



Compact Multi-signatures for Smaller Blockchains

Dan Boneh^{1(✉)}, Manu Drijvers^{2,3(✉)}, and Gregory Neven^{2(✉)}

¹ Stanford University, Stanford, USA

dabo@cs.stanford.edu

² DFINITY, Zurich, Switzerland

{manu,gregory}@dfinity.org

³ ETH Zurich, Zurich, Switzerland

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 κ 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* [29, 39] is a protocol that enables the n signers to jointly generate a short signature σ on m so that σ 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 σ . The algorithm either accepts or rejects σ . The multi-signature σ should be short – its length should be independent of the number of signers n . We define this concept more precisely in

M. Drijvers and G. Neven—Work partially done at IBM Research – Zurich.

the next section, where we also present the standard security model for such schemes [39]. 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 Sect. 1.4.

A more general concept called an *aggregate signature scheme* [13] lets each of the n parties sign a *different* message, but all these signatures can be aggregated into a single short signature σ . 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 [41]. In recent work, Maxwell, Poelstra, Seurin, and Wuille [36] 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, \dots, pk_n along with some number $t \in \{1, \dots, n\}$ called a threshold (see [2, 36] for details). To spend funds associated with this address, one creates a transaction containing all n public keys pk_1, \dots, 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 [36, Sect. 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. [36], 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 [22] 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 Sect. 5, we present a modification of the scheme by Maxwell et al. that we prove secure under the standard discrete-log assumption. Our *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. Independently from our work, Maxwell et al. [37] revised their work to use the same protocol we present here.

1.1 Better Constructions Using Pairings

Our main results show that we can do much better by replacing the Schnorr signature scheme in [36] by BLS signatures [14]. 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 $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_t$ 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 $H_0 : \mathcal{M} \rightarrow \mathbb{G}_1$. Now the BLS signature scheme works as follows:

- Key generation: choose a random $sk \leftarrow \mathbb{Z}_q$ and output (pk, sk) where $pk \leftarrow g_2^{sk} \in \mathbb{G}_2$.
- Sign(sk, m): output $\sigma \leftarrow H_0(m)^{sk} \in \mathbb{G}_1$.
- Verify(pk, m, σ): if $e(\sigma, g_2) \stackrel{?}{=} e(H_0(m), pk)$ output “accept”, otherwise output “reject”.

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

$$\sigma \leftarrow \sigma_1 \cdots \sigma_n \in \mathbb{G}_1. \quad (1)$$

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

$$e(\sigma, g_2) = e(H_0(m_1), pk_1) \cdots e(H_0(m_n), pk_n). \quad (2)$$

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

$$e(\sigma, g_2) \stackrel{?}{=} e(H_0(m_1), pk_1 \cdots pk_n). \quad (3)$$

Observe that 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 $\alpha \leftarrow \mathbb{Z}_q$ 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 $\sigma := H_0(m)^\alpha$. This signature verifies as an aggregate of two signatures, one from pk_1 and one from pk_2 , because

$$e(\sigma, g_2) = e(H_0(m)^\alpha, g_2) = e(H_0(m), g_2^\alpha) = e(H_0(m), pk_1 \cdot pk_2).$$

Hence, this σ 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, 33, 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 [36].
- Require that the messages being aggregated are distinct [7, 13], 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 fast verification and public-key aggregation as in (3) when aggregating signatures on a common message m .

1.2 Our Pairing-Based Results

In Sect. 3 we propose a different defense against the rogue public-key attack that retains all the benefits of both defenses above without the drawbacks. Our multi-signature scheme, called \mathcal{MSP} , 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 Sect. 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 [36] 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.

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 $S \subseteq \{1, \dots, n\}$, the 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 Sect. 4. This concept was previously studied by Micali et al. [39].

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 κ , the public key size in this trivial ASM is $O(n \times \kappa)$ bits. The signature size is $O(|S| \times \kappa)$ bits.

Our new ASM scheme, called \mathcal{ASM} , is presented in Sect. 4.2. It is the first ASM where signature size is only $O(\kappa)$ bits beyond the description of the set S , independent of n . The public key is only $O(\kappa)$ bits. Concretely, the 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. We also present an aggregate ASM, called \mathcal{AASM} , that partially aggregates w \mathcal{ASM} signatures into a single signature that is considerably smaller than w separate \mathcal{ASM} signatures.

To see how all 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. When a block contains w such transactions, our aggregate ASM in Sect. 4.4 can reduce this further to two (amortized) group elements per transaction. This is significantly better than the trivial linear size ASM scheme currently employed by Bitcoin.

Proofs of Possession. We further 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

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 [19] 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} , \mathcal{ASM} , and \mathcal{AASM} 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 .

