Simple Schnorr Multi-Signatures with Applications to Bitcoin

Gregory Maxwell, Andrew Poelstra[1], Yannick Seurin[2], and Pieter Wuille[1]

[1] Blockstream
[2] ANSSI, Paris, France
greg@xiph.org,
{apoelstra, pwuille}@blockstream.com,
yannick.seurin@m4x.org

January 15, 2018

Abstract. We describe a new Schnorr-based multi-signature scheme (i.e., a protocol which allows a group of signers to produce a short, joint signature on a common message), provably secure in the plain public-key model (meaning that signers are only required to have a public key, but do not have to prove knowledge of the private key corresponding to their public key to some certification authority or to other signers before engaging the protocol), which improves over the state-of-art scheme of Bellare and Neven (ACM-CCS 2006) and its variants by Bagherzandi *et al.* (ACM-CCS 2008) and Ma *et al.* (Des. Codes Cryptogr., 2010) in two respects: (i) it is simple and efficient, having only two rounds of communication instead of three for the Bellare-Neven scheme and the same key and signature size as standard Schnorr signatures; (ii) it allows *key aggregation*, which informally means that the joint signature can be verified exactly as a standard Schnorr signature with respect to a single "aggregated" public key which can be computed from the individual public keys of the signers. This comes at the cost of a stronger security assumption, namely the hardness of the One-More Discrete Logarithm problem, rather than the standard Discrete Logarithm problem, and a looser security reduction due to a double invocation of the Forking Lemma. As an application, we explain how our new multi-signature scheme could improve both performance and user privacy in Bitcoin.

Keywords: multi-signatures, Schnorr signatures, one-more discrete logarithm problem, forking lemma, Bitcoin

1 Introduction

Multi-signatures. Multi-signature protocols, first introduced by Itakura and Nakamura [IN83], allow a group of signers (each possessing its own private/public key pair) to produce a single signature σ on a message m. Verification of the validity of a purported signature σ can be publicly performed given the message and the set of public keys of all signers. A trivial way to transform a standard signature scheme into a multi-signature scheme is to have each signer produce a stand-alone signature for m with its private key and to concatenate all individual signatures. However, the size of the multi-signature in that case grows linearly with the number of signers. In order to be useful and practical, a multi-signature scheme should produce signatures whose size is (ideally) independent from the number of signers and close to the one of an ordinary signature scheme.

A serious concern when dealing with multi-signature schemes are *rogue-key attacks*, where a subset of t corrupted signers, $1 \leq t < n$, use public keys $\mathsf{pk}'_{n-t+1}, \ldots, \mathsf{pk}'_n$ computed as functions of public keys of honest users $\mathsf{pk}_1, \ldots, \mathsf{pk}_{n-t}$, allowing them to easily produce forgeries for the set of public keys $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_{n-t}, \mathsf{pk}'_{n-t+1}, \ldots, \mathsf{pk}'_n\}$ (even though they may not know the secret keys associated with $\mathsf{pk}'_{n-t+1}, \ldots, \mathsf{pk}'_n$). Such attacks decimated early proposals [LHL94, Har94, HMP95, OO91, Lan96, MH96, OO99] until a formal model was put forward together with a provably secure scheme by Micali, Ohta, and Reyzin [MOR01] (however, their solution relies on a costly and impractical interactive key generation protocol).

One way to generically prevent rogue-key attacks is to require that users prove knowledge (or possession [RY07]) of the secret key during public key registration with a certification authority, a setting known as the knowledge of secret key (KOSK) assumption. In particular, the pairing-based multi-signature schemes by Boldyreva [Bol03] and Lu *et al.* [LOS$^+$06] rely on this assumption for their security. We refer to [BN06, RY07] for a thorough discussion regarding why this assumption is problematic.

To date, the most practical multi-signature scheme provably secure without any assumption on the key setup has been proposed by Bellare and Neven (BN) [BN06] and is based on the Schnorr signature scheme [Sch91]. Following [BN06], we call this setting, where the only requirement is that each potential signer has a public key, the *plain public-key model*. Since our proposal can be seen as a simplification of BN's scheme, we recall it (as well as basic Schnorr signatures) first.

The Schnorr signature scheme [Sch91] uses 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 $R = g^r$, $c = H(X, R, m)$, and $s = r + cx$. The signature is the pair (R, s), and its validity can be checked by verifying whether $g^s = RX^c$. Note that what we just described is the so-called "key-prefixed" variant of the scheme where the public key is hashed together with R and m [BDL$^+$11]. This variant was argued to have a better multi-user security bound than the classic variant [Ber15], but key-prefixing was later shown to be in fact unnecessary for Schnorr signatures to enjoy good multi-user security [KMP16]. For our multi-signature scheme, key-prefixing seems to be required for the security proof to go through, even though we are not aware of any attack otherwise. In our use case, it also more closely matches reality, as the message being signed in Bitcoin transactions always indirectly commits to the public key.

The naive way to design a Schnorr multi-signature scheme 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 cosigner randomly generates and communicates to others a share $R_i = g^{r_i}$; 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 a traditional key-prefixed Schnorr signature with respect to public key $\widetilde{X}$, a property we call *key aggregation*. 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.

The Micali-Ohta-Reyzin multi-signature scheme [MOR01] solves this problem using a sophisticated interactive key generation protocol. In their scheme, Bellare and Neven [BN06] proceeded differently in order to avoid any key setup. Their main idea is to have each cosigner use a distinct "challenge" c_i when computing their partial signature $s_i = r_i + c_i x_i$, defined as $c_i = H(\langle L \rangle, X_i, R, m)$, where as before $R = \prod_{i=1}^{n} R_i$ and $\langle L \rangle$ is some unique encoding of the multiset of public keys $L = \{X_1, \ldots, X_n\}$. The verification equation for a signature (R, s) on message m for public keys L then becomes $g^s = R \prod_{i=1}^{n} X_i^{c_i}$. On top of that, they add a preliminary round in the signature protocol where each signer commits to its share R_i by sending $t_i = H'(R_i)$ to other cosigners first. This prevents any cosigner from setting $R = \prod_{i=1}^{r} R_i$ to some maliciously chosen value (and, on a more technical level, allows to simulate the signature oracle in the security proof). Bellare and Neven showed that this yields a multi-signature scheme provably secure in the plain public key model under the Discrete Logarithm assumption, modeling H and H' as random oracles. However, this scheme does not allow key aggregation anymore since the entire list of public keys is required for verification.

Our Contribution. We propose a new Schnorr-based multi-signature scheme which can be seen as a simpler and more efficient variant of the BN scheme. First, we remove the preliminary commitment phase, so that cosigners start right away by sending each others the shares R_i. Second, we change the way the challenges

[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.

c_i are computed from $c_i = H(\langle L \rangle, X_i, R, m)$ to

$$c_i = H_0(\langle L \rangle, X_i) \cdot H_1(\widetilde{X}, R, m),$$

where $\widetilde{X}$ is the so-called *aggregated public key* corresponding to the multiset of public keys $L = \{X_1, \ldots, X_n\}$, defined as

$$\widetilde{X} = \prod_{i=1}^{n} X_i^{a_i}$$

where $a_i = H_0(\langle L \rangle, X_i)$ (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 $c = H_1(\widetilde{X}, R, m)$. In other words, we have recovered the key aggregation property enjoyed by the naive scheme, albeit with respect to a more complex aggregated key $\widetilde{X} = \prod_{i=1}^{n} X_i^{a_i}$. Note that using $c = H_1(\langle L \rangle, R, m)$ yields a secure scheme as well, but does not allow key aggregation since verification is impossible without knowing all the individual signer keys.

These two simplifications come at the price of complications in the security proof. First, removing the preliminary commitment phase prevents us from using the same technique as Bellare and Neven for simulating the signature oracle. We overcome this problem by relying on the stronger One-More Discrete Logarithm (OMDL) assumption rather than the classic Discrete Logarithm assumption. Second, the "split" form of the challenges c_i prevents us from using directly the Forking Lemma [PS00] for extracting the discrete logarithm of the challenge public key by running the forger twice with distinct random oracle answers. We need a more elaborate strategy which consists in using the Forking Lemma *twice*, in a nested way (so that in total the reduction runs the forger four times), forking first with respect to answers of H_1, which allows to obtain some aggregated key (involving the challenge public key) together with its discrete logarithm, and then with respect to answers of H_0, which allows to retrieve the discrete logarithm of the challenge public key. The "general" Forking Lemma of Bellare and Neven (that we need to generalize even a bit further for our setting) comes in handy to keep the proof as modular as possible. On the downside, this double application of the Forking Lemma results in a rather loose overall security bound. This well-known shortcoming of rewinding-based proofs seems somehow inherent [PV05, GBL08, Seu12] and is often considered as an artifact of the technique rather than an indication of a real hardness gap between breaking the scheme and solving the underlying hard problem.

More on Key Aggregation. Let us elaborate a bit on the benefits of key aggregation. Say a group of n signers want to authorize an action (say, spend

Table 1. Comparison between DL-based multi-signature scheme secure in the plain public-key model when using a group $\mathbb{G}$ of order p and hash functions with ℓ-bit outputs.

scheme	sig. size	pk size	sk size	rounds	key agg.
[BN06]	$\lvert\mathbb{G}\rvert + \lvert p\rvert$	$\lvert\mathbb{G}\rvert$	$\lvert p\rvert$	3	No
[BCJ08]	$3\lvert\mathbb{G}\rvert + 3\lvert p\rvert$	$\lvert\mathbb{G}\rvert$	$\lvert p\rvert$	2	No
[MWLD10]	$\lvert\mathbb{G}\rvert + 2\lvert p\rvert$ or $\ell + 2\lvert p\rvert$	$\lvert\mathbb{G}\rvert$	$2\lvert p\rvert$	2	No
this paper	$\lvert\mathbb{G}\rvert + \lvert p\rvert$ or $\ell + \lvert p\rvert$	$\lvert\mathbb{G}\rvert$	$\lvert p\rvert$	2	Yes

some bitcoins) only if all of them agree, but do not necessarily wish to reveal their individual public keys. Then, they can privately compute the aggregated key $\widetilde{X}$ corresponding to their multiset of public keys and publish it as an ordinary (non-aggregated) key. Signers are ensured that all of them will need to cooperate to produce a signature which is valid under $\widetilde{X}$, whereas verifiers will not even learn that $\widetilde{X}$ is in fact an aggregated key. Moreover, $\widetilde{X}$ can be computed by a third party (say, someone sending bitcoins to the group of signers) just from the list of public keys, without interacting with the signers. As we will see, this property will prove instrumental for obtaining a more compact and privacy-preserving variant of so-called n-of-n multi-signature transactions in Bitcoin (see below).

Related Work. Two variants of the BN multi-signature scheme have been proposed previously. Bagherzandi *et al.* [BCJ08] reduced the number of rounds from three to two using an homomorphic commitment scheme. However, this increases the signature size and the computational cost of signing and verification. Ma *et al.* [MWLD10] proposed a variant based on Okamoto's signature scheme [Oka92] and a "double hashing" technique (the two hash functions being composed rather than multiplied as in our scheme), which allows to reduce the signature size compared to [BCJ08] while using again only two rounds. However, none of these two variants allows key aggregation. A comparison of the four discrete logarithm-based multi-signature schemes is provided in Table 1.

Interactive Aggregate Signatures. In some situations, it might be useful to allow each participant to sign a different message rather than a single common one. Such a protocol, where each signer has its own message m_i to sign, and the joint signature proves that the i-th signer has signed m_i, is called an interactive aggregate signature (IAS) scheme. IAS schemes are more general than multi-signature schemes, but less flexible than non-interactive aggregate signatures [BGLS03, BNN07] and sequential aggregate signatures [LMRS04]. Bellare and Neven [BN06] suggested a generic (i.e., black-box) way to turn any multi-signature scheme into an IAS scheme: the signers simply run the multi-signature protocol using as message the tuple of all public key/message pairs involved in the IAS protocol. (Note that for BN's scheme and ours, this does not increase the number of communication rounds since messages can be sent together with shares R_i.) However, a subtle problem arises with this generic construction in the plain public key model, as we explain in details in Appendix A. We also suggest a simple (yet non black-box) way to turn the BN multi-signature scheme into a secure IAS scheme.

