# Compact Multi-Signatures for Smaller Blockchains

Dan Boneh[1], Manu Drijvers[2,3], and Gregory Neven[2]

[1] Stanford University
dabo@cs.stanford.edu
[2] IBM Research – Zurich
{mdr,nev}@zurich.ibm.com
[3] ETH Zurich

**Abstract.** We construct new multi-signature schemes that provide new functionality. Our schemes are designed to reduce the size of the Bitcoin blockchain, but are useful in many other settings where multi-signatures are needed. All our constructions support both signature compression and public-key aggregation. Hence, to verify that a number of parties signed a common message $m$, the verifier only needs a short multi-signature, a short aggregation of their public keys, and the message $m$. We give new constructions that are derived from Schnorr signatures and from BLS signatures. Our constructions are in the plain public key model, meaning that users do not need to prove knowledge or possession of their secret key.

In addition, we construct the first short accountable-subgroup multi-signature (ASM) scheme. An ASM scheme enables any subset $S$ of a set of $n$ parties to sign a message $m$ so that a valid signature discloses which subset generated the signature (hence the subset $S$ is *accountable* for signing $m$). We construct the first ASM scheme where signature size is only $O(\kappa)$ bits over the description of $S$, where $\kappa$ is the security parameter. Similarly, the aggregate public key is only $O(\kappa)$ bits, independent of $n$. The signing process is non-interactive. Our ASM scheme is very practical and well suited for compressing the data needed to spend funds from a $t$-of-$n$ Multisig Bitcoin address, for any (polynomial size) $t$ and $n$.

## 1 Introduction

Consider $n$ parties where each party independently generates a key pair for a signature scheme. Some time later all $n$ parties want to sign the same message $m$. A *multi-signature scheme* [28, 38] is a protocol that enables the $n$ signers to jointly generate a short signature $\sigma$ on $m$ so that $\sigma$ convinces a verifier that all $n$ parties signed $m$. Specifically, the verification algorithm is given as input the $n$ public keys, the message $m$, and the multi-signature $\sigma$. The algorithm either accepts or rejects $\sigma$. The multi-signature $\sigma$ should be short; its length should be independent of the number of signers $n$. We define this concept more precisely in the next section, where we also present the standard security model for such

schemes [38]. Secure multi-signatures have been constructed from Schnorr signatures (e.g. [9]), from BLS signatures (e.g. [10]), and from many other schemes as discussed in Section 1.5.

A more general concept called an *aggregate signature scheme* [12] lets each of the $n$ parties sign a *different* message, but all these signatures can be aggregated into a single short signature $\sigma$. As before, this short signature should convince the verifier that all signers signed their designated message.

**Applications to Bitcoin.** Multi-signatures and aggregate signatures can be used to shrink the size of the Bitcoin blockchain [40]. In recent work, Maxwell, Poelstra, Seurin, and Wuille [35] suggest using multi-signatures to shrink the transaction data associated with Bitcoin Multisig addresses. Conceptually, a Multisig address is the hash of $n$ public keys $pk_1, \ldots, pk_n$ along with some number $t \in \{1, \ldots, n\}$ called a threshold (see [2, 35] for details). To spend funds associated with this address, one creates a transaction containing all $n$ public keys $pk_1, \ldots, pk_n$ followed by $t$ valid signatures from $t$ of the $n$ public keys, and writes this transaction to the blockchain. The message being signed is the same in all $t$ signatures, namely the transaction data.

In practice, Multisig addresses often use $t = n$, so that signatures from all $n$ public keys are needed to spend funds from this address. In this case, all $n$ signatures can be compressed using a multi-signature scheme into a single short signature. This shrinks the overall transaction size and reduces the amount of data written to the blockchain. This approach can also be made to work for $t < n$, when $\binom{n}{t}$ is small, by enumerating all $t$-size subsets [35, Sec. 5.2]. Multi-signatures can also be used to compress multi-input transactions, but for simplicity we will focus on Multisig addresses.

Notice that we still need to write all $n$ public keys to the blockchain, so compressing the signatures does not save too much. Fortunately, there is a solution to this as well. Maxwell et al. [35], building on the work on Bellare and Neven [9], construct a Schnorr-based multi-signature scheme that also supports public key aggregation; the verifier only needs a short aggregate public key instead of an explicit list of all $n$ public keys. With this approach, an $n$-of-$n$ Multisig address is simply the hash of the short aggregate public key, and the data written to the blockchain in a spending transaction is this single short aggregated public key, a single short compressed signature, and the message. This data is sufficient to convince the verifier that all $n$ signers signed the transaction. It shrinks the amount of data written to the blockchain by a factor of $n$.

Maxwell et al. call this primitive a multi-signature scheme with public key aggregation. Their signing protocol requires two rounds of communication among the signing parties, and they prove security of their scheme assuming the one-more discrete-log assumption (as assumption introduced in [8]). However, recent work [21] has shown that there is a gap in the security proof, and that security cannot be proven under this assumption. Whether their scheme can be proved secure under a different assumption or in a generic group model is currently an open problem.

In Section 5, we present a modification of the scheme by Maxwell et al. that we prove secure under the standard discrete-log assumption. Our $\mathcal{MSDL}$ scheme retains all the benefits of the original scheme, and in particular uses the same key aggregation technique, but we add one round to the signing protocol.

### 1.1 Better constructions using pairings

Our main results show that we can do much better by replacing the Schnorr signature scheme in [35] by BLS signatures [13]. The resulting schemes are an extremely good fit for Bitcoin, but are also very useful wherever multi-signatures are needed.

To describe our new constructions, we first briefly review the BLS signature scheme and its aggregation mechanism. Recall that the scheme needs: (1) An efficiently computable non-degenerate pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$ in groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ of prime order $q$. We let $g_1$ and $g_2$ be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. (2) A hash function $\mathsf{H}_0 : \mathcal{M} \to \mathbb{G}_1$. Now the BLS signature scheme works as follows:

- Key generation: choose a random $sk \xleftarrow{\$} \mathbb{Z}_q$ and output $(pk, sk)$ where $pk \leftarrow g_2^{sk} \in \mathbb{G}_2$.
- Sign$(sk, m)$: output $\sigma \leftarrow \mathsf{H}_0(m)^{sk} \in \mathbb{G}_1$.
- Verify$(pk, m, \sigma)$: if $e(\sigma, g_2) \stackrel{?}{=} e\big(\mathsf{H}_0(m), pk\big)$ output "accept", otherwise output "reject".

This signature scheme supports a simple signature aggregation procedure. Given triples $(pk_i,\ m_i,\ \sigma_i)$ for $i = 1, \ldots, n$, anyone can aggregate the signatures $\sigma_1, \ldots, \sigma_n \in \mathbb{G}_1$ into a short convincing aggregate signature $\sigma$ by computing

$$\sigma \leftarrow \sigma_1 \cdots \sigma_n \in \mathbb{G}_1. \tag{1}$$

To verify this aggregate signature $\sigma \in \mathbb{G}_1$ one checks that:

$$e(\sigma, g_2) = e\big(\mathsf{H}_0(m_1), pk_1\big) \cdots e\big(\mathsf{H}_0(m_n), pk_n\big). \tag{2}$$

Note that verification requires all $(pk_i,\ m_i)$ for $i = 1, \ldots, n$. When all the messages being signed are the same $(m_1 = \ldots = m_n)$ the verification relation (2) reduces to a simpler test that requires only two pairings:

$$e(\sigma, g_2) \stackrel{?}{=} e\Big(\mathsf{H}_0(m_1), pk_1 \cdots pk_n\Big). \tag{3}$$

Moreover, the verifier only needs to be given a short aggregate public-key $apk := pk_1 \cdots pk_n \in \mathbb{G}_2$.

**The rogue public-key attack.** The simple signature aggregation method in (1) is insecure on its own, and needs to be enhanced. To see why, consider the following rogue public-key attack: an attacker registers a rogue public key $pk_2 := g_2^{\alpha} \cdot (pk_1)^{-1} \in \mathbb{G}_2$, where $pk_1 \in \mathbb{G}_2$ is a public key of some unsuspecting user Bob, and $\alpha \xleftarrow{\$} \mathbb{Z}_q$ is chosen by the attacker. The attacker can then claim that both

it and Bob signed some message $m \in \mathcal{M}$ by presenting the aggregate signature $\sigma := \mathsf{H}_0(m)^\alpha$. This signature verifies as an aggregate of two signatures, one from $pk_1$ and one from $pk_2$, because

$$e(\sigma, g_2) = e\big(\mathsf{H}_0(m)^\alpha, g_2\big) = e\big(\mathsf{H}_0(m), g_2^\alpha\big) = e\big(\mathsf{H}_0(m), pk_1 \cdot pk_2\big).$$

Hence, this $\sigma$ satisfies (3). In effect, the attacker committed Bob to the message $m$, without Bob ever signing $m$.

**Defenses.** There are two standard defenses against the rogue public-key attack:

- Require every user to prove knowledge or possession of the corresponding secret key [10, 32, 47]. However, this is difficult to enforce in practice, as argued in [7, 47], and does not fit well with applications to crypto currencies, as explained in [35].
- Require that the messages being aggregated are distinct [12, 7], namely the verifier rejects an aggregate signature on non-distinct messages. This is sufficient to prevent the rogue key attack. Moreover, message distinctness can be enforced by always prepending the public key to every message prior to signing. However, because now all messages are distinct, we cannot take advantage of public-key aggregation as in (3) when aggregating signatures on a common message $m$.

### 1.2 Our pairing-based results

In Section 3 we propose a different defense against the rogue public-key attack that retains all the benefits of both defenses above without the drawbacks. In particular, our multi-signature scheme supports public key aggregation and fast verification as in (3). Moreover, the scheme is secure in the plain public-key model, which means that users do not need to prove knowledge or possession of their secret key. The scheme has two additional useful properties:

- The scheme supports batch verification where a set of multi-signatures can be verified as a batch faster than verifying them one by one.
- We show in Section 3.3 that given several multi-signatures on different messages, it is possible to aggregate all them using (1) into a single short signature. This can be used to aggregate signatures across many transactions and further shrink the data on the blockchain.

Our construction is based on the approach developed in [9] and [35] for securing Schnorr multi-signatures against the rogue public key attack.

Our BLS-based multi-signature scheme $\mathcal{MSP}$ is much easier to use than Schnorr multi-signatures. Recall that aggregation in Schnorr can only take place at the time of signing and requires a multi-round protocol between the signers. In our new scheme, aggregation can take place publicly by a simple multiplication, even long after all the signatures have been generated and the signers are no longer available. Concretely, in the context of Bitcoin this means that all the signers behind a Multisig address can simply send their signatures to one party

who aggregates all of them into a single signature. No interaction is needed, and the parties do not all need to be online at the same time. Moreover, we describe an aggregate multi-signature scheme $\mathcal{AMSP}$ that further compresses the multi-signatures on the different transactions into a single aggregate signature, yielding even more space savings.

**Accountable-Subgroup Multi-signatures.** Consider again $n$ parties where each party generates an independent signing key pair. An ASM enables any subset $S$ of the $n$ parties to jointly sign a message $m$, so that a valid signature implicates the subset $S$ that generated the signature; hence $S$ is accountable for signing $m$. The verifier in an ASM is given as input the (aggregate) ASM public key representing all $n$ parties, the set $S \subseteq \{1, \ldots, n\}$, the multi-signature generated by the set $S$, and the message $m$. It accepts or rejects the signature. Security should ensure that a set of signers $S' \not\supseteq S$ cannot issue a signature that will be accepted as if it were generated by $S$. We define ASMs and their security properties precisely in Section 4. This concept was previously studied by Micali et al. [38].

Any secure signature scheme gives a trivial ASM: every party generates an independent signing key pair. A signature by a set $S$ on message $m$ is simply the concatenation of all the signatures by the members of $S$. For a security parameter $\kappa$, the public key size in this trivial ASM is $O(n \times \kappa)$ bits. The signature size is $O(|S| \times \kappa)$ bits.

Our $\mathcal{ASM}$ scheme is the first ASM where signature size is only $O(\kappa)$ bits beyond the description of the set $S$, and the public key is only $O(\kappa)$ bits, independent of $n$. Concretely, the multi-signature is only two group elements, along with the description of $S$, and the public key is a single group element. The signing process is non-interactive, but initial key generation requires a simple one-round protocol between the all $n$ signers.

To see how this can be used, consider again a Bitcoin $n$-of-$n$ Multisig address. We already saw that multi-signatures with public key aggregation reduce the amount of data written to the blockchain to only $O(\kappa)$ bits when spending funds from this address (as opposed to $O(\kappa \times n)$ bits as currently done in Bitcoin). The challenge is to do the same for a $t$-of-$n$ Multisig address where $t < n$. Our ASM gives a complete solution; the only information that is written to the blockchain is a description of $S$ plus three additional group elements: one for the public key and two for the signature, even when $\binom{n}{t}$ is exponential. This is much better than the trivial linear size ASM scheme currently employed by Bitcoin.

### 1.3 Proofs of possession

Finally, in Section 6 we observe that all our schemes, both BLS-based and Schnorr-based, can be adapted to a setting where all users are required to provide a proof of possession (PoP) of their secret key. Proofs of possession increase the size of individual public keys, but there are applications where the size of individual keys is less relevant. For example, Multisig addresses in Bitcoin only need to store the aggregate public key on the blockchain, whereas the individual public keys are only relevant to the signers and can be kept off-chain, or

| | **Combined public key size** | **Combined signature size** | **Total size (KB)** | **Threshold support** |
|---|---|---|---|---|
| Bitcoin | $tx \cdot inp \cdot n \cdot \lvert\mathbb{G}\rvert$ | $tx \cdot inp \cdot n \cdot 2 \cdot \lvert\mathbb{Z}_q\rvert$ | 1296 | linear |
| MuSig ([35]) | $tx \cdot inp \cdot \lvert\mathbb{G}\rvert$ | $tx \cdot (\lvert\mathbb{G}\rvert + \lvert\mathbb{Z}_q\rvert)$ | 240 | small |
| $\mathcal{MSDL}$ (Sec. 5) | $tx \cdot inp \cdot \lvert\mathbb{G}\rvert$ | $tx \cdot (\lvert\mathbb{G}\rvert + \lvert\mathbb{Z}_q\rvert)$ | 240 | small |
| $\mathcal{MSP}$ (Sec. 3.1) | $tx \cdot inp \cdot \lvert\mathbb{G}_2\rvert$ | $tx \cdot \lvert\mathbb{G}_1\rvert$ | 360 | small |
| $\mathcal{AMSP}$ (Sec. 3.3) | $tx \cdot inp \cdot \lvert\mathbb{G}_2\rvert$ | $\lvert\mathbb{G}_1\rvert$ | 216 | small |
| $\mathcal{ASM}$ (Sec. 4) | $tx \cdot inp \cdot \lvert\mathbb{G}_2\rvert$ | $tx \cdot inp \cdot (\lvert\mathbb{G}_1\rvert + \lvert\mathbb{G}_2\rvert)$ | 864 | any |

**Table 1.** Comparison of the space required to authorize a block in the Bitcoin blockchain containing $tx$ transactions, each containing $inp$ inputs, all from $n$-out-of-$n$ multisig wallets. Here, $|\mathbb{G}|$ denotes the space required to represent an element of a group. The fourth column shows the concrete number of bytes taken in a bitcoin block by choosing some sample parameters ($tx = 1500$, $inp = 3$, $n = 3$), using secp256k1 [18] for Bitcoin, MuSig, and $\mathcal{MSDL}$ schemes ($|\mathbb{G}| = 32B$, $|\mathbb{Z}_q| = 32B$), and BLS381 [6] for the pairing-based $\mathcal{MSP}$, $\mathcal{AMSP}$, and $\mathcal{ASM}$ schemes ($|\mathbb{G}_1| = 96B$, $|\mathbb{G}_2| = 48B$, $|\mathbb{Z}_q| = 32B$). In the right-most column, "linear" denotes that $t$-of-$n$ thresholds are supported with key and signature sizes linear in $n$ and $t$, "small" denotes that support is limited to $\binom{n}{t}$ being small, and "any" denotes support for arbitrary (polynomial size) $t$ and $n$.

verified once and then discarded. Other applications may involve a more or less static set of signing nodes whose keys can be verified once and used in arbitrary combinations thereafter.

The PoP variants offer some advantages over our main schemes, such as simply using the product or hash of the public keys as the aggregate public key (as opposed to a multi-exponentiation), and having tighter security proofs to the underlying security assumption.

### 1.4 Efficiency comparison

Table 1 shows to what extent our constructions reduce the size of the Bitcoin blockchain. Our pairing-based scheme and $\mathcal{AMSP}$ and our discrete logarithm-based scheme $\mathcal{MSDL}$ both require less than 20% of the space to authenticate all transactions in a Bitcoin block compared to the currently deployed solution, assuming realistic parameters. While not immediately visible from the table, accountable-subgroup multi-signature schemes $\mathcal{ASM}$ is most useful for $t$-of-$n$ signatures when $\binom{n}{t}$ is very large. For instance, for a 50-out-of-100 multisig wallets, the currently deployed bitcoin solution would require 30 times more space than our $\mathcal{ASM}$ scheme. The other schemes support threshold signatures using Merkle trees [37] as outlined in [35, Sec. 5.2], but only when $\binom{n}{t}$ is small enough to generate the tree. This method would for example be infeasible for a 50-of-100 threshold scheme.