	Combined public key size	Combined signature size	Total size (KB)	Threshold support
Bitcoin	$tx \cdot inp \cdot n \cdot \mathbb{G} $	$tx \cdot inp \cdot n \cdot 2 \cdot \mathbb{Z}_q $	1296	linear
MuSig ([36])	$tx \cdot inp \cdot \mathbb{G} $	$tx \cdot (\mathbb{G} + \mathbb{Z}_q)$	240	small
\mathcal{MSDL} (Sec. 5)	$tx \cdot inp \cdot \mathbb{G} $	$tx \cdot (\mathbb{G} + \mathbb{Z}_q)$	240	small
\mathcal{MSP} (Sec. 3.1)	$tx \cdot inp \cdot \mathbb{G}_2 $	$tx \cdot \mathbb{G}_1 $	360	small
\mathcal{AMSP} (Sec. 3.3)	$tx \cdot inp \cdot \mathbb{G}_2 $	$ \mathbb{G}_1 $	216	small
\mathcal{ASM} (Sec. 4)	$tx \cdot inp \cdot \mathbb{G}_2 $	$tx \cdot inp \cdot (\mathbb{G}_1 + \mathbb{G}_2)$	864	any
\mathcal{AASM} (Sec. 4.4)	$tx \cdot inp \cdot \mathbb{G}_2 $	$tx \cdot inp \cdot \mathbb{G}_2 + \mathbb{G}_1 $	432	any

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 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. Due to space constraints, the PoP variants are only presented in the full version of this work [12].

1.3 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{AASM} 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 almost 60 times more space than our \mathcal{ASM} scheme. The other schemes support threshold signatures using Merkle trees [38] as outlined in [36, Sect. 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.

1.4 Related Work

Multi-signatures have been studied extensively based on RSA [29, 30, 43, 45], discrete logarithms [3, 4, 9, 17, 18, 20, 22, 26–28, 32, 35, 36, 39, 44], pairings [10, 11, 31, 33, 47], and lattices [5]. Defending against rogue public-key attacks has always been a primary concern in the context of multi-signature schemes based on discrete-log and pairings [7, 9, 13, 28, 39, 40, 47], and is the main reason for the added complexity in discrete-log-based multi-signature systems. Aggregate signatures [1, 13, 25] are a closely related concept where signatures by different signers on different messages can be compressed together. Sequential aggregate signatures [15, 24, 33, 34, 42] 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 [36] 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 κ 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 $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_t$, 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 $\text{Adv}_{\mathbb{G}}^{\text{dl}}$ of an adversary \mathcal{A} as

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

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 τ and has $\text{Adv}_{\mathbb{G}}^{\text{dl}} \geq \epsilon$. Discrete log is (τ, ϵ) -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 $\text{Adv}_{\mathbb{G}_1, \mathbb{G}_2}^{\text{co-CDH}}$ of an adversary \mathcal{A} as

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

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

Definition 3 (Computational ψ -co-Diffie-Hellman Problem). For 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 $g_2^x \in \mathbb{G}_2$ returns $g_1^x \in \mathbb{G}_1$. Define $\text{Adv}_{\mathbb{G}_1, \mathbb{G}_2}^{\psi\text{-co-CDH}}$ of an adversary \mathcal{A} as

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

where the probability is taken over the random choices of \mathcal{A} and the random selection of (α, β) . \mathcal{A} (τ, ϵ) -breaks the ψ -co-CDH problem if it runs in time at most τ and has $\text{Adv}_{\mathbb{G}_1, \mathbb{G}_2}^{\psi\text{-co-CDH}} \geq \epsilon$. ψ -co-CDH is (τ, ϵ) -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 [3, 9]. 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 $H : \{0, 1\}^* \rightarrow \mathbb{Z}_q$. Let $f = (\rho, h_1, \dots, h_{q_H})$ be the randomness involved in an execution of \mathcal{A} , where ρ is \mathcal{A} 's random tape, h_i is the response to \mathcal{A} 's i -th query to H , and q_H is its maximal number of random-oracle queries. Let Ω be the space of all such vectors f and let $f|_i = (\rho, h_1, \dots, h_{i-1})$. 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 $\{1, \dots, q_H\}$ 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 ϵ be the probability that $\mathcal{A}(in, f)$ is successful for fresh randomness $f \xleftarrow{s} \Omega$ and for an input $in \xleftarrow{s} \text{IG}$ generated by an input generator IG .

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

```

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

```

We say that \mathcal{GF}_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 IG be a randomized algorithm and A be a randomized algorithm running in time τ making at most q_H random-oracle queries that succeeds with probability ϵ . If $q > 8nq_H/\epsilon$, then $\mathcal{GF}_A(\text{in})$ runs in time at most $\tau \cdot 8n^2q_H/\epsilon \cdot \ln(8n/\epsilon)$ and succeeds with probability at least $\epsilon/8$, where the probability is over the choice of $\text{in} \xleftarrow{\$} \text{IG}$ and over the coins of \mathcal{GF}_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 Pg , Kg , Sign , KAg , and Vf . A trusted party generates the system parameters $\text{par} \leftarrow \text{Pg}$. Every signer generates a key pair $(pk, sk) \xleftarrow{\$} \text{Kg}(\text{par})$, and signers can collectively sign a message m by each calling the interactive algorithm $\text{Sign}(\text{par}, \mathcal{PK}, sk, m)$, 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 σ . Algorithm 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 σ on message m under an aggregate public key apk by running $\text{Vf}(\text{par}, apk, m, \sigma)$ 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 $(pk_i, sk_i) \leftarrow \text{Kg}(\text{par})$ for $i = 1, \dots, n$, and for any message m , if all signers input $\text{Sign}(\text{par}, \{pk_1, \dots, pk_n\} sk_i, m)$, then every signer will output a signature σ such that $\text{Vf}(\text{par}, \text{KAg}(\text{par}, \{pk_i\}_{i=1}^n), m, \sigma) = 1$. Second, a multisignature scheme should satisfy unforgeability. Unforgeability of a multisignature scheme $\mathcal{MS} = (\text{Pg}, \text{Kg}, \text{Sign}, \text{KAg}, \text{Vf})$ is defined by a three-stage game.

