Designs, Codes and Cryptography (2019) 87:2139–2164
https://doi.org/10.1007/s10623-019-00608-x

Simple Schnorr multi-signatures with applications to Bitcoin

Gregory Maxwell[1] · **Andrew Poelstra**[2] · **Yannick Seurin**[3] · **Pieter Wuille**[2]

Received: 31 May 2018 / Revised: 7 January 2019 / Accepted: 10 January 2019 / Published online: 2 February 2019

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) called MuSig, provably secure under the Discrete Logarithm assumption and 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). MuSig improves over the state-of-art scheme of Bellare and Neven (ACM Conference on Computer and Communications Security-CCS 2006) and its variants by Bagherzandi et al. (ACM Conference on Computer and Communications Security-CCS 2008) and Ma et al. (Des Codes Cryptogr 54(2):121–133, 2010) in two respects: (i) it is simple and efficient, having 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. To the best of our knowledge, this is the first multi-signature scheme provably secure under the Discrete Logarithm assumption in the plain public-key model which allows key aggregation. 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 · Key aggregation · Discrete logarithm problem · Forking lemma · Bitcoin

Communicated by S. D. Galbraith.

✉ Yannick Seurin
yannick.seurin@m4x.org

Gregory Maxwell
greg@xiph.org

Andrew Poelstra
apoelstra@blockstream.com

Pieter Wuille
pwuille@blockstream.com

1 San Francisco, USA

2 Blockstream, Mountain View, USA

3 ANSSI, Paris, France

Mathematics Subject Classification 94A60

1 Introduction

Multi-signatures Multi-signature protocols, first introduced by Itakura and Nakamura [22], 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 [20,21,24,25,33,38,39] until a formal model was put forward together with a provably secure scheme by Micali et al. [34] (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 [43]) 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 [13] and Lu et al. [27] rely on this assumption for their security. We refer to [9,43] 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) [9] and is based on the Schnorr signature scheme [44]. Following [9], 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 variant of BN's scheme, we recall it (as well as basic Schnorr signatures) first.

The Schnorr signature scheme [44] 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 [5]. This variant was argued to have a better multi-user security bound than the classic variant [4], but key-prefixing was later shown to be in fact unnecessary for Schnorr signatures to enjoy good multi-user security [23]. 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[1] 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 [21,24,33,34], 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 [34] solves this problem using a sophisticated interactive key generation protocol. In their scheme, Bellare and Neven [9] 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 the reduction 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 variant of the BN scheme allowing key aggregation in the plain public-key model. Our scheme has three rounds, the first two being exactly the same as in BN. However, we change the way the challenges c_i are computed from $c_i = H(\langle L\rangle, X_i, R, m)$ to

$$c_i = H_{\text{agg}}(\langle L\rangle, X_i) \cdot H_{\text{sig}}(\widetilde{X}, R, m),$$

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 $a_i = H_{\text{agg}}(\langle L\rangle, X_i)$ (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

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

[1] 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.

where $c = H_{\text{sig}}(\widetilde{X}, R, m)$. 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 $c = H_{\text{sig}}(\langle L \rangle, R, m)$ yields a secure scheme as well, but does not allow key aggregation since verification is impossible without knowing all the individual signer keys.

This modification comes at the price of complications in the security proof. Indeed, the "split" form of the challenges c_i prevents us from using directly the Forking Lemma [41] 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 of *using the Forking Lemma twice*, in a nested way (so that in total the reduction runs the forger four times). In more details, the reduction forks the execution of the adversary first with respect to answers of H_{sig} in order to obtain some aggregated key (involving the challenge public key) together with its discrete logarithm, and then with respect to answers of H_{agg} in order to retrieve the discrete logarithm of the challenge public key. The generalized Forking Lemma of Bellare and Neven [9] 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 [17,40,45] 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 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 valid signature for $\widetilde{X}$, 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).

Applications to Bitcoin Bitcoin [35] 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*[2] 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 [2], 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 [1,36] over the `secp256k1` curve [14] to authenticate transactions. As Bitcoin nodes fully verify all transactions, signature size and verification

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

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.
[9]	$\lvert\mathbb{G}\rvert + \lvert p\rvert$	$\lvert\mathbb{G}\rvert$	$\lvert p\rvert$	3	No
[3]	$3\lvert\mathbb{G}\rvert + 3\lvert p\rvert$	$\lvert\mathbb{G}\rvert$	$\lvert p\rvert$	2	No
[29]	$\lvert\mathbb{G}\rvert + 2\lvert p\rvert$ 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 $\ell + \lvert p\rvert$	$\lvert\mathbb{G}\rvert$	$\lvert p\rvert$	3	Yes

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 [18,26,30] 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.

Related Work Two variants of the BN multi-signature scheme have been proposed previously. Bagherzandi et al. [3] reduced the number of rounds from three to two using a homomorphic commitment scheme. However, this increases the signature size and the computational cost of signing and verification. Ma et al. [29] proposed a variant based on Okamoto's signature scheme [37] and a "double hashing" technique (the two hash functions being composed rather than multiplied as in our scheme), which allows reducing the signature size compared to [3] while using again only two rounds. However, none of these two variants allow key aggregation. A comparison of the four discrete logarithm-based multi-signature schemes provably secure in the plain public-key model is provided in Table 1.

A multi-signature scheme derived from the BLS signature scheme [7] and allowing key aggregation using the technique introduced in this paper was recently proposed by Boneh et al. [8]. It requires a single communication round but relies on the use of pairings.

Multi-signature schemes supporting key aggregation are easier to come by in the KOSK model. In particular, Syta et al. [46] proposed the CoSi scheme which can be seen as the naive Schnorr multi-signature scheme described earlier where the cosigners are organized in a tree structure for fast signature generation.

Other multi-signature schemes relying of different types of hard problems have been envisioned for Bitcoin, such as lattice-based schemes [16].

Besides Bitcoin, multi-signatures have applications in many other distributed applications. See [15,46] for more examples.

Revisions A preliminary version of this paper, dated January 15, 2018, proposed a 2-round variant of MuSig, where the initial commitment round is omitted, claiming provable security under the One More Discrete Logarithm (OMDL) assumption [10,11]. However, Drijvers et al. [15] have discovered a flaw in the security proof (as well as in the proof of the other 2-round DL-based multi-signature schemes by Bagherzandi et al. [3] and Ma et al. [29]). They also showed through a meta-reduction that these schemes cannot be proved secure using an algebraic black-box reduction under the DL or OMDL assumption (assuming the OMDL problem is hard).