### 1.5 Related Work

Multi-signatures have been studied extensively based on RSA [28, 42, 45, 29], discrete logarithms [25, 31, 26, 27, 19, 43, 16, 38, 17, 9, 3, 4, 34, 35, 21], pair-

ings [10, 32, 47, 11, 30], and lattices [5]. Defending against rogue public-key attacks has always been a primary concern in the context of multisignature schemes based on discrete-log and pairings [27, 39, 38, 12, 9, 7, 47], and is the main reason for the added complexity in discrete-log-based multi-signature systems. Aggregate signatures [12, 24, 1] are a closely related concept where signatures by different signers on different messages can be compressed together. Sequential aggregate signatures [33, 32, 41, 14, 23] are a variant where signers take turns adding their own signature onto the aggregate. The concept of public-key aggregation in addition to signature compression has not been explicitly discussed in the plain public key model until [35] and this work. This concept greatly reduces the combined length of the data needed to verify a multi-signature.

## 2 Preliminaries

### 2.1 Bilinear Groups

Let $\mathcal{G}$ be a bilinear group generator that takes as an input a security parameter $\kappa$ and outputs the descriptions of multiplicative groups $(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2)$ where $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_t$ are groups of prime order $q$, $e$ is an efficient, non-degenerating bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$, and $g_1$ and $g_2$ are generators of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively.

### 2.2 Computational Problems

**Definition 1 (Discrete Log Problem).** *For a group $\mathbb{G} = \langle g \rangle$ of prime order $q$, we define $\mathsf{Adv}^{\mathsf{dl}}_{\mathbb{G}}$ of an adversary $\mathcal{A}$ as*

$$\Pr\left[y = g^x : y \xleftarrow{\$} \mathbb{G}, x \xleftarrow{\$} \mathcal{A}(y)\right],$$

*where the probability is taken over the random choices of $\mathcal{A}$ and the random selection of $y$. $\mathcal{A}$ $(\tau, \epsilon)$-breaks the discrete log problem if it runs in time at most $\tau$ and has $\mathsf{Adv}^{\mathsf{dl}}_{\mathbb{G}} \geq \epsilon$. Discrete log is $(\tau, \epsilon)$-hard if no such adversary exists.*

**Definition 2 (Computational co-Diffie-Hellman Problem).** *For a groups $\mathbb{G}_1 = \langle g_1 \rangle, \mathbb{G}_2 = \langle g_2 \rangle$ of prime order $q$, define $\mathsf{Adv}^{\mathsf{co\text{-}CDH}}_{\mathbb{G}_1,\mathbb{G}_2}$ of an adversary $\mathcal{A}$ as*

$$\Pr\left[y = g_1^{\alpha\beta} : (\alpha, \beta) \xleftarrow{\$} \mathbb{Z}_q^2, y \leftarrow \mathcal{A}(g_1^{\alpha}, g_1^{\beta}, g_2^{\beta})\right],$$

*where the probability is taken over the random choices of $\mathcal{A}$ and the random selection of $(\alpha, \beta)$. $\mathcal{A}$ $(\tau, \epsilon)$-breaks the $\mathsf{co\text{-}CDH}$ problem if it runs in time at most $\tau$ and has $\mathsf{Adv}^{\mathsf{co\text{-}CDH}}_{\mathbb{G}_1,\mathbb{G}_2} \geq \epsilon$. $\mathsf{co\text{-}CDH}$ is $(\tau, \epsilon)$-hard if no such adversary exists.*

**Definition 3 (Computational $\psi$-co-Diffie-Hellman Problem).** *For a groups $\mathbb{G}_1 = \langle g_1 \rangle, \mathbb{G}_2 = \langle g_2 \rangle$ of prime order $q$, let $\mathcal{O}^{\psi}(\cdot)$ be an oracle that on input $g_2^{\alpha} \in \mathbb{G}_2$ returns $g_1^{\alpha} \in \mathbb{G}_1$. Define $\mathsf{Adv}^{\psi\text{-}\mathsf{co\text{-}CDH}}_{\mathbb{G}_1,\mathbb{G}_2}$ of an adversary $\mathcal{A}$ as*

$$\Pr\left[y = g_1^{\alpha\beta} : (\alpha, \beta) \xleftarrow{\$} \mathbb{Z}_q^2, y \leftarrow \mathcal{A}^{\mathcal{O}^{\psi}(\cdot)}(g_1^{\alpha}, g_1^{\beta}, g_2^{\beta})\right],$$

*where the probability is taken over the random choices of $\mathcal{A}$ and the random selection of $(\alpha, \beta)$. $\mathcal{A}$ $(\tau, \epsilon)$-breaks the $\psi$-co-CDH problem if it runs in time at most $\tau$ and has $\mathsf{Adv}^{\psi\text{-co-CDH}}_{\mathbb{G}_1,\mathbb{G}_2} \geq \epsilon$. $\psi$-co-CDH is $(\tau, \epsilon)$-hard if no such adversary exists.*

### 2.3 Generalized Forking Lemma

The forking lemma of Pointcheval and Stern [46] is commonly used to prove the security of schemes based on Schnorr signatures [48] in the random-oracle model. Their lemma was later generalized to apply to a wider class of schemes [9, 3]. We recall the version due to Bagherzandi, Cheon, and Jarecki [3] here.

Let $\mathcal{A}$ be an algorithm that on input *in* interacts with a random oracle $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_q$. Let $f = (\rho, h_1, \ldots, h_{q_\mathrm{H}})$ be the randomness involved in an execution of $\mathcal{A}$, where $\rho$ is $\mathcal{A}$'s random tape, $h_i$ is the response to $\mathcal{A}$'s $i$-th query to $\mathsf{H}$, and $q_\mathrm{H}$ is its maximal number of random-oracle queries. Let $\Omega$ be the space of all such vectors $f$ and let $f|_i = (\rho, h_1, \ldots, h_{i-1})$. We consider an execution of $\mathcal{A}$ on input *in* and randomness $f$, denoted $\mathcal{A}(in, f)$, as *successful* if it outputs a pair $(J, \{out_j\}_{j\in J})$, where $J$ is a multi-set that is a non-empty subset of $\{1, \ldots, q_\mathrm{H}\}$ and $\{out_j\}_{j\in J}$ is a multi-set of side outputs. We say that $\mathcal{A}$ failed if it outputs $J = \emptyset$. Let $\epsilon$ be the probability that $\mathcal{A}(in, f)$ is successful for fresh randomness $f \xleftarrow{\$} \Omega$ and for an input $in \xleftarrow{\$} \mathsf{IG}$ generated by an input generator $\mathsf{IG}$.

For a given input *in*, the generalized forking algorithm $\mathcal{GF}_\mathcal{A}$ is defined as follows:

$\mathcal{GF}_\mathcal{A}(in)$:
  $f = (\rho, h_1, \ldots, h_{q_\mathrm{H}}) \xleftarrow{\$} \Omega$
  $(J, \{out_j\}_{j\in J}) \leftarrow \mathcal{A}(in, f)$
  If $J = \emptyset$ then output `fail`
  Let $J = \{j_1, \ldots, j_n\}$ such that $j_1 \leq \ldots \leq j_n$
  For $i = 1, \ldots, n$ do
    $succ_i \leftarrow 0$ ; $k_i \leftarrow 0$ ; $k_{\max} \leftarrow 8nq_\mathrm{H}/\epsilon \cdot \ln(8n/\epsilon)$
    Repeat until $succ_i = 1$ or $k_i > k_{\max}$
      $f'' \xleftarrow{\$} \Omega$ such that $f'|_{j_i} = f|_{j_i}$
      Let $f'' = (\rho, h_1, \ldots, h_{j_i-1}, h''_{j_i}, \ldots, h''_{q_\mathrm{H}})$
      $(J'', \{out''_j\}_{j\in J''}) \leftarrow \mathcal{A}(in, f'')$
      If $h''_{j_i} \neq h_{j_i}$ and $J'' \neq \emptyset$ and $j_i \in J''$ then
        $out'_{j_i} \leftarrow out''_{j_i}$ ; $succ_i \leftarrow 1$
  If $succ_i = 1$ for all $i = 1, \ldots, n$
  Then output $(J, \{out_j\}_{j\in J}, \{out'_j\}_{j\in J})$ else output `fail`

We say that $\mathcal{GF}_\mathcal{A}$ succeeds if it doesn't output `fail`. Bagherzandi et al. proved the following lemma for this forking algorithm.

**Lemma 1 (Generalized Forking Lemma [3]).** *Let $\mathsf{IG}$ be a randomized algorithm and $\mathcal{A}$ be a randomized algorithm running in time $\tau$ making at most*

$q_{\mathrm{H}}$ *random-oracle queries that succeeds with probability* $\epsilon$*. If* $q > 8nq_{\mathrm{H}}/\epsilon$*, then* $\mathcal{GF}_{\mathcal{A}}(in)$ *runs in time at most* $\tau \cdot 8n^2 q_{\mathrm{H}}/\epsilon \cdot \ln(8n/\epsilon)$ *and succeeds with probability at least* $\epsilon/8$*, where the probability is over the choice of* $in \xleftarrow{\$} \mathsf{IG}$ *and over the coins of* $\mathcal{GF}_{\mathcal{A}}$*.*

### 2.4 Multi-Signatures and Aggregate Multi-Signatures

We follow the definition of Bellare and Neven [9] and define a multisignature scheme as algorithms $\mathsf{Pg}$, $\mathsf{Kg}$, $\mathsf{Sign}$, $\mathsf{KAg}$, and $\mathsf{Vf}$. A trusted party generates the system parameters $par \leftarrow \mathsf{Pg}$. Every signer generates a key pair $(pk, sk) \xleftarrow{\$} \mathsf{Kg}(par)$, and signers can collectively sign a message $m$ by each calling the interactive algorithm $\mathsf{Sign}(par, \mathcal{PK}, sk, m)$, where $\mathcal{PK}$ is the set of the public keys of the signers, and $sk$ is the signer's individual secret key. At the end of the protocol, every signer outputs a signature $\sigma$. Algorithm $\mathsf{KAg}$ on input a set of public keys $\mathcal{PK}$ outputs a single aggregate public key $apk$. A verifier can check the validity of a signature $\sigma$ on message $m$ under an aggregate public key $apk$ by running $\mathsf{Vf}(par, apk, m, \sigma)$ which outputs 0 or 1 indicating that the signatures is invalid or valid, respectively.

A multisignature scheme should satisfy completeness, meaning that for any $n$, if we have $(pk_i, sk_i) \leftarrow \mathsf{Kg}(par)$ for $i = 1, \ldots, n$, and for any message $m$, if all signers input $\mathsf{Sign}(par, sk_i, m)$, then every signer will output a signature $\sigma$ such that $\mathsf{Vf}(par, \mathsf{KAg}(par, \{pk_i\}_{i=1}^n), m, \sigma) = 1$. Second, a multisignature scheme should satisfy unforgeability. Unforgeability of a multisignature scheme $\mathcal{AMS} = (\mathsf{Pg}, \mathsf{Kg}, \mathsf{Sign}, \mathsf{KAg}, \mathsf{Vf})$ is defined by a three-stage game.

*Setup.* The challenger generates the parameters $par \leftarrow \mathsf{Pg}$ and a challenge key pair $(pk^*, sk^*) \xleftarrow{\$} \mathsf{Kg}(par)$. It runs the adversary on the public key $\mathcal{A}(par, pk^*)$.

*Signature queries.* $\mathcal{A}$ is allowed to make signature queries on any message $m$ for any set of signer public keys $\mathcal{PK}$ with $pk^* \in \mathcal{PK}$, meaning that it has access to oracle $\mathcal{O}^{\mathsf{Sign}(par, \cdot, sk^*, \cdot)}$ that will simulate the honest signer interacting in a signing protocol with the other signers of $\mathcal{PK}$ to sign message $m$. Note that $\mathcal{A}$ may make any number of such queries concurrently.

*Output.* Finally, the adversary outputs a multisignature forgery $\sigma$, a message $m$, and a set of public keys $\mathcal{PK}$. The adversary wins if $pk^* \in \mathcal{PK}$, $\mathcal{A}$ made no signing queries on $m^*$, and $\mathsf{Vf}(par, \mathsf{KAg}(par, \mathcal{PK}), m)\}, \sigma) = 1$.

**Definition 4.** *We say* $\mathcal{A}$ *is a* $(\tau, q_{\mathrm{S}}, q_{\mathrm{H}}, \epsilon)$*-forger for multisignature scheme* $\mathcal{AMS} = (\mathsf{Pg}, \mathsf{Kg}, \mathsf{Sign}, \mathsf{Vf}, \mathsf{SAg}, \mathsf{AVf})$ *if it runs in time* $\tau$*, makes* $q_{\mathrm{S}}$ *signing queries, makes* $q_{\mathrm{H}}$ *random oracle queries, and wins the above game with probability at least* $\epsilon$*.* $\mathcal{AMS}$ *is* $(\tau, q_{\mathrm{S}}, q_{\mathrm{H}}, \epsilon)$*-unforgeable if no* $(\tau, q_{\mathrm{S}}, q_{\mathrm{H}}, \epsilon)$*-forger exists.*

### 2.5 Aggregate Multi-Signatures

We now introduce aggregate multi-signatures, combining the concepts of aggregate signatures and multisignatures, allowing for multiple multisignatures to be

aggregated into one. More precisely, we extend the definition of multisignatures with two algorithms. $\mathsf{SAg}$ takes input a set of tuples, each tuple containing an aggregate public key $apk$, a message $m$, and a multisignature $\sigma$, and outputs a single aggregate multisignature $\Sigma$. $\mathsf{AVf}$ takes input a set of tuples, each tuple containing an aggregate public key $apk$ and a message $m$, and an aggregate multisignature $\Sigma$, and outputs 0 or 1 indicating that the aggregate multisignatures is invalid or valid, respectively. Observe that any multisignature scheme can be transformed into an aggregate multisignature scheme in a trivial manner, by implementing $\mathsf{SAg}(par, \{apk_i, m_i, \sigma_i\})$ to output $\Sigma \leftarrow (\sigma_1, \ldots, \sigma_n)$, and $\mathsf{AVf}\big(par, \{apk_i, m_i\}, (\sigma_1, \ldots, \sigma_n)\big)$ to output 1 if all individual multisignatures are valid. The goal however is to have $\Sigma$ much smaller than the concatenation of the individual multisignatures, and ideally of constant size.

The security of aggregate multisignatures is very similar to the security of multisignatures. First, an aggregate multisignature scheme should satisfy completeness, meaning that 1) for any $n$, if we have $(pk_i, sk_i) \leftarrow \mathsf{Kg}(par)$ for $i = 1, \ldots, n$, and for any message $m$, if all signers input $\mathsf{Sign}(par, sk_i, m)$, then every signer will output a signature $\sigma$ such that $\mathsf{Vf}(par, \mathsf{KAg}(par, \{pk_i\}_{i=1}^n), m, \sigma) = 1$, and 2) for any set of multisignatures $\{(apk_i, m_i, \sigma_i)\}$ where every element is valid ($\mathsf{Vf}(par, apk_i, m_i, \sigma_i) = 1$ for every element), the aggregated multisignature is also valid: $\mathsf{AVf}(par, \{apk_i, m_i\}, \mathsf{SAg}(par, \{(apk_i, m_i, \sigma_i)\})) = 1$. Second, an aggregate multisignature scheme should satisfy unforgeability. Unforgeability of an aggregate multisignature scheme $\mathcal{AMS} = (\mathsf{Pg}, \mathsf{Kg}, \mathsf{Sign}, \mathsf{KAg}, \mathsf{Vf}, \mathsf{SAg}, \mathsf{AVf})$ is defined by a three-stage game, where the setup stage and the signature queries stage are the same as in the multisignature unforgeability game. The output stage is changed as follows:

*Output.* Finally, the adversary halts by outputting an aggregate multisignature forgery $\Sigma$, set of aggregate public keys a message pairs $\{apk_i, m_i\}$, a set of public keys $\mathcal{PK}$, and a message $m^*$. The adversary wins if $pk^* \in \mathcal{PK}$, $\mathcal{A}$ made no signing queries on $m^*$, and $\mathsf{AVf}(par, \{(apk_i, m_i)\} \cup \{(\mathsf{KAg}(par, \mathcal{PK}), m^*)\}, \Sigma) = 1$.

**Definition 5.** *We say $\mathcal{A}$ is a $(\tau, q_{\mathrm{S}}, q_{\mathrm{H}}, \epsilon)$-forger for aggregate multisignature scheme $\mathcal{AMS} = (\mathsf{Pg}, \mathsf{Kg}, \mathsf{Sign}, \mathsf{Vf}, \mathsf{SAg}, \mathsf{AVf})$ if it runs in time $\tau$, makes $q_{\mathrm{S}}$ signing queries, makes $q_{\mathrm{H}}$ random oracle queries, and wins the above game with probability at least $\epsilon$. $\mathcal{AMS}$ is $(\tau, q_{\mathrm{S}}, q_{\mathrm{H}}, \epsilon)$-unforgeable if no $(\tau, q_{\mathrm{S}}, q_{\mathrm{H}}, \epsilon)$-forger exists.*

## 3 Multi-Signatures with Key Aggregation from Pairings

We begin by presenting our new pairing-based multi-signature scheme that supports public-key aggregation. Bilinear groups are typically asymmetric, in the sense that one of the two groups has a more compact representation. The pairing-based schemes below require public keys and signatures to live in different groups. For standard signatures, a single public key is used to sign many messages, so it would make sense to use the more compact group for signatures. Because our

schemes below enable aggregation of both signatures and public keys, however, this may no longer be true, and the best choice of groups may depend strongly on the concrete application. We describe our schemes below placing signatures in $\mathbb{G}_1$ and public keys in $\mathbb{G}_2$, but leave it open which of those two groups has the more compact representation. Note that efficient hash functions exist mapping into either of the groups [49, 22, 15].