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

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 $\mathcal{O}^{\text{Sign}(\text{par}, \cdot, sk^*, \cdot)}$ 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 σ , 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 $\text{Vf}(\text{par}, \text{KAg}(\text{par}, \mathcal{PK}), m) \cdot \sigma = 1$.

Definition 4. *We say \mathcal{A} is a $(\tau, q_S, q_H, \epsilon)$ -forger for multisignature scheme $\mathcal{MS} = (\text{Pg}, \text{Kg}, \text{Sign}, \text{KAg}, \text{Vf})$ if it runs in time τ , makes q_S signing queries, makes q_H random oracle queries, and wins the above game with probability at least ϵ . \mathcal{MS} is $(\tau, q_S, q_H, \epsilon)$ -unforgeable if no $(\tau, q_S, q_H, \epsilon)$ -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. SAG takes input a set of tuples, each tuple containing an aggregate public key apk , a message m , and a multisignature σ , and outputs a single aggregate multisignature Σ . AVf takes input a set of tuples, each tuple containing an aggregate public key apk and a message m , and an aggregate multisignature Σ , 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 $\text{SAG}(\text{par}, \{apk_i, m_i, \sigma_i\})$ to output $\Sigma \leftarrow (\sigma_1, \dots, \sigma_n)$, and $\text{AVf}(\text{par}, \{apk_i, m_i\}, (\sigma_1, \dots, \sigma_n))$ to output 1 if all individual multisignatures are valid. The goal however is to have Σ 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 $(pk_i, sk_i) \leftarrow \text{Kg}(\text{par})$ for $i = 1, \dots, n$, and for any message m , if all signers input $\text{Sign}(\text{par}, \{pk_1, \dots, pk_n\} sk_i, m)$, then every signer will output a signature σ such that $\text{Vf}(\text{par}, \text{KAg}(\text{par}, \{pk_i\}_{i=1}^n, m, \sigma) = 1$, and 2) for any set of valid multisignatures $\{(apk_i, m_i, \sigma_i)\}$ (with $\text{Vf}(\text{par}, apk_i, m_i, \sigma_i) = 1$), the aggregated multisignature is also valid: $\text{AVf}(\text{par}, \{apk_i, m_i\}, \text{SAG}(\text{par}, \{(apk_i, m_i, \sigma_i)\})) = 1$. Second, an aggregate multisignature scheme should satisfy unforgeability. Unforgeability of an aggregate multisignature scheme $\mathcal{AMS} = (\text{Pg}, \text{Kg}, \text{Sign}, \text{KAg}, \text{Vf}, \text{SAG}, \text{AVf})$ 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 Σ , 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 $\text{AVf}(\text{par}, \{(apk_i, m_i)\} \cup \{(\text{KAg}(\text{par}, \mathcal{PK}), m^*)\}, \Sigma) = 1$.

Definition 5. We say \mathcal{A} is a $(\tau, qs, q_H, \epsilon)$ -forger for aggregate multisignature scheme $\mathcal{AMS} = (\text{Pg}, \text{Kg}, \text{Sign}, \text{KAg}, \text{Vf}, \text{SAG}, \text{AVf})$ if it runs in time τ , makes qs signing queries, makes q_H random oracle queries, and wins the above game with probability at least ϵ . \mathcal{AMS} is $(\tau, qs, q_H, \epsilon)$ -unforgeable if no $(\tau, qs, q_H, \epsilon)$ -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 [16, 23, 49].

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 [14]. The scheme is secure in the plain public key model, and assumes hash functions $H_0 : \{0, 1\}^* \rightarrow \mathbb{G}_1$ and $H_1 : \{0, 1\}^* \rightarrow \mathbb{Z}_q$.

Parameters Generation. $Pg(\kappa)$ 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 $Kg(par)$ chooses $sk \xleftarrow{\$} \mathbb{Z}_q$, computes $pk \leftarrow g_2^{sk}$, and outputs (pk, sk) .

Key Aggregation. $KAg(\{pk_1, \dots, pk_n\})$ outputs

$$apk \leftarrow \prod_{i=1}^n pk_i^{H_1(pk_i, \{pk_1, \dots, pk_n\})}.$$

Signing. Signing is a single round protocol. $Sign(par, \{pk_1, \dots, pk_n\}, sk_i, m)$ computes $s_i \leftarrow H_0(m)^{a_i \cdot sk_i}$, where $a_i \leftarrow H_1(pk_i, \{pk_1, \dots, pk_n\})$. 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. $Vf(par, apk, m, \sigma)$ outputs 1 iff

$$e(\sigma, g_2^{-1}) \cdot e(H_0(m), apk) \stackrel{?}{=} 1_{\mathbb{G}_t}.$$

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, σ_i, apk_i) for $i = 1, \dots, b$, where apk_i is the aggregated public-key used to verify the multi-signature σ_i on m_i . If all the messages m_1, \dots, 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