In more details, observe that in the 2-round variant of MuSig, an adversary (controlling public keys $X_2, \ldots, X_n$) can impose the value of $R = \prod_{i=1}^{n} R_i$ used in signature protocols since he can choose $R_2, \ldots, R_n$ after having received R_1 from the honest signer (controlling public key $X_1 = g^{x_1}$). This forbids to use the textbook way of simulating the honest signer in the Random Oracle model without knowing x_1 by randomly drawing s_1 and c, computing $R_1 = g^{s_1}(X_1)^{-a_1 c}$, and later programming $H_{\text{sig}}(\widetilde{X}, R, m) := c$, since the adversary might have made the random oracle query $H_{\text{sig}}(\widetilde{X}, R, m)$ *before* engaging the corresponding signature protocol. To solve this problem without a commitment round, our hope was that the reduction could compute s_1 by querying the discrete logarithm of $R_1(X_1)^{a_1 c}$ to the DL oracle available in the formulation of the OMDL problem, using a fresh DL challenge as R_1 in each signature query, whose discrete logarithm could be computed once x_1 had been retrieved. This however does not combine well with the Forking Lemma, since the adversary might be forked in the middle of a signature protocol, when it has received R_1 but has not returned $R_2, \ldots, R_n$ to the reduction yet. This implies that in order to correctly simulate the signature oracle in the forked execution, the reduction must "waste" a second query to the DL oracle related to the same challenge R_1, foregoing any hope to solve the OMDL problem. Using an initial commitment round where each signer commits to its share R_i before receiving the shares of other signers as in the BN scheme allows us to simulate the honest signer in the standard way by programming H_{sig} and to restore the provable security of MuSig (under the standard DL assumption).

The security of the 3-round variant of MuSig was proved independently by Boneh et al. [8].

Organization of the paper We start in Sect. 2 by providing definitions and recalling the generalized Forking Lemma. In Sect. 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 Sect. 5 we expose the applications of multi-signatures to Bitcoin.

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.

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.

```
**algorithm** $\mathsf{Fork}^{\mathcal{A}}(\mathsf{inp})$
    pick random coins $\rho$ **for** $\mathcal{A}$
    $h_1, \ldots, h_q \leftarrow_\$ \{0,1\}^\ell$
    $\alpha \leftarrow \mathcal{A}(\mathsf{inp}, h_1, \ldots, h_q; \rho)$
    **if** $\alpha = \perp$ **then return** $\perp$
    **else** parse $\alpha$ as $(i, \mathsf{out})$
    $h'_i, \ldots, h'_q \leftarrow_\$ \{0,1\}^\ell$
    $\alpha' \leftarrow \mathcal{A}(\mathsf{inp}, h_1, \ldots, h_{i-1}, h'_i, \ldots, h'_q; \rho)$
    **if** $\alpha' = \perp$ **then return** $\perp$
    **else** parse $\alpha'$ as $(i', \mathsf{out}')$
    **if** $(i = i'$ and $h_i \neq h'_i)$ **then return** $(i, \mathsf{out}, \mathsf{out}')$
    **else return** $\perp$
```

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

The Discrete Logarithm Problem We recall the formal definition of the Discrete Logarithm (DL) problem.

Definition 1 *(DL problem)* Let $(\mathbb{G}, p, g)$ be group parameters. An algorithm $\mathcal{A}$ is said to (t, ε)-solve the DL problem w.r.t. $(\mathbb{G}, p, g)$ if on input a random group element X, it runs in time at most t and returns $x \in \{0, \ldots, p-1\}$ such that $X = g^x$ with probability at least ε, where the probability is taken over the random draw of X and the random coins of $\mathcal{A}$.

The Generalized Forking Lemma As in [9], our security proof will rely on the following generalized Forking Lemma extending Pointcheval and Stern's Forking Lemma [41] and which does not mention signatures nor forgers and only deals with the outputs of an algorithm $\mathcal{A}$ run twice on related inputs. See [9] for the proof of this result.

Lemma 1 *Fix integers q and ℓ. Let $\mathcal{A}$ be a randomized algorithm which takes as input some main input inp and ℓ-bit strings $h_1, \ldots, h_q$ and returns either a distinguished failure symbol $\perp$ or a pair (i, out), where $i \in \{1, \ldots, q\}$ 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 (according to some well-understood distribution), $h_1, \ldots, h_q \leftarrow_\$ \{0,1\}^\ell$, 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, described on Fig. 1. Let frk be the probability (over the draw of inp and the random coins of $\mathsf{Fork}^{\mathcal{A}}$) that $\mathsf{Fork}^{\mathcal{A}}$ returns a non-$\perp$ output. Then*

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

2.2 Syntax and security definition of multi-signature schemes

Syntax A multi-signature scheme Π consists of three algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver})$. 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 $L = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$, 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 $(\mathsf{sk}_1, \mathsf{pk}_1), \ldots, (\mathsf{sk}_n, \mathsf{pk}_n)$ and any message m, if a group of signers with public keys $L = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ 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 [9] and requires that it be infeasible to forge multi-signatures involving at least one honest signer. As in previous work [9,13,34], 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 $L = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ 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 $L = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$, 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,[3] and wins the above security

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

game with probability at least ε, the size of L in any signature query and in the forgery being at most N.

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 three hash functions[4] H_{com}, H_{agg}, and H_{sig} from $\{0,1\}^*$ 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 Sect. 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.[5] For $i \in \{1, \ldots, n\}$, the signer computes

$$a_i = H_{\mathrm{agg}}(L, X_i) \tag{1}$$

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 $R_1 = g^{r_1}$, $t_1 = H_{\mathrm{com}}(R_1)$, and sends t_1 to all other cosigners. Upon reception of commitments $t_2, \ldots, t_n$ from other cosigners, it sends R_1. Upon reception of $R_2, \ldots, R_n$ from other cosigners, it checks that $t_i = H_{\mathrm{com}}(R_i)$ for all $i \in \{2, \ldots, n\}$ and aborts the protocol if this is not the case; otherwise, it computes

$$\begin{aligned} R &= \prod_{i=1}^{n} R_i, \\ c &= H_{\mathrm{sig}}(\widetilde{X}, R, m), \\ s_1 &= r_1 + c a_1 x_1 \bmod p, \end{aligned}$$

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 $a_i = H_{\mathrm{agg}}(L, X_i)$ for $i \in \{1, \ldots, n\}$, $\widetilde{X} = \prod_{i=1}^{n} X_i^{a_i}$, $c = H_{\mathrm{sig}}(\widetilde{X}, R, m)$ and accepts the signature if $g^s = R \prod_{i=1}^{n} X_i^{a_i c} = R\widetilde{X}^c$.

[4] Hash function H_{com} is used in the commitment phase, H_{agg} to compute the aggregated key, and H_{sig} to compute the signature. These hash functions can be constructed from a single one using proper domain separation.

[5] As in [9], indices $1, \ldots, n$ are local references to cosigners, defined within the specific signer instance at hand.

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 Sect. 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 $H_{\mathrm{agg}}(X_i)$ 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 $\widetilde{X}_h = X_1^{H_{\mathrm{agg}}(X_1)}$. The attacker can then use Wagner's algorithm [47] to find $n-1$ integers $y_2, \ldots, y_n$ such that $\sum_{i=2}^{n} H_{\mathrm{agg}}(\widetilde{X}_h g^{y_i}) = -1 \bmod p$. 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

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

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 [42], 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

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

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 $c' = H_{\mathrm{sig}}(\widetilde{X}, R_1 R_2', m)$ 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 $(\mathbb{G}, p, g)$ where p is k-bit long and hash functions $H_{\text{com}}, H_{\text{agg}}, H_{\text{sig}} : \{0,1\}^* \to \{0,1\}^\ell$ are modeled as random oracles. Then, there exists an algorithm $\mathcal{C}$ which (t', ε')-solves the DL problem for $(\mathbb{G}, p, g)$, with $t' = 4t + 4Nt_{\exp} + O(N(q_h + q_s + 1))$ where $t_{\exp}$ is the time of an exponentiation in $\mathbb{G}$ and*

$$\varepsilon' \geq \frac{\varepsilon^4}{(q_h + q_s + 1)^3} - \frac{16q_s(q_h + Nq_s)}{2^k} - \frac{16(q_h + Nq_s)^2 + 3}{2^\ell}.$$

Before proving the theorem, we start with an informal explanation of the key technique used in the proof.