### 3.1 Description of our Pairing-Based Scheme

Our pairing-based multi-signature with public-key aggregation $\mathcal{MSP}$ is built from the BLS signature scheme [13]. The scheme is secure in the plain public key model, and assumes hash functions $\mathsf{H}_0 : \{0,1\}^* \to \mathbb{G}_2$ and $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{Z}_q$.

**Parameters Generation.** $\mathsf{Pg}(\kappa)$ sets up bilinear group $(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2) \leftarrow \mathcal{G}(\kappa)$ and outputs $par \leftarrow (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2)$.

**Key Generation.** The key generation algorithm $\mathsf{Kg}(par)$ chooses $sk \xleftarrow{\$} \mathbb{Z}_q$, computes $pk \leftarrow g_2^{sk}$, and outputs $(pk, sk)$.

**Key Aggregation.** $\mathsf{KAg}(\{pk_1, \ldots, pk_n\})$ outputs

$$apk \leftarrow \prod_{i=1}^{n} pk_i^{\mathsf{H}_1(pk_i, \{pk_1, \ldots, pk_n\})} .$$

**Signing.** Signing is a single round protocol. $\mathsf{Sign}(par, \{pk_1, \ldots, pk_n\}, sk_i, m)$ computes $s_i \leftarrow \mathsf{H}_0(m)^{a_i \cdot sk_i}$, where $a_i \leftarrow \mathsf{H}_1(pk_i, \{pk_1, \ldots, pk_n\})$. Send $s_i$ to a designated combiner who computes the final signature as $\sigma \leftarrow \prod_{j=1}^{n} s_j$. This designated combiner can be one of the signers or it can be an external party.

**Multi-Signature Verification.** $\mathsf{Vf}(par, apk, m, \sigma)$ outputs 1 iff

$$e(\sigma, g_2^{-1}) \cdot e(\mathsf{H}_0(m), apk) \stackrel{?}{=} 1_{\mathbb{G}_t} .$$

**Batch verification.** We note that a set of $b$ multi-signatures can be verified as a batch faster than verifying them one by one. To see how, suppose we are given triples $(m_i, \sigma_i, apk_i)$ for $i = 1, \ldots, b$, where $apk_i$ is the aggregated public-key used to verify the multi-signature $\sigma_i$ on $m_i$. If all the messages $m_1, \ldots, m_b$ are distinct then we can use signature aggregation as in (1) to verify all these triples as a batch:

- Compute an aggregate signature $\tilde{\sigma} = \sigma_1 \cdots \sigma_b \in \mathbb{G}_1$,
- Accept all $b$ multi-signature tuples as valid iff

$$e(\tilde{\sigma}, g_2) \stackrel{?}{=} e\big(\mathsf{H}_0(m_1), apk_1\big) \cdots e\big(\mathsf{H}_0(m_b), apk_b\big).$$

This way, verifying the $b$ multi-signatures requires only $b+1$ pairings instead of $2b$ pairings to verify them one by one. This simple batching procedure can only be used when all the messages $m_1, \ldots, m_b$ are distinct. If some messages are repeated then batch verification can be done by first choosing random exponents $\rho_1, \ldots, \rho_b \xleftarrow{\$} \{1, \ldots, 2^\kappa\}$, where $\kappa$ is a security parameter, computing $\tilde{\sigma} = \sigma_1^{\rho_1} \cdots \sigma_b^{\rho_b} \in \mathbb{G}_2$, and checking that

$$e(\tilde{\sigma}, g_2) \stackrel{?}{=} e\big(\mathsf{H}_0(m_1), apk_1^{\rho_1}\big) \cdots e\big(\mathsf{H}_0(m_b), apk_b^{\rho_b}\big).$$

Of course the pairings on the right hand side can be coalesced for repeated messages.

### 3.2 Security Proof

**Theorem 1.** *$\mathcal{MSP}$ is an unforgeable multisignature scheme (as defined in Def 4) under the computational co-Diffie-Hellman problem in the random-oracle model. More precisely, $\mathcal{MSP}$ is $(\tau, q_S, q_H, \epsilon)$-unforgeable in the random-oracle model if $q > 8q_H/\epsilon$ and if $\mathsf{co\text{-}CDH}$ is $((\tau + q_H\tau_{\exp_1} + q_S(\tau_{\exp_2^l} + \tau_{\exp_1}) + \tau_{\exp_2^l}) \cdot 8q_H^2/\epsilon \cdot \ln(8q_H/\epsilon), \epsilon/(8q_H))$-hard, where $l$ is the maximum number of signers involved in a single multisignature, $\tau_{\exp_1}$ and $\tau_{\exp_2}$ denote the time required to compute exponentiations in $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively, and $\tau_{\exp_1^i}$ and $\tau_{\exp_2^i}$ denote the time required to compute $i$-multiexponentiations in $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively.*

*Proof.* Suppose we have a $(\tau, q_S, q_H, \epsilon)$ forger $\mathcal{F}$ against the $\mathcal{MSP}$ multisignature scheme. Then consider an input generator $\mathsf{IG}$ that generates random tuples $(A, B_1, B_2) = (g_1^\alpha, g_1^\beta, g_2^\beta)$ where $\alpha, \beta \xleftarrow{\$} \mathbb{Z}_q$, and an algorithm $\mathcal{A}$ that on input $(A, B_1, B_2)$ and randomness $f = (\rho, h_1, \ldots, h_{q_S})$ proceeds as follows.

Algorithm $\mathcal{A}$ picks an index $k \xleftarrow{\$} \{1, \ldots, q_H\}$ and runs the forger $\mathcal{F}$ on input $pk^* \leftarrow B_2$ with random tape $\rho$. It responds to $\mathcal{F}$'s $i$-th $\mathsf{H}_0$ query by choosing $r_i \xleftarrow{\$} \mathbb{Z}_q$ and returning $g_1^{r_i}$ if $i \neq k$. The $k$-th $\mathsf{H}_0$ query is answered by returning $A$. We assume w.l.o.g. that $\mathcal{F}$ makes no repeated $\mathsf{H}_0$ queries. $\mathcal{A}$ responds to $\mathcal{F}$'s $\mathsf{H}_1$ queries as follows.We distinguish three types of $\mathsf{H}_1$ queries:

1. A query on $(pk, \mathcal{PK})$ with $pk \in \mathcal{PK}$ and $pk^* \in \mathcal{PK}$, and this is the first such query with $\mathcal{PK}$.
2. A query on $(pk, \mathcal{PK})$ with $pk \in \mathcal{PK}$ and $pk^* \in \mathcal{PK}$, and and a prior query of this form with $\mathcal{PK}$ has been made.
3. Queries of any other form.

$\mathcal{A}$ handles the $i$-th query of type (1) by choosing a random value for $\mathsf{H}_1(pk_i, \mathcal{PK})$ for every $pk_i \neq pk^* \in \mathcal{PK}$. It fixes $\mathsf{H}_1(pk^*, \mathcal{PK})$ to $h_i$, and returns the $\mathsf{H}_1(pk, \mathcal{PK})$. $\mathcal{A}$ handles a type (2) query by returning the values chosen earlier when the type (1) query for $\mathcal{PK}$ was made. $\mathcal{A}$ handles a type (3) query by simply returning a random value in $\mathbb{Z}_q$.

When $\mathcal{F}$ makes a signing query on message $m$, with signers $\mathcal{PK}$, $\mathcal{A}$ computes $apk \leftarrow \mathsf{KAg}(par, \mathcal{PK})$ and looks up $\mathsf{H}_0(m)$. If this is $A$, then $\mathcal{A}$ aborts with output $(0, \perp)$. Else, it must be of form $g_1^r$, and $\mathcal{A}$ can simulate the honest signer

by computing $s_i \leftarrow B_1^r$. When $\mathcal{F}$ fails to output a successful forgery, then $\mathcal{A}$ outputs $(0, \perp)$. If $\mathcal{F}$ successfully outputs a forgery for a message $m$ so that $\mathsf{H}_0(m) \neq A$), then $\mathcal{A}$ also outputs $(0, \perp)$. Otherwise, $\mathcal{F}$ has output a forgery $(\sigma, \mathcal{PK}, m)$ such that

$$e(\sigma, g_2) = e(A, \mathsf{KAg}(par, \mathcal{PK})).$$

Let $j_\mathrm{f}$ be the index such that $\mathsf{H}_1(pk^*, \mathcal{PK}) = h_{j_\mathrm{f}}$, let $apk \leftarrow \mathsf{KAg}(par, \mathcal{PK})$, and let $a_j \leftarrow \mathsf{H}_1(pk_j, \mathcal{PK})$ for $\mathcal{PK} = \{pk_1, \ldots, pk_n\}$. Then $\mathcal{A}$ outputs $(J = \{j_\mathrm{f}\}, \{(\sigma, \mathcal{PK}, apk, a_1, \ldots, a_n)\})$.

The running time of $\mathcal{A}$ is that of $\mathcal{F}$ plus the additional computation $\mathcal{A}$ makes. Let $q_\mathrm{H}$ denote the total hash queries $\mathcal{F}$ makes, i.e., the queries to $\mathsf{H}_0$ and $\mathsf{H}_1$ combined. $\mathcal{A}$ needs one exponentiation in $\mathbb{G}_1$ to answer $\mathsf{H}_0$ queries, so it spends at most $q_\mathrm{H} \cdot \tau_{\mathrm{exp}_1}$ to answer the hash queries. For signing queries with a $\mathcal{PK}$ of size at most $l$, $\mathcal{A}$ computes one multi-exponentiation costing time $\tau_{\mathrm{exp}_2^l}$, and one exponentiation in $\mathbb{G}_1$ costing $\tau_{\mathrm{exp}_1}$, giving a total of $q_\mathrm{S} \cdot (\tau_{\mathrm{exp}_2^l} + \tau_{\mathrm{exp}_1})$. Finally, $\mathcal{A}$ computes the output values, which costs an additional $\tau_{\mathrm{exp}_2^l}$ to compute $apk$. $\mathcal{A}$'s runtime is therefore $\tau + q_\mathrm{H}\tau_{\mathrm{exp}_1} + q_\mathrm{S}(\tau_{\mathrm{exp}_2^l} + \tau_{\mathrm{exp}_1}) + \tau_{\mathrm{exp}_2^l}$. The success probability of $\mathcal{A}$ is the probability that $\mathcal{F}$ succeeds and that it guessed the hash index of $\mathcal{F}$'s forgery correctly, which happens with probability at least $1/q_\mathrm{H}$, making $\mathcal{A}$'s overall success probability $\epsilon_\mathcal{A} = \epsilon/q_\mathrm{H}$.

We prove the theorem by constructing an algorithm $\mathcal{B}$ that, on input a co-CDH instance $(A, B_1, B_2) \in \mathbb{G}_1 \times \mathbb{G}_1 \times \mathbb{G}_2$ and a forger $\mathcal{F}$, solves the co-CDH problem in $(\mathbb{G}_1, \mathbb{G}_2)$. Namely, $\mathcal{B}$ runs the generalized forking algorithm $\mathcal{GF}_\mathcal{A}$ from Lemma 1 on input $(A, B_1, B_2)$ with the algorithm $\mathcal{A}$ described above. Observe that the co-CDH-instance is distributed indentically to the output of $\mathsf{IG}$. If $\mathcal{GF}_\mathcal{A}$ outputs $(0, \perp)$, then $\mathcal{B}$ outputs `fail`. If $\mathcal{GF}_\mathcal{A}$ outputs $(\{j_\mathrm{f}\}, \{out\}, \{out'\})$, then $\mathcal{B}$ proceeds as follows. $\mathcal{B}$ parses $out$ as $(\sigma, \mathcal{PK}, apk, a_1, \ldots, a_n)$ and $out'$ as $(\sigma', \mathcal{PK}', apk', a'_1, \ldots, a'_{n'})$. From the construction of $\mathcal{GF}_\mathcal{A}$, we know that $out$ and $out'$ were obtained from two executions of $\mathcal{A}$ with randomness $f$ and $f'$ such that $f|_{j_\mathrm{f}} = f'|_{j_\mathrm{f}}$, meaning that these executions are identical up to the $j_\mathrm{f}$-th $\mathsf{H}_1$ query of type (1). In particular, this means that the arguments of this query are identical, i.e., $\mathcal{PK} = \mathcal{PK}'$ and $n = n'$. If $i$ is the index of $pk^*$ in $\mathcal{PK}$, then again by construction of $\mathcal{GF}_\mathcal{A}$, we have $a_i = h_{j_\mathrm{f}}$ and $a'_i = h'_{j_\mathrm{f}}$, and by the forking lemma it holds that $a_i \neq a_i$. By construction of $\mathcal{A}$, we know that $apk = \prod_{j=1}^n pk_j^{a_j}$ and $apk' = \prod_{j=1}^n pk_j^{a'_j}$. Since $\mathcal{A}$ assigned $\mathsf{H}_1(pk_j, \mathcal{PK}) \leftarrow a_j$ for all $j \neq i$ *before* the forking point, we have that $a_j = a'_j$ for $j \neq i$, and therefore that $apk/apk' = pk^{*a_i - a'_i}$. We know that $\mathcal{A}$'s output satisfies $e(\sigma, g_2) = e(A, apk)$ and $e(\sigma', g_2) = e(A, apk')$, so that $e(\sigma/\sigma', g_2) = e(A, {B_2}^{a_i - a'_i})$, showing that $(\sigma/\sigma')^{1/(a_i - a'_i)}$ is a solution to the co-CDH instance.

Using Lemma 1, we know that if $q > 8q_\mathrm{H}/\epsilon$, then $\mathcal{B}$ runs in time at most $(\tau + q_\mathrm{H}\tau_{\mathrm{exp}_1} + q_\mathrm{S}(\tau_{\mathrm{exp}_2^l} + \tau_{\mathrm{exp}_1}) + \tau_{\mathrm{exp}_2^l}) \cdot 8q_\mathrm{H}^2/\epsilon \cdot \ln(8q_\mathrm{H}/\epsilon)$ and succeeds with probability $\epsilon' \geq \epsilon/(8q_\mathrm{H})$.

### 3.3 Aggregating Multi-Signatures

It is possible to further aggregate the multi-signatures of the $\mathcal{MSP}$ scheme by multiplying them together, as long as the messages of the aggregated multi-signatures are different. The easiest way to guarantee that messages are different is by including the aggregate public key in the message to be signed, which is how we define the aggregate multisignature scheme $\mathcal{AMSP}$ here. That is, $\mathcal{AMSP}$ and $\mathcal{MSP}$ share the $\mathsf{Pg}$, $\mathsf{Kg}$, and $\mathsf{KAg}$, algorithms, but $\mathcal{AMSP}$ has slightly modified $\mathsf{Sign}$ and $\mathsf{Vf}$ algorithms that include $apk$ in the signed message, and has additional algorithms $\mathsf{SAg}$ and $\mathsf{AVf}$ to aggregate signatures and verify aggregate signatures, respectively.

**Signing.** $\mathsf{Sign}(par, \mathcal{PK}, sk_i, m)$ computes $s_i \leftarrow \mathsf{H}_0(apk, m)^{a_i \cdot sk_i}$, where $apk \leftarrow \mathsf{KAg}(par, \mathcal{PK})$ and $a_i \leftarrow \mathsf{H}_1(pk_i, \{pk_1, \ldots, pk_n\})$. The designated combiner collect all signatures $s_i$ and computes the final signature $\sigma \leftarrow \prod_{j=1}^{n} s_j$.

**Multi-Signature Verification.** $\mathsf{Vf}(par, apk, m, \sigma)$ outputs 1 if and only if $e(\sigma, g_2^{-1}) \cdot e(\mathsf{H}_0(apk, m), apk) \stackrel{?}{=} 1_{\mathbb{G}_t}$.

**Signature Aggregation.** $\mathsf{SAg}(par, \{(apk_i, m_i, \sigma_i)\}_{i=1}^{n})$ outputs $\Sigma \leftarrow \prod_{i=1}^{n} \sigma_i$.

**Aggregate Signature Verification.** $\mathsf{AVf}(\{(apk_i, m_i)\}_{i=1}^{n}, \Sigma)$ outputs 1 if and only if $e(\Sigma, g_2^{-1}) \cdot \prod_{i=1}^{n} e(\mathsf{H}_0(apk_i, m_i), apk_i) \stackrel{?}{=} 1_{\mathbb{G}_t}$.

The security proof is almost identical to that of $\mathcal{MSP}$, but now requires an isomorphism $\psi$ between $\mathbb{G}_1$ and $\mathbb{G}_2$. We therefore prove security under the stronger $\psi$-$\mathsf{co}$-$\mathsf{CDH}$ assumption, which is equivalent to $\mathsf{co}$-$\mathsf{CDH}$ but offers this isomorphism as an oracle to the adversary.