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

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, \dots, m_b are distinct. If some messages are repeated then batch verification can be done by first choosing random exponents $\rho_1, \dots, \rho_b \stackrel{?}{\leftarrow} \{1, \dots, 2^\kappa\}$, where κ is a security parameter, computing $\tilde{\sigma} = \sigma_1^{\rho_1} \cdots \sigma_b^{\rho_b} \in \mathbb{G}_2$, and checking that

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

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 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 co-CDH is $((\tau + q_H\tau_{\exp_1} + q_S(\tau_{\exp_1^l} + \tau_{\exp_1}) + \tau_{\exp_2^l}) \cdot 8q_H^2/\epsilon \cdot \ln(8q_H/\epsilon), \epsilon/(8q_H))$ -hard, where l is the maximum number of signers involved in a single multisignature, τ_{\exp_1} and τ_{\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 IG that generates random tuples $(A, B_1, B_2) = (g_1^\alpha, g_1^\beta, g_2^\beta)$ where $\alpha, \beta \stackrel{?}{\leftarrow} \mathbb{Z}_q$, and an algorithm \mathcal{A} that on input (A, B_1, B_2) and randomness $f = (\rho, h_1, \dots, h_{q_S})$ proceeds as follows.

Algorithm \mathcal{A} picks an index $k \stackrel{?}{\leftarrow} \{1, \dots, q_H\}$ and runs the forger \mathcal{F} on input $pk^* \leftarrow B_2$ with random tape ρ . It responds to \mathcal{F} 's i -th H_0 query by choosing $r_i \stackrel{?}{\leftarrow} \mathbb{Z}_q$ and returning $g_1^{r_i}$ if $i \neq k$. The k -th H_0 query is answered by returning A . We assume w.l.o.g. that \mathcal{F} makes no repeated H_0 queries. \mathcal{A} responds to \mathcal{F} 's H_1 queries as follows. We distinguish three types of 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 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_f be the index such that $\mathsf{H}_1(pk^*, \mathcal{PK}) = h_{j_f}$, let $apk \leftarrow \mathsf{KAg}(par, \mathcal{PK})$, and let $a_j \leftarrow \mathsf{H}_1(pk_j, \mathcal{PK})$ for $\mathcal{PK} = \{pk_1, \dots, pk_n\}$. Then \mathcal{A} outputs $(J = \{j_f\}, \{(\sigma, \mathcal{PK}, apk, a_1, \dots, a_n)\})$.

The running time of \mathcal{A} is that of \mathcal{F} plus the additional computation \mathcal{A} makes. Let q_H denote the total hash queries \mathcal{F} makes, i.e., the queries to H_0 and H_1 combined. \mathcal{A} needs one exponentiation in \mathbb{G}_1 to answer H_0 queries, so it spends at most $q_H \cdot \tau_{\text{exp}_1}$ 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 $\tau_{\text{exp}_2^l}$, and one exponentiation in \mathbb{G}_1 costing τ_{exp_1} , giving a total of $qs \cdot (\tau_{\text{exp}_2^l} + \tau_{\text{exp}_1})$. Finally, \mathcal{A} computes the output values, which costs an additional $\tau_{\text{exp}_2^l}$ to compute apk . \mathcal{A} 's runtime is therefore $\tau + q_H \tau_{\text{exp}_1} + qs(\tau_{\text{exp}_2^l} + \tau_{\text{exp}_1}) + \tau_{\text{exp}_2^l}$. 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 $1/q_H$, making \mathcal{A} 's overall success probability $\epsilon_{\mathcal{A}} = \epsilon/q_H$.

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 identically to the output of IG. If $\mathcal{GF}_{\mathcal{A}}$ outputs $(0, \perp)$, then \mathcal{B} outputs **fail**. If $\mathcal{GF}_{\mathcal{A}}$ outputs $(\{j_f\}, \{out\}, \{out'\})$, then \mathcal{B} proceeds as follows. \mathcal{B} parses out as $(\sigma, \mathcal{PK}, apk, a_1, \dots, a_n)$ and out' as $(\sigma', \mathcal{PK}', apk', a'_1, \dots, a'_{n'})$. 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 $f|_{j_f} = f'|_{j_f}$, meaning that these executions are identical up to the j_f -th 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 $a_i = h_{j_f}$ and $a'_i = h'_{j_f}$, 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 $q > 8q_H/\epsilon$, then \mathcal{B} runs in time at most $(\tau + q_H \tau_{\text{exp}_1} + q_S(\tau_{\text{exp}_2^l} + \tau_{\text{exp}_1}) + \tau_{\text{exp}_2^l}) \cdot 8q_H^2/\epsilon \cdot \ln(8q_H/\epsilon)$ and succeeds with probability $\epsilon' \geq \epsilon/(8q_H)$.

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 Pg, Kg, and KAg, algorithms, but \mathcal{AMSP} has slightly modified Sign and Vf algorithms that include apk in the signed message, and has additional algorithms SAg and AVf to aggregate signatures and verify aggregate signatures, respectively.

Signing. $\text{Sign}(par, \mathcal{PK}, sk_i, m)$ computes $s_i \leftarrow H_0(apk, m)^{a_i \cdot sk_i}$, where $apk \leftarrow \text{KAg}(par, \mathcal{PK})$ and $a_i \leftarrow H_1(pk_i, \{pk_1, \dots, pk_n\})$. The designated combiner collects all signatures s_i and computes the final signature $\sigma \leftarrow \prod_{j=1}^n s_j$.