The Double-Forking Technique Let us recall the security game defined in Sect. 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 signature oracle parses L as $\{X_1 = X^*, X_2, \dots, X_n\}$, draws a random $r_1 \leftarrow_\$ \mathbb{Z}_p$, computes $R_1 = g^{r_1}$, and sends $t_1 = H_{\text{com}}(R_1)$ to the forger which sends back commitments $t_2, \dots, t_n$; the signature oracle then sends R_1 to the forger which answers with $R_2, \dots, R_n$; the signature oracle aborts the protocol if $t_i \neq H_{\text{com}}(R_i)$ for some $i \in \{2, \dots, n\}$, otherwise it computes

$$R = \prod_{i=1}^{r} R_i,$$
$$c = H_{\text{sig}}(\widetilde{X}, R, m),$$
$$s_1 = r_1 + ca_1x^* \bmod p,$$

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

The crux of the proof is how 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 $R = R'$, $H_{\text{sig}}(\widetilde{X}, R, m)$ was programmed in both executions to the same value h_1, $H_{\text{agg}}(L, X_i)$ was programmed in both executions to the same value a_i for each i such that $X_i \neq X^*$, and $H_{\text{agg}}(L, X^*)$ 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 $H_{\text{agg}}(L, X^*)$ does not work: indeed, at this moment, the relevant query $H_{\text{sig}}(\widetilde{X}, R, m)$ might not have been made yet by the forger,[6] 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_{sig} query. In order to remedy this situation, we fork the execution of the forger *twice*: once on the answer to the query $H_{\text{sig}}(\widetilde{X}, R, m)$, which allows us 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 $H_{\text{agg}}(L, X^*)$, which allows us 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_{com}, H_{agg}, and H_{sig} uniformly at random and the signature oracle by programming H_{sig}, and returns a forgery together with some information about the forger execution, unless some bad events happen.[7] Then, we use $\mathcal{A}$ to construct an algorithm $\mathcal{B}$ which runs the forking algorithm $\mathsf{Fork}^{\mathcal{A}}$ as defined in Sect. 2 (where the fork is w.r.t. the answer to the H_{sig} 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 DL problem by running $\mathsf{Fork}^{\mathcal{B}}$ (where the fork is now w.r.t. the answer to the H_{agg} query related to the forgery). Throughout the proof, the reader might find helpful to refer to Fig. 2 which illustrates the inner working of $\mathcal{C}$.

[6] 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 $H_{\text{agg}}(L, X_i)$ for $X_i \in L$, so that the query $H_{\text{sig}}(\widetilde{X}, R, m)$ can only come after the relevant queries $H_{\text{agg}}(L, X_i)$ except with negligible probability.

[7] 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 $\mathsf{KeyColl}$ defined in the proof of Lemma 2.

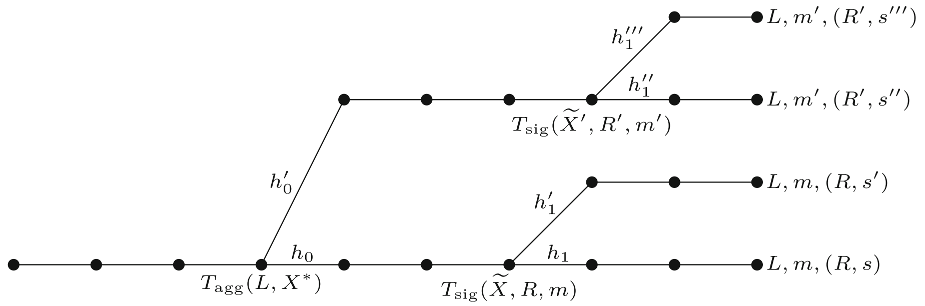

Fig. 2 A possible execution of algorithm $\mathcal{C}$. Each path from the leftmost root to one of the four rightmost leaves represent an execution of the forger. Each vertex symbolizes an assignment to tables $T_{\rm agg}$ or $T_{\rm sig}$ used to program $H_{\rm agg}$ and $H_{\rm sig}$, and the edge 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 $H_{\rm agg}(L, X)$ (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 a uniformly random group element X^* and uniformly random ℓ-bit strings $h_{0,1}, \ldots, h_{0,q}$ and $h_{1,1}, \ldots, h_{1,q}$,*[8] *and, with accepting probability (as defined in Lemma 1) at least*

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

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

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

Proof We construct algorithm $\mathcal{A}$ as follows. It initializes three empty tables $T_{\rm com}$, $T_{\rm agg}$, and $T_{\rm sig}$ for storing programmed values for respectively $H_{\rm com}$, $H_{\rm agg}$, and $H_{\rm sig}$ and two counters ctr_0 and ctr_1 (initially zero) that will be incremented respectively each time an entry of the form $T_{\rm agg}(\cdot, X^*)$ is assigned and each time an assignment is made in $T_{\rm sig}$. It also initializes six flags $\mathsf{BadCom}_i, i \in \{1, 2, 3\}$, $\mathsf{BadProg}$, $\mathsf{BadOrder}$, and $\mathsf{KeyColl}$ (that will help keep track of bad events) and a special flag Alert to `false`. Then, it picks random coins ρ_F and runs the forger $\mathcal{F}$ on input the public key X^*, answering its queries as follows.

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

- Hash query $H_{\mathrm{com}}(R)$: If $T_{\mathrm{com}}(R)$ is undefined, then $\mathcal{A}$ randomly assigns $T_{\mathrm{com}}(R) \leftarrow_{\$} \{0,1\}^{\ell}$; then, it returns $T_{\mathrm{com}}(R)$.
- Hash query $H_{\mathrm{agg}}(L, X)$: (Recall that by assumption $X^* \in L$ and $X \in L$.) If $T_{\mathrm{agg}}(L, X)$ is undefined, then $\mathcal{A}$ increments ctr_0, randomly assigns $T_{\mathrm{agg}}(L, X') \leftarrow_{\$} \{0,1\}^{\ell}$ for all $X' \in L \setminus \{X^*\}$, and assigns $T_{\mathrm{agg}}(L, X^*) := h_{0,\mathsf{ctr}_0}$. Then, it returns $T_{\mathrm{agg}}(L, X)$.
- Hash query $H_{\mathrm{sig}}(\widetilde{X}, R, m)$: If $T_{\mathrm{sig}}(\widetilde{X}, R, m)$ is undefined, then $\mathcal{A}$ increments ctr_1 and assigns $T_{\mathrm{sig}}(\widetilde{X}, R, m) := h_{1,\mathsf{ctr}_1}$. Then, it returns $T_{\mathrm{sig}}(\widetilde{X}, R, m)$.
- 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 $T_{\mathrm{agg}}(L, X^*)$ is undefined,[9] it makes an "internal" query to $H_{\mathrm{agg}}(L, X^*)$ which ensures that $T_{\mathrm{agg}}(L, X_i)$ is defined for each $i \in \{1, \ldots, n\}$, sets $a_i := T_{\mathrm{agg}}(L, X_i)$, and computes $\widetilde{X} := \prod_{i=1}^{n} X_i^{a_i}$. Then, $\mathcal{A}$ increases ctr_1, lets $c := h_{1,\mathsf{ctr}_1}$, draws $s_1 \leftarrow_{\$} \mathbb{Z}_p$, and computes $R_1 := g^{s_1}(X^*)^{-a_1 c}$. If $T_{\mathrm{com}}(R_1)$ is already defined, it sets $\mathsf{BadCom}_1 := \mathtt{true}$ and halts returning $\perp$. Otherwise, $\mathcal{A}$ makes an internal query to $H_{\mathrm{com}}(R_1)$ and sends $t_1 := T_{\mathrm{com}}(R_1)$ to the forger. Upon reception of commitments $t_2, \ldots, t_n$ from the forger, $\mathcal{A}$ looks for values R_i such that $T_{\mathrm{com}}(R_i) = t_i$ for $i \in \{2, \ldots, n\}$. Three cases may occur:

  1. If more than one such value can be found for some $i \in \{2, \ldots, n\}$, then $\mathcal{A}$ sets $\mathsf{BadCom}_2 := \mathtt{true}$ and halts returning $\perp$.
  2. If no such value can be found for some $i \in \{2, \ldots, n\}$, then $\mathcal{A}$ sets $\mathsf{Alert} := \mathtt{true}$ and sends R_1 to the forger.
  3. If exactly one such value exists for each $i \in \{2, \ldots, n\}$, then $\mathcal{A}$ computes $R := \prod_{i=1}^{n} R_i$. If $T_{\mathrm{sig}}(\widetilde{X}, R, m)$ has already been defined, then $\mathcal{A}$ sets $\mathsf{BadProg} := \mathtt{true}$ and halts returning $\perp$. Otherwise, it assigns $T_{\mathrm{sig}}(\widetilde{X}, R, m) := c$ and sends R_1 to the forger.

  Upon reception of $R_2, \ldots, R_n$ from the forger, $\mathcal{A}$ checks that $H_{\mathrm{com}}(R_i) = t_i$ for all $i \in \{2, \ldots, n\}$ (making internal call(s) to H_{com} if we were in case 2 above) and aborts the signature protocol if this is not the case. (Note that $\mathcal{A}$ does not halt returning $\perp$ in this case since this is just the expected behavior of the honest signer.) If $\mathsf{Alert} = \mathtt{true}$ and $H_{\mathrm{com}}(R_i) = t_i$ for all $1 \le i \le n$, then $\mathcal{A}$ sets $\mathsf{BadCom}_3 := \mathtt{true}$ and halts returning $\perp$. Otherwise, $\mathcal{A}$ sends s_1 to the forger, completing the simulation of the honest signature oracle.

If $\mathcal{F}$ returns $\perp$, $\mathcal{A}$ outputs $\perp$ as well. Otherwise, the forger returns a purported forgery (R, s) for a public key multiset L such that $X^* \in L$ and a message m. Then, $\mathcal{A}$ parses L as $\{X_1 = X^*, \ldots, X_n\}$ and checks the validity of the forgery as follows. If $T_{\mathrm{agg}}(L, X^*)$ is undefined, it makes an internal query to $H_{\mathrm{agg}}(L, X^*)$ which ensures that $T_{\mathrm{agg}}(L, X_i)$ is defined for each $i \in \{1, \ldots, n\}$, sets $a_i := T_{\mathrm{agg}}(L, X_i)$, and computes $\widetilde{X} := \prod_{i=1}^{n} X_i^{a_i}$. If $T_{\mathrm{sig}}(\widetilde{X}, R, m)$ is undefined, it makes an internal query to $H_{\mathrm{sig}}(\widetilde{X}, R, m)$ and lets $c := T_{\mathrm{sig}}(\widetilde{X}, R, m)$.[10] 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 $T_{\mathrm{agg}}(L, X^*) = h_{0,i_0}$ and i_1 be the index such that $T_{\mathrm{sig}}(\widetilde{X}, R, m) = h_{1,i_1}$. If the assignment $T_{\mathrm{agg}}(L, X^*) := h_{0,i_0}$ occurred *after* the assignment $T_{\mathrm{sig}}(\widetilde{X}, R, m) := h_{1,i_1}$, $\mathcal{A}$ sets $\mathsf{BadOrder} := \mathtt{true}$ and returns $\perp$. If there exists another multiset of public keys L' such that, at the end of the execution, $T_{\mathrm{agg}}(L', X')$ is defined for each $X' \in L'$ and the aggregated keys corresponding to L and L' are equal, $\mathcal{A}$ sets $\mathsf{KeyColl} := \mathtt{true}$ and returns $\perp$. Otherwise, it returns $(i_0, i_1, L, R, s, \mathbf{a})$, where

[9] This holds *iff* L never appeared in a previous query to H_{agg} or a previous signature query.

[10] 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.

$\mathbf{a} = (a_1, \ldots, a_n)$. By construction, $a_i = h_{0,i_0}$ for each i such that $X_i = X^*$, and the validity of the forgery implies Eq. (2).

We must now lower bound the accepting probability of $\mathcal{A}$. Since $h_{0,1}, \ldots, h_{0,q}$ and $h_{1,1}, \ldots, h_{1,q}$ are uniformly random, $\mathcal{A}$ perfectly simulates the security experiment to the forger, unless BadCom_i, $i \in \{1, 2, 3\}$ or $\mathsf{BadProg}$ is set to `true` and $\mathcal{A}$ halts the simulation. Moreover, when the forger eventually returns a forgery, $\mathcal{A}$ returns a non-$\perp$ output unless $\mathsf{BadOrder}$ or $\mathsf{KeyColl}$ is set to `true`. Hence, one has $\mathsf{acc}(\mathcal{A}) \geq \varepsilon - \Pr\left[\mathsf{Bad}\right]$, where Bad is the event that one of the six flags capturing bad events is set to `true`. It remains to upper bound $\Pr\left[\mathsf{Bad}\right]$.

We start with BadCom_i, $i \in \{1, 2, 3\}$. First, note that there are at most $q_h + Nq_s$ assignments to T_{com} in total. Since for each signature query, s_1 is drawn uniformly at random, $R_1 = g^{s_1}(X^*)^{-a_1 c}$ is uniform in a set of at least $p \geq 2^{k-1}$ values, so that the probability that $T_{\text{com}}(R_1)$ is defined is at most $(q_h + Nq_s)/2^{k-1}$. By the union bound, BadCom_1 is set to `true` with probability at most $q_s(q_h + Nq_s)/2^{k-1}$. For BadCom_2 to be set to `true`, two distinct entries of T_{com} must be set to the same value. Since T_{com} entries are set uniformly at random, this happens with probability at most $(q_h + Nq_s)^2/2^{\ell+1}$. Finally, BadCom_3 is set to `true` if during a signature query, the forger sends R_i such that $T_{\text{com}}(R_i)$ is undefined and gets set by chance to the value t_i of the corresponding commitment. Hence, this happens with probability at most $Nq_s/2^{\ell}$.