Applications to Bitcoin. Bitcoin [Nak08] is a digital currency scheme in which all participants (are able to) validate transactions. These transactions consist of *outputs*, which have a verification key and amount, and *inputs*[4] which are references to outputs of earlier transactions. Each input contains a signature of a modified version of the transaction to be validated with its referenced output's key. In fact, some outputs even require multiple signatures to be spent. Transactions spending such an output are often referred to as m-of-n multi-signature transactions [And11], and the current implementation corresponds to the trivial way of building a multi-signature scheme by concatenating individual signatures. Additionally, a threshold policy can be enforced where only m valid signatures out of the n possible ones are needed to redeem the transaction (again, this is the most straightforward way to turn a multi-signature scheme into some kind of basic threshold signature scheme).

Today, Bitcoin uses ECDSA signatures [ANS05, NIS13] over the `secp256k1` curve [SEC10] to authenticate transactions. As Bitcoin nodes fully verify all transactions, signature size and verification time are important design considerations, while signing time is much less so. Besides, signatures account for a large part of the size of Bitcoin transactions. Because of this, using multi-signatures seems appealing. However, designing multiparty ECDSA signature schemes is notably cumbersome [MR01, GGN16, Lin17] due to the modular inversion involved in signing, and moving to Schnorr signatures would definitely help deploying compact multi-signatures. While several multi-signature schemes could offer an improvement over the currently available method, two properties increase the possible impact:

- The availability of key aggregation removes the need for verifiers to see all the involved keys, improving bandwidth, privacy, and validation cost.
- Security under the plain public key model enables multi-signatures *across* multiple inputs of a transaction, where the choice of signers cannot be committed to in advance. This greatly increases the number of situations in which multi-signatures are beneficial.

Our contribution is novel in combining these two properties. The removal of BN's commitment phase further improves its convenience, requiring only two rather than three interaction rounds.

Organization of the Paper. We start in Section 2 by providing definitions and stating our version of the general Forking Lemma. In Section 3, we specify our new multi-signature protocol, and also describe some attacks on simpler variants. Section 4 is then entirely devoted to the security proof of the scheme. Finally, in Section 5 we expose the applications of multi-signatures to Bitcoin.

[4] All Bitcoin transactions have at least one input except coinbase transactions which reward miners when they validate blocks and bootstrap the currency supply.

2 Preliminaries

2.1 Notation and Definitions

Notation. 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 $\mathcal{A}$ is a randomized algorithm, we let $y \leftarrow \mathcal{A}(x_1, \ldots; \rho)$ denote the operation of running $\mathcal{A}$ on inputs $x_1, \ldots$ and random coins ρ and assigning its output to y, and $y \leftarrow_\$ \mathcal{A}(x_1, \ldots)$ when coins ρ are chosen uniformly at random. Given a random variable Y, we let $\mathbf{E}[Y]$ denote its expected value.

In all the following, we let $\mathbb{G}$ be a cyclic group of order p, where p is a k-bit integer, and g be a generator of $\mathbb{G}$. The group $\mathbb{G}$ will be denoted multiplicatively, and we will conflate group elements and their representation when given as input to hash functions. We call the triplet $(\mathbb{G}, p, g)$ the *group parameters*. We adopt the concrete security approach, i.e., we view $(\mathbb{G}, p, g)$ as fixed, but the bit length k of p can be regarded as a security parameter if need be.

The One-More Discrete Logarithm Problem. The One-More Discrete Logarithm (OMDL) problem is an extension of the standard Discrete Logarithm (DL) problem which consists in finding the discrete logarithm of $q+1$ group elements by making at most q calls to an oracle solving the discrete logarithm problem. It was introduced in [BNPS03] and used for example to prove the security of the Schnorr identification protocol against active and concurrent attacks [BP02]. The formal definition follows.

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 $\mathcal{A}$.*

2.2 Syntax and Security Definition of Multi-Signature Schemes

Syntax. A multi-signature scheme Π consists of three algorithms (KeyGen, Sign, Ver). System-wide parameters are selected by a setup algorithm taking as input the security parameter. We assume that this setup phase is performed correctly (or at least that correctness can be checked efficiently, so that it does not have to be run by a trusted entity) and we do not mention 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 signature algorithm Sign is run by each participant on input its key pair $(\mathsf{sk}, \mathsf{pk})$, a multiset of public keys $L = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ containing at least once its own public key pk, and a message m, and returns a signature σ for L and m. The deterministic verification algorithm

Ver takes as input a multiset of public keys $L = \{\mathsf{pk}_1, \dots, \mathsf{pk}_n\}$, a message m, and a signature σ, and returns 1 is the signature is valid for L and m and 0 otherwise.

Correctness requires that for any private/public key pairs $(\mathsf{sk}_1, \mathsf{pk}_1), \dots, (\mathsf{sk}_n, \mathsf{pk}_n)$ and any message m, if a group of signers with public keys $L = \{\mathsf{pk}_1, \dots, \mathsf{pk}_n\}$ run the signature protocol for m without deviating from the specification, then all signers output the same signature σ which is valid for L and m, i.e., $\mathsf{Ver}(L, m, \sigma) = 1$. Our syntax assumes that each cosigner 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 output σ.

Security. Our security model is the same as the one of [BN06] and requires that it be 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 adversary can engage in any number of (concurrent) signature protocols with the honest signer before returning a forgery attempt.

More formally, the security game involving an adversary (forger) $\mathcal{F}$ proceeds as follows:

– A key pair for the honest signer $(\mathsf{sk}^*, \mathsf{pk}^*) \leftarrow_\$ \mathsf{KeyGen}()$ is generated at random and the public key pk^* is given as input to $\mathcal{F}$.
– The forger can engage arbitrary signing protocols with the honest user. Formally, it has access to a signature oracle which must be called on input a multiset of public keys $L = \{\mathsf{pk}_1, \dots, \mathsf{pk}_n\}$ where pk^* occurs at least once and a message m. The oracle implements the signing algorithm corresponding to the honest user's secret key sk^*, while the forger plays the role of all other signers in L (potentially deviating from the protocol). Note that pk^* might appear $k \geq 2$ times in L, in which case the forger plays the role of $k - 1$ instances of pk^*, but since the signing algorithm only depends on the multiset L there is no need to specify which instances are associated with the honest oracle and the forger. The forger can interact concurrently with as many independent instances of the honest signature oracle as it wishes.
– At the end of its execution, the forger returns a multiset of public keys $L = \{\mathsf{pk}_1, \dots, \mathsf{pk}_n\}$, a message m, and a signature σ. The forger wins if $\mathsf{pk}^* \in L$, the forgery is valid, i.e., $\mathsf{Ver}(L, m, \sigma) = 1$, and $\mathcal{F}$ never initiated a signature protocol 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 defined as follows.

Definition 2. *Let $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver})$ be a multi-signature scheme. We say that an adversary $\mathcal{F}$ is a $(t, q_s, q_h, N, \varepsilon)$-forger in the random oracle model against the multi-signature scheme Π if it runs in time at most t, initiates at most*

q_s signature protocols with the honest signer, makes at most q_h random oracle queries,[5] *and wins the above security game with probability at least ε, the size of L in any signature query and in the forgery being at most N.*

2.3 A General Forking Lemma

As in [BN06], our security proof will rely on a "general 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, for reasons which should become clear later, the general Forking Lemma of Bellare and Neven [BN06] is not general enough for our setting. In short, in BN's lemma, only the values $h_i, \dots, h_q$ (i.e., the post-fork random oracle answers used by the reduction) are refreshed in the second execution of $\mathcal{A}$, whereas we will need to refresh another part of the input of $\mathcal{A}$ (jumping ahead, in our case the reduction must use fresh OMDL challenges, used as shares R_1 sent to the signature oracle in signature queries, after the two executions of $\mathcal{A}$ have forked; see Footnote 15).

Since the proof of the lemma below is very similar to the one of [BN06, Lemma 1], it is deferred to Appendix B.

Lemma 1. *Fix integers q, m, and ℓ. Let $\mathcal{A}$ be a randomized algorithm which takes as input some main input inp following some unspecified distribution, ℓ-bit strings $h_1, \dots, h_q$, and elements $V_1, \dots, V_m$ from some arbitrary set S, and returns either a distinguished failure symbol $\perp$, or a tuple (i, j, out), where $i \in \{1, \dots, q\}$, $j \in \{0, \dots, m\}$, 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 draw of inp, $h_1, \dots, h_q \leftarrow_\$ \{0,1\}^\ell$, $V_1, \dots, V_m \leftarrow_\$ S$, 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 $V_1, V'_1, \dots, V_m, V'_m \in S$, described on Figure 1. Let frk be the probability (over the draw of inp, $V_1, V'_1, \dots, V_m, V'_m \leftarrow_\$ S$, and the random coins of $\mathsf{Fork}^{\mathcal{A}}$) that $\mathsf{Fork}^{\mathcal{A}}$ returns a non-$\perp$ output. Then*

$$\mathsf{frk} \geq \mathsf{acc}(\mathcal{A}) \left(\frac{\mathsf{acc}(\mathcal{A})}{q} - \frac{1}{2^\ell} \right).$$

3 Our New Multi-Signature Scheme

3.1 Description

Our new multi-signature scheme, denoted MuSig in all the following, is parameterized by group parameters $(\mathbb{G}, p, g)$ where p is a k-bit integer, $\mathbb{G}$ is a cyclic group of order p, and g is a generator of G, and by two hash functions[6]

[5] If the scheme relies on several random oracles, we assume that $\mathcal{F}$ makes at most q_h queries to each of them.

[6] These hash functions can be constructed from a single one using proper domain separation.

```
algorithm $\mathsf{Fork}^{\mathcal{A}}(\mathsf{inp}, V_1, V_1', \ldots, V_m, V_m')$
  pick random coins $\rho$ for $\mathcal{A}$
  $h_1, \ldots, h_q \leftarrow_{\$} \{0,1\}^{\ell}$
  $\alpha \leftarrow \mathcal{A}(\mathsf{inp}, h_1, \ldots, h_q, V_1, \ldots, V_m; \rho)$
  if $\alpha = \bot$ then return $\bot$
  else parse $\alpha$ as $(i, j, \mathsf{out})$
  $h_i', \ldots, h_q' \leftarrow_{\$} \{0,1\}^{\ell}$
  $\alpha' \leftarrow \mathcal{A}(\mathsf{inp}, h_1, \ldots, h_{i-1}, h_i', \ldots, h_q', V_1, \ldots, V_j, V_{j+1}', \ldots, V_m'; \rho)$
  if $\alpha' = \bot$ then return $\bot$
  else parse $\alpha'$ as $(i', j', \mathsf{out}')$
  if $(i = i'$ and $h_i \neq h_i')$ then return $(i, \mathsf{out}, \mathsf{out}')$
  else return $\bot$
```

Fig. 1. The “forking” algorithm $\mathsf{Fork}^{\mathcal{A}}$ built from $\mathcal{A}$.

$H_0, H_1 : \{0,1\}^* \to \{0,1\}^{\ell}$. Note that for groups used in cryptography (multiplicative subgroups of prime fields and elliptic curve groups), correctness of the group parameters generation can be efficiently checked, as discussed in Section 2.2.

For notational simplicity, in all the following, we drop notation $\langle L \rangle$ used in the introduction: when a multiset of public keys (in our case, group elements) $L = \{\mathsf{pk}_1 = X_1, \ldots, \mathsf{pk}_n = X_n\}$ is given as input to a hash function, we assume it is uniquely encoded first, e.g. using the lexicographical order.

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

Signing. Let X_1 and x_1 be the public and private key of a specific signer, 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.[7] For $i \in \{1, \ldots, n\}$, the signer computes

$$a_i = H_0(L, X_i) \tag{1}$$

and then the “aggregated” public key $\widetilde{X} = \prod_{i=1}^{n} X_i^{a_i}$. Then, the signer generates a random $r_1 \leftarrow_{\$} \mathbb{Z}_p$, computes $R_1 = g^{r_1}$, and sends R_1 to all other cosigners. Upon reception of $R_2, \ldots, R_n$ from other cosigners, it computes

$$R = \prod_{i=1}^{n} R_i,$$
$$c = H_1(\widetilde{X}, R, m),$$
$$s_1 = r_1 + c a_1 x_1 \bmod p,$$

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

and sends s_1 to all other cosigners. Finally, upon reception of $s_2, \ldots, s_n$ from 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 $a_i = H_0(L, X_i)$ for $i \in \{1, \ldots, n\}$, $\widetilde{X} = \prod_{i=1}^{n} X_i^{a_i}$, $c = H_1(\widetilde{X}, R, m)$ 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 similar to standard Schnorr signatures (with the public key included in the hash call) with respect to the "aggregated" public key $\widetilde{X} = \prod_{i=1}^{n} X_i^{a_i}$. We discuss (secure) variants of the scheme in Section 4.3.