Multi-signature Verification. $\text{Vf}(par, apk, m, \sigma)$ outputs 1 if and only if $e(\sigma, g_2^{-1}) \cdot e(H_0(apk, m), apk) \stackrel{?}{=} 1_{\mathbb{G}_t}$.

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

Aggregate Signature Verification. $\text{AVf}(\{(apk_i, m_i)\}_{i=1}^n, \Sigma)$ outputs 1 if and only if $e(\Sigma, g_2^{-1}) \cdot \prod_{i=1}^n e(H_0(apk_i, m_i), apk_i) \stackrel{?}{=} 1_{\mathbb{G}_t}$.

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

Theorem 2. \mathcal{AMSP} is a secure aggregate multisignature scheme under the computational ψ -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 ψ -co-Diffie-Hellman problem is $((\tau + q_H \tau_{\text{exp}_1} + q_S(\tau_{\text{exp}_2^l} + \tau_{\text{exp}_1} + \tau_{\text{exp}_2^n}) + \tau_{\text{exp}_2^l}) \cdot 8q_H^2/\epsilon \cdot \ln(8q_H/\epsilon), \epsilon/(8q_H))$ -hard, where l is the maximum number of signers involved in a single multisignature, n is the amount of multisignatures aggregated into the forgery, τ_{exp_1} and τ_{exp_2} denote the time required to compute exponentiations in \mathbb{G}_1 and \mathbb{G}_2 respectively, and $\tau_{\text{exp}_1^i}$ and $\tau_{\text{exp}_2^i}$ 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 forgery instead of a plain multisignature forgery. That is, \mathcal{F} outputs an aggregate multisignature Σ , a set of aggregate public keys and message pairs $\{(apk_1, m_1), \dots, (apk_n, m_n)\}$, a set

of public keys \mathcal{PK} , and a message m^* . Let $apk^* \leftarrow \text{KAg}(\text{par}, \mathcal{PK})$. If \mathcal{A} correctly guessed that the k -th 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 σ , which costs $\tau_{\exp_1^n}$.

4 Accountable-Subgroup Multisignatures

Micali, Ohta, and Reyzin [39] 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 [39] to support public-key aggregation and an interactive group setup procedure.

An ASM scheme consists of algorithms Pg , Kg , GSetup , Sign , KAg , and Vf . The common system parameters are generated as $\text{par} \xleftarrow{\$} \text{Pg}$. Each signer generates a key pair $(pk, sk) \xleftarrow{\$} \text{Kg}(\text{par})$. To participate in a group of signers $\mathcal{PK} = \{pk_1, \dots, pk_n\}$, each signer in \mathcal{PK} runs the interactive algorithm $\text{GSetup}(sk, \mathcal{PK})$ to obtain a membership key mk . We assume that each signer in

\mathcal{PK} is assigned a publicly computable index $i \in \{1, \dots, |\mathcal{PK}|\}$, e.g., the index of pk in a sorted list of \mathcal{PK} . Any subgroup of signers $S \subseteq \{1, \dots, |\mathcal{PK}|\}$ of \mathcal{PK} can then collectively sign a message m by each calling the interactive algorithm $\text{Sign}(par, \mathcal{PK}, S, sk, mk, m)$, where mk is the signer's membership key for this group of signers, to obtain a signature σ . The key aggregation algorithm, on input the public keys of a group of signers \mathcal{PK} , outputs an aggregate public key apk . A signature σ is verified by running $\text{Vf}(par, apk, S, m, \sigma)$ which outputs 0 or 1.

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

Security. Unforgeability is described by the following game.

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

Group Setup. The adversary can perform the group setup protocol $\text{GSetup}(sk^*, \mathcal{PK})$ 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 $S \subseteq \{1, \dots, |\mathcal{PK}|\}$ so that $i \in S$, where i is the index of pk^* in \mathcal{PK} . The challenger runs $\text{Sign}(par, \mathcal{PK}, S, sk^*, mk^*, m)$ to play the role of the i -th signer and hands the resulting signature σ to \mathcal{A} .

Output. The adversary outputs a set of public keys \mathcal{PK} , a set $S \subseteq \{1, \dots, |\mathcal{PK}|\}$, a message m and an ASM signature σ . It wins the game if $\text{Vf}(par, apk, S, m, \sigma) = 1$, where $apk \leftarrow \text{KAg}(\mathcal{PK})$, $pk^* \in \mathcal{PK}$ 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 $(\tau, q_G, q_S, q_H, \epsilon)$ -forger for accountable-subgroup multisignature scheme \mathcal{ASM} if it runs in time τ , makes q_G group setup queries, q_S signing queries, q_H random-oracle queries, and wins the above game with probability at least ϵ . \mathcal{ASM} is $(\tau, q_G, q_S, q_H, \epsilon)$ -unforgeable if no $(\tau, q_G, q_S, q_H, \epsilon)$ -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-signature now consists of the aggregation of the individual signers’ signatures and their membership keys and the aggregate public key of the subgroup 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 $H_0 : \{0, 1\}^* \rightarrow \mathbb{G}_1$, $H_1 : \{0, 1\}^* \rightarrow \mathbb{Z}_q$, and $H_2 : \{0, 1\}^* \rightarrow \mathbb{G}_1$. Parameter generation, key generation, and key aggregation are the same as for the aggregate multi-signature scheme in Sect. 3.