Next, we consider $\mathsf{BadProg}$, but we only upper bound the probability that it is set to `true` conditioned on BadCom_1 not being set to `true`. Let A, resp. B, be the event that BadCom_1, resp. $\mathsf{BadProg}$ is set to `true`. Then, $\Pr[A \vee B] \leq \Pr[A] + \Pr\left[B|\bar{A}\right]$, where $\Pr[A]$ has already been upper bounded above. If BadCom_1 never gets set to `true` during the execution, then for each signature query, the view of the forger when it returns $R_2, \ldots, R_n$ is independent of R_1, so that $R = \prod_{i=1} R_i$ is uniform in a set of at least $p \geq 2^{k-1}$ values. Since there are at most q assignments to T_{sig} in total, R "hits" a defined entry in T_{sig} with probability at most $q/2^{k-1}$. Hence, $\Pr\left[B|\bar{A}\right] \leq q_s q/2^{k-1}$.

It remains to consider $\mathsf{BadOrder}$ and $\mathsf{KeyColl}$, for which it will be convenient to introduce the following wording. Note that by construction of $\mathcal{A}$, for any multiset L' appearing at some point in the queries of the forger or in its forgery, assignments $T_{\text{agg}}(L', X')$ for all $X' \in L'$ are concomitant and occur the first time L' appears either in a query to H_{agg}, or in a signature query, or in the forgery. We will refer to the set of assignments $\{T_{\text{agg}}(L', X') := a', X' \in L'\}$ as the *set of* T_{agg} *assignments related to* L'. Note that there are at most q sets of T_{agg} assignments and that each of them contains a unique assignment $T_{\text{agg}}(L', X^*) := h_{0,i}$ for some $i \in \{1, \ldots, q\}$.

In order to upper bound the probability that $\mathsf{BadOrder}$ is set to `true`, we upper bound the probability that some set of T_{agg} assignments related to some multiset L' (not necessarily the one returned in the forgery) results in the aggregated key $\widetilde{X}'$ corresponding to L' being equal to the first argument of a defined entry in table T_{sig} (which is clearly a necessary condition for $\mathsf{BadOrder}$ to be set to `true`). Considering the i-th set of T_{agg} 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_{sig} and at most q sets of T_{agg} assignments, BadOrder is set to `true` with probability at most $q^2/2^\ell$.[11]

In order to upper bound the probability that KeyColl is set to `true`, we upper bound the probability that some set of T_{agg} assignments related to some multiset L' (not necessarily the one returned in the forgery) results in the aggregated key $\widetilde{X}'$ corresponding to L' being equal to the aggregated key $\widetilde{X}''$ corresponding to some previous set of T_{agg} assignments related to some other multiset L'' (again, neither L' nor L'' need be the multiset returned in the forgery). Since each 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 $q^2/2^\ell$.

Combining all of the above, we obtain

$$\begin{aligned}\mathsf{acc}(\mathcal{A}) \leq \varepsilon &- \frac{q_s(q_h + Nq_s)}{2^{k-1}} - \frac{q_s(q_h + q_s + 1)}{2^{k-1}} \\ &- \frac{(q_h + Nq_s)^2}{2^{\ell+1}} - \frac{Nq_s}{2^\ell} - \frac{2(q_h + q_s + 1)^2}{2^\ell} \\ \leq \varepsilon &- \frac{4q_s(q_h + Nq_s)}{2^k} - \frac{4(q_h + Nq_s)^2}{2^\ell},\end{aligned}$$

assuming without loss of generality that $q_s \geq 1$ and $N \geq 2$. □

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 a uniformly random group element X^* and uniformly random ℓ-bit strings $h_{0,1}, \dots, h_{0,q}$ and, with accepting probability (as defined in Lemma 1) at least*

$$\frac{\varepsilon^2}{q_h + q_s + 1} - \frac{8q_s(q_h + Nq_s)}{2^k} - \frac{8(q_h + Nq_s)^2 + 1}{2^\ell},$$

outputs a tuple $(i_0, L, \mathbf{a}, \tilde{x})$ where $i_0 \in \{1, \dots, q\}$, $L = \{X_1, \dots, X_n\}$ 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 $\tilde{x}$ 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 $h_{0,1}, \dots, h_{0,q}$ play the role of inp,
– $h_{1,1}, \dots, h_{1,q}$ play the role of $h_1, \dots, h_q$,
– i_1 plays the role of i,
– $(i_0, L, R, s, \mathbf{a})$ plays the role of out.

In more details, $\mathcal{B}$ picks random coins ρ_A and runs algorithm $\mathcal{A}$ on coins ρ_A, group element X^*, and ℓ-bit strings $h_{0,1}, \dots, h_{0,q}$ and $h_{1,1}, \dots, h_{1,q}$, where $h_{1,1}, \dots, h_{1,q}$ are drawn uniformly at random by $\mathcal{B}$ (recall that $h_{0,1}, \dots, h_{0,q}$ are part of the *input* of $\mathcal{B}$ and will be the same in both runs of $\mathcal{A}$). If $\mathcal{A}$ returns $\perp$, $\mathcal{B}$ returns $\perp$ as well. Otherwise, $\mathcal{A}$ returns a tuple $(i_0, i_1, L, R, s, \mathbf{a})$, where $L = \{X_1, \dots, X_n\}$ and $\mathbf{a} = (a_1, \dots, a_n)$.

[11] Note that for this argument to go through, we rely on $\widetilde{X}$ being included in the call to H_{sig}. 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_{sig}, even though we are not aware of any attack.

Then, $\mathcal{B}$ runs $\mathcal{A}$ again with the same random coins on input X^*, $h_{0,1}, \ldots, h_{0,q}$, and $h_{1,1}, \ldots, h_{1,i_1-1}, h'_{1,i_1}, \ldots, h'_{1,q}$, where $h'_{1,i_1}, \ldots, h'_{1,q}$ are uniformly random. If $\mathcal{A}$ returns $\perp$ in this second run, $\mathcal{B}$ returns $\perp$ as well. If $\mathcal{A}$ returns another tuple $(i'_0, i'_1, L', R', s', \mathbf{a}')$, 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

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

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 $(i_0, L, \mathbf{a}, \tilde{x})$.

It is easy to see that $\mathcal{B}$ returns a non-$\perp$ output *iff* $\mathsf{Fork}^{\mathcal{A}}$ does, so that by Lemmas 1 and 2, and letting

$$\alpha = \frac{4q_s(q_h + Nq_s)}{2^k} + \frac{4(q_h + Nq_s)^2}{2^\ell},$$

the accepting probability of $\mathcal{B}$ is at least

$$\begin{aligned}
\mathsf{acc}(\mathcal{A})\left(\frac{\mathsf{acc}(\mathcal{A})}{q} - \frac{1}{2^\ell}\right) &\geq \frac{(\varepsilon - \alpha)^2}{q} - \frac{\varepsilon - \alpha}{2^\ell} \\
&\geq \frac{\varepsilon^2}{q} - 2\alpha - \frac{1}{2^\ell} \\
&= \frac{\varepsilon^2}{q_h + q_s + 1} - \frac{8q_s(q_h + Nq_s)}{2^k} - \frac{8(q_h + Nq_s)^2 + 1}{2^\ell}.
\end{aligned}$$