**Theorem 2.** *$\mathcal{AMSP}$ is a secure aggregate multisignature scheme under the computational $\psi$-co-Diffie-Hellman problem in the random-oracle model. More precisely, $\mathcal{AMSP}$ is $(\tau, q_S, q_H, \epsilon)$-unforgeable in the random-oracle model if $q > 8q_H/\epsilon$ and if the computational $\psi$-co-Diffie-Hellman problem is $((\tau + q_H\tau_{\exp_1} + q_S(\tau_{\exp_2^l} + \tau_{\exp_1}) + \tau_{\exp_2^l} + \tau_{\exp_1^n}) \cdot 8q_H^2/\epsilon \cdot \ln(8q_H/\epsilon), \epsilon/(8q_H))$-hard, where $l$ is the maximum number of signers involved in a single multisignature, $n$ is the amount of multisignatures aggregated into the forgery, $\tau_{\exp_1}$ and $\tau_{\exp_2}$ denote the time required to compute exponentiations in $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively, and $\tau_{\exp_1^i}$ and $\tau_{\exp_2^i}$ denote the time required to compute $i$-multiexponentiations in $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively.*

*Proof.* Suppose we have a $(\tau, q_S, q_H, \epsilon)$ forger $\mathcal{F}$ against the $\mathcal{AMSP}$ multisignature scheme. We construct $\mathcal{A}$ exactly as in the proof of Theorem 1, except that $\mathcal{F}$ now outputs an aggregate multisignature signature forgery instead of a plain multisignature forgery. That is, $\mathcal{F}$ outputs an aggregate multisignature $\Sigma$, a set of aggregate public keys and message pairs $\{(apk_1, m_1), \ldots, (apk_n, m_n)\}$, a set of public keys $\mathcal{PK}$, and a message $m^*$. Let $apk^* \leftarrow \mathsf{KAg}(par, \mathcal{PK})$. If $\mathcal{A}$ correctly guessed that the $k$-th $\mathsf{H}_0$ query is $\mathsf{H}_0(apk^*, m^*)$, then we have that

$$e(\Sigma, g_2^{-1}) \cdot e(A, apk^*) \cdot \prod_{i=1}^{n} e(\mathsf{H}_0(apk_i, m_i), apk_i) = 1_{\mathbb{G}_t} \ .$$

$\mathcal{A}$ looks up $r_i$ for every $(apk_i, m_i)$ such that $\mathsf{H}_0(apk_i, m_i) = g_1^{r_i}$. It computes $\sigma \leftarrow \Sigma \cdot \prod_{i=1}^{n} \mathcal{O}^{\psi}(apk_i^{-r_i})$, so that

$$e(\sigma, g_2) = e(y, apk^*).$$

Note that $\mathcal{A}$ has now extracted a $\mathcal{MSP}$ forgery, meaning that the rest of the reduction is exactly as in the proof of Theorem 1. The success probability of the reduction is therefore the same, and the runtime is only increased by the extra steps required to compute $\sigma$, which costs $\tau_{\exp_1^n}$.

## 4 Accountable-Subgroup Multisignatures

Micali, Ohta, and Reyzin [38] defined an accountable-subgroup multisignature scheme as a multisignature scheme where any subset $S$ of a group of signers $\mathcal{PK}$ can create a valid multisignature that can be verified against the public keys of signers in the subset. An ASM scheme can be combined with an arbitrary access structure over $\mathcal{PK}$ to determine whether the subset $S$ is authorized to sign on behalf of $\mathcal{PK}$. For example, requiring that $|S| \geq t$ turns the ASM scheme into a type of threshold signature scheme whereby the signature also authenticates the set of signers that participated.

Verification of an ASM scheme obviously requires a description of the set $S$ of signers which can be described by their indices in the group $\mathcal{PK}$ using $\min(|\mathcal{PK}|,\ |S| \times \lceil \log_2 |\mathcal{PK}| \rceil)$ bits. We describe the first ASM scheme that, apart from the description of $S$, requires no data items with sizes depending on $|S|$ or $|\mathcal{PK}|$. Verification is performed based on a compact aggregate public key and signature. The aggregate public key is publicly computable from the individual signers' public keys, but we do require all members of $\mathcal{PK}$ to engage in a one-time group setup after which each signer obtains a group-specific membership key that it needs to sign messages for the group $\mathcal{PK}$.

### 4.1 Definition of ASM Schemes

We adapt the original syntax and security definition of ASM schemes [38] to support public-key aggregation and an interactive group setup procedure.

An ASM scheme consists of algorithms $\mathsf{Pg}$, $\mathsf{Kg}$, $\mathsf{GSetup}$, $\mathsf{Sign}$, $\mathsf{KAg}$, and $\mathsf{Vf}$. The common system parameters are generated as $par \xleftarrow{\$} \mathsf{Pg}$. Each signer generates a key pair $(pk, sk) \xleftarrow{\$} \mathsf{Kg}(par)$. To paricipate in a group of signers $\mathcal{PK} = \{pk_1, \ldots, pk_n\}$, each signer in $\mathcal{PK}$ runs the interactive algorithm $\mathsf{GSetup}(sk, \mathcal{PK})$ to obtain a membership key $mk$. We assume that each signer in $\mathcal{PK}$ is assigned a publicly computable index $i \in \{1, \ldots, |\mathcal{PK}|\}$, e.g., the index of $pk$ in a sorted list of $\mathcal{PK}$. Any subgroup of signers $S \subseteq \{1, \ldots, |\mathcal{PK}|\}$ of $\mathcal{PK}$ can then collectively sign a message $m$ by each calling the interactive algorithm $\mathsf{Sign}(par, \mathcal{PK}, S, sk, mk, m)$, where $mk$ is the signer's membership key for this group of signers, to obtain a signature $\sigma$. The key aggregation algorithm, on input the public keys of a group of signers $\mathcal{PK}$, outputs an aggregate public key

$apk$. A signature $\sigma$ is verified by running $\mathsf{Vf}(par, apk, S, m, \sigma)$ which outputs 0 or 1.

Correctness requires that for all $n > 0$, for all $S \subseteq \{1, \ldots, n\}$, and for all $m \in \{0,1\}^*$ it holds that $\mathsf{Vf}(par, apk, S, m, \sigma) = 1$ with probability one when $par \xleftarrow{\$} \mathsf{Pg}$, $(pk_i, sk_i) \xleftarrow{\$} \mathsf{Kg}(par)$, $mk_i \xleftarrow{\$} \mathsf{GSetup}(sk_i, \{pk_1, \ldots, pk_n\})$, and $\sigma \xleftarrow{\$} \mathsf{Sign}(par, \{pk_1, \ldots, pk_n\}, S, sk_i, mk_i, m)$, where $\mathsf{GSetup}$ is executed by all signers $1, \ldots, n$ while $\mathsf{Sign}$ is only executed by the members of $S$.

**Security.** Unforgeability is described by the following game.

*Setup.* The challenger generates $par \leftarrow \mathsf{Pg}$ and $(pk^*, sk^*) \xleftarrow{\$} \mathsf{Kg}(par)$, and runs the adversary $\mathcal{A}(par, pk^*)$.

*Group Setup.* The adversary can perform the group setup protocol $\mathsf{GSetup}(sk^*, \mathcal{PK})$ for any set of public keys $\mathcal{PK}$ so that $pk^* \in \mathcal{PK}$, where the challenger plays the role of the target signer $pk^*$. The challenger stores the resulting membership key $mk^*_{\mathcal{PK}}$, but doesn't hand it to $\mathcal{A}$.

*Signature queries.* The adversary can also engage in arbitrarily many concurrent signing protocols for any message $m$, for any group of signers $\mathcal{PK}$ for which $pk^* \in \mathcal{PK}$ and $mk^*_{\mathcal{PK}}$ is defined, and for any $S \subseteq \{1, \ldots, |\mathcal{PK}|\}$ so that $i \in S$, where $i$ is the index of $pk^*$ in $\mathcal{PK}$. The challenger runs $\mathsf{Sign}(par, \mathcal{PK}, S, sk^*, mk^*, m)$ to play the role of the $i$-th signer and hands the resulting signature $\sigma$ to $\mathcal{A}$.

*Output.* The adversary outputs a set of public keys $\mathcal{PK}$, a set $S \subseteq \{1, \ldots, |\mathcal{PK}|\}$, a message $m$ and an ASM signature $\sigma$. It wins the game if $\mathsf{Vf}(par, apk, S, m, \sigma) = 1$, where $apk \leftarrow \mathsf{KAg}(\mathcal{PK})$, $pk^* \in \mathcal{PK}$ and $i$ is the index of $pk^*$ in $\mathcal{PK}$, $i \in S$, and $\mathcal{A}$ never submitted $m$ as part of a signature query.

**Definition 6.** *We say that $\mathcal{A}$ is a $(\tau, q_{\mathrm{G}}, q_{\mathrm{S}}, q_{\mathrm{H}}, \epsilon)$-forger for accountable-subgroup multisignature scheme $\mathcal{ASM}$ if it runs in time $\tau$, makes $q_{\mathrm{G}}$ group setup queries, $q_{\mathrm{S}}$ signing queries, $q_{\mathrm{H}}$ random-oracle queries, and wins the above game with probability at least $\epsilon$. $\mathcal{ASM}$ is $(\tau, q_{\mathrm{G}}, q_{\mathrm{S}}, q_{\mathrm{H}}, \epsilon)$-unforgeable if no $(\tau, q_{\mathrm{G}}, q_{\mathrm{S}}, q_{\mathrm{H}}, \epsilon)$-forger exists.*

### 4.2 Our ASM Scheme

Key generation and key aggregation in our ASM scheme are the same as for our aggregatable multi-signature scheme in the previous section. We construct an ASM scheme by letting all signers, during group setup, contribute to multi-signatures on the aggregate public key and the index of every signer, such that the $i$-th signer in $\mathcal{PK}$ has a "membership key" which is a multi-signature on $(apk, i)$. On a high level, an accountable-subgroup multi-siganture now consists of the aggregation of the individual signers' signatures and their membership keys and the aggregate public key of the subroup $S$. To verify whether a subgroup $S$ signed a message, one checks that the signature is a valid aggregate signature

where the aggregate public key of the subgroup signed the message and the membership keys corresponding to $S$.

The scheme uses hash functions $\mathsf{H}_0 : \{0,1\}^* \to \mathbb{G}_1$, $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{Z}_q$, and $\mathsf{H}_2 : \{0,1\}^* \to \mathbb{G}_1$. Parameter generation, key generation, and key aggregation are the same as for the aggregate multi-signature scheme in Section 3.

**Group Setup.** $\mathsf{GSetup}(sk_i, \mathcal{PK} = \{pk_1, \ldots, pk_n\})$ checks that $pk_i \in \mathcal{PK}$ and that $i$ is the index of $pk_i$ in $\mathcal{PK}$. Signer $i$ computes the aggregate public key $apk \leftarrow \mathsf{KAg}(\mathcal{PK})$ as well as $a_i \leftarrow \mathsf{H}_1(pk_i, \mathcal{PK})$. It then sends $\mu_{j,i} = \mathsf{H}_2(apk, j)^{a_i \cdot sk_i}$ to signer $j$ for $j \neq i$, or simply publishes these values. After having received $\mu_{i,j}$ from all other signers $j \neq i$, it computes $\mu_{i,i} \leftarrow \mathsf{H}_2(apk, i)^{a_i \cdot sk_i}$ and returns the membership key $mk_i \leftarrow \prod_{j=1}^{n} \mu_{i,j}$. Note that if all signers behave honestly, we have that

$$e(g_1, mk_i) = e(apk, \mathsf{H}_2(apk, i)).$$

In other words, this $mk_i$ is a valid multi-signature on the message $\mathsf{H}_2(apk, i))$ by all $n$ parties, as defined in the scheme in Section 3.1.

**Signing.** $\mathsf{Sign}(par, \mathcal{PK}, S, sk_i, mk_i, m)$ computes $apk \leftarrow \mathsf{KAg}(\mathcal{PK})$ and

$$s_i \leftarrow \mathsf{H}_0(apk, m)^{sk_i} \cdot mk_i \ ,$$

and sends $(pk_i, s_i)$ to a designated combiner (either one of the members of $S$ or an external party). The combiner computes

$$pk \leftarrow \prod_{j \in S} pk_j, \qquad s \leftarrow \prod_{j \in S} s_j,$$

and outputs the multisignature $\sigma := (pk, s)$. Note that the set $S$ does not have to be fixed at the beginning of the protocol, but can be determined as partial signatures are collected.

**Verification.** $\mathsf{Vf}(par, apk, S, m, \sigma)$ parses $\sigma$ as $(pk, s)$ and outputs 1 iff

$$e(\mathsf{H}_0(apk, m), pk) \cdot e(\prod_{j \in S} \mathsf{H}_2(apk, j), apk) \stackrel{?}{=} e(s, g_2)$$

and $S$ is a set authorized to sign.

The presented ASM scheme satisfies correctness. If parties honestly execute the group setup and and signing protocols, we have $pk = g_2^{\sum_{i \in S} sk_i}$, $apk = g_2^{\sum_{i=1,\ldots,n} a_i \cdot pk_i}$, and $s = \mathsf{H}_0(apk, m)^{\sum_{i \in S} sk_i} \cdot \prod_{i \in S} \mathsf{H}_2(apk, i)^{\sum_{j \in 1,\ldots,n} a_j \cdot sk_j}$, which passes verification:

$$\begin{aligned}
e(s, g_2) &= e\big(\mathsf{H}_0(apk, m)^{\sum_{i \in S} sk_i} \cdot \prod_{i \in S} \mathsf{H}_2(apk, i)^{\sum_{j \in 1,\ldots,n} a_j \cdot sk_j}, g_2\big) \\
&= e\big(\mathsf{H}_0(apk, m), pk\big) \cdot e\big(\prod_{i \in S} \mathsf{H}_2(apk, i), g_2^{\sum_{j \in 1,\ldots,n} a_j \cdot sk_j}\big) \\
&= e\big(\mathsf{H}_0(apk, m), pk\big) \cdot e\big(\prod_{i \in S} \mathsf{H}_2(apk, i), apk\big)
\end{aligned}$$

### 4.3 Security of our ASM Scheme

**Theorem 3.** *Our ASM scheme is unforgeable under the hardness of the computational $\psi$-co-Diffie-Hellman problem in the random-oracle model. More precisely, it is $(\tau, q_S, q_H, \epsilon)$-unforgeable in the random-oracle model if $q > 8q_H/\epsilon$ and if $\psi$-co-CDH is $(\tau + q_H \cdot \max(\tau_{\exp_2^l}, \tau_{\exp_1^2}) + q_G \cdot (l-1)\tau_{\exp_1} + q_S \cdot (\tau_{\exp_2^l} + \tau_{\exp_1}) + 2\tau_{\text{pair}} + \tau_{\exp_1^3}) \cdot 8q_H^2/((1-(q_S+q_H)/q) \cdot \epsilon) \cdot \ln(8q_H/((1-(q_S+q_H)/q) \cdot \epsilon)), (1-(q_S+q_H)/q) \cdot \epsilon/(8q_H))$-hard, where $l$ is the maximum number of signers involved in any group setup, $\tau_{\exp_1}$ and $\tau_{\exp_2}$ denote the time required to compute exponentiations in $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively, and $\tau_{\exp_1^i}$ and $\tau_{\exp_2^i}$ denote the time required to compute $i$-multiexponentiations in $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively, and $\tau_{\text{pair}}$ denotes the time required to compute a pairing operation.*

*Proof.* Given a forger $\mathcal{F}$ against the ASM scheme, we construct a wrapper algorithm $\mathcal{A}$ that can be used by the generalized forking algorithm $\mathcal{GF}_{\mathcal{A}}$. We then give an adversary $\mathcal{B}$ that can solve the $\psi$-co-CDH problem by running $\mathcal{GF}_{\mathcal{A}}$. The proof essentially combines techniques related to the non-extractability of BGLS aggregate signatures [12, 20] with Maxwell et al. 's key aggregation technique [35].