3.2 Attacks against Simpler Variants and Derandomization

Before proving the security of the scheme described above, we explain why simpler ways to compute the aggregated public key do not work, and why derandomized signing cannot be applied.

Simpler Key Aggregation Variants. As already pointed out in the introduction, a simplified version of the scheme where a_i is defined to be 1 would be insecure. In this case, the aggregated public key would be $\widetilde{X} = \prod_{i=1}^{n} X_i$. This is clearly vulnerable to a rogue-key attack where the last signer reveals his key as $X_n (\prod_{i=1}^{n-1} X_i)^{-1}$, resulting in an aggregated key $\widetilde{X} = X_n$, which the last signer clearly can forge signatures for.

A more complicated scheme where a_i is defined as $H_0(X_i)$ is insecure when multiple keys are controlled by the attacker. Assume that the honest signer controls key X_1. The aggregate of just that key alone is $\widetilde{X}_h = X_1^{H_0(X_1)}$. The attacker can then use Wagner's algorithm [Wag02] to find $n-1$ integers $y_2, \ldots, y_n$ such that $\sum_{i=2}^{n} H_0(\widetilde{X}_h g^{y_i}) = -1 \bmod p$. For sufficiently large values of $n-1$, this can be done in $O(2^{2\sqrt{k}})$ time, where k is the bit-length of p. The attacker then reveals public keys $X_i = \widetilde{X}_h g^{y_i}$ for $i = 2 \ldots n$. The overall aggregated key in that case becomes

$$\begin{aligned}
\widetilde{X} &= \widetilde{X}_h \prod_{i=2}^{n} \left(\widetilde{X}_h g^{y_i}\right)^{H_0(\widetilde{X}_h g^{y_i})} \\
&= g^{\sum_{i=2}^{n} y_i H_0(\widetilde{X}_h g^{y_i})} \widetilde{X}_h^{1+\sum_{i=2}^{n} H_0(\widetilde{X}_h g^{y_i})} \\
&= g^{\sum_{i=2}^{n} y_i H_0(\widetilde{X}_h g^{y_i})}
\end{aligned}$$

which just the attacker can forge signatures for.

Derandomized Signing. To avoid the need for a strong random number generation at signing time, the creation of the random values r_i is often done using an algorithm like RFC6979 [Por13], which computes them using a deterministic function $f(x_i, m)$. When multiple signers are cooperating, one must ensure that the same random value is not reused when other signers change their random values in a repeated signing attempt. Otherwise signers can recover others' private keys.

Assume Alice and Bob, holding respective key pairs (x_1, g^{x_1}) and (x_2, g^{x_2}), want to jointly produce a signature. Alice produces r_1 and sends $R_1 = g^{r_1}$ to Bob. In a first attempt, Bob responds with R_2. Alice computes

$$\begin{aligned} R &= R_1 R_2, \\ c &= H_1(\widetilde{X}, R, m), \\ s_1 &= r_1 + c a_1 x_1 \bmod p, \end{aligned}$$

and sends s_1 over to Bob. Bob chooses not to produce a valid s_2, and thus subsequent protocol steps fail. A new signing attempt takes place, and Alice again sends R_1. Bob responds with $R'_2 \neq R_2$. Alice computes $c' = H_1(\widetilde{X}, R_1 R'_2, m)$ and $s'_1 = r_1 + c' a_1 x_1$, and sends s'_1 over. Bob can now derive

$$x_1 = \frac{s_1 - s'_1}{a_1(c - c')} \bmod p.$$

To avoid this problem, each signer must ensure that whenever any R_j sent by other cosigners or the message m changes, his r_i value changes unpredictably. As long as f is deterministic, this implies a circular dependency in the choice of random values. It can be solved by introducing (non-repeating) randomness or a counter into the function f. Unfortunately, this requires a secure random number generator at signing time, or state that is kept between signing attempts.

4 Security of the New Multi-Signature Scheme

4.1 Preliminaries

In this section, we prove the security of the MuSig scheme, expressed by the following theorem.

Theorem 1. *Assume that there exists a $(t, q_s, q_h, N, \varepsilon)$-forger $\mathcal{F}$ against the multi-signature scheme* MuSig *with group parameters $(\mathbb{G}, p, g)$ and hash functions $H_0, H_1 : \{0,1\}^* \to \{0,1\}^\ell$ modeled as random oracles. Then there exists an algorithm $\mathcal{C}$ which $(4q_s, t', \varepsilon')$-solves the OMDL problem for $(\mathbb{G}, p, g)$, with $t' = 4t + 4Nt_{\mathrm{exp}} + O(N(q_h + q_s + 1))$ where t_{exp} is the time of an exponentiation in $\mathbb{G}$ and*

$$\varepsilon' \geq \frac{\varepsilon^4}{(q_h + q_s + 1)^3} - \frac{11}{2^\ell}.$$

In other words, $\mathcal{C}$ takes as input $4q_s + 1$ uniformly random group elements

$$X^*, U_1, U'_1, U''_1, U'''_1, \ldots, U_{q_s}, U'_{q_s}, U''_{q_s}, U'''_{q_s},$$

makes at most $4q_s$ queries to a discrete logarithm oracle $\mathsf{DL}_g(\cdot)$, and returns the discrete logarithm of all its challenges in time at most t' and with probability at least ε'.

Before proving the theorem, we start with a number of observations.

Obstacles on the Way to the Security Proof. Let us recall the security game defined in Section 2.2, 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 forger $\mathcal{F}$. Then, the forger 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^*. More concretely, it has access to an interactive signature oracle working as follows: the forger sends a multiset L of public keys with $X^* \in L$ (we assume the oracle returns $\perp$ if $X^* \notin L$) and a message m to sign; the signing oracle parses L as $\{X_1 = X^*, X_2, \ldots, X_n\}$, draws a random $r_1 \leftarrow_\$ \mathbb{Z}_p$, and sends $R_1 = g^{r_1}$ to the forger; the forger sends group elements $R_2, \ldots, R_n$; the signing oracle computes

$$\begin{aligned} R &= \prod_{i=1}^{r} R_i, \\ c &= H_1(\widetilde{X}, R, m), \\ s_1 &= r_1 + c a_1 x^* \bmod p, \end{aligned}$$

and returns s_1 to the forger. Note that the remaining of the protocol, where $\mathcal{F}$ sends $s_2, \ldots, s_n$ to the signature oracle which outputs $s = \sum_{i=1}^{n} s_i \bmod p$, can be omitted since the oracle's behavior does not depend on x^* and can be perfectly simulated by the forger.

Observe that the forger "controls" the value of R used in signature queries since he can choose $R_2, \ldots, R_n$ after having received R_1 from the signature oracle. This stands in contrast to standard Schnorr signatures, where R is randomly chosen by the signature oracle. This also forbids to use the textbook way of simulating the signature oracle by randomly drawing s_1 and c, computing $R_1 = g^{s_1}(X^*)^{-a_1 c}$, and later programming $H_1(\widetilde{X}, R, m) := c$, since the forger might have made the random oracle query $H_1(\widetilde{X}, R, m)$ *before* engaging the corresponding signature protocol.[8] To solve this problem, we let the simulator compute s_1 by querying $\mathsf{DL}_g(R_1(X^*)^{a_1 c})$ to the DL oracle available in the formulation of the OMDL problem. We use a fresh DL challenge as R_1 in each signature query, and the reduction will be able to compute its discrete logarithm once x^* has been retrieved.

The second 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 forger 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 $R = R'$, $H_1(\widetilde{X}, R, m)$ was programmed in both executions to some common value h_1, $H_0(L, X_i)$ was programmed in both executions to the same

[8] This is the reason why Bellare and Neven [BN06] add an extra round at the beginning of the protocol where each signer commits to its share R_i before receiving the shares of other signers.

value a_i for each i such that $X_i \neq X^*$, and $H_0(L, X^*)$ was programmed to two distinct values h_0 and h'_0 in the two executions, implying that

$$g^s = R(X^*)^{n^* h_0 h_1} \prod_{\substack{i \in \{1,\ldots,n\} \\ X_i \neq X^*}} X_i^{a_i h_1}$$
$$g^{s'} = R(X^*)^{n^* h'_0 h_1} \prod_{\substack{i \in \{1,\ldots,n\} \\ X_i \neq X^*}} X_i^{a_i h_1},$$

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 $H_0(L, X^*)$ does not work: indeed, at this moment, the relevant query $H_1(\widetilde{X}, R, m)$ might not have been made yet by the forger,[9] and there is no guarantee that the forger will ever make this same query again in the second execution, let alone return a forgery corresponding to the same H_1 query. In order to remedy this situation, we fork the execution of the forger *twice*: once on the answer to the query $H_1(\widetilde{X}, R, m)$, which allows to retrieve the discrete logarithm of the aggregated public key $\widetilde{X}$ with respect to which the adversary returns a forgery, and then on the answer to $H_0(L, X^*)$, which allows to retrieve the discrete logarithm of X^*.

4.2 Security Proof

Proof Sketch. We first construct a "wrapping" algorithm $\mathcal{A}$ which essentially runs the forger, simulating H_0 and H_1 uniformly at random and the signature oracle thanks to a DL oracle, and returns a forgery together with some information about the forger execution unless a couple of bad events happen.[10] Then, we use $\mathcal{A}$ to construct an algorithm $\mathcal{B}$ which runs the forking algorithm $\mathsf{Fork}^{\mathcal{A}}$ as defined in Section 2.3 (where the fork is w.r.t. the answer to the H_1 query related to the forgery), allowing him to return a multiset of public keys L together with the discrete logarithm of the corresponding aggregated public key. Finally, we use $\mathcal{B}$ to construct an algorithm $\mathcal{C}$ solving the OMDL problem by running $\mathsf{Fork}^{\mathcal{B}}$ (where the fork is now w.r.t. the answer to the H_0 query related to the forgery). Throughout the proof, the reader might find helpful to refer to Figure 2 which illustrates the inner working of $\mathcal{C}$.

[9] In fact, it is easy to see that the forger can only guess the value of the aggregated public key $\widetilde{X}$ corresponding to L at random before making the relevant queries $H_0(L, X_i)$ for $X_i \in L$, so that the query $H_1(\widetilde{X}, R, m)$ can only come after the relevant queries $H_0(L, X_i)$ except with negligible probability.

[10] 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 aggregated public keys are equal, since when this happens the forger can make a signature query for (L, m) and return the resulting signature σ as a forgery for (L', m). Jumping ahead, this will correspond to bad event AKColl defined in the proof of Lemma 2.

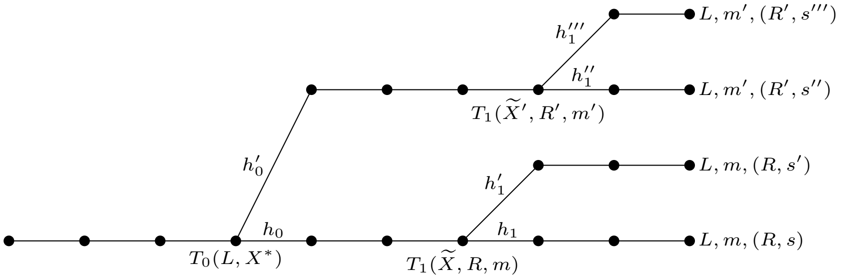

Fig. 2. A possible execution of algorithm $\mathcal{C}$. Each path from the leftmost root to one of the four rightmost leaves represent an execution of the forger. Each vertex symbolizes an assignment to tables T_0 or T_1 used to program H_0 and H_1, and the edge originating from this vertex symbolizes the value used for the assignment. Leaves symbolize the forgery returned by the forger. Only vertices and edges that are relevant to the forgery are labeled.

Normalizing Assumptions. In all the following, we assume that the forger never repeats a query, and only makes "well-formed" queries, meaning that $X^* \in L$ and $X \in L$ for any query $H_0(L, X)$ (this is without loss of generality, since "ill-formed" queries are irrelevant and could simply be answered uniformly at random in the simulation). We also assume *wlog* that the adversary makes exactly q_h queries to each random oracle and exactly q_s signature queries.