It remains to prove the equalities of Eq. (3). In $\mathcal{A}$'s first execution, h_{1,i_1} is assigned to $T_{\text{sig}}(\widetilde{X}, R, m)$, while is $\mathcal{A}$'s second execution, h'_{1,i_1} is assigned to $T_{\text{sig}}(\widetilde{X}', R', m')$. Note that these two assignments can happen either because of a direct query to H_{sig} 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 $T_{\text{agg}}(\cdot, X^*)$ assignments, the same values $h_{1,1}, \ldots, h_{1,i_1-1}$ for T_{sig} assignments, and the same random coins for T_{com} assignments, $T_{\text{agg}}(\cdot, X \neq X^*)$ assignments, and random draws of s_1 in signature queries. Since both executions are identical up to the two assignments $T_{\text{sig}}(\widetilde{X}, R, m) := h_{1,i_1}$ and $T_{\text{sig}}(\widetilde{X}', R', m') := h'_{1,i_1}$, 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 $\mathsf{KeyColl}$ is set to `true` in both executions, a contradiction since $\mathcal{A}$ returns a non-$\perp$ output in both executions. Hence, $L = L'$. Since in both executions of $\mathcal{A}$, $\mathsf{BadOrder}$ is not set to `true`, assignments $T_{\text{agg}}(L, X^*) := h_{0,i_0}$ and $T_{\text{agg}}(L', X^*) := h_{0,i'_0}$ necessarily happened *before* the fork. This implies that $i_0 = i'_0$ 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 DL 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 $(\mathbb{G}, p, g)$ where p is k-bit long and hash functions $H_{\text{com}}, H_{\text{agg}}, H_{\text{sig}} : \{0,1\}^* \to \{0,1\}^\ell$ modeled as random oracles. Then, there exists an algorithm $\mathcal{C}$ which (t', ε')-solves the DL 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*

$$\varepsilon' \geq \frac{\varepsilon^4}{(q_h + q_s + 1)^3} - \frac{16 q_s (q_h + N q_s)}{2^k} - \frac{16(q_h + N q_s)^2 + 3}{2^\ell}.$$

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,
- $h_{0,1}, \dots, h_{0,q}$ play the role of $h_1, \dots, h_q$,
- i_0 plays the role of i,
- $(L, \mathbf{a}, \tilde{x})$ plays the role of out.

In more details, on input X^*, $\mathcal{C}$ picks random coins ρ_B and runs algorithm $\mathcal{B}$ on coins ρ_B, group element X^*, and uniformly random ℓ-bit strings $h_{0,1}, \dots, h_{0,q}$. If $\mathcal{B}$ returns $\perp$, $\mathcal{C}$ returns $\perp$ as well. Otherwise, $\mathcal{B}$ returns a tuple $(i_0, L, \mathbf{a}, \tilde{x})$. Then, $\mathcal{C}$ runs $\mathcal{B}$ again with the same random coins ρ_B on input X^* and

$$h_{0,1}, \dots, h_{0,i_0-1}, h'_{0,i_0}, \dots, h'_{0,q},$$

where $h'_{0,i_0}, \dots, h'_{0,q}$ are uniformly random. If $\mathcal{B}$ returns $\perp$ in this second run, $\mathcal{C}$ returns $\perp$ as well. If $\mathcal{B}$ returns another tuple $(i'_0, L', \mathbf{a}', \tilde{x}')$, $\mathcal{C}$ proceeds as follows. Let $L = \{X_1, \dots, X_n\}$, $\mathbf{a} = (a_1, \dots, a_n)$, $L' = \{X'_1, \dots, X'_{n'}\}$, and $\mathbf{a}' = (a'_1, \dots, a'_{n'})$. Let also n^* be the number of times X^* appears in L. If $i_0 \neq i'_0$, or $i_0 = i'_0$ and $h_{0,i_0} = h'_{0,i_0}$, $\mathcal{C}$ 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

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

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 time of $\mathcal{F}$ plus the time needed to maintain tables T_{com}, T_{agg}, and T_{sig} (we assume each assignment takes unit time) and answer signature queries. The size of T_{com}, T_{agg}, and T_{sig} are respectively at most $q_h + Nq_s$, qN, and 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 Lemmas 1 and 3, and letting

$$\beta = \frac{8q_s(q_h + Nq_s)}{2^k} + \frac{8(q_h + Nq_s)^2 + 1}{2^\ell},$$

the success probability ε' of $\mathcal{C}$ is at least

$$\begin{aligned}
\varepsilon' &\geq \mathsf{acc}(\mathcal{B})\left(\frac{\mathsf{acc}(\mathcal{B})}{q} - \frac{1}{2^\ell}\right)\\
&\geq \frac{(\varepsilon^2/q - \beta)^2}{q} - \frac{\varepsilon^2/q - \beta}{2^\ell}\\
&\geq \frac{\varepsilon^4}{q^3} - 2\beta - \frac{1}{2^\ell}\\
&= \frac{\varepsilon^4}{(q_h + q_s + 1)^3} - \frac{16q_s(q_h + Nq_s)}{2^k} - \frac{16(q_h + Nq_s)^2 + 3}{2^\ell}.
\end{aligned}$$

It remains to prove the equalities of Eq. (4). In the two executions of $\mathcal{A}$ run within the first execution of $\mathcal{B}$, h_{0,i_0} is assigned to $T_{\text{agg}}(L, X^*)$, while in the two executions of $\mathcal{A}$ run within the second execution of $\mathcal{B}$, h'_{0,i_0} is assigned to $T_{\text{agg}}(L', X^*)$. Note that these two assignments can happen either because of a direct query $H_{\text{agg}}(L, X)$ 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 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 $h_{0,1}, \ldots, h_{0,i_0-1}$ for $T_{\text{agg}}(\cdot, X^*)$ assignments, the same values $h_{1,1}, \ldots, h_{1,q}$ for T_{sig} assignments, and the same random coins for T_{com} assignments, $T_{\text{agg}}(\cdot, X \neq X^*)$ assignments, and the random draws of s_1 in signature queries (note that this relies on the fact that in the four executions of $\mathcal{A}$, $\mathsf{BadOrder}$ is not set to $\texttt{true}$). Since the four executions of $\mathcal{A}$ are identical up to the assignments $T_{\text{agg}}(L, X^*) := h_{0,i_0}$ and $T_{\text{agg}}(L', X^*) := h'_{0,i_0}$, the arguments of these two assignments must be the same, which implies that $L = L'$. Besides, all values $T_{\text{agg}}(L, X)$ 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 $c = H_{\text{sig}}(\widetilde{X}, R, m)$. 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_{sig} 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 $H'_{\mathrm{agg}}(L)$ when computing coefficients a_i in Eq. (1). This does not affect the security of the scheme but the security proof becomes slightly more complex. Informally, any query $H_{\mathrm{agg}}(h, X)$ where h is distinct from all previous answers to H'_{agg} queries is useless to the adversary unless a subsequent H'_{agg} 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 $H_{\mathrm{agg}}(h, X)$, $\mathcal{A}$ checks whether a previous H'_{agg} queries was answered with h; if not, it answers randomly; otherwise, $\mathcal{A}$ looks for the multiset L such that $H'_{\mathrm{agg}}(L)$ 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 $H_{\mathrm{agg}}(L, X)$.