Given a forger $\mathcal{F}$, consider the following algorithm $\mathcal{A}$. On input $in = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2, A = g_1^\alpha, B_1 = g_1^\beta, B_2 = g_2^\beta)$ and randomness $f = (\rho, h_1, \ldots, h_{q_H})$, and given access to a homomorphism oracle $\mathcal{O}^\psi(\cdot)$, $\mathcal{A}$ proceeds as follows. It guesses a random index $k \xleftarrow{\$} \{1, \ldots, q_H\}$ and runs $\mathcal{F}$ on input $par \leftarrow (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2)$ and $pk^* \leftarrow B_2$, answering its oracle queries using initially empty lists $L_0, L_2$ as follows:

- $\mathsf{H}_1(x)$: If $x$ can be parsed as $(pk, \mathcal{PK})$ and $pk^* \in \mathcal{PK}$ and $\mathcal{F}$ did not make any previous query $\mathsf{H}_1(pk', \mathcal{PK})$, then it sets $\mathsf{H}_1(pk^*, \mathcal{PK})$ to the next unused value $h_i$ and, for all $pk \in \mathcal{PK} \setminus \{pk^*\}$, assigns a random value in $\mathbb{Z}_q$ to $\mathsf{H}_1(pk, \mathcal{PK})$. Let $apk \leftarrow \prod_{pk \in \mathcal{PK}} pk^{\mathsf{H}_1(pk, \mathcal{PK})}$ and let $i$ be the index of $pk^*$ in $\mathcal{PK}$. If $\mathcal{F}$ previously made any random-oracle or signing queries involving $apk$, then we say that event **bad** happened and $\mathcal{A}$ gives up by outputting $(0, \perp)$. If $\mathsf{H}_1(x)$ did not yet get assigned a value, then $\mathcal{A}$ assigns a random value $\mathsf{H}_1(x) \xleftarrow{\$} \mathbb{Z}_q$.
- $\mathsf{H}_2(x)$: If $x$ can be parsed as $(apk, i)$ such that there exist defined entries for $\mathsf{H}_1$ such that $apk = \prod_{pk \in \mathcal{PK}} pk^{\mathsf{H}_1(pk, \mathcal{PK})}$, $pk^* \in \mathcal{PK}$, and $i$ is the index of $pk^*$ in $\mathcal{PK}$, then $\mathcal{A}$ chooses $r \xleftarrow{\$} \mathbb{Z}_q$, adds $((apk, i), r, 1)$ to $L_2$ and assigns $\mathsf{H}_2(x) \leftarrow g_1^r A^{-1/a_i}$ where $a_i = \mathsf{H}_1(pk^*, \mathcal{PK})$. If not, then $\mathcal{A}$ chooses $r \xleftarrow{\$} \mathbb{Z}_q$, adds $(x, r, 0)$ to $L_2$ and assigns $\mathsf{H}_2(x) \leftarrow g_1^r$.
- $\mathsf{H}_0(x)$: If this is $\mathcal{F}$'s $k$-th random-oracle query, then $\mathcal{A}$ sets $m^* \leftarrow x$, hoping that $\mathcal{F}$ will forge on message $m^*$. It then chooses $r \xleftarrow{\$} \mathbb{Z}_q$, adds $(m^*, r, 1)$ to $L_0$ and assigns $\mathsf{H}_0(m^*) \leftarrow g_1^r$. If this is not $\mathcal{F}$'s $k$-th random-oracle query, then $\mathcal{A}$ chooses $r \xleftarrow{\$} \mathbb{Z}_q$, adds $(x, r, 0)$ to $L_0$ and assigns $\mathsf{H}_0(x) \leftarrow g_1^r A$.
- $\mathsf{GSetup}(\mathcal{PK})$: If $pk^* \notin \mathcal{PK}$, then $\mathcal{A}$ ignores this query. Otherwise, it computes $apk \leftarrow \prod_{pk \in \mathcal{PK}} pk^{\mathsf{H}_1(pk, \mathcal{PK})}$, internally simulating the random-oracle queries $\mathsf{H}_1(pk, \mathcal{PK})$ if needed. It also internally simulates queries $\mathsf{H}_2(apk, j)$ for $j =$

$1, \ldots, |\mathcal{PK}|, j \neq i$, to create entries $((apk, j), r_j, 0) \in L_2$, as well as $a_i \leftarrow \mathsf{H}_1(pk^*, \mathcal{PK})$, where $i$ is the index of $pk^*$ in $\mathcal{PK}$. Since $\mathsf{H}_2(apk, j) = g_1^{r_j}$, $\mathcal{A}$ can simulate the values $\mu_{j,i} = \mathsf{H}_2(apk, j)^{a_i \cdot sk^*} = \mathsf{H}_2(apk, j)^{a_i \cdot \beta}$ for $j \neq i$ as $\mu_{j,i} \leftarrow B_1^{a_i \cdot r_j}$.
After having received $\mu_{i,j}$ from all other signers $j \neq i$, $\mathcal{A}$ internally stores $\mu_{apk} \leftarrow \prod_{j \neq i} \mu_{i,j}$.

– $\mathsf{Sign}(\mathcal{PK}, S, m)$: If $\mathcal{F}$ did not perform group setup for $\mathcal{PK}$, then $\mathcal{A}$ ignores this query. If $m = m^*$, then $\mathcal{A}$ gives up by outputting $(0, \perp)$. Otherwise, it recomputes $apk \leftarrow \mathsf{KAg}(\mathcal{PK})$ and looks up $((apk, m), r_0, 0) \in L_0$ and $((apk, i), r_2, 1) \in L_2$, internally simulating queries $\mathsf{H}_0(apk, m)$ and $\mathsf{H}_2(apk, i)$ to create them if needed, where $i$ is the index of $pk^*$ in $\mathcal{PK}$. Now $\mathcal{A}$ must simulate the partial signature $s_i = \mathsf{H}_0(apk, m)^{sk^*} \cdot \mu_{apk} \cdot \mathsf{H}_2(apk, i)^{a_i \cdot sk^*}$, where $a_i = \mathsf{H}_1(pk^*, \mathcal{PK})$. From the way $\mathcal{A}$ responded to random-oracle queries, we know that $\mathsf{H}_0(apk, m) = g_1^{r_0} A = g_1^{r_0 + \alpha}$ and $\mathsf{H}_2(apk, i) = g_1^{r_2} A^{-1/a_i} = g_1^{r_2 - \alpha/a_i}$, so that $\mathcal{A}$ has to simulate $s_i = g_1^{\beta(r_0+\alpha)} \cdot \mu_{apk} \cdot g_1^{\beta(a_i r_2 - \alpha)} = \mu_{apk} \cdot g_1^{\beta(r_0 + a_i r_2)}$, which it can easily compute as $s_i \leftarrow \mu_{apk} \cdot B_1^{r_0 + a_i r_2}$.

When $\mathcal{F}$ eventually outputs its forgery $(\mathcal{PK}, S, m, \sigma)$, $\mathcal{A}$ recomputes $apk^* \leftarrow \mathsf{KAg}(\mathcal{PK}) = \prod_{j=1}^{|\mathcal{PK}|} pk_j^{a_j}$, where $pk_j$ is the $j$-th public key in $\mathcal{PK}$ and $a_j = \mathsf{H}_1(apk, j)$, and checks that the forgery is valid, i.e., $\mathsf{Vf}(par, apk, S, m, \sigma) = 1$, $pk^* \in \mathcal{PK}$, $i \in S$ where $i$ is the index of $pk^* \in \mathcal{PK}$, and $\mathcal{F}$ never made a signing query for $m$. If any of these checks fails, $\mathcal{A}$ outputs $(0, \perp)$. If $m \neq m^*$, then $\mathcal{A}$ also outputs $(0, \perp)$. Else, observe that $\sigma = (pk, s)$ such that

$$s \;=\; \mathsf{H}_0(apk, m^*)^{\log pk} \cdot \prod_{j \in S} \mathsf{H}_2(apk, j)^{\log apk^*} \;.$$

Because of how $\mathcal{A}$ simulated $\mathcal{F}$'s random-oracle queries, it can look up $((apk^*, m^*), r_0, 1) \in L_0$, $((apk^*, j), r_{2,j}, 0) \in L_2$ for $j \in S \setminus \{i\}$, and $((apk^*, i), r_{2,i}, 1) \in L_2$, where $i$ is the index of $pk^*$ in $\mathcal{PK}$, such that

$$\begin{aligned} \mathsf{H}_0(apk, m^*) &= g_1^{r_0} \\ \mathsf{H}_2(apk, j) &= g_1^{r_{2,j}} \text{ for } j \in S \setminus \{i\} \\ \mathsf{H}_2(apk, i) &= g_1^{r_{2,i}} A^{-1/a_i} \end{aligned}$$

so that we have that

$$s = g_1^{\log pk \cdot r_0} \cdot g_1^{\log apk^* \cdot \sum_{j \in S} r_{2,j}} \cdot A^{-\log apk^*/a_i}$$

If we let

$$t \leftarrow \left(s^{-1} \cdot \mathcal{O}^{\psi}(pk)^{r_0} \cdot \mathcal{O}^{\psi}(apk^*)^{\sum_{j \in S} r_{2,j}}\right)^{a_i}$$

then we have that

$$t \;=\; A^{\log apk^*} \;=\; A^{\sum_{j=1}^{|\mathcal{PK}|} a_j \log pk_j} \;.$$

If $I$ is the index such that $\mathsf{H}(pk^*, \mathcal{PK}) = h_I$, then algorithm $\mathcal{A}$ outputs $(I, (t, \mathcal{PK}, a_1, \ldots, a_n))$.

$\mathcal{A}$'s runtime is $\mathcal{F}$'s runtime plus the additional computation $\mathcal{A}$ performs. Let $q_{\mathrm{H}}$ denote the total hash queries $\mathcal{F}$ makes, i.e., the queries to $\mathsf{H}_0$, $\mathsf{H}_1$, and $\mathsf{H}_2$ combined. To answer a $\mathsf{H}_1$ query, $\mathcal{A}$ computes $apk$ which costs at most $\tau_{\exp_2^l}$ for groups consisting of up to $l$ signers. To answer $\mathsf{H}_0$ and $\mathsf{H}_2$ queries, $\mathcal{A}$ performs at most $\tau_{\exp_1^2}$. $\mathcal{A}$ therefore spends at most $q_{\mathrm{H}} \cdot \max(\tau_{\exp_2^l}, \tau_{\exp_1^2})$ answering hash queries. For every group-setup query with $l$ signers, $\mathcal{A}$ computes $apk$ costing $\tau_{\exp_2^l}$, and $\mathcal{A}$ computes $\mu_{j,i}$ costing $(l-1)\tau_{\exp_1}$, meaning $\mathcal{A}$ spends $q_{\mathrm{G}} \cdot (l-1)\tau_{\exp_1}$ answering group setup queries. For signing queries with a $\mathcal{PK}$ of size at most $l$, $\mathcal{A}$ computes $apk$ costing time $\tau_{\exp_2^l}$, and one exponentiation in $\mathbb{G}_1$ costing $\tau_{\exp_1}$, giving a total of $q_{\mathrm{S}} \cdot (\tau_{\exp_2^l} + \tau_{\exp_1})$. Finally, $\mathcal{A}$ computes the output values, which involves verifying the forgery (costing $2\tau_{\mathrm{pair}}$) and computing $t$ (costing $\tau_{\exp_1^3}$), giving $\mathcal{A}$ a total runtime of $\tau + q_{\mathrm{H}} \cdot \max(\tau_{\exp_2^l}, \tau_{\exp_1^2}) + q_{\mathrm{G}} \cdot (l-1)\tau_{\exp_1} + q_{\mathrm{S}} \cdot (\tau_{\exp_2^l} + \tau_{\exp_1}) + 2\tau_{\mathrm{pair}} + \tau_{\exp_1^3}$. $\mathcal{A}$ successfully outputs if the **bad** event does not happen, it guesses the index of the forgery correctly, and $\mathcal{F}$ successfully forges. Event **bad** happens with probability at most $(q_{\mathrm{S}} + q_{\mathrm{H}})/q$ for every hash query, so it happens with probability $q_{\mathrm{H}}(q_{\mathrm{S}} + q_{\mathrm{H}})/q$. $\mathcal{A}$ guesses the forgery index correctly with prbability $1/q_{\mathrm{H}}$, and $\mathcal{F}$ forges with probability $\epsilon$, giving $\mathcal{A}$ success probability $(1 - (q_{\mathrm{S}} + q_{\mathrm{H}})/q) \cdot \epsilon/q_{\mathrm{H}}$.

Using the generalized forking lemma from Lemma 1, we can build an algorithm $\mathcal{B}$ that solves the $\psi$-co-CDH problem by, on input $(A = g_1^{\alpha}, B_1 = g_1^{\beta}, B_2 = g_2^{\beta})$, running $\mathcal{GF}_{\mathcal{A}}(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_{\mathrm{t}}, e, g_1, g_2, A, B_1, B_2)$ to obtain two outputs $(I, (t, \mathcal{PK}, a_1, \ldots, a_n))$ and $(I, (t', \mathcal{PK}', a_1', \ldots, a_n'))$, giving $\mathcal{GF}_{\mathcal{A}}$ access to the homomorphism oracle $\mathcal{O}^{\psi}(\cdot)$ offered by $\psi$-co-CDH. Since the two executions of $\mathcal{A}$ are identical up to the first query $\mathsf{H}_1(pk, \mathcal{PK})$ involving the forged set of signers $\mathcal{PK}$, we have that $\mathcal{PK} = \mathcal{PK}'$. Also, from the way $\mathcal{A}$ assigns values to outputs of $\mathsf{H}_1$, one can see that $a_j = a_j'$ for $j \neq i$ and $a_i \neq a_i'$, where $i$ is the index of $pk^*$ in $\mathcal{PK}$. We therefore have that

$$t/t' = A^{(a_i - a_i') \log pk^*} = g_1^{\alpha\beta(a_i - a_i')} ,$$

so that $\mathcal{B}$ can output its solution $g_1^{\alpha \cdot \beta} = (t/t')^{1/(a_i - a_i')}$.

Using Lemma 1, we know that if $q > 8q_{\mathrm{H}}/\epsilon$, then $\mathcal{B}$ runs in time at most $(\tau + q_{\mathrm{H}} \cdot \max(\tau_{\exp_2^l}, \tau_{\exp_1^2}) + q_{\mathrm{G}} \cdot (l-1)\tau_{\exp_1} + q_{\mathrm{S}} \cdot (\tau_{\exp_2^l} + \tau_{\exp_1}) + 2\tau_{\mathrm{pair}} + \tau_{\exp_1^3}) \cdot 8q_{\mathrm{H}}^2/((1 - (q_{\mathrm{S}} + q_{\mathrm{H}})/q) \cdot \epsilon) \cdot \ln(8q_{\mathrm{H}}/((1 - (q_{\mathrm{S}} + q_{\mathrm{H}})/q) \cdot \epsilon))$ and succeeds with probability $(1 - (q_{\mathrm{S}} + q_{\mathrm{H}})/q) \cdot \epsilon/(8q_{\mathrm{H}})$.

## 5 A Scheme from Discrete Logarithms

The basic key aggregation technique of our pairing-based schemes is due to Maxwell et al. [35], who presented a Schnorr-based multi-signature scheme that uses the same key aggregation technique and that also saves one round of interaction in the signing protocol with respect to Bellare-Neven's scheme [9]. Unfortunately, their security proof was found to be flawed due to a problem in the simulation of the signing protocol [21]. In the following, we recover Maxwell et al.'s key aggregation technique for ordinary (i.e., non-pairing-friendly) curves

by combining it with Bellare-Neven's preliminary round of hashes. The resulting scheme achieves the same space savings as Maxwell et al.'s original scheme, but is provably secure under the hardness of the discrete-logarithm assumption. Independently from our work, Maxwell et al. [36] revised their work to use the same protocol we present here.

### 5.1 Description of our Discrete-Logarithm Scheme

Our discrete-logarithm based multi-signature scheme $\mathcal{MSDL}$ uses hash functions $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2 : \{0,1\}^* \to \mathbb{Z}_q$, which can be instantiated from a single hash function using domain separation.

**Parameters generation.** $\mathsf{Pg}(\kappa)$ sets up a group $\mathbb{G}$ of order $q$ with generator $g$, where $q$ is a $\kappa$-bit prime, and output $par \leftarrow (\mathbb{G}, g, q)$.

**Key generation.** The key generation algorithm $\mathsf{Kg}(par)$ chooses $sk \xleftarrow{\$} \mathbb{Z}_q$ and computes $pk \leftarrow g^{sk}$. Output $(pk, sk)$.

**Key Aggregation.** $\mathsf{KAg}(\{pk_1, \ldots, pk_n\})$ outputs

$$apk \leftarrow \prod_{i=1}^{n} pk_i^{\mathsf{H}_1(pk_i, \{pk_1, \ldots, pk_n\})} .$$

**Signing.** Signing is an interactive three-round protocol. On input $\mathsf{Sign}(par, \{pk_1, \ldots, pk_n\}, sk, m)$, signer $i$ behaves as follows:

Round 1. Choose $r_i \xleftarrow{\$} \mathbb{Z}_q$ and compute $R_i \leftarrow g^{r_i}$. Let $t_i \leftarrow \mathsf{H}_2(R_i)$. Send $t_i$ to all other signers corresponding to $pk_1, \ldots, pk_n$ and wait to receive $t_j$ from all other signers $j \neq i$.

Round 2. Send $R_i$ to all other signers corresponding to $pk_1, \ldots, pk_n$ and wait to receive $R_j$ from all other signers $j \neq i$. Check that $t_j = \mathsf{H}_2(R_j)$ for all $j = 1, \ldots, n$.

Round 3. Compute $apk \leftarrow \mathsf{KAg}(\{pk_1, \ldots, pk_n\})$ and let $a_i \leftarrow \mathsf{H}_1(pk_i, \{pk_1, \ldots, pk_n\})$. Note that when multiple messages are signed with the same set of signers, $apk$ and $a_i$ can be stored rather than recomputed.

Compute $\bar{R} \leftarrow \prod_{j=1}^{n} R_j$ and $c \leftarrow \mathsf{H}_0(\bar{R}, apk, m)$. Compute $s_i \leftarrow r_i + c \cdot sk_i \cdot a_i \bmod q$. Send $s_i$ to all other signers and wait to receive $s_j$ from all other signers $j \neq i$. Compute $s \leftarrow \sum_{j=1}^{n} s_j$ and output $\sigma \leftarrow (\bar{R}, s)$ as the final signature.

**Verification.** $\mathsf{Vf}(par, apk, m, \sigma)$ parses $\sigma$ as $(\bar{R}, s) \in \mathbb{G} \times \mathbb{Z}_q$, computes $c \leftarrow \mathsf{H}_0(\bar{R}, apk, m)$ and outputs 1 iff $g^s \cdot apk^{-c} \stackrel{?}{=} \bar{R}$.

The scheme allows for more efficient batch verification, which allows a verifier to check the validity of $n$ signatures with one $3n$-multi-exponentiation instead of $n$ 2-multi-exponentiations. To verify that every signature in a list of $n$ signatures $\{(apk_i, m_i, (\bar{R}_i, s_i))\}_{i=1}^{n}$ is valid, compute $c_i \leftarrow \mathsf{H}_1(\bar{R}, apk_i, m_i)$, pick $\alpha_i \xleftarrow{\$} \mathbb{Z}_q$ for $i = 1, \ldots, n$, and accept iff

$$\prod_{i=1}^{n} g^{\alpha_i s_i} apk_i^{-\alpha_i c_i} \bar{R}_i^{-\alpha_i} \stackrel{?}{=} 1_{\mathbb{G}} .$$

### 5.2 Security Proof

The security proof follows that of [35] by applying the forking lemma twice: once by forking on a random-oracle query $\mathsf{H}_0(\bar{R}, apk, m)$ to obtain two forgeries from which the discrete logarithm $w$ of $apk$ can be extracted, and then once again by forking on a query $\mathsf{H}_1(pk_i, \{pk_1, \ldots, pk_n\}$ to obtain two such pairs $(apk, w)$ and $(apk', w')$ from which the discrete logarithm of the target public key can be extracted.