We start with the construction of the wrapping algorithm $\mathcal{A}$.

Lemma 2. *Assume that there exists a $(t, q_s, q_h, N, \varepsilon)$-forger $\mathcal{F}$ in the random oracle model against the multi-signature scheme* MuSig *with group parameters $(\mathbb{G}, p, g)$ and let $q = q_h + q_s + 1$. Then there exists an algorithm $\mathcal{A}$ that takes as input $q_s + 1$ uniformly random group elements $X^*, U_1, \dots, U_{q_s}$ and uniformly random ℓ-bit strings $h_{0,1}, \dots, h_{0,q}$ and $h_{1,1}, \dots, h_{1,q}$,*[11] *makes at most q_s queries to a discrete logarithm oracle $\mathsf{DL}_g(\cdot)$, and, with accepting probability (as defined in Lemma 1) at least*

$$\varepsilon - \frac{2(q_h + q_s + 1)^2}{2^\ell},$$

outputs $(i_0, j_0, i_1, j_1, L, R, s, \mathbf{a})$ where $i_0, i_1 \in \{1, \dots, q\}$, $j_0, j_1 \in \{0, \dots, q_s\}$, $L = \{X_1, \dots, X_n\}$ is a multiset of public keys such that $X^* \in L$, $\mathbf{a} = (a_1, \dots, a_n)$ is a tuple of ℓ-bit values such that $a_i = h_{0,i_0}$ for any i such that $X_i = X^*$, and

$$g^s = R \prod_{i=1}^{n} X_i^{a_i h_{1,i_1}}. \tag{2}$$

[11] Strings $h_{0,i}$, resp. $h_{1,i}$ will be used to answers queries to H_0, resp. H_1. We need $q_h + q_s + 1$ answers for each random oracle because one query to H_0 and one query to H_1 may be incurred by each signature query and by the final verification of the validity of the forgery.

Proof. We construct algorithm $\mathcal{A}$ as follows. It initializes two empty tables T_0 and T_1 for storing simulated values for respectively H_0 and H_1 and three counters ctrh_0, ctrh_1, and ctrs (initially zero) that will be incremented respectively each time an entry of the form $T_0(\cdot, X^*)$ is assigned, each time an assignment is made in T_1, and each time the forger makes a signature query. Then, it picks random coins ρ_F and runs the forger $\mathcal{F}$ on input the public key X^*, answering its queries as follows.

- Hash query $H_0(L, X)$: (Recall that by assumption $X^* \in L$ and $X \in L$.) If $T_0(L, X)$ is undefined, $\mathcal{A}$ increments ctrh_0, assigns $T_0(L, X^*) := h_{0,\mathsf{ctrh}_0}$, and assigns random values to $T_0(L, X')$ for all $X' \in L \setminus \{X^*\}$. Then, it returns $T_0(L, X)$.

- Hash query $H_1(\widetilde{X}, R, m)$: If $T_1(\widetilde{X}, R, m)$ is undefined, then $\mathcal{A}$ increments ctrh_1 and assigns $T_1(\widetilde{X}, R, m) := h_{1,\mathsf{ctrh}_1}$. Then, it returns $T_1(\widetilde{X}, R, m)$.

- Signature query (L, m): If $X^* \notin L$, then $\mathcal{A}$ returns $\perp$ to the forger. Otherwise, it parses L as $\{X_1 = X^*, X_2, \ldots, X_n\}$. If $T_0(L, X^*)$ is undefined,[12] it makes an "internal" query to $H_0(L, X^*)$ which ensures that $T_0(L, X_i)$ is defined for each $i \in \{1, \ldots, n\}$, sets $a_i := T_0(L, X_i)$, and computes $\widetilde{X} := \prod_{i=1}^{n} X_i^{a_i}$. Then, it increments ctrs, lets $R_1 := U_{\mathsf{ctrs}}$, and sends R_1 to the forger. Upon reception of $R_2, \ldots, R_n$, $\mathcal{A}$ computes $R := \prod_{i=1}^{n} R_i$. If $T_1(\widetilde{X}, R, m)$ is undefined, it makes an "internal" query to $H_1(\widetilde{X}, R, m)$ and lets $c := T_1(\widetilde{X}, R, m)$. Finally, it queries the DL oracle on $R_1(X^*)^{a_1 c}$, obtaining an answer s_1 that it returns to the forger.

If $\mathcal{F}$ returns $\perp$, $\mathcal{A}$ outputs $\perp$ as well. Otherwise, if the forger returns a purported forgery (R, s) for a public key multiset L such that $X^* \in L$ and a message m, $\mathcal{A}$ parses L as $\{X_1 = X^*, \ldots, X_n\}$ and checks the validity of the forgery as follows. If $T_0(L, X^*)$ is undefined, it makes an "internal" query to $H_0(L, X^*)$ which ensures that $T_0(L, X_i)$ is defined for each $i \in \{1, \ldots, n\}$, sets $a_i := T_0(L, X_i)$, and computes $\widetilde{X} := \prod_{i=1}^{n} X_i^{a_i}$. If $T_1(\widetilde{X}, R, m)$ is undefined, it makes an "internal" query to $H_1(\widetilde{X}, R, m)$ and lets $c := T_1(\widetilde{X}, R, m)$.[13] If $g^s \neq R\widetilde{X}^c$, i.e., the forgery is invalid, $\mathcal{A}$ outputs $\perp$, otherwise it takes the following additional steps. Let

- i_0 be the index such that $T_0(L, X^*) = h_{0,i_0}$,
- j_0 be the value of ctrs at the moment $T_0(L, X^*)$ is assigned,
- i_1 be the index such that $T_1(\widetilde{X}, R, m) = h_{1,i_1}$,
- j_1 be the value of ctrs at the moment $T_1(\widetilde{X}, R, m)$ is assigned.

If the assignment $T_0(L, X^*) := h_{0,i_0}$ occurred *after* the assignment $T_1(\widetilde{X}, R, m) := h_{1,i_1}$, we say that bad event $\mathsf{BadOrder}$ happened. If there exists another multiset

[12] This is true *iff* L never appeared in a previous query to H_0 or a previous signature query.

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

of public keys L' such that, at the end of the execution, $T_0(L', X')$ is defined for each $X' \in L'$ and the aggregated keys corresponding to L and L' are equal, we say that event AKColl (*aggregated key collision*) happened. In both cases, $\mathcal{A}$ returns $\perp$. Otherwise, it returns $(i_0, j_0, i_1, j_1, L, R, s, \mathbf{a})$, where $\mathbf{a} = (a_1, \ldots, a_n)$. By construction, $a_i = h_{0,i_0}$ for each i such that $X_i = X^*$, and the validity of the forgery implies Equation (2).

We must now lower bound the accepting probability of $\mathcal{A}$. It is easy to see that $\mathcal{A}$ perfectly simulates the security experiment to the forger when $h_{0,1}, \ldots, h_{0,q}, h_{1,1}, \ldots, h_{1,q}$ are uniformly random. Hence, with probability at least ε, the forger eventually returns a valid forgery. It remains to upper bound the probability that $\mathsf{BadOrder}$ or AKColl occurs.

Note that by construction of $\mathcal{A}$, for any multiset L' appearing at some point in the queries of the adversary or in its forgery, assignments $T_0(L', X')$ for all $X' \in L'$ are concomitant and occur the first time L' appears either in a query to H_0, or in a signature query, or in the forgery. We will refer to the set of assignments $\{T_0(L', X') := a', X' \in L'\}$ as the *set of* T_0 *assignments related to* L'. Note that there are at most q sets of T_0 assignments and that each of them contains a unique assignment $T_0(L', X^*) := h_{0,i}$ for some $i \in \{1, \ldots, q\}$.

In order to upper bound the probability that $\mathsf{BadOrder}$ happens, we upper bound the probability that some set of T_0 assignments related to some multiset L' (not necessarily the one returned in the forgery) results in the aggregated key $\widetilde{X}'$ corresponding to L' being equal to the first argument of a defined entry in table T_1 (which is clearly a necessary condition for $\mathsf{BadOrder}$ to happen). Considering the i-th set of T_0 assignments, one has

$$\widetilde{X}' = (X^*)^{n^* h_{0,i}} \cdot Z$$

where $n^* \geq 1$ is the number of times X^* appears in L' and $h_{0,i}$ is uniformly random in $\{0,1\}^\ell$ and independent from Z which accounts for public keys different from X^* in L'. Hence, $\widetilde{X}'$ is uniformly random in a set of at least 2^ℓ group elements. Since there are always at most q defined entries in T_1 and at most q sets of assignments, $\mathsf{BadOrder}$ happens with probability at most $q^2/2^\ell$.[14]

In order to upper bound the probability that AKColl happens, we upper bound the probability that some set of T_0 assignments related to some multiset L' (not necessarily the one returned in the forgery) results in the aggregated key $\widetilde{X}'$ corresponding to L' being equal to the aggregated key $\widetilde{X}''$ corresponding to some previous set of T_0 assignments related to some other multiset L'' (again, neither L' nor L'' need be the multiset returned in the forgery). Since each aggregated key is uniform in a set of at least 2^ℓ group elements and independent from other aggregated keys, this happens with probability at most $q^2/2^\ell$. □

[14] Note that for this argument to go through, we rely on $\widetilde{X}$ being included in the call to H_1. As already said in introduction, we do not know how to prove the security for the variant without "aggregate key-prefixing" where $\widetilde{X}$ is omitted from the call to H_1, even though we are not aware of any attack.

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

Lemma 3. *Assume that there exists a* $(t, q_s, q_h, N, \varepsilon)$*-forger* $\mathcal{F}$ *in the random oracle model against the multi-signature scheme* MuSig *with group parameters* $(\mathbb{G}, p, g)$ *and let* $q = q_h + q_s + 1$*. Then there exists an algorithm* $\mathcal{B}$ *that takes as input* $2q_s + 1$ *uniformly random group elements* $X^*, U_1, U_1', \ldots, U_{q_s}, U_{q_s}'$ *and uniformly random* ℓ*-bit strings* $h_{0,1}, \ldots, h_{0,q}$*, makes at most* $2q_s$ *queries to a discrete logarithm oracle* $\mathsf{DL}_g(\cdot)$*, and, with accepting probability (as defined in Lemma 1) at least*

$$\frac{\varepsilon^2}{q_h + q_s + 1} - \frac{4(q_h + q_s + 1) + 1}{2^\ell},$$

outputs a tuple $(i_0, j_0, L, \mathbf{a}, \tilde{x})$ *where* $i_0 \in \{1, \ldots, q\}$*,* $j_0 \in \{0, \ldots, q_s\}$*,* $L = \{X_1, \ldots, X_n\}$ *is a multiset of public keys such that* $X^* \in L$*,* $\mathbf{a} = (a_1, \ldots, a_n)$ *is a tuple of* ℓ*-bit values such that* $a_i = h_{0,i_0}$ *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 $\mathcal{B}$ runs $\mathsf{Fork}^{\mathcal{A}}$ with $\mathcal{A}$ as defined in Lemma 2 and takes a few additional steps described below. The mapping with notation of Lemma 1 is as follows:

- X^* and $h_{0,1}, \ldots, h_{0,q}$ play the role of inp,
- $h_{1,1}, \ldots, h_{1,q}$ play the role of $h_1, \ldots, h_q$,
- $U_1, U_1', \ldots, U_{q_s}, U_{q_s}'$ play the role of $V_1, V_1', \ldots, V_m, V_m'$,
- (i_1, j_1) play the role of (i, j),
- $(i_0, j_0, L, R, s, \mathbf{a})$ play the role of out.

