduction. Let c and c' with $c \neq c'$ be the signature hashes of the sessions and a_1 be the MuSig coefficient of the reduction's public key. Then the meta-reduction computes the secret key x_1 of the reduction as $(s_1 - s_1')/((c - c')a_1)$. In order to prevent the reduction from guessing the H_{sig} query and programming it such that $c = c'$, the simulated forger actually opens multiple concurrent signing sessions to make sure that the reduction guessed wrong for at least one of the sessions with high probability. Note how in the concrete attack with Wagner's algorithm the attacker similarly opens multiple signing sessions and controls their R values.

With the secret key x_1 the meta-reduction uses the s-value from the DL oracle query to compute r such that $g^r = R$, creates a signature with x_1 and r and returns it as the forgery in the second execution. The meta-reduction used one DL query and obtained two OMDL challenges that are answered with x_1 and r. Therefore, if the reduction is successful, the meta-reduction solves the OMDL problem without access to an actual forger for $\mathsf{InsecureMuSig}$ which implies that the OMDL problem was not hard if such a reduction existed.

However, the idea behind this meta-reduction does not apply to $\mathsf{MuSig2}$. Assume that the reduction now provides $\nu = 2$ nonces $R_{1,1}$ and $R_{1,2}$ at the start of a signing session. If the forger replies with values such that $c \neq c'$ we have that $b_2 \neq b_2'$. As a result, the meta-reduction can not extract the secret key because it obtains two linearly independent signature equations with three unknowns $x_1, r_{1,1}, r_{1,2}$. That means that in order to create a forgery in the second execution the meta-reduction needs to use another DL query which prevents it from winning the OMDL game. More generally, this explains why the meta-reduction can only be applied to reductions that execute the forger more than ν times. In Section 5 we provide a reduction for $\mathsf{MuSig2}$ with $\nu \geq 4$ that requires exactly four executions of the forger.