**Theorem 4.** *$\mathcal{MSDL}$ is an unforgeable multisignature scheme (as defined in Def 4) in the random-oracle model if the discrete log problem is hard. More precisely, $\mathcal{MSDL}$ is $(\tau, q_S, q_H, \epsilon)$-unforgeable in the random-oracle model if $q > 8q_H/\epsilon$ and if discrete log is $((\tau + 4lq_T \cdot \tau_{exp} + O(lq_T)) \cdot 512q_T^2/(\epsilon-\delta) \cdot \ln^2(64/(\epsilon-\delta)), (\epsilon-\delta)/64)$-hard, where $l$ is the maximum number of signers involved in a single multisignature, $q_T = q_H + q_S + 1$, $\delta = 4lq_T^2/q$, and $\tau_{exp}$ is the time required to compute an exponentiation in $\mathbb{G}$.*

*Proof.* We first wrap the forger $\mathcal{F}$ into an algorithm $\mathcal{A}$ that can be used in the forking lemma. We then describe an algorithm $\mathcal{B}$ that runs $\mathcal{GF}_{\mathcal{A}}$ to obtain an aggregated public key $apk$ and its discrete logarithm $w$. We finally describe a discrete-logarithm algorithm $\mathcal{D}$ that applies the forking lemma again to $\mathcal{B}$ by running $\mathcal{GF}_{\mathcal{B}}$ and using its output to compute the wanted discrete logarithm.

Algorithm $\mathcal{A}$, on input $in = (y, h_{1,1}, \ldots, h_{1,q_H})$ and randomness $f = (\rho, h_{0,1}, \ldots, h_{0,q_H})$ runs $\mathcal{F}$ on input $pk^* = y$ and random tape $\rho$, responding to its queries as follows:

- $\mathsf{H}_0(\bar{R}, apk, m)$: Algorithm $\mathcal{A}$ returns the next unused value $h_{0,i}$ from its randomness $f$.
- $\mathsf{H}_1(pk_i, \mathcal{PK})$: If $pk^* \in \mathcal{PK}$ and $\mathcal{F}$ did not make any previous query $\mathsf{H}_1(pk', \mathcal{PK})$, then $\mathcal{A}$ sets $\mathsf{H}_1(pk^*, \mathcal{PK})$ to the next unused value $h_{1,i}$ from its input and assigns $\mathsf{H}_1(pk, \mathcal{PK}) \xleftarrow{\$} \mathbb{Z}_q$ for all $pk \in \mathcal{PK} \backslash \{pk^*\}$. Let $apk \leftarrow \prod_{pk \in \mathcal{PK}} pk^{\mathsf{H}_1(pk, \mathcal{PK})}$. If $\mathcal{F}$ already made any random-oracle or signing queries involving $apk$, then we say that event $\mathbf{bad}_1$ happened and $\mathcal{A}$ gives up by outputting $(0, \perp)$.
- $\mathsf{H}_2(R)$: $\mathcal{A}$ simply chooses a random value $t \xleftarrow{\$} \mathbb{Z}_q$ and assigns $\mathsf{H}_2(R) \leftarrow t$. If there exists another $R' \neq R$ such that $\mathsf{H}_2(R') = t$, or if $t$ has already been used (either by $\mathcal{F}$ or in $\mathcal{A}$'s simulation) in the first round of a signing query, then we say that event $\mathbf{bad}_2$ happened and $\mathcal{A}$ gives up by outputting $(0, \perp)$.
- $\mathsf{Sign}(\mathcal{PK}, m)$: Algorithm $\mathcal{A}$ first computes $apk \leftarrow \mathsf{KAg}(\mathcal{PK})$, simulating internal queries to $\mathsf{H}_1$ as needed. In the first round of the protocol, $\mathcal{A}$ returns a random value $t_i \xleftarrow{\$} \mathbb{Z}_q$.
  After receiving values $t_j$ from all other signers, it looks up the corresponding values $R_j$ such that $\mathsf{H}_2(R_j) = t_j$. If not all such values can be found, then $\mathcal{A}$ sends $R_i \xleftarrow{\$} \mathbb{G}$ to all signers; unless $\mathbf{bad}_2$ happens, the signing protocol finishes in the next round. If all values $R_j$ are found, then $\mathcal{A}$ chooses $s_i, c \xleftarrow{\$} \mathbb{Z}_q$, simulates an internal query $a_i \leftarrow \mathsf{H}_1(pk^*, \mathcal{PK})$, computes $R_i \leftarrow g^{s_i} pk^{*-a_i \cdot c}$ and $\bar{R} \leftarrow \prod_{j=1}^{n} R_j$, assigns $\mathsf{H}_2(R_i) \leftarrow t_i$ and $\mathsf{H}_0(\bar{R}, apk, m) \leftarrow c$, and sends $R_i$ to all signers. If the latter assignment failed because the entry was taken,

we say that event $\mathbf{bad}_3$ happened and $\mathcal{A}$ gives up by outputting $(0, \perp)$. (Note that the first assignment always succeeds, unless $\mathbf{bad}_2$ occurs.)
After it received the values $R_j$ from all other signers, $\mathcal{A}$ sends $s_i$.

When $\mathcal{F}$ outputs a valid forgery $(\bar{R}, s)$ on message $m$ for a set of signers $\mathcal{PK} = \{pk_1, \ldots, pk_n\}$, $\mathcal{A}$ computes $apk \leftarrow \mathsf{KAg}(\mathcal{PK})$, $c \leftarrow \mathsf{H}_0(\bar{R}, apk, m)$, and $a_i \leftarrow \mathsf{H}_1(pk_i, \mathcal{PK})$ for $i = 1, \ldots, n$. If $j$ is the index such that $c = h_{0,j}$, then $\mathcal{A}$ returns $(j, (\bar{R}, c, s, apk, \mathcal{PK}, a_1, \ldots, a_n))$.

Note that $apk = \prod_{i=1}^{n} pk_i^{a_i}$ and, because the forgery is valid, $g^s = \bar{R} \cdot apk^c$. If $\mathcal{F}$ is a $(\tau, q_\mathrm{S}, q_\mathrm{H}, \epsilon)$-forger, then $\mathcal{A}$ succeeds with probability

$$\begin{aligned}
\epsilon_{\mathcal{A}} &= \Pr[\mathcal{F} \text{ succeeds} \wedge \overline{\mathbf{bad}_1} \wedge \overline{\mathbf{bad}_2} \wedge \overline{\mathbf{bad}_3}] \\
&\geq \Pr[\mathcal{F} \text{ succeeds}] - \Pr[\mathbf{bad}_1] - \Pr[\mathbf{bad}_2] - \Pr[\mathbf{bad}_3] \\
&\geq \epsilon - \frac{q_\mathrm{H}(q_\mathrm{H} + q_\mathrm{S} + 1)}{q} - \left( \frac{(q_\mathrm{H} + q_\mathrm{S})^2}{2q} + \frac{l q_\mathrm{H} q_\mathrm{S}}{q} \right) - \frac{q_\mathrm{H}(q_\mathrm{H} + q_\mathrm{S} + 1)}{q} \\
&\geq \epsilon - \frac{4 l q_\mathrm{T}^2}{q} \;=\; \epsilon - \delta
\end{aligned}$$

where $q_\mathrm{T} = q_\mathrm{H} + q_\mathrm{S} + 1$ and $\delta = 4l(q_\mathrm{H} + q_\mathrm{S} + 1)^2/q$. The running time of $\mathcal{A}$ is $\tau_{\mathcal{A}} = \tau + 4 l q_\mathrm{T} \cdot \tau_{\exp} + O(l q_\mathrm{T})$.

We now construct algorithm $\mathcal{B}$ that runs the forking algorithm $\mathcal{GF}_{\mathcal{A}}$ on algorithm $\mathcal{A}$, but that itself is a wrapper algorithm around $\mathcal{GF}_{\mathcal{A}}$ that can be used in the forking lemma. Algorithm $\mathcal{B}$, on input $in = y$ and randomness $f = (\rho, h_{1,1}, \ldots, h_{1,q_\mathrm{H}})$, runs $\mathcal{GF}_{\mathcal{A}}$ on input $in' = (y, h_{1,1}, \ldots, h_{1,q_\mathrm{H}})$ to obtain output

$$\big(j,\ (\bar{R}, c, s, apk, \mathcal{PK}, a_1, \ldots, a_n),\ \ (\bar{R}', c', s', apk', \mathcal{PK}', a_1', \ldots, a_n')\big).$$

In its two executions by $\mathcal{GF}_{\mathcal{A}}$, $\mathcal{F}$'s view is identical up to the $j$-th $\mathsf{H}_0$ query $\mathsf{H}_0(\bar{R}, apk, m)$, meaning that also the arguments of that query are identical in both executions, and hence $\bar{R} = \bar{R}'$ and $apk = apk$. From the way $\mathcal{A}$ answers $\mathcal{F}$'s $\mathsf{H}_1$ queries by aborting when $\mathbf{bad}_1$ happens, the fact that $apk = apk'$ also means that $\mathcal{PK} = \mathcal{PK}'$ and that $a_i = a_i'$ for $i = 1, \ldots, n$. The forking algorithm moreover guarantees that $c \neq c'$.

By dividing the two verification equations $g^s = \bar{R} \cdot apk^c$ and $g^{s'} = \bar{R}' \cdot apk'^{c'} = \bar{R} \cdot apk^{c'}$, one can see that $w \leftarrow (s - s')/(c - c') \bmod q$ is the discrete logarithm of $apk$. If $i$ is the index such that $\mathsf{H}_1(pk^*, \mathcal{PK}) = h_{1,i}$, then $\mathcal{B}$ outputs $(i, (w, \mathcal{PK}, a_1, \ldots, a_n))$. It does so whenever $\mathcal{GF}_{\mathcal{A}}$ is successful, which according to Lemma 1 occurs with probability $\epsilon_{\mathcal{B}}$ and running time $\tau_{\mathcal{B}}$:

$$\begin{aligned}
\epsilon_{\mathcal{B}} &\geq \frac{\epsilon_{\mathcal{A}}}{8} \;\geq\; \frac{\epsilon - \delta}{8} \\
\tau_{\mathcal{B}} &= \tau_{\mathcal{A}} \cdot 8 q_\mathrm{H} / \epsilon_{\mathcal{A}} \cdot \ln(8/\epsilon_{\mathcal{A}}) \\
&\leq (\tau + 4 l q_\mathrm{T} \cdot \tau_{\exp} + O(l q_\mathrm{T})) \cdot \frac{8 q_\mathrm{T}}{\epsilon - \delta} \cdot \ln \frac{8}{\epsilon - \delta} \ .
\end{aligned}$$

Now consider the discrete-logarithm algorithm $\mathcal{D}$ that, on input $y$, runs $\mathcal{GF}_{\mathcal{B}}$ on input $y$ to obtain output $(i, (w, \mathcal{PK}, a_1, \ldots, a_n), (w, \mathcal{PK}', a'_1, \ldots, a'_n))$. Both executions of $\mathcal{B}$ in $\mathcal{GF}_{\mathcal{B}}$ are identical up to the $i$-th $\mathsf{H}_1$ query $\mathsf{H}_1(pk, \mathcal{PK})$, so we have that $\mathcal{PK} = \mathcal{PK}'$. Because $\mathcal{A}$ immediately assigns outputs of $\mathsf{H}_1$ for all public keys in $\mathcal{PK}$ as soon as the first query for $\mathcal{PK}$ is made, and because it uses $h_{1,i}$ to answer $\mathsf{H}_1(pk^*, \mathcal{PK})$, we also have that $a_i = a'_i$ for $pk_i \neq pk^*$ and $a_i \neq a'_i$ for $pk_i = pk^*$. By dividing the equations $apk = \prod_{i=1}^{n} pk_i^{a_i} = g^w$ and $apk' = \prod_{i=1}^{n} pk_i^{a'_i} = g^{w'}$, one can see that $\mathcal{D}$ can compute the discrete logarithm of $pk^* = y$ as $x \leftarrow (w - w')/(a_i - a'_i) \bmod q$ , where $i$ is the index such that $pk_i = pk^*$. By Lemma 1, it can do so with the following success probability $\epsilon_{\mathcal{D}}$ and running time $\tau_{\mathcal{D}}$:

$$\begin{aligned}
\epsilon_{\mathcal{D}} &\geq \frac{\epsilon_{\mathcal{B}}}{8} \geq \frac{\epsilon - \delta}{64} \\
\tau_{\mathcal{D}} &= \tau_{\mathcal{B}} \cdot 8q_{\mathrm{H}}/\epsilon_{\mathcal{B}} \cdot \ln(8/\epsilon_{\mathcal{B}}) \\
&\leq (\tau + 4lq_{\mathrm{T}} \cdot \tau_{\exp} + O(lq_{\mathrm{T}})) \cdot \frac{512q_{\mathrm{T}}^2}{\epsilon - \delta} \cdot \ln^2 \frac{64}{\epsilon - \delta} \ .
\end{aligned}$$

## 6 Schemes with Proofs of Possession

As first observed by Ristenpart and Yilek [47], rogue-key attacks in multi-signatures can for some schemes be prevented by letting signers add a so-called *proof of possession* (PoP) to their public keys, which is simply a signature on their own public key. In particular, they showed this to be the case for Boldyreva's [10, 13] and Lu et al.'s [32] multisignatures. We show that all of our schemes can also be strengthened with PoPs instead of with Maxwell et al.'s key aggregation [35], yielding an even simpler key aggregation where, after having verified the PoPs, the aggregate key is simply the product of all individual keys.

The main disadvantage of schemes using PoPs is of course the growth in public key size due to the PoPs. There are situations, however, where the size of individual public keys is not crucial. For example, Multisig addresses in principle do not need to reveal the individual public keys of all signers, as long as the aggregate public key is known and agreed upon by all signers. Signers can build trust in the aggregate key by exchanging and verifying each others' PoPs when the Multisig wallet is created, but only advertise the aggregate key to the outside world. In distributed ledgers (e.g., Blockchain), the signers may be a relatively small and static set of nodes whose public keys can be verified once and then aggregated in arbitrary combinations.

The PoP variants of our schemes do offer some advantages that could make them worth considering. Namely, the aggregate public keys are a simple multiplication and/or hash of the individual keys, as opposed to a multi-exponentiation, which could be relevant if the set of active co-signers changes frequently. Also, their security proofs avoid (one layer of) forking, yielding much tighter security reductions, and therefore shorter key and signature sizes if concrete security is taken into account.

Below, we only summarize the differences of the PoP variants to the original schemes and their security. The security proofs can be found in the supplementary material.

### 6.1 Pairing-Based Schemes with PoPs

The *MSP-pop* and *AMSP-pop* schemes are PoP variants of the *MSP* and *AMSP* schemes, respectively. In fact, *MSP-pop* is known [47, 10], but we recall it here for completeness. The schemes use an additional hash function $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{G}_1$ that could be constructed from $\mathsf{H}_0$ using domain separation. Key generation differs in that it computes $y \leftarrow g_2^{sk}$ as well as a proof of possession $\pi \leftarrow \mathsf{H}_1(y)^x$, and sets $pk \leftarrow (y, \pi)$. To aggregate keys $\{(y_1, \pi_1), \ldots, (y_n, \pi_n)\}$), one first checks that $e(\mathsf{H}_1(y_i), y_i) \stackrel{?}{=} e(\pi_i, g_2)$ for $i = 1, \ldots, n$. If so, then $\mathsf{KAg}$ outputs $apk \leftarrow \prod_{i=1}^{n} y_i$, otherwise it outputs $\perp$.

The signing protocol of *MSP-pop* lets a signers with secret key $sk_i$ compute its partial signature as $s_i \leftarrow \mathsf{H}_0(m)^{sk_i}$, while that of *AMSP-pop* computes it as $s_i \leftarrow \mathsf{H}_0(apk, m)^{sk_i}$. Signature combination, aggregation, and verification are otherwise identical to *MSP* and *AMSP*.

**Theorem 5.** *MSP-pop and AMSP-pop are unforgeable multi-signature and aggregate multi-signature schemes under the hardness of the* co-CDH *and* $\psi$-co-CDH *problems, respectively, in the random-oracle model. More precisely, MSP-pop is* $(\tau, q_S, q_H, \epsilon)$*-unforgeable in the random-oracle model if* co-CDH *is* $(\tau + (q_H + 2q_S + l + 3)\tau_{\exp_1} + l \cdot \tau_{\text{pair}} + O(l(q_H + q_S + 1)), \epsilon/(q_H + q_S + 1))$*-hard, where* $l$ *is the maximum number of signers involved in a single multisignature,* $\tau_{\exp_1}$ *is the time required to compute an exponentiation in* $\mathbb{G}_1$*, and* $\tau_{\text{pair}}$ *is the time required to compute a pairing. AMSP-pop is* $(\tau, q_S, q_H, \epsilon)$*-unforgeable in the random-oracle model if* $\psi$-co-CDH *is* $(\tau + (q_H + 2q_S + l + 3)\tau_{\exp_1} + \tau_{\exp_1^n} + l \cdot \tau_{\text{pair}} + O(l(q_H + q_S + 1)), \epsilon/(q_H + q_S + 1))$*-hard, where* $\tau_{\exp_1^n}$ *is the time required to compute an* $n$*-multiexponentiation in* $\mathbb{G}_1$*.*

### 6.2 Accountable-Subgroup Scheme with PoPs

Key generation in *ASM-pop*, our ASM scheme with PoPs, is the same as for *MSP-pop* and *AMSP-pop* above. Key aggregation is slightly different in that the aggregate of a set of keys $\mathcal{PK} = \{(y_1, \pi_1), \ldots, (y_n, \pi_n)\}$) not only contains the product $Y \leftarrow \prod_{i=1}^{n} y_i$, but also the hash of all public keys $h \leftarrow \mathsf{H}_3(\mathcal{PK})$, where $\mathsf{H}_3 : \{0,1\}^* \to \mathbb{Z}_q$ is another hash function. The aggregate public key is given by the pair $apk \leftarrow (Y, h)$. The reason to include this hash is that when simulating a response to $\mathsf{H}_2(apk, i)$, the simulator must be able to tell whether $i$ is the index of the target signer in the set $\mathcal{PK}$ for which $apk$ is the aggregate public key.