In more details, $\mathcal{B}$ picks random coins ρ_A and runs algorithm $\mathcal{A}$ on coins ρ_A, group elements $X^*, U_1, \ldots, U_{q_s}$ and ℓ-bit strings $h_{0,1}, \ldots, h_{0,q}$ and $h_{1,1}, \ldots, h_{1,q}$, where $h_{1,1}, \ldots, h_{1,q}$ are drawn uniformly at random by $\mathcal{B}$ (recall that $h_{0,1}, \ldots, h_{0,q}$ are part of the *input* of $\mathcal{B}$ and will be the same in both runs of $\mathcal{A}$). All DL oracle queries made by $\mathcal{A}$ are relayed by $\mathcal{B}$ to its own DL oracle. If $\mathcal{A}$ returns $\perp$, $\mathcal{B}$ returns $\perp$ as well. Otherwise, if $\mathcal{A}$ returns a tuple $(i_0, j_0, i_1, j_1, L, R, s, \mathbf{a})$, where $L = \{X_1, \ldots, X_n\}$ and $\mathbf{a} = (a_1, \ldots, a_n)$, $\mathcal{B}$ runs $\mathcal{A}$ again with the same random coins on input

$$\begin{aligned} &X^*, U_1, \ldots, U_{j_1}, U_{j_1+1}', \ldots, U_{q_s}', \\ &h_{0,1}, \ldots, h_{0,q}, \\ &h_{1,1}, \ldots, h_{1,i_1-1}, h_{1,i_1}', \ldots, h_{1,q}', \end{aligned}$$

where $h_{1,i_1}', \ldots, h_{1,q}'$ are uniformly random. We will see shortly that the first j_1 DL oracle queries made by $\mathcal{A}$ in this second execution are the same as in the first one, so that $\mathcal{B}$ only relays the $q_s - j_1$ last DL oracle queries made by $\mathcal{A}$ to its own DL oracle. If $\mathcal{A}$ returns $\perp$ in this second run, $\mathcal{B}$ returns $\perp$ as well. If $\mathcal{A}$ returns another tuple $(i_0', j_0', i_1', j_1', L', R', s', \mathbf{a}')$, where $L' = \{X_1', \ldots, X_{n'}'\}$ and $\mathbf{a}' = (a_1', \ldots, a_{n'}')$, $\mathcal{B}$ 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 aggregated public keys corresponding to the two forgeries. If $i_1 \neq i'_1$, or $i_1 = i'_1$ and $h_{1,i_1} = h'_{1,i_1}$, then $\mathcal{B}$ returns $\perp$. Otherwise, if $i_1 = i'_1$ and $h_{1,i_1} \neq h'_{1,i_1}$, we will prove shortly that necessarily

$$i_0 = i'_0,\ j_0 = j'_0,\ L = L',\ R = R',\ \text{and } \mathbf{a} = \mathbf{a}', \tag{3}$$

which implies in particular that $\widetilde{X} = \widetilde{X}'$. By Lemma 2, the two outputs returned by $\mathcal{A}$ are such that

$$g^s = R\widetilde{X}^{h_{1,i_1}} \quad \text{and} \quad g^{s'} = R'(\widetilde{X}')^{h'_{1,i_1}} = R\widetilde{X}^{h'_{1,i_1}},$$

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

$$\tilde{x} = (s - s')(h_{1,i_1} - h'_{1,i_1})^{-1} \bmod p.$$

Then $\mathcal{B}$ returns $(i_0, j_0, L, \mathbf{a}, \tilde{x})$.

It is easy to see that $\mathcal{B}$ returns a non-$\perp$ output *iff* $\mathsf{Fork}^{\mathcal{A}}$ does, so that by Lemma 1 and Lemma 2, $\mathcal{B}$'s accepting probability is at least

$$\begin{aligned} \mathsf{acc}(\mathcal{A}) \left(\frac{\mathsf{acc}(\mathcal{A})}{q} - \frac{1}{2^\ell} \right) &= \frac{(\varepsilon - 2q^2/2^\ell)^2}{q} - \frac{\varepsilon - 2q^2/2^\ell}{2^\ell} \\ &\geq \frac{\varepsilon^2}{q_h + q_s + 1} - \frac{4(q_h + q_s + 1) + 1}{2^\ell}. \end{aligned}$$

It remains to prove the equalities of Equation (3). In $\mathcal{A}$'s first execution, h_{1,i_1} is assigned to $T_1(\widetilde{X}, R, m)$, while is $\mathcal{A}$'s second execution, h'_{1,i_1} is assigned to $T_1(\widetilde{X}', R', m')$. Note that these two assignments can happen either because of a direct query to H_1 by the forger, during a signature query, or during the final verification of the validity of the forgery. Up to these two assignments, the two executions are identical since $\mathcal{A}$ runs $\mathcal{F}$ on the same random coins and input, uses the same values $h_{0,1}, \ldots, h_{0,q}$ for $T_0(\cdot, X^*)$ assignments, the same random values for $T_0(\cdot, X \neq X^*)$ assignments, the same values $h_{1,1}, \ldots, h_{1,i_1-1}$ for T_1 assignments, and the same shares $U_1, \ldots, U_{j_1}$ for signature queries. This implies in particular that the first j_1 DL oracle queries made by $\mathcal{A}$ in this second execution are the same as in the first one, as claimed above. Since both executions are identical up to the two assignments $T_1(\widetilde{X}, R, m) := h_{1,i_1}$ and $T_1(\widetilde{X}', R', m') := h'_{1,i_1}$, 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 the bad event AKColl happened in both executions, a contradiction since $\mathcal{A}$ returns a non-$\perp$ output in both executions. Hence, $L = L'$. Since in both executions of $\mathcal{A}$, the bad event $\mathsf{BadOrder}$ does not happen, assignments $T_0(L, X^*) := h_{0,i_0}$ and $T_0(L', X^*) := h_{0,i'_0}$ necessarily happened *before* the fork. This implies that $i_0 = i'_0$, $j_0 = j'_0$, and $\mathbf{a} = \mathbf{a}'$. □

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

Theorem. *Assume that there exists a* $(t, q_s, q_h, N, \varepsilon)$*-forger* $\mathcal{F}$ *against the multi-signature scheme* MuSig *with group parameters* $(\mathbb{G}, p, g)$ *and hash functions* $H_0, H_1 : \{0,1\}^* \to \{0,1\}^\ell$ *modeled as random oracles. Then there exists an algorithm* $\mathcal{C}$ *which* $(4q_s, t', \varepsilon')$*-solves the OMDL problem for* $(\mathbb{G}, p, g)$*, with* $t' = 4t + 4Nt_{\text{exp}} + O(N(q_h + q_s + 1))$ *where* t_{exp} *is the time of an exponentiation in* $\mathbb{G}$ *and*

$$\varepsilon' \geq \frac{\varepsilon^4}{(q_h + q_s + 1)^3} - \frac{11}{2^\ell}.$$

In other words, $\mathcal{C}$ *takes as input* $4q_s + 1$ *uniformly random group elements*

$$X^*, U_1, U_1', U_1'', U_1''', \ldots, U_{q_s}, U_{q_s}', U_{q_s}'', U_{q_s}''',$$

makes at most $4q_s$ *queries to a discrete logarithm oracle* $\mathsf{DL}_g(\cdot)$*, and returns the discrete logarithm of all its challenges in time at most* t' *and with probability at least* ε'*.*

Proof. In the following we let $q = q_h + q_s + 1$. Algorithm $\mathcal{C}$ runs $\mathsf{Fork}^{\mathcal{B}}$ with $\mathcal{B}$ as defined in Lemma 3 and takes a few additional steps described below. The mapping with notation of Lemma 1 is as follows:

- X^* plays the role of inp,
- $h_{0,1}, \ldots, h_{0,q}$ play the role of $h_1, \ldots, h_q$,
- $(U_1, U_1'), (U_1'', U_1'''), \ldots, (U_{q_s}, U_{q_s}'), (U_{q_s}'', U_{q_s}''')$ play the role of $V_1, V_1', \ldots, V_m$, V_m',
- (i_0, j_0) play the role of (i, j),
- $(L, \mathbf{a}, \tilde{x})$ play the role of out.

In more details, $\mathcal{C}$ picks random coins ρ_B and runs algorithm $\mathcal{B}$ on coins ρ_B, group elements $X^*, U_1, U_1', \ldots, U_{q_s}, U_{q_s}'$, and uniformly random ℓ-bit strings $h_{0,1}, \ldots, h_{0,q}$. It relays all DL oracle queries made by $\mathcal{B}$ to its own DL oracle. If $\mathcal{B}$ returns $\perp$, $\mathcal{C}$ returns $\perp$ as well. Otherwise, if $\mathcal{B}$ returns a tuple $(i_0, j_0, L, \mathbf{a}, \tilde{x})$, $\mathcal{C}$ runs $\mathcal{B}$ again with the same random coins ρ_B on input

$$X^*, U_1, U_1', \ldots, U_{j_0}, U_{j_0}', U_{j_0+1}'', U_{j_0+1}''', \ldots, U_{q_s}'', U_{q_s}'''$$

and

$$h_{0,1}, \ldots, h_{0,i_0-1}, h_{0,i_0}', \ldots, h_{0,q}',$$

where $h_{0,i_0}', \ldots, h_{0,q}'$ are uniformly random. We will see shortly that the first j_0 DL oracle queries made by $\mathcal{B}$ in this second execution are the same as in the first one, so that $\mathcal{C}$ only relays the DL oracle queries made by $\mathcal{B}$ to its own DL oracle starting from the $j_0 + 1$-th one. If $\mathcal{B}$ returns $\perp$ in this second run, $\mathcal{C}$ returns $\perp$ as well. If $\mathcal{B}$ returns another tuple $(i_0', j_0', L', \mathbf{a}', \tilde{x}')$, $\mathcal{B}$ proceeds as follows. Let $L = \{X_1, \ldots, X_n\}$, $\mathbf{a} = (a_1, \ldots, a_n)$, and $\mathbf{a}' = (a_1', \ldots, a_n')$. Let also n^* be the number of times X^* appears in L. If $i_0 \neq i_0'$, or $i_0 = i_0'$ and $h_{0,i_0} = h_{0,i_0}'$, $\mathcal{B}$ returns $\perp$. Otherwise, if $i_0 = i_0'$ and $h_{0,i_0} \neq h_{0,i_0}'$, we will prove shortly that necessarily

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

By Lemma 3, we have that

$$g^{\tilde{x}} = \prod_{i=1}^{n} X_i^{a_i} = (X^*)^{n^* h_{0,i_0}} \prod_{\substack{i \in \{1,\dots,n\} \\ X_i \neq X^*}} X_i^{a_i},$$

$$g^{\tilde{x}'} = \prod_{i=1}^{n} X_i^{a'_i} = (X^*)^{n^* h'_{0,i_0}} \prod_{\substack{i \in \{1,\dots,n\} \\ X_i \neq X^*}} X_i^{a_i}.$$

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

$$x^* = (\tilde{x} - \tilde{x}')(n^*)^{-1}(h_{0,i_0} - h'_{0,i_0})^{-1} \bmod p.$$

Once x^* has been computed, $\mathcal{C}$ can deduce the discrete logarithm of all challenges which were used in signature queries as follows: if $U_i^{(j)}$ was assigned to R_1 in a signature query and the DL oracle was queried on $R_1(X^*)^{a_1 c}$, returning s_1, then the discrete logarithm of $U_i^{(j)}$ is $s_1 - a_1 c x^* \bmod p$. Each challenge was used for at most one signature query,[15] hence $\mathcal{C}$ can retrieve the discrete logarithm of challenges which have not been used by directly querying the DL oracle: the total number of DL oracle queries is then exactly $4q_s$.

Neglecting the time needed to compute discrete logarithms, the running time t' of $\mathcal{C}$ is twice the running time of $\mathcal{B}$, which itself is twice the running time of $\mathcal{A}$. The running time of $\mathcal{A}$ is the running of $\mathcal{F}$ plus the time needed to maintain tables T_0 and T_1 (we assume each assignment takes unit time) and answer signature queries. The size of T_0, resp. T_1 is always at most qN, resp. q, and answering signature queries is dominated by the time needed to compute the aggregated key which is at most Nt_{exp}. Therefore, $t' = 4t + 4Nt_{\text{exp}} + O(N(q_h + q_s + 1))$.