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 for both 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 $\mathsf{KeyColl}$), 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 [35] is a digital currency with no trusted issuer or transaction processor, which works by means of a publicly verifiable distributed ledger, called the blockchain.[12] It contains every transaction since the system's inception, resulting in a final state, the set of unspent coins (also called the UTXO[13] 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. 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[14] correspond to a single ECDSA verification.

This is also how Bitcoin supports a naive version of multi-signatures with a threshold policy[15]: 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 [19] 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,[16] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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 [32] 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 [34, 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 [13,27], they rely on mechanisms that are usually not implemented by certification authorities [9,43].

5.3 Cross-input signature aggregation

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. In this case, signatures contained in each input are on a different message. An interactive 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 [6,12] and sequential aggregate signatures [28]. Bellare and Neven [9] 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.) The only caveat is that each signer must check that public keys of other participants are different from its own public key, and abort if this is not the case.

Cross-input signature aggregation 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-key/message pairs. Overall validity requires all returned booleans to be True and an aggregate signature of the transaction with L the union of all returned key/message pairs.

More concretely, this can be implemented by providing an alternative to the signature checking opcode `OP_CHECKSIG`[17] and related opcodes in the Script language. Instead of

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

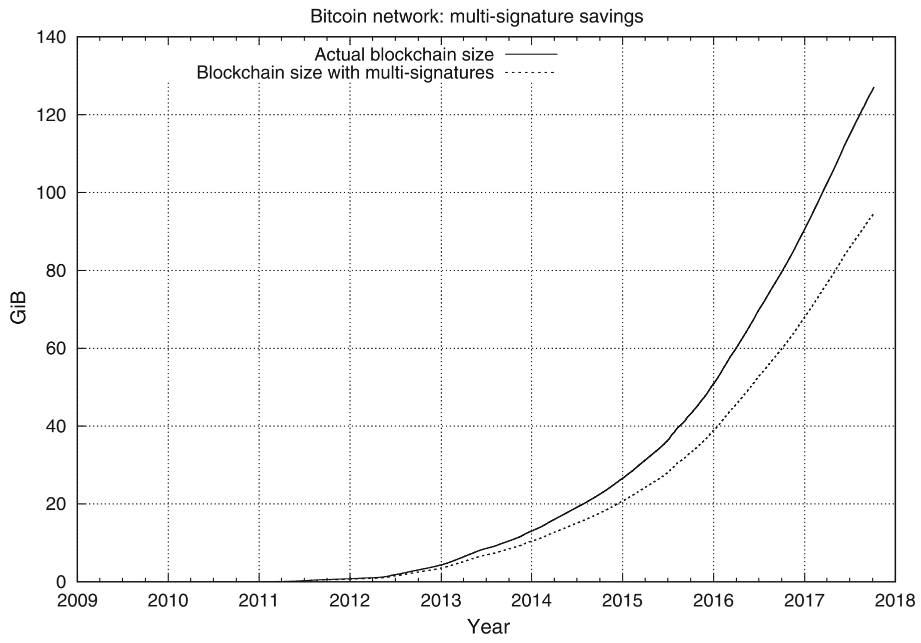

Fig. 3 Size of the Bitcoin blockchain with and without multi-signatures

returning the result of an actual ECDSA verification, they always return True, but additionally add the public-key/message pair for which the verification would have taken place to a transaction-wide multiset of key/message pairs. Finally, after all inputs are verified, an aggregate 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 aggregate 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 [31]. 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 Sect. 3.2. An attacker could identify an arbitrary number of outputs he wants to steal, with public keys $X_1, \ldots, X_{n-t}$, and then use the rogue-key attack to determine $X_{n-t+1}, \ldots, X_n$ such that he can sign for keys $\{X_1, \ldots, X_n\}$. He would then send a small amount of his own money to outputs with predicates corresponding to the keys $X_{n-t+1}, \ldots, X_n$. 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 signature aggregation 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.

5.4 Space savings

To analyze the impact of multi-signatures, an estimation 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.

References

1. 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).
2. Andresen G.: M-of-N standard transactions. Bitcoin Improvement Proposal. https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki (2011).
3. Bagherzandi A., Cheon J.H., Stanislaw J.: Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In: Ning, P., Syverson, P.F., Jha, S. (eds.) ACM Conference on Computer and Communications Security-CCS 2008, pp. 449–458. ACM (2008).
4. Bernstein D.J.: Multi-user Schnorr security, revisited. IACR Cryptology ePrint Archive, Report 2015/996 (2015). http://eprint.iacr.org/2015/996.
5. Bernstein D.J., Duif N., Lange T., Schwabe P., Yang B.-Y.: High-speed high-security signatures. In: Preneel, B., Takagi, T. (eds.) Cryptographic Hardware and Embedded Systems-CHES 2011, LNCS, vol. 6917, pp. 124–142. Springer, Berlin (2011).
6. Boneh D., Gentry C., Lynn B., Shacham H.: Aggregate and verifiably encrypted signatures from bilinear maps. In: Biham, E. (ed.) Advances in Cryptology-EUROCRYPT 2003, LNCS, vol. 2656, pp. 416–432. Springer, Berlin (2003).
7. Boneh D., Lynn B., Shacham H.: Short signatures from the Weil pairing. J. Cryptol. **17**(4), 297–319 (2004).
8. Boneh D., Drijvers M., Neven G.: Compact multi-signatures for smaller blockchains. In: Peyrin, T., Galbraith, S.D. (eds.) Advances in Cryptology-ASIACRYPT 2018 (Proceedings, Part II), LNCS, vol. 11273, pp. 435–464. Springer, Berlin (2018).
9. Bellare M., Neven G.: Multi-signatures in the plain public-key model and a general forking lemma. In: Juels, A., Wright, R.N., De Capitani di Vimercati, S. (eds.) ACM Conference on Computer and Communications Security-CCS 2006, pp. 390–399. ACM (2006).
10. Bellare M., Palacio A.: GQ and Schnorr identification schemes: proofs of security against impersonation under active and concurrent attacks. In: Yung, M. (ed.) Advances in Cryptology-CRYPTO 2002, LNCS, vol. 2442, pp. 162–177. Springer, Berlin (2002).
11. Bellare M., Namprempre C., Pointcheval D., Semanko M.: The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. J. Cryptol. **16**(3), 185–215 (2003).
12. Bellare M., Namprempre C., Neven G.: Unrestricted aggregate signatures. In: Arge, L., Cachin, C., Jurdzinski, T., Tarlecki, A. (eds.) Automata, Languages and Programming-ICALP 2007, LNCS, vol. 4596, pp. 411–422. Springer, Berlin (2007).
13. Boldyreva A.: Threshold signatures, multisignatures and blind signatures based on the Gap-Diffie-Hellman-Group Signature Scheme. In: Desmedt, Y. (ed.) Public Key Cryptography-PKC 2003, LNCS, vol. 2567, pp. 31–46. Springer, Berlin (2003).
14. Certicom Research: SEC 2: recommended elliptic curve domain parameters, v2.0 (2010). http://www.secg.org/sec2-v2.pdf.
15. Drijvers M., Edalatnejad K., Ford B., Neven G.: On the provable security of two-round multi-signatures. IACR Cryptology ePrint Archive, Report 2018/417 (2018). http://eprint.iacr.org/2018/417.