Group setup $\mathsf{GSetup}(sk_i, \mathcal{PK} = \{pk_1, \ldots, pk_n\})$ computes $apk = (Y, h) \leftarrow \mathsf{KAg}(par, \mathcal{PK})$ and sends $\mu_{j,i} = \mathsf{H}_2(apk, j)^{sk_i}$ to signer $j$. After having received $\mu_{i,j}$ from all other signers $j \neq i$, signer $i$ computes $\mu_{i,i} \leftarrow \mathsf{H}_2(apk, i)^{sk_i}$ and outputs the membership key $mk_i \leftarrow \prod_{j=1}^{n} \mu_{i,j}$. If all signers behave honestly, it holds that $e(g_1, mk_i) = e(Y, \mathsf{H}_2(apk, i))$.

Signature and verification are the same as for $\mathcal{ASM}$, except that the product of public keys $pk \leftarrow \prod_{j \in S} pk_j$ is replaced with $y \leftarrow \prod_{j \in S} y_j$.

**Theorem 6.** *Our $\mathcal{ASM}$-pop scheme is unforgeable under the hardness of the computational $\psi$-co-Diffie-Hellman problem in the random-oracle model. More precisely, it is $(\tau, q_S, q_H, \epsilon)$-unforgeable in the random-oracle model if $\psi$-co-CDH is $(\tau + (q_H + q_G(l-1) + q_S + 1) \cdot \tau_{\exp_1} + (2l+2) \cdot \tau_{\text{pair}} + \tau_{\exp_1^{l+2}}, \epsilon / q_H - q_H(q_S + q_H)/q)$-hard, where $l$ is the maximum number of signers involved in any group setup, $\tau_{\exp_1}$ and $\tau_{\exp_1^i}$ denote the time required to compute an exponentiation and $i$-multiexponentiation in $\mathbb{G}_1$, respectively, and $\tau_{\text{pair}}$ denotes the time required to compute a pairing operation.*

### 6.3 Schemes from Discrete Logarithms with PoPs

Drijvers et al. [21] presented the $\mathcal{DG}$-$\mathcal{CoSi}$ scheme, which combines double-generator Okamoto signatures [44] over non-pairing curves with proofs of possession to protect against rogue-key attacks. The use of Okamoto signatures avoids the extra round of hashes in the signing protocol of Bellare-Neven [9], but increases the signature size with an additional element in $\mathbb{Z}_q$. Alternatively, one could consider the $\mathcal{MSDL}$-*pop* scheme, a PoP variant of our $\mathcal{MSDL}$ that uses the extra round of hashes.

To generate a key pair for $\mathcal{MSDL}$-*pop*, the signer chooses $sk \xleftarrow{\$} \mathbb{Z}_q$, computes $y \leftarrow g^{sk}$, and adds proof of possession that is a Schnorr signature [48] on $y$ using $\mathsf{H}_1$ as a hash function. Meaning, it chooses $r \xleftarrow{\$} \mathbb{Z}_q$ and computes $t \leftarrow g^r$, $c \leftarrow \mathsf{H}_1(t, y)$, and $s \leftarrow r + c \cdot sk \bmod q$. The public key is $pk \leftarrow (y, c, s)$.

To aggregate keys $\{(y_1, c_1, s_1), \ldots, (y_n, c_n, s_n)\}$, one first checks that $c_i \stackrel{?}{=} \mathsf{H}_1(g^s y_i^{-c_i}, y_i)$ for all $i = 1, \ldots, n$. If so, then the aggregate public key is $apk \leftarrow \prod_{i=1}^n y_i$, otherwise it is returned as $\perp$. The signing protocol is mostly the same as $\mathcal{MSDL}$, except that the signer in the third round computes $s_i \leftarrow r_i + c \cdot sk_i \bmod q$.

The $\mathcal{MSDL}$-*pop* is secure under the discrete-logarithm assumption in the random-oracle model. The proof is similar to that of $\mathcal{DG}$-$\mathcal{CoSi}$ [21] and hence omitted.

## References


1. Ahn, J.H., Green, M., Hohenberger, S.: Synchronized aggregate signatures: new definitions, constructions and applications. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) ACM CCS 10: 17th Conference on Computer and Communications Security. pp. 473–484. ACM Press, Chicago, Illinois, USA (Oct 4–8, 2010)
2. Andresen, G.: $m$-of-$n$ standard transactions. Bitcoin improvement proposal (BIP) 0011 (2011)
3. Bagherzandi, A., Cheon, J.H., Jarecki, S.: Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In: Ning, P., Syverson, P.F., Jha, S. (eds.) ACM CCS 08: 15th Conference on Computer and Communications Security. pp. 449–458. ACM Press, Alexandria, Virginia, USA (Oct 27–31, 2008)

4. Bagherzandi, A., Jarecki, S.: Multisignatures using proofs of secret key possession, as secure as the Diffie-Hellman problem. In: Ostrovsky, R., Prisco, R.D., Visconti, I. (eds.) SCN 08: 6th International Conference on Security in Communication Networks. Lecture Notes in Computer Science, vol. 5229, pp. 218–235. Springer, Heidelberg, Germany, Amalfi, Italy (Sep 10–12, 2008)
5. Bansarkhani, R.E., Sturm, J.: An efficient lattice-based multisignature scheme with applications to bitcoins. In: Foresti, S., Persiano, G. (eds.) CANS 16: 15th International Conference on Cryptology and Network Security. Lecture Notes in Computer Science, vol. 10052, pp. 140–155. Springer, Heidelberg, Germany, Milan, Italy (Nov 14–16, 2016)
6. Barreto, P.S.L.M., Lynn, B., Scott, M.: On the selection of pairing-friendly groups. In: Matsui, M., Zuccherato, R.J. (eds.) SAC 2003: 10th Annual International Workshop on Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 3006, pp. 17–25. Springer, Heidelberg, Germany, Ottawa, Ontario, Canada (Aug 14–15, 2004)
7. Bellare, M., Namprempre, C., Neven, G.: Unrestricted aggregate signatures. In: Arge, L., Cachin, C., Jurdzinski, T., Tarlecki, A. (eds.) ICALP 2007: 34th International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 4596, pp. 411–422. Springer, Heidelberg, Germany, Wroclaw, Poland (Jul 9–13, 2007)
8. Bellare, M., Namprempre, C., Pointcheval, D., Semanko, M.: The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. Journal of Cryptology 16(3), 185–215 (Jun 2003)
9. Bellare, M., Neven, G.: Multi-signatures in the plain public-key model and a general forking lemma. In: Juels, A., Wright, R.N., Vimercati, S. (eds.) ACM CCS 06: 13th Conference on Computer and Communications Security. pp. 390–399. ACM Press, Alexandria, Virginia, USA (Oct 30 – Nov 3, 2006)
10. Boldyreva, A.: Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In: Desmedt, Y. (ed.) PKC 2003: 6th International Workshop on Theory and Practice in Public Key Cryptography. Lecture Notes in Computer Science, vol. 2567, pp. 31–46. Springer, Heidelberg, Germany, Miami, FL, USA (Jan 6–8, 2003)
11. Boldyreva, A., Gentry, C., O'Neill, A., Yum, D.H.: Ordered multisignatures and identity-based sequential aggregate signatures, with applications to secure routing. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) ACM CCS 07: 14th Conference on Computer and Communications Security. pp. 276–285. ACM Press, Alexandria, Virginia, USA (Oct 28–31, 2007)
12. 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. Lecture Notes in Computer Science, vol. 2656, pp. 416–432. Springer, Heidelberg, Germany, Warsaw, Poland (May 4–8, 2003)
13. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the Weil pairing. In: Boyd, C. (ed.) Advances in Cryptology – ASIACRYPT 2001. Lecture Notes in Computer Science, vol. 2248, pp. 514–532. Springer, Heidelberg, Germany, Gold Coast, Australia (Dec 9–13, 2001)
14. Brogle, K., Goldberg, S., Reyzin, L.: Sequential aggregate signatures with lazy verification from trapdoor permutations - (extended abstract). In: Wang, X., Sako, K. (eds.) Advances in Cryptology – ASIACRYPT 2012. Lecture Notes in Computer Science, vol. 7658, pp. 644–662. Springer, Heidelberg, Germany, Beijing, China (Dec 2–6, 2012)

15. Budroni, A., Pintore, F.: Efficient hash maps to $\mathbb{G}_2$ on BLS curves. Cryptology ePrint Archive, Report 2017/419 (2017), http://eprint.iacr.org/2017/419
16. Burmester, M., Desmedt, Y., Doi, H., Mambo, M., Okamoto, E., Tada, M., Yoshifuji, Y.: A structured ElGamal-type multisignature scheme. In: Imai, H., Zheng, Y. (eds.) PKC 2000: 3rd International Workshop on Theory and Practice in Public Key Cryptography. Lecture Notes in Computer Science, vol. 1751, pp. 466–483. Springer, Heidelberg, Germany, Melbourne, Victoria, Australia (Jan 18–20, 2000)
17. Castelluccia, C., Jarecki, S., Kim, J., Tsudik, G.: A robust multisignatures scheme with applications to acknowledgment aggregation. In: Blundo, C., Cimato, S. (eds.) SCN 04: 4th International Conference on Security in Communication Networks. Lecture Notes in Computer Science, vol. 3352, pp. 193–207. Springer, Heidelberg, Germany, Amalfi, Italy (Sep 8–10, 2005)
18. Certicom Research: Sec 2: Recommended elliptic curve domain parameters. Tech. rep., Certicom Research (2010)
19. Chang, C.C., Leu, J.J., Huang, P.C., Lee, W.B.: A scheme for obtaining a message from the digital multisignature. In: Imai, H., Zheng, Y. (eds.) PKC'98: 1st International Workshop on Theory and Practice in Public Key Cryptography. Lecture Notes in Computer Science, vol. 1431, pp. 154–163. Springer, Heidelberg, Germany, Pacifico Yokohama, Japan (Feb 5–6, 1998)
20. Coron, J.S., Naccache, D.: Boneh et al.'s k-element aggregate extraction assumption is equivalent to the Diffie-Hellman assumption. In: Laih, C.S. (ed.) Advances in Cryptology – ASIACRYPT 2003. Lecture Notes in Computer Science, vol. 2894, pp. 392–397. Springer, Heidelberg, Germany, Taipei, Taiwan (Nov 30 – Dec 4, 2003)
21. Drijvers, M., EdalatNejad, K., Ford, B., Neven, G.: Okamoto beats Schnorr: On the provable security of multi-signatures. Cryptology ePrint Archive, Report 2018/417 (2018), https://eprint.iacr.org/2018/417
22. Fuentes-Castañeda, L., Knapp, E., Rodríguez-Henríquez, F.: Faster hashing to $\mathfrak{d}_2$. In: Miri, A., Vaudenay, S. (eds.) SAC 2011: 18th Annual International Workshop on Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 7118, pp. 412–430. Springer, Heidelberg, Germany, Toronto, Ontario, Canada (Aug 11–12, 2012)
23. Gentry, C., O'Neill, A., Reyzin, L.: A unified framework for trapdoor-permutation-based sequential aggregate signatures. In: Abdalla, M., Dahab, R. (eds.) PKC 2018: 21st International Conference on Theory and Practice of Public Key Cryptography, Part II. Lecture Notes in Computer Science, vol. 10770, pp. 34–57. Springer, Heidelberg, Germany, Rio de Janeiro, Brazil (Mar 25–29, 2018)
24. Gentry, C., Ramzan, Z.: Identity-based aggregate signatures. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography. Lecture Notes in Computer Science, vol. 3958, pp. 257–273. Springer, Heidelberg, Germany, New York, NY, USA (Apr 24–26, 2006)
25. Hardjono, T., Zheng, Y.: A practical digital multisignature scheme based on discrete logarithms. In: Seberry, J., Zheng, Y. (eds.) Advances in Cryptology – AUSCRYPT'92. Lecture Notes in Computer Science, vol. 718, pp. 122–132. Springer, Heidelberg, Germany, Gold Coast, Queensland, Australia (Dec 13–16, 1993)
26. Harn, L.: Group-oriented (t, n) threshold digital signature scheme and digital multisignature. IEE Proceedings-Computers and Digital Techniques 141(5), 307–313 (1994)

27. Horster, P., Michels, M., Petersen, H.: Meta-multisignature schemes based on the discrete logarithm problem. In: Information Securitythe Next Decade. pp. 128–142. Springer (1995)
28. Itakura, K., Nakamura, K.: A public-key cryptosystem suitable for digital multisignatures. Tech. rep., NEC Research and Development (1983)
29. Komano, Y., Ohta, K., Shimbo, A., Kawamura, S.: Formal security model of multisignatures. In: Katsikas, S.K., Lopez, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) ISC 2006: 9th International Conference on Information Security. Lecture Notes in Computer Science, vol. 4176, pp. 146–160. Springer, Heidelberg, Germany, Samos Island, Greece (Aug 30 – Sep 2, 2006)
30. Le, D.P., Bonnecaze, A., Gabillon, A.: Multisignatures as secure as the Diffie-Hellman problem in the plain public-key model. In: Shacham, H., Waters, B. (eds.) PAIRING 2009: 3rd International Conference on Pairing-based Cryptography. Lecture Notes in Computer Science, vol. 5671, pp. 35–51. Springer, Heidelberg, Germany, Palo Alto, CA, USA (Aug 12–14, 2009)
31. Li, C.M., Hwang, T., Lee, N.Y.: Threshold-multisignature schemes where suspected forgery implies traceability of adversarial shareholders. In: Santis, A.D. (ed.) Advances in Cryptology – EUROCRYPT'94. Lecture Notes in Computer Science, vol. 950, pp. 194–204. Springer, Heidelberg, Germany, Perugia, Italy (May 9–12, 1995)
32. 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. Lecture Notes in Computer Science, vol. 4004, pp. 465–485. Springer, Heidelberg, Germany, St. Petersburg, Russia (May 28 – Jun 1, 2006)
33. 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. Lecture Notes in Computer Science, vol. 3027, pp. 74–90. Springer, Heidelberg, Germany, Interlaken, Switzerland (May 2–6, 2004)
34. Ma, C., Weng, J., Li, Y., Deng, R.: Efficient discrete logarithm based multi-signature scheme in the plain public key model. Designs, Codes and Cryptography 54(2), 121–133 (2010)
35. Maxwell, G., Poelstra, A., Seurin, Y., Wuille, P.: Simple schnorr multi-signatures with applications to bitcoin. Cryptology ePrint Archive, Report 2018/068 (2018), `https://eprint.iacr.org/2018/068/20180118:124757`
36. Maxwell, G., Poelstra, A., Seurin, Y., Wuille, P.: Simple schnorr multi-signatures with applications to bitcoin. Cryptology ePrint Archive, Report 2018/068 (2018), `https://eprint.iacr.org/2018/068/20180520:191909`
37. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology – CRYPTO'87. Lecture Notes in Computer Science, vol. 293, pp. 369–378. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 16–20, 1988)
38. Micali, S., Ohta, K., Reyzin, L.: Accountable-subgroup multisignatures: Extended abstract. In: ACM CCS 01: 8th Conference on Computer and Communications Security. pp. 245–254. ACM Press, Philadelphia, PA, USA (Nov 5–8, 2001)
39. Michels, M., Horster, P.: On the risk of disruption in several multiparty signature schemes. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 334–345. Springer (1996)
40. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), `http://bitcoin.org/bitcoin.pdf`

41. Neven, G.: Efficient sequential aggregate signed data. In: Smart, N.P. (ed.) Advances in Cryptology – EUROCRYPT 2008. Lecture Notes in Computer Science, vol. 4965, pp. 52–69. Springer, Heidelberg, Germany, Istanbul, Turkey (Apr 13–17, 2008)
42. 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. Lecture Notes in Computer Science, vol. 739, pp. 139–148. Springer, Heidelberg, Germany, Fujiyoshida, Japan (Nov 11–14, 1993)
43. Ohta, K., Okamoto, T.: Multi-signature schemes secure against active insider attacks. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 82(1), 21–31 (1999)
44. Okamoto, T.: Provably secure and practical identification schemes and corresponding signature schemes. In: Brickell, E.F. (ed.) Advances in Cryptology – CRYPTO'92. Lecture Notes in Computer Science, vol. 740, pp. 31–53. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 16–20, 1993)
45. Park, S., Park, S., Kim, K., Won, D.: Two efficient RSA multisignature schemes. In: Han, Y., Okamoto, T., Qing, S. (eds.) ICICS 97: 1st International Conference on Information and Communication Security. Lecture Notes in Computer Science, vol. 1334, pp. 217–222. Springer, Heidelberg, Germany, Beijing, China (Nov 11–14, 1997)
46. Pointcheval, D., Stern, J.: Security arguments for digital signatures and blind signatures. Journal of Cryptology 13(3), 361–396 (2000)
47. 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. Lecture Notes in Computer Science, vol. 4515, pp. 228–245. Springer, Heidelberg, Germany, Barcelona, Spain (May 20–24, 2007)
48. Schnorr, C.P.: Efficient signature generation by smart cards. Journal of Cryptology 4(3), 161–174 (1991)
49. Scott, M., Benger, N., Charlemagne, M., Perez, L.J.D., Kachisa, E.J.: Fast hashing to $g_2$ on pairing-friendly curves. In: Shacham, H., Waters, B. (eds.) PAIRING 2009: 3rd International Conference on Pairing-based Cryptography. Lecture Notes in Computer Science, vol. 5671, pp. 102–113. Springer, Heidelberg, Germany, Palo Alto, CA, USA (Aug 12–14, 2009)

# A Security Proofs for Schemes with Proofs of Possession

## A.1 Security Proof for $\mathcal{MSP}$-pop and $\mathcal{AMSP}$-pop

*Proof (Theorem 5).* Given a $(\tau, q_S, q_H, \epsilon)$ forger $\mathcal{F}$ against $\mathcal{MSP}$-pop, consider the following co-CDH algorithm $\mathcal{A}$. On input $A = g_1^{\alpha}$, $B_1 = g_1^{\beta}$, $B_2 = g_2^{\beta}$, algorithm $\mathcal{A}$ first chooses a random index $k \xleftarrow{\$} \{1, \ldots, q_H + q_S + 1\}$. It then chooses $r \xleftarrow{\$} \mathbb{Z}_q$, assigns $\mathsf{H}_1(B_2) \leftarrow g_1^r$, and runs $\mathcal{F}$ on target public key $pk^* \leftarrow (B_2, B_1^r)$, responding to its queries as follows:

- $\mathsf{H}_0(m)$: If this is $\mathcal{F}$'s $k$-th random-oracle query, then $\mathcal{A}$ assigns $\mathsf{H}_0(m) \leftarrow A$ and adds $(m, \perp, 1)$ to $L_0$. If not, then it chooses $r \xleftarrow{\$} \mathbb{Z}_q$, assigns $\mathsf{H}_0(m) \leftarrow g_1^r$, and adds $(m, r, 0)$ to $L_0$.
- $\mathsf{H}_1(y)$: $\mathcal{A}$ chooses $r \xleftarrow{\$} \mathbb{Z}_q$, assigns $\mathsf{H}_1(y) \leftarrow B_1^r$, and adds $(y, r)$ to $L_1$.