Group Setup. $GSetup(sk_i, \mathcal{PK} = \{pk_1, \dots, pk_n\})$ 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 $apk \leftarrow KAg(\mathcal{PK})$ as well as $a_i \leftarrow H_1(pk_i, \mathcal{PK})$. It then sends $\mu_{j,i} = H_2(apk, j)^{a_i \cdot sk_i}$ 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 $\mu_{i,i} \leftarrow H_2(apk, i)^{a_i \cdot sk_i}$ 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

$$e(mk_i, g_2) = e(H_2(apk, i), apk).$$

In other words, this mk_i is a valid multi-signature on the message (apk, i) by all n parties, as defined in the scheme in Sect. 3.1.

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

$$s_i \leftarrow H_0(apk, m)^{sk_i} \cdot mk_i,$$

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

$$PK \leftarrow \prod_{j \in S} pk_j, \quad s \leftarrow \prod_{j \in S} s_j,$$

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. $Vf(par, apk, S, m, \sigma)$ parses σ as (PK, s) and outputs 1 iff

$$e(H_0(apk, m), PK) \cdot e\left(\prod_{j \in S} H_2(apk, j), apk\right) \stackrel{?}{=} e(s, g_2)$$

and S is a set authorized to sign.

The presented ASM scheme satisfies correctness. If parties honestly execute the group setup and signing protocols, we have $PK = g_2^{\sum_{i \in S} sk_i}$, $apk = g_2^{\sum_{i=1, \dots, n} a_i \cdot pk_i}$, and $s = H_0(apk, m)^{\sum_{i \in S} sk_i} \cdot \prod_{i \in S} H_2(apk, i)^{\sum_{j \in S} a_j \cdot sk_j}$, which passes verification:

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

4.3 Security of Our ASM Scheme

Theorem 3. *Our ASM scheme is unforgeable under the hardness of the computational ψ -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 ψ -co-CDH is (τ', ϵ') -hard for*

$$\begin{aligned}
\tau' &= (\tau + \tau'') \cdot \frac{8q_H^2 q}{(q - q_S - q_H) \cdot \epsilon} \cdot \ln \frac{8q_H q}{(q - q_S - q_H) \cdot \epsilon}, \\
\tau'' &= q_H \cdot \max(\tau_{\text{exp}_2^1}, \tau_{\text{exp}_1^2}) + (lq_G + q_S) \cdot \tau_{\text{exp}_1} + q_S \cdot \tau_{\text{exp}_2^1} + 2 \cdot \tau_{\text{pair}} + \tau_{\text{exp}_1^3}, \\
\epsilon' &= \frac{\epsilon}{8q_H} - \frac{q_S + q_H}{8q_H q},
\end{aligned}$$

where l is the maximum number of signers involved in any group setup, τ_{exp_1} and τ_{exp_2} denote the time required to compute exponentiations in \mathbb{G}_1 and \mathbb{G}_2 respectively, and $\tau_{\text{exp}_1^i}$ and $\tau_{\text{exp}_2^i}$ denote the time required to compute i -multi-exponentiations in \mathbb{G}_1 and \mathbb{G}_2 respectively, and τ_{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 ψ -co-CDH problem by running $\mathcal{GF}_{\mathcal{A}}$. The proof essentially combines techniques related to the non-extractability of BGLS aggregate signatures [13, 21] with Maxwell et al.’s key aggregation technique [36].

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 $f = (\rho, h_1, \dots, h_{q_H})$, and given access to a homomorphism oracle $\mathcal{O}^\psi(\cdot)$, \mathcal{A} proceeds as follows. It guesses a random index $k \xleftarrow{\$} \{1, \dots, q_H\}$ 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 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 $r \xleftarrow{s} \mathbb{Z}_q$, 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 $r \xleftarrow{s} \mathbb{Z}_q$, 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 $r \xleftarrow{s} \mathbb{Z}_q$, 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 $r \xleftarrow{s} \mathbb{Z}_q$, 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 $j = 1, \dots, |\mathcal{PK}|, j \neq i$, 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., $\mathsf{Vf}(\mathit{par}, apk, S, m, \sigma) = 1$, $pk^* \in \mathcal{PK}$, $i \in S$ 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

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

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 $H(pk^*, \mathcal{PK}) = h_I$, then algorithm \mathcal{A} outputs $(I, (t, \mathcal{PK}, a_1, \dots, a_n))$.