Clearly, $\mathcal{C}$ is successful *iff* $\mathsf{Fork}^{\mathcal{B}}$ returns a non-$\perp$ answer. By Lemma 1 and Lemma 3, the success probability of $\mathcal{C}$ is at least

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

It remains to prove the equalities of Equation (4). In the two executions of $\mathcal{A}$ run within the first execution of $\mathcal{B}$, h_{0,i_0} is assigned to $T_0(L, X^*)$, while in the two executions of $\mathcal{A}$ run within the second execution of $\mathcal{B}$, h'_{0,i_0} is assigned to $T_0(L', X^*)$. Note that these two assignments can happen either because of a direct query $H_0(L, X)$ made by the forger for some key $X \in L$ (not necessarily X^*), during a signature query, or during the final verification of the validity of

[15] Note that this is why we need to use fresh values $U_i^{(j)}$ after each fork.

the forgery. Up to these two assignments, the four executions of $\mathcal{F}$ are identical since $\mathcal{A}$ runs $\mathcal{F}$ on the same random coins and the same input, uses the same values $h_{0,1}, \dots, h_{0,i_0-1}$ for $T_0(\cdot, X^*)$ assignments, the same random values for $T_0(\cdot, X \neq X^*)$ assignments, the same values $h_{1,1}, \dots, h_{1,q}$ for T_1 assignments, and the same shares $U_1, \dots, U_{j_0}$ for signature queries (the last two claims following from the fact that in the four executions of $\mathcal{A}$, bad event $\mathsf{BadOrder}$ does not happen). This implies in particular that the first j_0 DL oracle queries made by $\mathcal{B}$ in the second execution are the same as in the first execution, as claimed above. Since the four executions of $\mathcal{A}$ are identical up to the assignments $T_0(L, X^*) := h_{0,i_0}$ and $T_0(L', X^*) := h'_{0,i_0}$, the arguments of these two assignments must be the same, which implies that $L = L'$. Besides, all values $T_0(L, X)$ for $X \in L \setminus \{X^*\}$ are chosen uniformly at random by $\mathcal{A}$ using the same coins in the four executions, which implies that $a_i = a'_i$ for each i such that $X_i \neq X^*$. □

4.3 Discussion

We conclude this section with a number of remarks.

Optimized Signatures. It is customary to output (c, s) rather than (R, s) as the signature of message m for multiset of public keys L with aggregated public key $\widetilde{X}$, where $c = H_1(\widetilde{X}, R, m)$. This makes the signature shorter when working in subgroups of prime finite fields, where group elements are integers of 2048 bits or more, whereas H_1 outputs are typically 256-bit long. (In elliptic curve groups, where group elements representation is more compact, this optimization is not as interesting.) It is straightforward to verify that our security proof applies *mutatis mutandis* to this variant as well (the simulation of signature queries by $\mathcal{A}$ is unaffected, only the final verification of the forgery by $\mathcal{A}$ must be slightly adapted but this does not modify Lemma 2).

Hashing L. For efficiency reasons, it might be interesting to replace L by a hash $H'_0(L)$ when computing coefficients a_i in Equation (1). This does not affect the security of the scheme but the security proof becomes slightly more complex. Informally, any query $H_0(h, X)$ where h is distinct from all previous answers to H'_0 queries is useless to the adversary unless a subsequent H'_0 queries returns h, which happens only with negligible probability. Hence, we can modify $\mathcal{A}$ as follows: at the moment the forger makes a query $H_0(h, X)$, $\mathcal{A}$ checks whether a previous H'_0 queries was answered with h; if not, it answers randomly; otherwise, $\mathcal{A}$ looks for the multiset L such that $H'_0(L)$ was queried and answered with h (in the unlikely case where there is more than one such multiset we let $\mathcal{A}$ return $\perp$) and behaves as described in Lemma 2 on query $H_0(L, X)$.

Modified Verification Algorithm. For efficiency and privacy, we can replace the verification algorithm with the one that takes as input $\widetilde{X}$ rather than computing it from L (in other words, with the verification algorithm for the standard signature scheme). This also better matches key aggregation use cases where the signers agree beforehand on L before publishing $\widetilde{X}$ as their "joint public key". At first sight, this seems to impact how a successful forgery is defined, which in turns impacts the security model and the security proof. However, note that we cannot simply allow the attacker to freely choose the public key for which it returns a forgery (this would imply trivial wins where the attacker picks a secret key and returns a signature for the corresponding public key). Hence, in this case, we still require the adversary to output, along with a message m and a signature σ, the multiset of public keys L (containing the challenge public key X^*) for which the forgery is intended. The forgery is considered valid if the modified verification algorithm returns 1 on input m, σ, and the aggregated public key $\widetilde{X}$ corresponding to L, and if the attacker never initiated the signature protocol for L and m. Hence, we are brought back to the original security model and the security proof applies.

Using the Same Key for Normal Signatures and Multi-Signatures. Signers might be tempted to use their key both for issuing normal signatures and participating to multi-signatures. Signing "normally" under public key X is not equivalent to creating a multi-signature under the singleton key set $L = \{X\}$. Hence, this situation is not captured by our security model where only the latter is allowed.

In order to take this possibility into account, we must augment the security game with an additional oracle and an extra winning condition: first, the adversary is granted access to a special signature oracle taking as input a message m and returning a standard signature for key X^*; second, the adversary is considered successful if it returns a valid forgery under key X^*. It is easy to adapt the security proof to this modified security model: the special signature oracle can be simulated using the standard strategy for simulating Schnorr signatures since $\mathcal{A}$ gets to select R randomly in this case (and moreover, this does not modify the probability of events $\mathsf{BadOrder}$ and AKColl), and if the adversary returns a forgery under key X^*, the standard way of extracting the discrete logarithm of X^* with a single application of the Forking Lemma can be applied.

5 Applications to Bitcoin

5.1 Introduction

Deployed in 2009, Bitcoin [Nak08] is a digital currency with no trusted issuer or transaction processor, which works by means of a publicly verifiable distributed ledger, called the blockchain.[16] It contains every transaction since the system's inception, resulting in a final state, the set of unspent coins (also called the UTXO[17] set). Each unspent coin has an associated value (expressed as a multiple of the currency unit, 10^{-8} bitcoin) and a *programmable* public key of the owner.

[16] While temporary disagreement between nodes is possible about which chain is to be accepted, we use *the* blockchain to refer to the chain that an individual node currently considers its best one.

[17] UTXO is an abbreviation of Unspent Transaction (TX) Output.

Every transaction consumes one or more coins, providing a signature for each to authorize its spending, and creates one or more new coins, with a total value not larger than the value of the consumed coins.

Programmable Public Keys and Signatures. Bitcoin uses a programmable generalization of a digital signature scheme. Instead of a public key, a predicate that determines spendability is included in every output (implemented in a concise programming language, called *Bitcoin Script*). When spending, instead of a signature, a witness that satisfies the predicate is provided. In practice, most output predicates effectively[18] correspond to a single ECDSA verification.

This is also how Bitcoin supports a naive version of multi-signatures with a threshold policy[19]: coins can be assigned a predicate that requires valid signatures for multiple public keys. Several use cases for this exist, including low-trust escrow services [GBGN17] and split-device security. While using the predicate language to implement multi-signatures is very flexible, it is inefficient in terms of size, computational cost, and privacy.

Challenges. As a global consensus system, kept in check by the ability for every participant to validate all updates to the ledger, the size of signatures and predicates[20], and the computational cost for verifying them are the primary limiting factors for its scalability. The computational requirements for signing, or the communication overhead between different signers are far less constrained. Bitcoin does not have any central trusted party, so it is not generally possible to introduce new cryptographic schemes that require a trusted setup. Finally, to function as a currency, a high degree of fungibility and privacy is desirable. Among other things, this means that ideally the predicates of coins do not leak information about the owner. In particular, if several styles of predicates are in use, the choice may reveal what software or service is being used to manage it.

5.2 Native Multi-Signature Support

An obvious improvement is to replace the need for implementing n-of-n multi-signatures in an ad-hoc fashion with a constant-size multi-signature primitive like Bellare-Neven. While this is on itself an improvement in terms of size, it still requires the predicate itself — whose size also matters — to contain all of the signers' public keys. Key aggregation improves upon this further, as a single-key predicate can be used instead which is both smaller and has lower computational cost for verification. It also improves privacy, as the participant keys and their count remain private to the signers.

[18] Specifically, a function is used that takes a public key and a signature, and requires that the hash of the public key is a fixed constant and that the signature verifies with that key.

[19] Note that the term “multisig” in the context of Bitcoin is used to refer to any spending policy that requires signatures with m out of n public keys.

[20] The size of predicates is even more important, as they are part of the UTXO set that is maintained by every node.

When generalizing to the m-of-n scenario, several options exist. One is to forego key aggregation, and still include all potential signer keys in the predicates while still only producing a single signature for the chosen combination of keys. Alternatively, a Merkle tree [Mer87] where the leaves are permitted combinations of keys (in aggregated form) can be employed. The predicate in this case would take as input an aggregated public key, a signature with it, and a proof. Its validity would depend on the signature being valid with the provided key, and the proof establishing that the key is in fact one of the leaves of the Merkle tree, identified by its root hash. This approach is very generic, as it works for any subset of combinations of keys, and as a result has very good privacy as the exact policy is not visible from the proof. It is only feasible however when the total number of combinations is tractable.

Alternatives. Some key aggregation schemes that do not protect against rogue-key attacks can be used instead in the above cases, under the assumption that the sender is given a proof of knowledge/possession for the receivers' private keys. However, these schemes are difficult to prove secure except by using very large proofs of knowledge [MOR01, Problem 5]. As those proofs of knowledge/possession do not need to be seen by verifiers — they are effectively certified by the sender's signature on the transaction which includes the predicate — they do not burden validation. However, passing them around to senders is inconvenient, and easy to get wrong. Using a scheme that is secure in the plain public-key model categorically avoids these concerns.

Another alternative is to use an algorithm whose key generation requires a trusted setup, for example in the KOSK model. While many of these schemes have been proven secure [Bol03, LOS$^+$06], they rely on mechanisms that are usually not implemented by certification authorities [BN06, RY07].

5.3 Cross-Input Multi-Signatures

The previous sections explained how the number of signatures per input can generally be reduced to one, but we would like to go further, and replace it with a single signature per transaction. Doing so requires a fundamental change in validation semantics, as the validity of separate inputs is no longer independent. As a result, the outputs can no longer be modeled as predicates. Instead, we model them as functions that return a boolean plus a set of zero or more public keys. Overall validity requires all returned booleans to be True and a multi-signature of the transaction with L the union of all returned keys.

More concretely, this can be implemented by providing an alternative to the signature checking opcode `OP_CHECKSIG`[21] and related opcodes in the Script language. Instead of returning the result of an actual ECDSA verification, they always return True, but additionally add the public key with which the verification would have taken place to a transaction-wide multiset of keys. Finally, after all

[21] See https://bitcoin.org/en/developer-reference#term-op-checksig for more information.

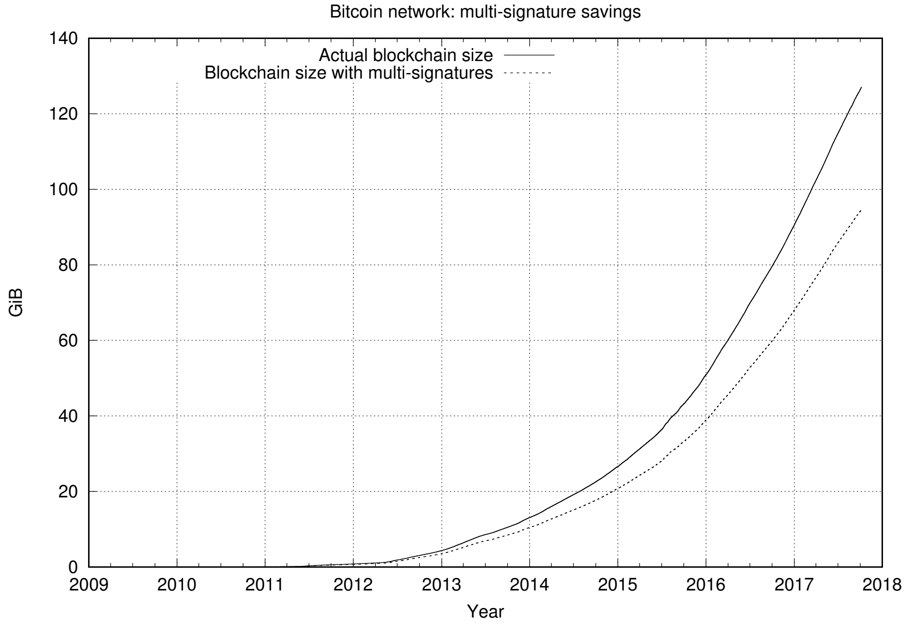

Fig. 3. Size of the Bitcoin blockchain with and without multi-signatures.

inputs are verified, a multi-signature present in the transaction is verified against that multiset. In case the transaction spends inputs from multiple owners, they will need to collaborate to produce the multi-signature, or choose to only use the original opcodes. Adding these new opcodes is possible in a backward-compatible way.