– $\mathsf{Sign}(m)$: $\mathcal{A}$ first simulates an internal query $\mathsf{H}_0(m)$. If $m = m^*$ then $\mathcal{A}$ gives up. Otherwise, it looks up $(m, r, 0) \in L_0$ and returns $s_i \leftarrow B_1^r$.

When $\mathcal{F}$ outputs its forgery $\sigma$ for message $m^*$ and public keys $\mathcal{PK} = \{pk_1, \ldots, pk_n\}$, $\mathcal{A}$ computes $apk \leftarrow \mathsf{KAg}(\mathcal{PK})$ and checks that $\mathsf{Vf}(par, apk, m^*, \sigma) = 1$. If $\mathsf{H}_0(m^*)$ was not $\mathcal{F}$'s $k$-th random-oracle query, then $\mathcal{A}$ aborts. Otherwise, it parses $pk_j$ as $(y_j, \pi_j)$ and looks up $(y_j, r_j) \in L_1$ for $j = 1, \ldots, n$. Let $i$ be the index such that $pk_i = pk^*$. For $j \neq i$, we have that $\pi_j = A^{r_j \cdot \log pk_j}$, so that $\mathcal{A}$ can return $\sigma \cdot \prod_{j=1, j\neq i}^{n} \pi_j^{-r_j}$ as its $\mathsf{co\text{-}CDH}$ solution. Then $\mathcal{A}$ succeeds with probability $\epsilon_{\mathcal{A}} = \epsilon/(q_{\mathrm{H}} + q_{\mathrm{S}} + 1)$ and has running time $\tau_{\mathcal{A}} = \tau + (q_{\mathrm{H}} + 2q_{\mathrm{S}} + l + 3)\tau_{\mathrm{exp}_1} + l \cdot \tau_{\mathrm{pair}} + O(l(q_{\mathrm{H}} + q_{\mathrm{S}} + 1))$.

Given a forger $\mathcal{F}$ against the *AMSP-pop* scheme, one can construct an algorithm $\mathcal{A}$ solving the $\psi$-$\mathsf{co\text{-}CDH}$ problem along the same lines. Its forgery consists of an aggregate multi-signature $\Sigma$, a set of aggregate public keys and message pairs $\{(apk_1, m_1), \ldots, (apk_n, m_n)\}$, a set of public keys $\mathcal{PK}$, and a message $m^*$. Let $apk^* \leftarrow \mathsf{KAg}(par, \mathcal{PK})$. If $\mathsf{H}_0(apk^*, m^*)$ was $\mathcal{F}$'s $k$-th random-oracle query, then we have that

$$e(\Sigma, g_2^{-1}) \cdot e(A, apk^*) \cdot \prod_{i=1}^{n} e(\mathsf{H}_0(apk_i, m_i), apk_i) = 1_{\mathbb{G}_{\mathrm{t}}} \ .$$

$\mathcal{A}$ looks up $r_i$ for every $(apk_i, m_i)$ such that $\mathsf{H}_0(apk_i, m_i) = g_1^{r_i}$. It computes $\sigma \leftarrow \Sigma \cdot \prod_{i=1}^{n} \mathcal{O}^{\psi}(apk_i^{-r_i})$, so that

$$e(\sigma, g_2) = e(y, apk^*).$$

Note that $\mathcal{A}$ has now extracted a *MSP* forgery, meaning that the rest of the reduction is exactly as for the *MSP-pop* scheme. The success probability of the reduction is therefore the same, and the runtime is only increased by the extra steps required to compute $\sigma$, which costs $\tau_{\mathrm{exp}_1^{\mathrm{n}}}$.

### A.2 Security Proof for *ASM-pop*

*Proof (Theorem 6).* Given a forger $\mathcal{F}$ against the ASM scheme, we construct the following algorithm $\mathcal{A}$ solving the $\psi$-$\mathsf{co\text{-}CDH}$ problem. On input $(A = g_1^{\alpha}, B_1 = g_1^{\beta}, B_2 = g_2^{\beta})$ and given access to a homomorphism oracle $\mathcal{O}^{\psi}(\cdot)$, $\mathcal{A}$ proceeds as follows. It guesses a random index $k \xleftarrow{\$} \{1, \ldots, q_{\mathrm{H}}\}$, chooses $r \xleftarrow{\$} \mathbb{Z}_q$, assigns $\mathsf{H}_1(B_2) \leftarrow g_1^r$, and runs $\mathcal{F}$ on input $par \leftarrow (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_{\mathrm{t}}, e, g_1, g_2)$ and $pk^* \leftarrow (B_2, B_1^r)$, answering its oracle queries using initially empty lists $L_0, L_2$ as follows:

– $\mathsf{H}_1(y)$: Choose $r \xleftarrow{\$} \mathbb{Z}_q$, assign $\mathsf{H}_1(x) \leftarrow A^r$, and add $(y, r)$ to $L_1$.
– $\mathsf{H}_3(x)$: Choose a random value $h \xleftarrow{\$} \mathbb{Z}_q$ and assign $\mathsf{H}_3(x) \leftarrow h$. If $\mathcal{F}$ previously made any random-oracle or signing queries involving $h$ as part of an aggregated public key, then we say that event **bad** happened and $\mathcal{A}$ gives up.
– $\mathsf{H}_2(x)$: If $x$ can be parsed as $(apk, i)$ such that $apk = (Y, h)$ and there exists a defined entry $\mathsf{H}_3(\mathcal{PK}) = h$ such that $Y = \prod_{(y,\pi)\in\mathcal{PK}} y$, $pk^* \in \mathcal{PK}$, and $i$ is the index of $pk^*$ in $\mathcal{PK}$, then $\mathcal{A}$ chooses $r \xleftarrow{\$} \mathbb{Z}_q$, adds $((apk, i), r, 1)$ to $L_2$ and assigns $\mathsf{H}_2(x) \leftarrow g_1^r A^{-1}$. If not, then $\mathcal{A}$ chooses $r \xleftarrow{\$} \mathbb{Z}_q$, adds $(x, r, 0)$ to $L_2$ and assigns $\mathsf{H}_2(x) \leftarrow g_1^r$.

– $\mathsf{H}_0(x)$: If this is $\mathcal{F}$'s $k$-th random-oracle query, then $\mathcal{A}$ sets $m^* \leftarrow x$, hoping that $\mathcal{F}$ will forge on message $m^*$. It then chooses $r \xleftarrow{\$} \mathbb{Z}_q$, adds $(m^*, r, 1)$ to $L_0$ and assigns $\mathsf{H}_0(m^*) \leftarrow g_1^r$. If this is not $\mathcal{F}$'s $k$-th random-oracle query, then $\mathcal{A}$ chooses $r \xleftarrow{\$} \mathbb{Z}_q$, adds $(x, r, 0)$ to $L_0$ and assigns $\mathsf{H}_0(x) \leftarrow g_1^r A$.
– $\mathsf{GSetup}(\mathcal{PK})$: If $pk^* \notin \mathcal{PK}$, then $\mathcal{A}$ ignores this query. Otherwise, it computes $apk \leftarrow \mathsf{KAg}(par, \mathcal{PK})$, internally simulating the random-oracle queries $\mathsf{H}_1(y)$ and $\mathsf{H}_3(\mathcal{PK})$ if needed. It also internally simulates queries $\mathsf{H}_2(apk, j)$ for $j = 1, \ldots, |\mathcal{PK}|, j \neq i$, to create entries $((apk, j), r_j, 0) \in L_2$, where $i$ is the index of $pk^*$ in $\mathcal{PK}$. Since $\mathsf{H}_2(apk, j) = g_1^{r_j}$, $\mathcal{A}$ can simulate the values $\mu_{j,i} = \mathsf{H}_2(apk, j)^{sk^*} = \mathsf{H}_2(apk, j)^{\beta}$ for $j \neq i$ as $\mu_{j,i} \leftarrow B_1^{r_j}$.
After having received $\mu_{i,j}$ from all other signers $j \neq i$, $\mathcal{A}$ internally stores $\mu_{apk} \leftarrow \prod_{j \neq i} \mu_{i,j}$.
– $\mathsf{Sign}(\mathcal{PK}, S, m)$: If $\mathcal{F}$ did not perform group setup for $\mathcal{PK}$, then $\mathcal{A}$ ignores this query. If $m = m^*$, then $\mathcal{A}$ gives up. Otherwise, it recomputes $apk \leftarrow \mathsf{KAg}(\mathcal{PK})$ and looks up $(m, r_0, 0) \in L_0$ and $((apk, i), r_2, 1) \in L_2$, internally simulating queries $\mathsf{H}_0(m)$ and $\mathsf{H}_2(apk, i)$ to create them if needed, where $i$ is the index of $pk^*$ in $\mathcal{PK}$. Now $\mathcal{A}$ must simulate the partial signature $s_i = \mathsf{H}_0(m)^{sk^*} \cdot \mu_{apk} \cdot \mathsf{H}_2(apk, i)^{sk^*}$. From the way $\mathcal{A}$ responded to random-oracle queries, we know that $\mathsf{H}_0(m) = g_1^{r_0} A = g_1^{r_0 + \alpha}$ and $\mathsf{H}_2(x) = g_1^{r_2} A^{-1} = g_1^{r_2 - \alpha}$, so that $\mathcal{A}$ has to simulate $s_i = g_1^{\beta(r_0+\alpha)} \cdot \mu_{apk} \cdot g_1^{\beta(r_2-\alpha)} = \mu_{apk} \cdot g_1^{\beta(r_0+r_2)}$, which it can easily compute as $s_i \leftarrow \mu_{apk} \cdot B_1^{r_0+r_2}$.

When $\mathcal{F}$ eventually outputs its forgery $(\mathcal{PK}, S, m, \sigma)$, $\mathcal{A}$ recomputes $apk^* \leftarrow \mathsf{KAg}(\mathcal{PK}) = (Y, h)$ and checks that the forgery is valid, i.e., $\mathsf{Vf}(par, apk, S, m, \sigma) = 1$, $pk^* \in \mathcal{PK}$, $i \in S$ where $i$ is the index of $pk^* \in \mathcal{PK}$, and $\mathcal{F}$ never made a signing query for $m$. If any of these checks fails, $\mathcal{A}$ outputs $(0, \perp)$. If $m \neq m^*$, then $\mathcal{A}$ also outputs $(0, \perp)$. Else, observe that $\sigma = (y, s)$ such that

$$s \;=\; \mathsf{H}_0(m^*)^{\log y} \cdot \prod_{j \in S} \mathsf{H}_2(apk, j)^{\log Y} \;.$$

Because of how $\mathcal{A}$ simulated $\mathcal{F}$'s random-oracle queries, it can look up $(m^*, r_0, 1) \in L_0$, $(y_j, r_{1,j}) \in L_1$ and $((apk^*, j), r_{2,j}, 0) \in L_2$ for $j \in S \backslash \{i\}$, and $((apk^*, i), r_{2,i}, 1) \in L_2$, where $i$ is the index of $pk^*$ in $\mathcal{PK}$, such that

$$\begin{aligned} \mathsf{H}_0(m^*) &= g_1^{r_0} \\ \mathsf{H}_1(y_j) &= A^{r_{1,j}} \text{ for } j = 1, \ldots, |\mathcal{PK}| \\ \mathsf{H}_2(apk, j) &= g_1^{r_{2,j}} \text{ for } j \in S \setminus \{i\} \\ \mathsf{H}_2(apk, i) &= g_1^{r_{2,i}} A^{-1/a_i} \end{aligned}$$

so that we have that

$$s = g_1^{\log y \cdot r_0} \cdot g_1^{\log apk^* \cdot \sum_{j \in S} r_{2,j}} \cdot A^{-\log apk^*} \;.$$

If we let

$$t \leftarrow s^{-1} \cdot \mathcal{O}^{\psi}(pk)^{r_0} \cdot \mathcal{O}^{\psi}(apk^*)^{\sum_{j \in S} r_{2,j}}$$

then we have that

$$t \;=\; A^{\log apk^*} \;=\; A^{\sum_{j=1}^{|\mathcal{PK}|} \log y_j} \;=\; A^{\beta} \cdot \prod_{j=1, j\neq i}^{|\mathcal{PK}|} A^{\log y_j} \;.$$

From the PoPs in $\mathcal{PK}$, we also have that

$$\pi_j = \mathsf{H}_1(y_j)^{\log y_j} = A^{r_{1,j} \log y_j}$$

so that $\mathcal{A}$ can compute its solution to the $\psi$-co-CDH problem as

$$g_1^{\alpha\beta} \;=\; t \cdot \prod_{j=1, j\neq i}^{|\mathcal{PK}|} \pi_j^{-r_{1,j}} \;.$$

$\mathcal{A}$'s runtime is $\mathcal{F}$'s runtime plus the additional computation $\mathcal{A}$ performs. Let $q_{\mathrm{H}}$ denote the total hash queries $\mathcal{F}$ makes, i.e., the queries to $\mathsf{H}_0$, $\mathsf{H}_1$, and $\mathsf{H}_2$ combined. To answer $\mathsf{H}_0$, $\mathsf{H}_1$, and $\mathsf{H}_2$ queries, $\mathcal{A}$ performs at most $\tau_{\exp_1}$. $\mathcal{A}$ therefore spends at most $q_{\mathrm{H}} \cdot \tau_{\exp_1}$ answering hash queries. For every group-setup query with $l$ signers, $\mathcal{A}$ computes $\mu_{j,i}$ costing $(l-1)\tau_{\exp_1}$, meaning $\mathcal{A}$ spends $q_{\mathrm{G}}(l-1) \cdot \tau_{\exp_1}$ answering group setup queries. For signing queries with a $\mathcal{PK}$ of size at most $l$, $\mathcal{A}$ computes one exponentiation in $\mathbb{G}_1$ costing $\tau_{\exp_1}$, giving a total of $q_{\mathrm{S}} \cdot \tau_{\exp_1}$. Finally, $\mathcal{A}$ computes the output values, which involves verifying the forgery (costing $2\tau_{\mathrm{pair}}$) and the PoPs (costing $2l\tau_{\mathrm{pair}}$) and computing the final solution (costing $\tau_{\exp_1^{l+2}}$), giving $\mathcal{A}$ a total runtime of $\tau + (q_{\mathrm{H}} + q_{\mathrm{G}}(l-1) + q_{\mathrm{S}} + 1) \cdot \tau_{\exp_1} + (2l+2) \cdot \tau_{\mathrm{pair}} + \tau_{\exp_1^{l+2}}$. $\mathcal{A}$ successfully outputs if the **bad** event does not happen, it guesses the index of the forgery correctly, and $\mathcal{F}$ successfully forges. Event **bad** happens with probability at most $(q_{\mathrm{S}} + q_{\mathrm{H}})/q$ for every hash query, so it happens with probability $q_{\mathrm{H}}(q_{\mathrm{S}} + q_{\mathrm{H}})/q$. $\mathcal{A}$ guesses the forgery index correctly with prbability $1/q_{\mathrm{H}}$, and $\mathcal{F}$ forges with probability $\epsilon$, giving $\mathcal{A}$ success probability at least $\epsilon/q_{\mathrm{H}} - q_{\mathrm{H}}(q_{\mathrm{S}} + q_{\mathrm{H}})/q$.