\mathcal{A} 's runtime is \mathcal{F} 's runtime plus the additional computation \mathcal{A} performs. Let q_H denote the total hash queries \mathcal{F} makes, i.e., the queries to H_0 , H_1 , and H_2 combined. To answer a H_1 query, \mathcal{A} computes apk which costs at most $\tau_{\exp_2^1}$ for groups consisting of up to l signers. To answer H_0 and H_2 queries, \mathcal{A} performs at most $\tau_{\exp_1^2}$. \mathcal{A} therefore spends at most $q_H \cdot \max(\tau_{\exp_2^1}, \tau_{\exp_1^2})$ answering hash queries. For every group-setup query with l signers, \mathcal{A} computes apk costing $\tau_{\exp_1^1}$, and \mathcal{A} computes $\mu_{j,i}$ costing $(l-1)\tau_{\exp_1}$, meaning \mathcal{A} spends $q_G \cdot (l-1)\tau_{\exp_1}$ answering group setup queries. For signing queries with a \mathcal{PK} of size at most l , \mathcal{A} computes apk costing time $\tau_{\exp_2^1}$, and one exponentiation in \mathbb{G}_1 costing τ_{\exp_1} , giving a total of $q_S \cdot (\tau_{\exp_2^1} + \tau_{\exp_1})$. Finally, \mathcal{A} computes the output values, which involves verifying the forgery (costing $2\tau_{\text{pair}}$) and computing t (costing $\tau_{\exp_1^3}$), giving \mathcal{A} a total runtime of $\tau + q_H \cdot \max(\tau_{\exp_2^1}, \tau_{\exp_1^2}) + q_G \cdot (l-1)\tau_{\exp_1} + q_S \cdot (\tau_{\exp_2^1} + \tau_{\exp_1}) + 2\tau_{\text{pair}} + \tau_{\exp_1^3}$.

\mathcal{A} is successful if the **bad** event does not happen, if it guesses the index of the forgery correctly, and if \mathcal{F} successfully forges. Event **bad** happens with probability at most $(q_S + q_H)/q$ for every hash query, so it happens with probability $q_H(q_S + q_H)/q$. \mathcal{A} guesses the forgery index correctly with probability $1/q_H$, and \mathcal{F} forges with probability ϵ , giving \mathcal{A} success probability $(1 - (q_S + q_H)/q) \cdot \epsilon / q_H$.

Using the generalized forking lemma from Lemma 1, we can build an algorithm \mathcal{B} that solves the ψ -co-CDH problem by, on input $(A = g_1^\alpha, B_1 = g_1^\beta, B_2 = g_2^\beta)$, running $\mathcal{GF}_{\mathcal{A}}(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2, A, B_1, B_2)$ to obtain two outputs $(I, (t, \mathcal{PK}, a_1, \dots, a_n))$ and $(I, (t', \mathcal{PK}', a'_1, \dots, a'_n))$, giving $\mathcal{GF}_{\mathcal{A}}$ access to the homomorphism oracle $\mathcal{O}^\psi(\cdot)$ offered by ψ -co-CDH. Since the two executions of \mathcal{A} are identical up to the first query $H_1(pk, \mathcal{PK})$ 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 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 $q > 8q_H/\epsilon$, then \mathcal{B} runs in time at most $(\tau + q_H \cdot \max(\tau_{\exp_2^1}, \tau_{\exp_1^2}) + q_G \cdot (l-1)\tau_{\exp_1} + q_S \cdot (\tau_{\exp_2^1} + \tau_{\exp_1}) + 2\tau_{\text{pair}} + \tau_{\exp_1^3}) \cdot 8q_H^2 / ((1 - (q_S + q_H)/q) \cdot \epsilon) \cdot \ln(8q_H / ((1 - (q_S + q_H)/q) \cdot \epsilon))$ and succeeds with probability $(1 - (q_S + q_H)/q) \cdot \epsilon / (8q_H)$, proving the bounds in the theorem.

4.4 Partial Aggregation of ASM Signatures

Looking at the description of the ASM scheme above, one would expect that one can further aggregate the second components when given several such ASM signatures. The first components are needed separately for verification, though, so even though we don't obtain full aggregation to constant-size signatures, we would shave a factor two off of the total signature length.

The straightforward way to partially aggregate ASM signatures is insecure, however, because the link between membership keys and signed messages is lost. For example, an aggregate ASM signature (PK_1, PK_2, s) for a set of tuples $\{(apk, S_1, m_1), (apk, S_2, m_2)\}$ would also be a valid signature for $\{(apk, S_1, m_2), (apk, S_2, m_1)\}$, leading to easy forgery attacks.

We show that a variation on Maxwell et al.'s key aggregation technique [36] can be used to create a provably secure scheme. We define an aggregate accountable-subgroup multi-signature (AASM) scheme as an ASM scheme with two additional algorithms SAg and AVf , where SAg takes as input a set of tuples $\{(apk_i, S_i, m_i, \sigma_i)_{i=1}^n\}$ where apk_i is an aggregate public key, S_i is a set of signers, m_i is a message, and σ_i is an accountable-subgroup multi-signature, and outputs an aggregate multi-signature Σ , while AVf takes a set of tuples (apk, S, m) and an AASM signature Σ , and outputs 0 or 1 indicating that the signature is invalid or valid, respectively.

Apart from satisfying the natural correctness definition, AASM schemes must satisfy an unforgeability notion that is similar to that of ASM schemes, but where the adversary outputs a signature Σ , a set of tuples $\{(apk_i, S_i, m_i)\}$, a set of public keys \mathcal{PK}^* , a set of signers S^* , and a message m^* . The adversary wins if $pk^* \in \mathcal{PK}^*$, \mathcal{A} made no signing queries on m^* , and $\text{AVf}(\text{par}, \{(apk_i, S_i, m_i)\} \cup \{(\text{KAg}(\text{par}, \mathcal{PK}^*), S^*, m^*)\}, \Sigma) = 1$.