Protection against Rogue-Key Attacks. In the case of cross-input signatures, there is no published commitment to the set of signers, as each transaction input can independently spend an output that requires authorization from distinct participants. We do not wish to restrict this functionality, as it would interfere with fungibility improvements like CoinJoin [Max13]. Due to the lack of certification, security against rogue-key attacks is essential here.

Assume transactions used a single multi-signature that was vulnerable to rogue-key attacks, like the simpler schemes described in Section 3.2. An attacker could identify an arbitrary number of outputs he wants to steal, with public keys $X_1, \dots, X_{n-t}$, and then use the rogue-key attack to determine $X_{n-t+1}, \dots, X_n$ such that he can sign for the aggregated key $\widetilde{X}$. He would then send a small amount of his own money to outputs with predicates corresponding to the keys $X_{n-t+1}, \dots, X_n$. Finally, he can create a transaction that spends all of the victim coins together with the ones he just created by forging a multi-signature for the whole transaction.

We observe that in the case of multi-signatures across inputs, theft can occur by merely being able to forge a signature over a set of keys that includes at least one key not controlled by the attacker — exactly what the plain public key model considers a win for the attacker. This is in contrast to the single-input multi-signature case, where theft is only possible by forging a signature for the exact (aggregated) keys contained in an existing output. As a result, it is no longer possible to rely on proofs of knowledge/possession that are private to the signers.

Space Savings. To analyze the impact of multi-signatures, a simulation on Bitcoin's historical blockchain was performed to determine the potential space savings. Figure 3 shows the cumulative blockchain size together with what the size would be if all transactions' signatures were replaced with just one per transaction, giving an indication of what could have been saved if MuSig had been used since the beginning. Note that this only encompasses the savings from using multi-signatures, and does not include the savings that are possible from key aggregation.

Acknowledgments

We are grateful to Russell O'Connor for sharing with us the attack against the generic way to turn a multi-signature scheme into a IAS scheme described in Appendix A.2.

References

[And11] Gavin Andresen. M-of-N Standard Transactions. Bitcoin Improvement Proposal, 2011. See https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki.

[ANS05] Accredited Standards Committee X9. *American National Standard X9.62-2005, Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)*, 2005.

[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 Conference on Computer and Communications Security - CCS 2008*, pages 449–458. ACM, 2008.

[BDL$^+$11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-Speed High-Security Signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011*, volume 6917 of *LNCS*, pages 124–142. Springer, 2011.

[Ber15] Daniel J. Bernstein. Multi-user Schnorr security, revisited. IACR Cryptology ePrint Archive, Report 2015/996, 2015. Available at http://eprint.iacr.org/2015/996.

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In Eli Biham, editor, *Advances in Cryptology - EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 416–432. Springer, 2003.

[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 Conference on Computer and Communications Security - CCS 2006*, pages 390–399. ACM, 2006.

[BNN07] Mihir Bellare, Chanathip Namprempre, and Gregory Neven. Unrestricted Aggregate Signatures. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming - ICALP 2007*, volume 4596 of *LNCS*, pages 411–422. Springer, 2007.

[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. *J. Cryptology*, 16(3):185–215, 2003.

[Bol03] Alexandra Boldyreva. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In Yvo Desmedt, editor, *Public Key Cryptography - PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, 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, *Advances in Cryptology - CRYPTO 2002*, volume 2442 of *LNCS*, pages 162–177. Springer, 2002.

[GBGN17] Steven Goldfeder, Joseph Bonneau, Rosario Gennaro, and Arvind Narayanan. Escrow protocols for cryptocurrencies: How to buy physical goods using Bitcoin. In *Financial Cryptography and Data Security - FC 2017*, 2017. Available at http://www.jbonneau.com/doc/GBGN17-FC-physical_escrow.pdf.

[GBL08] Sanjam Garg, Raghav Bhaskar, and Satyanarayana V. Lokam. Improved Bounds on Security Reductions for Discrete Log Based Signatures. In David Wagner, editor, *Advances in Cryptology - CRYPTO 2008*, volume 5157 of *LNCS*, pages 93–107. Springer, 2008.

[GGN16] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *Applied Cryptography and Network Security - ACNS 2016*, volume 9696 of *LNCS*, pages 156–174. Springer, 2016.

[Har94] Lein Harn. Group-oriented (t, n) threshold digital signature scheme and digital multisignature. *IEE Proceedings - Computers and Digital Techniques*, 141(5):307–313, 1994.

[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.

[KMP16] Eike Kiltz, Daniel Masny, and Jiaxin Pan. Optimal Security Proofs for Signatures from Identification Schemes. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 (Proceedings, Part II)*, volume 9815 of *LNCS*, pages 33–61. Springer, 2016.

[Lan96] Susan K. Langford. Weakness in Some Threshold Cryptosystems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96*, volume 1109 of *LNCS*, pages 74–82. Springer, 1996.

[LHL94] Chuan-Ming Li, Tzonelih Hwang, and Narn-Yih Lee. Threshold-Multisignature Schemes where Suspected Forgery Implies Traceability of Adversarial Shareholders. In Alfredo De Santis, editor, *Advances in Cryptology - EUROCRYPT '94*, volume 950 of *LNCS*, pages 194–204. Springer, 1994.

[Lin17] Yehuda Lindell. Fast Secure Two-Party ECDSA Signing. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 (Proceedings, Part II)*, volume 10402 of *LNCS*, pages 613–644. Springer, 2017.

[LMRS04] Anna Lysyanskaya, Silvio Micali, Leonid Reyzin, and Hovav Shacham. Sequential Aggregate Signatures from Trapdoor Permutations. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 74–90. Springer, 2004.

[LOS$^+$06] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential Aggregate Signatures and Multisignatures Without Random Oracles. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 465–485. Springer, 2006.

[Max13] Gregory Maxwell. CoinJoin: Bitcoin privacy for the real world, 2013. BitcoinTalk post, https://bitcointalk.org/index.php?topic=279249.0.

[Mer87] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87*, volume 293 of *LNCS*, pages 369–378. Springer, 1987.

[MH96] Markus Michels and Patrick Horster. On the Risk of Disruption in Several Multiparty Signature Schemes. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology - ASIACRYPT '96*, volume 1163 of *LNCS*, pages 334–345. Springer, 1996.

[MOR01] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-Subgroup Multisignatures. In Michael K. Reiter and Pierangela Samarati, editors, *ACM Conference on Computer and Communications Security - CCS 2001*, pages 245–254. ACM, 2001.

[MR01] Philip D. MacKenzie and Michael K. Reiter. Two-Party Generation of DSA Signatures. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *LNCS*, pages 137–154. Springer, 2001.

[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 Cryptography*, 54(2):121–133, 2010.

[Nak08] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. Available at http://bitcoin.org/bitcoin.pdf.

[NIS13] National Institute of Standards and Technology. *FIPS 186-4: Digital Signature Standard (DSS)*, 2013. Available at http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[Oka92] Tatsuaki Okamoto. Provably Secure and Practical Identification Schemes and Corresponding Signature Schemes. In Ernest F. Brickell, editor, *Advances in Cryptology - CRYPTO '92*, volume 740 of *LNCS*, pages 31–53. Springer, 1992.

[OO91] Kazuo Ohta and Tatsuaki Okamoto. A Digital Multisignature Scheme Based on the Fiat-Shamir Scheme. In Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto, editors, *Advances in Cryptology - ASIACRYPT '91*, volume 739 of *LNCS*, pages 139–148. Springer, 1991.

[OO99] Kazuo Ohta and Tatsuaki Okamoto. Multi-Signature Schemes Secure against Active Insider Attacks. *IEICE Transactions on Fundamentals* *of Electronics, Communications and Computer Science*, E82-A(1):21–31, 1999.

[Por13] Thomas Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979, 2013. Available at https://rfc-editor.org/rfc/rfc6979.txt.

[PS00] David Pointcheval and Jacques Stern. Security Arguments for Digital Signatures and Blind Signatures. *J. Cryptology*, 13(3):361–396, 2000.

[PV05] Pascal Paillier and Damien Vergnaud. Discrete-Log-Based Signatures May Not Be Equivalent to Discrete Log. In Bimal K. Roy, editor, *Advances in Cryptology - ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 1–20. Springer, 2005.

[RY07] Thomas Ristenpart and Scott Yilek. The Power of Proofs-of-Possession: Securing Multiparty Signatures against Rogue-Key Attacks. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 228–245. Springer, 2007.

[Sch91] Claus-Peter Schnorr. Efficient Signature Generation by Smart Cards. *J. Cryptology*, 4(3):161–174, 1991.

[SEC10] Certicom Research. *SEC 2: Recommended Elliptic Curve Domain Parameters, v2.0*, 2010. Available at http://www.secg.org/sec2-v2.pdf.

[Seu12] Yannick Seurin. On the Exact Security of Schnorr-Type Signatures in the Random Oracle Model. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 554–571. Springer, 2012.

[Wag02] David A. Wagner. A Generalized Birthday Problem. In Moti Yung, editor, *Advances in Cryptology - CRYPTO 2002*, volume 2442 of *LNCS*, pages 288–303. Springer, 2002.

A Interactive Aggregate Signatures

Interactive Aggregate Signature (IAS) schemes extend multi-signature schemes by allowing each signer to sign a different message. Bellare and Neven [BN06] suggested a generic way to transform any multi-signature scheme into an IAS scheme. Here, we take a closer look at this transformation and show that it subtly fails to provide the strongest security guarantees in the plain public key model.

A.1 Syntax and Security Model

The syntax is adapted as follows. An IAS scheme Π is a triple of algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver})$. The randomized key generation algorithm takes no input and returns a private/public key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow_{\$} \mathsf{KeyGen}()$. The signature algorithm Sign is run by each participant on input its key pair $(\mathsf{sk}, \mathsf{pk})$, a message m, and a set of public key/message pairs for other cosigners $S' = \{(\mathsf{pk}'_1, m'_1), \ldots, (\mathsf{pk}'_{n-1}, m'_{n-1})\}$ such that $(\mathsf{pk}, m) \notin S'$; it returns a signature σ for the set of public key/message pairs $S = S' \cup \{(\mathsf{pk}, m)\}$.[22] The deterministic verification algorithm Ver takes as input a set of public key/message pairs

[22] S is a set, i.e., no public key/message pair repeats, even though the same public key might appear multiple times. This is without loss of generality since any repeated public key/message pair in S can be deleted by the signature/verification algorithm.

$S = \{(\mathsf{pk}_1, m_1), \ldots, (\mathsf{pk}_n, m_n)\}$ and a signature σ, and returns 1 if the signature is valid for S and 0 otherwise.

Alternatively, one can specify the signature algorithm to take as input a key pair $(\mathsf{sk}, \mathsf{pk})$, an ordered set of public key/message pairs $S = \{(\mathsf{pk}_1, m_1), \ldots, (\mathsf{pk}_n, m_n)\}$ such that the public key pk of the signer appears at least once in S, and a *message index* i such that $\mathsf{pk}_i = \mathsf{pk}$ specifying which message m_i is "really" signed by the signer. We require the set S to be ordered in this case mainly because it makes the definition of the message index i simpler. If S is given as input to the signing algorithm as an unordered set, one can still define the message index with respect to some specified ordering, e.g. the lexicographical one. Note that the message index is necessary to make both definitions equivalent since in the plain public key model, an attacker can copy the public key of an honest signer, in which case there is no way to define which message is intended to be signed by the honest signer from S alone.