16. El Bansarkhani R., Jan S.: An efficient lattice-based multisignature scheme with applications to bitcoins. In: Foresti, S., Persiano, G. (eds.) Cryptology and Network Security-CANS 2016, LNCS, vol. 10052, pp. 140–155. Springer, Berlin (2016).
17. Garg S., Bhaskar R., Lokam S.V.: Improved bounds on security reductions for discrete log based signatures. In: Wagner, D. (ed.) Advances in Cryptology-CRYPTO 2008, LNCS, vol. 5157, pp. 93–107. Springer, Berlin (2008).
18. Gennaro R., Goldfeder S., Narayanan A.: Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In: Manulis, M., Sadeghi, A.-R., Schneider, S. (eds.) Applied Cryptography and Network Security-ACNS 2016, LNCS, vol. 9696, pp. 156–174. Springer, Berlin (2016).
19. Goldfeder S., Bonneau J., Gennaro R., Narayanan A.: Escrow protocols for cryptocurrencies: how to buy physical goods using Bitcoin. In: Financial Cryptography and Data Security-FC 2017 (2017). http://www.jbonneau.com/doc/GBGN17-FC-physical_escrow.pdf.
20. Harn L.: Group-oriented (t, n) threshold digital signature scheme and digital multisignature. IEE Proc. Comput. Digit. Tech. **141**(5), 307–313 (1994).
21. Horster P., Michels M., Petersen H.: Meta-multisignature schemes based on the discrete logarithm problem. In: IFIP/Sec '95, IFIP Advances in Information and Communication Technology, pp. 128–142. Springer, Berlin (1995).
22. Itakura K., Nakamura K.: A public-key cryptosystem suitable for digital multisignatures. NEC Res. Dev. **71**, 1–8 (1983).
23. Kiltz E., Masny D., Pan J.: Optimal security proofs for signatures from identification schemes. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology-CRYPTO 2016 (Proceedings, Part II), LNCS, vol. 9815, pp. 33–61. Springer, Berlin (2016).
24. Langford S.K.: Weakness in some threshold cryptosystems. In: Koblitz, N. (ed.) Advances in Cryptology-CRYPTO '96, LNCS, vol. 1109, pp. 74–82. Springer, Berlin (1996).
25. Li C.-M., Hwang T., Lee N.-Y.: Threshold-multisignature schemes where suspected forgery implies traceability of adversarial shareholders. In: De Santis, A. (ed.), Advances in Cryptology - EUROCRYPT '94, LNCS, vol. 950, pp. 194–204. Springer, Berlin (1994).
26. Lindell Y.: Fast secure two-party ECDSA signing. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology-CRYPTO 2017 (Proceedings, Part II), LNCS, vol. 10402, pp. 613–644. Springer, Berlin (2017).
27. Lu S., Ostrovsky R., Sahai A., Shacham H., Waters B.: Sequential aggregate signatures and multisignatures without random oracles. In: Vaudenay, S. (ed.) Advances in Cryptology-EUROCRYPT 2006, LNCS, vol. 4004, pp. 465–485. Springer, Berlin (2006).
28. Lysyanskaya A., Micali S., Reyzin L., Shacham H.: Sequential aggregate signatures from trapdoor permutations. In: Cachin, C., Camenisch, J. (eds.) Advances in Cryptology - EUROCRYPT 2004, LNCS, vol. 3027, pp. 74–90. Springer, Berlin (2004).
29. Ma C., Weng J., Li Y., Deng R.H.: Efficient discrete logarithm based multi-signature scheme in the plain public key model. Des. Codes Cryptogr. **54**(2), 121–133 (2010).
30. MacKenzie P.D., Reiter M.K.: Two-party generation of DSA signatures. In: Kilian, J. (ed.), Advances in Cryptology-CRYPTO 2001, LNCS, vol. 2139, pp. 137–154. Springer, Berlin (2001).
31. Maxwell G.: CoinJoin: Bitcoin privacy for the real world. (2013). BitcoinTalk post. https://bitcointalk.org/index.php?topic=279249.0.
32. Merkle R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology-CRYPTO '87, LNCS, vol. 293, pp. 369–378. Springer, Berlin (1987).
33. Michels M., Horster P.: On the risk of disruption in several multiparty signature schemes. In: Kim, K., Matsumoto, T. (eds.) Advances in Cryptology-ASIACRYPT '96, LNCS, vol. 1163, pp. 334–345. Springer, Berlin (1996).
34. Micali S., Ohta K., Reyzin L.: Accountable-subgroup multisignatures. In: Reiter, M.K., Samarati, P. (eds.) ACM Conference on Computer and Communications Security-CCS 2001, pp. 245–254. ACM (2001).
35. Nakamoto S.: Bitcoin: a peer-to-peer electronic cash system (2008). http://bitcoin.org/bitcoin.pdf.
36. National Institute of Standards and Technology. FIPS 186-4: digital signature standard (DSS) (2013). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.
37. Okamoto T.: Provably secure and practical identification schemes and corresponding signature schemes. In: Brickell, E.F. (ed.) Advances in Cryptology-CRYPTO'92, LNCS, vol. 740, pp. 31–53. Springer, Berlin (1992).
38. Ohta K., Okamoto T.: A digital multisignature scheme based on the Fiat-Shamir scheme. In: Imai, H., Rivest, R. L., Matsumoto, T. (eds.) Advances in Cryptology-ASIACRYPT '91, LNCS, vol. 739, pp. 139–148. Springer, Berlin (1991).
39. Ohta K., Okamoto T.: Multi-signature schemes secure against active insider attacks. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E82–A**(1), 21–31 (1999).

40. Paillier P., Vergnaud D.: Discrete-log-based signatures may not be equivalent to discrete log. In: Roy, B.K. (ed.) Advances in Cryptology-ASIACRYPT 2005, LNCS, vol. 3788, pp. 1–20. Springer, Berlin (2005).
41. Pointcheval D., Stern J.: Security arguments for digital signatures and blind signatures. J. Cryptol. **13**(3), 361–396 (2000).
42. Pornin T.: Deterministic usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC 6979 (2013). https://rfc-editor.org/rfc/rfc6979.txt.
43. Ristenpart T., Yilek S.: The power of proofs-of-possession: securing multiparty signatures against rogue-key attacks. In: Naor, M. (ed.) Advances in Cryptology-EUROCRYPT 2007, LNCS, vol. 4515, pp. 228–245. Springer, Berlin (2007).
44. Schnorr C.-P.: Efficient signature generation by smart cards. J. Cryptol. **4**(3), 161–174 (1991).
45. Seurin Y.: On the exact security of Schnorr-type signatures in the random oracle model. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology-EUROCRYPT 2012, LNCS, vol. 7237, pp. 554–571. Springer, Berlin (2012).
46. Syta E., Tamas I., Visher D., Wolinsky D.I., Jovanovic P., Gasser L., Gailly N., Khoffi I., Ford B.: Keeping authorities ”Honest or Bust” with decentralized witness cosigning. In: IEEE Symposium on Security and Privacy, SP 2016, pp. 526–545. IEEE Computer Society (2016).
47. Wagner D.A.: A generalized birthday problem. In: Yung, M. (ed.) Advances in Cryptology-CRYPTO 2002, LNCS, vol. 2442, pp. 288–303. Springer, Berlin (2002).