The security model for an IAS scheme is very similar to the one of a multi-signature scheme. Formally, the security game involving an adversary (forger) $\mathcal{F}$ proceeds as follows:

- A key pair for the honest signer $(\mathsf{sk}^*, \mathsf{pk}^*) \leftarrow_{\$} \mathsf{KeyGen}()$ is generated at random and the public key pk^* is given as input to $\mathcal{F}$.
- The forger can engage arbitrary signing protocols with the honest user. Formally, it has access to a signature oracle which must be called on input an ordered set of public key/message pairs $S = \{(\mathsf{pk}_1, m_1), \ldots, (\mathsf{pk}_n, m_n)\}$ where pk^* occurs at least once and a message index i such that $\mathsf{pk}_i = \mathsf{pk}^*$. The oracle implements an execution of the signing algorithm on input $(\mathsf{sk}^*, \mathsf{pk}^*)$, S, and message index i while the forger plays the role of all other signers in S (potentially deviating from the protocol). Note that pk^* might appear multiple times in S, in which case the forger plays the role of all instances of pk^* except the one corresponding to message index i. The forger can interact concurrently with as many independent instances of the honest signature oracle as it wishes.
- At the end of its execution, the forger returns a set of public key/message pairs $S = \{(\mathsf{pk}_1, m_1), \ldots, (\mathsf{pk}_n, m_n)\}$ and a signature σ. The forger wins if the forgery is valid, i.e., $\mathsf{Ver}(S, \sigma) = 1$, and there exists $i \in \{1, \ldots, n\}$ such that $\mathsf{pk}_i = \mathsf{pk}^*$ and $\mathcal{F}$ never initiated a signature protocol with the oracle on input S and message index i.

A.2 BN's Generic Conversion Method

Bellare and Neven [BN06] suggested a generic way to transform a multi-signature scheme Π into an IAS scheme Π' which simply consists of using the tuple of all public key/message pairs as message in the multi-signature scheme. More precisely, using the second syntactic definition of an IAS signature algorithm, on input $(\mathsf{sk}, \mathsf{pk})$, $S = \{(\mathsf{pk}_1, m_1), \ldots, (\mathsf{pk}_n, m_n)\}$, and message index i, the algorithm $\Pi'.\mathsf{Sign}$ runs $\Pi.\mathsf{Sign}$ on input $(\mathsf{sk}, \mathsf{pk})$, $L = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$, and $m = \langle S \rangle$ where $\langle S \rangle$ is some unique encoding of S (e.g. lexicographical ordering of the public key/message pairs) and outputs the signature σ returned by $\Pi.\mathsf{Sign}$. Note that the execution of the signature algorithm $\Pi'.\mathsf{Sign}((\mathsf{sk}, \mathsf{pk}), S, i)$ is independent of the message index i, which allows the following attack.[23]

Consider an adversary proceeding as follows against an honest signer with key pair $(\mathsf{sk}^*, \mathsf{pk}^*)$. The adversary chooses two distinct messages m_1 and m_2, and queries the signature oracle on input $S = \{(\mathsf{pk}^*, m_1), (\mathsf{pk}^*, m_2)\}$ with message index 1 (i.e., the honest signer intends to sign m_1). Note that the alleged cosigner would run the signing algorithm on input $(\mathsf{sk}^*, \mathsf{pk}^*)$, $S = \{(\mathsf{pk}^*, m_1), (\mathsf{pk}^*, m_2)\}$, and message index 2. If the execution of the signature algorithm is independent from the message index, the adversary can perfectly emulate the cosigner by simply copying all messages received from the signature oracle. Hence, the adversary can compute a valid signature for $S = \{(\mathsf{pk}^*, m_1), (\mathsf{pk}^*, m_2)\}$. This is a valid forgery since the signature oracle was never called on S with message index 2.

For concreteness, we detail the attack against the IAS scheme obtained from the BN multi-signature scheme with the above generic conversion. Let (x^*, X^*) be the key pair of the honest signer. On input $S = \{(X^*, m_1), (X^*, m_2)\}$ and message index 1, the signature oracle chooses $r_1 \leftarrow_{\$} \mathbb{Z}_p$, computes $R_1 = g^{r_1}$ and $t_1 = H'(R_1)$, and sends t_1 to the adversary. The adversary simply sends back t_1. The oracle then sends R_1, and the adversary similarly answers with R_1. Finally, the oracle computes $R = (R_1)^2$, $c_1 = H(\langle L \rangle, X^*, R, \langle S \rangle)$ where $L = \{X^*, X^*\}$, and $s_1 = r_1 + c_1 x^* \bmod p$, and returns s_1 to the adversary. The adversary is now able to compute $s = 2s_1 \bmod p$ and the forgery $\sigma = (R, s)$ for S.

It is easy to see that BN generic conversion method turns a secure multi-signature scheme into an IAS scheme which is secure in a security model weaker than the one described in Section A.1, differing only in the winning condition, namely where the adversary $\mathcal{F}$ is successful if it returns a valid signature for a set $S = \{(\mathsf{pk}_1, m_1), \ldots, (\mathsf{pk}_n, m_n)\}$ such that *for any* $i \in \{1, \ldots, n\}$ such that $\mathsf{pk}_i = \mathsf{pk}^*$, $\mathcal{F}$ never initiated a signature protocol for S and message index i. Such a security model is sufficient if one makes the assumption that each signer checks other participants' public keys and aborts the protocol if his own public key has been copied (indeed, in that case, the winning conditions of the two security models become equivalent since any public key appears at most once in S). However, given how many vulnerabilities result from incorrect assumptions under which a cryptographic primitive can be used, an IAS scheme that remains secure without this assumption would be preferable.

A.3 Turning BN's Scheme into a Secure IAS

In order to transform the BN multi-signature scheme into an IAS scheme secure in the model of Section A.1, we propose the following simple fix which makes the execution of the signing algorithm dependent on the message index (unfortunately the transformation is not black-box anymore).

[23] This attack was communicated to us privately by Russell O'Connor.

Let $S = \{(X_1, m_1), \ldots, (X_n, m_n)\}$ be the ordered set of public key/message pairs of all participants, where $X_i = g^{x_i}$. In practice, if X is the public key of a specific signer and m the message he wants to sign, and $S' = \{(X'_1, m'_1), \ldots, (X'_{n-1}, m'_{n-1})\}$ is the set of the public key/message pairs of other signers, this specific signer merges (X, m) and S' into the ordered set $S = \{(X_1, m_1), \ldots, (X_n, m_n)\}$ and retrieves the resulting message index i such that $(X_i, m_i) = (X, m)$.

Then, as in the BN multi-signature scheme, each signer draws $r_i \leftarrow_\$ \mathbb{Z}_p$, computes $R_i = g^{r_i}$, sends commitment $t_i = H'(R_i)$ in a first round and then R_i in a second round, and computes $R = \prod_{i=1}^{n} R_i$. The signer with message index i then computes $c_i = H(R, \langle S \rangle, i)$ and $s_i = r_i + c_i x_i \bmod p$ and sends s_i to other signers. All signers can compute $s = \sum_{i=1}^{n} s_i \bmod p$. The signature is $\sigma = (R, s)$.

Given an ordered set $S = \{(X_1, m_1), \ldots, (X_n, m_n)\}$ and a signature $\sigma = (R, s)$, σ is valid for S *iff*

$$g^s = R \prod_{i=1}^{n} X_i^{H(R, \langle S \rangle, i)}.$$

Note that there is no need to include in the hash computation an encoding of the multiset $L = \{X_1, \ldots, X_n\}$ of public keys nor the public key X_i of the local signer since they are already "accounted for" through S and the message index i.

We leave a complete security analysis of this scheme for future work.

B Proof of Lemma 1

As in [BN06], we will need the following two lemmas which are consequences of Jensen's inequality.

Lemma 4. *Let Y be a real-valued random variable. Then* $\mathbf{E}[Y^2] \geq \mathbf{E}[Y]^2$.

Lemma 5. *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 1. 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,

$$\mathsf{frk}(\mathsf{inp}) \geq \mathsf{acc}(\mathsf{inp}) \left(\frac{\mathsf{acc}(\mathsf{inp})}{q} - \frac{1}{2^\ell} \right). \tag{5}$$

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

$$\begin{aligned}
\mathsf{frk} = \mathbf{E}[\mathsf{frk}(\mathsf{inp})] &\geq \mathbf{E}\left[\mathsf{acc}(\mathsf{inp})\left(\frac{\mathsf{acc}(\mathsf{inp})}{q} - \frac{1}{2^{\ell}}\right)\right] && (6)\\
&= \frac{\mathbf{E}[\mathsf{acc}(\mathsf{inp})^2]}{q} - \frac{\mathbf{E}[\mathsf{acc}(\mathsf{inp})]}{2^{\ell}} && (7)\\
&\geq \frac{\mathbf{E}[\mathsf{acc}(\mathsf{inp})]^2}{q} - \frac{\mathbf{E}[\mathsf{acc}(\mathsf{inp})]}{2^{\ell}} && (8)\\
&= \mathsf{acc}(\mathcal{A})\left(\frac{\mathsf{acc}(\mathcal{A})}{q} - \frac{1}{2^{\ell}}\right), && (9)
\end{aligned}$$

where we used Lemma 4 for Equation (8). It remains to prove Equation (5).

From now on, 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 $V_1, V_1', \ldots, V_m, V_m' \leftarrow_{\$} S$ 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

$$\begin{aligned}
\mathsf{frk}(\mathsf{inp}) &= \Pr\left[(I > 0) \wedge (I = I') \wedge (h_I \neq h_I')\right]\\
&\geq \Pr\left[(I > 0) \wedge (I = I')\right] - \Pr\left[(I > 0) \wedge (h_I = h_{I'})\right]\\
&= \Pr\left[(I > 0) \wedge (I = I')\right] - \frac{\Pr\left[I > 0\right]}{2^{\ell}}\\
&= \underbrace{\Pr\left[(I > 0) \wedge (I = I')\right]}_{\overset{\text{def}}{=} \mathsf{pr}} - \frac{\mathsf{acc}(\mathsf{inp})}{2^{\ell}}.
\end{aligned}$$

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 $Y_{i,j} : R \times (\{0,1\}^{\ell})^{i-1} \times S^j \to [0,1]$ as

$$Y_{i,j}(\rho, \mathbf{h}, \mathbf{V}) \overset{\text{def}}{=} \Pr\left[\left\{\begin{array}{l} h_i, \ldots, h_q \leftarrow_{\$} \{0,1\}^{\ell},\\ V_{j+1}, \ldots, V_m \leftarrow_{\$} S,\\ (I, J, \mathsf{out}) \leftarrow \mathcal{A}(\mathsf{inp}, h_1, \ldots, h_q, V_1, \ldots, V_m; \rho) \end{array}\right\} : I = i\right]$$

for all $\rho \in R$, $\mathbf{h} = (h_1, \ldots, h_{i-1}) \in (\{0,1\}^{\ell})^{i-1}$, and $\mathbf{V} = (V_1, \ldots, V_j) \in S^j$.

Then

$$\begin{aligned}
\mathsf{pr} &= \sum_{j=0}^{m}\sum_{i=1}^{q} \Pr\left[(I = i) \wedge (J = j) \wedge (I' = i)\right] && (10)\\
&= \sum_{j=0}^{m}\sum_{i=1}^{q} \Pr\left[J = j\right] \cdot \Pr\left[I = i | J = j\right] \cdot \Pr\left[I' = i | (I = i) \wedge (J = j)\right] && (11)
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{j=0}^{m} \sum_{i=1}^{q} \Pr\left[J=j\right] \sum_{\substack{\rho \\ \mathbf{h}=(h_1,\ldots,h_{i-1}) \\ \mathbf{V}=(V_1,\ldots,V_j)}} \frac{Y_{i,j}(\rho,\mathbf{h},\mathbf{V})^2}{|R| \cdot 2^{\ell(i-1)} \cdot |S|^j} && (12) \\
&= \sum_{j=0}^{m} \Pr\left[J=j\right] \sum_{i=1}^{q} \mathbf{E}[Y_{i,j}^2] && (13) \\
&\geq \sum_{j=0}^{m} \Pr\left[J=j\right] \sum_{i=1}^{q} \mathbf{E}[Y_{i,j}]^2 && (14) \\
&\geq \frac{1}{q} \sum_{j=0}^{m} \Pr\left[J=j\right] \left(\sum_{i=1}^{q} \mathbf{E}[Y_{i,j}] \right)^2 && (15) \\
&\geq \frac{1}{q} \left(\sum_{j=0}^{m} \Pr\left[J=j\right] \sum_{i=1}^{q} \mathbf{E}[Y_{i,j}] \right)^2 && (16) \\
&= \frac{\mathsf{acc}(\mathsf{inp})^2}{q}. && (17)
\end{aligned}
$$

Above we used Lemma 4 to derive Equation (14) and Equation (16) and Lemma 5 for each j with $y_i = \mathbf{E}[Y_{i,j}]$ to derive Equation (15).

□