Our \mathcal{AASM} scheme uses the Pg , Kg , GSetup , Sign , KAg , and Vf algorithms of \mathcal{ASM} , and adds the following two algorithms as well as a hash function $H_3 : \{0, 1\} \rightarrow \mathbb{Z}_q$.

Signature Aggregation. $\text{SAg}(\text{par}, \{(apk_i, S_i, m_i, \sigma_i)_{i=1}^n\})$ parses σ_i as (PK_i, s_i) and for $i = 1, \dots, n$ computes

$$b_i \leftarrow H_3((apk_i, S_i, m_i, PK_i), \{(apk_j, S_j, m_j, PK_j)_{j=1}^n\}).$$

It aggregates the signatures by computing $s \leftarrow \prod_{i=1}^n s_i^{b_i}$, and outputs $\Sigma \leftarrow (PK_1, \dots, PK_n, s)$.

Aggregate Signature Verification. $\text{AVf}(\{(apk_i, S_i, m_i)_{i=1}^n\}, \Sigma)$ parses Σ as (PK_1, \dots, PK_n, s) , computes

$$b_i \leftarrow H_3((apk_i, S_i, m_i, PK_i), \{(apk_j, S_j, m_j, PK_j)_{j=1}^n\})$$

for $i = 1, \dots, n$, and outputs 1 if and only if

$$\prod_{i=1}^n \left(e(H_0(apk_i, m_i), PK_i^{b_i}) \cdot e(\prod_{j \in S_i} H_2(apk_i, j), apk_i^{b_i}) \right) \stackrel{?}{=} e(s, g_2).$$

Theorem 4. Our \mathcal{AASM} scheme is unforgeable in the random-oracle model if \mathcal{ASM} is unforgeable in the random-oracle model. More precisely, it is $(\tau, q_S, q_H, \epsilon)$ -unforgeable in the random-oracle model if $q > 8lq_H/\epsilon$ and \mathcal{ASM} is $((\tau + \tau_{\text{exp}_1} + O(lq_H)) \cdot 8lq_H/\epsilon \cdot \ln(8l/\epsilon), q_S, q_H, \epsilon/8l)$ -unforgeable in the random-oracle model, where l is the maximum number of multi-signatures that can be aggregated and τ_{exp_1} denotes the time required to compute an exponentiation in \mathbb{G}_1 .

Proof. Given a forger \mathcal{F} for \mathcal{AASM} , we construct a forger \mathcal{G} for \mathcal{ASM} as follows. We first build a wrapper algorithm \mathcal{A} to be used in the forking lemma, and then construct \mathcal{G} based on $\mathcal{GF}_{\mathcal{A}}$. We actually use a slight variation on the forking lemma (Lemma 1) by giving \mathcal{A} access to oracles. To ensure that the executions of \mathcal{A} are identical up to the forking points j_i , the forking algorithm $\mathcal{GF}_{\mathcal{A}}$ remembers the oracle responses during \mathcal{A} 's first run, and returns the same responses in all subsequent runs up to the respective forking points j_i . One can see that the same bounds hold as in Lemma 1.

Algorithm \mathcal{A} , on input a target public key pk^* and $f = (\rho, h_1, \dots, h_{q_H})$ and given access to oracles $H_0, H_1, H_2, \text{GSetup}$, and Sign , runs the forger \mathcal{F} on input pk^* by relaying queries and responses for the mentioned oracles, and responding to H_3 queries as:

- $H_3(x)$: If x can be parsed as (y, \mathcal{APK}) with $\mathcal{APK} = \{(apk_i, S_i, m_i, PK_i)\}_{i=1}^n$ and \mathcal{F} did not make any previous query $H_1(y', \mathcal{APK})$, then \mathcal{A} guesses an index $i^* \xleftarrow{\$} \{1, \dots, n\}$ and sets $H_3((apk_{i^*}, S_{i^*}, m_{i^*}, PK_{i^*}), \mathcal{APK})$ to the next unused value from h_1, \dots, h_{q_H} . For all other indices $j \in \{1, \dots, n\} \setminus \{i^*\}$, it assigns $H_3((apk_j, S_j, m_j, PK_j), \mathcal{APK}) \xleftarrow{\$} \mathbb{Z}_q$. If $H_3(x)$ did not yet get assigned a value, then \mathcal{A} assigns a random value $H_3(x) \xleftarrow{\$} \mathbb{Z}_q$. Finally, \mathcal{A} makes queries $H_0(apk_i, m_i)$ and $H_2(apk_i, j)$ for all $i = 1, \dots, n$ and $j \in S_i$, just to fix their values at this point.

When \mathcal{F} outputs a valid forgery $\Sigma, \{(apk_i, S_i, m_i)\}_{i=1}^n, \mathcal{PK}^*, S^*$, and m^* , \mathcal{A} looks up in its records for H_3 to check whether $H_3((apk^*, S^*, m^*, PK^*), \mathcal{APK}^*)$ was the random-oracle query for which \mathcal{A} returned a value from h_1, \dots, h_{q_H} , where $\mathcal{APK}^* = \{(apk_i, S_i, m_i)\}_{i=1}^n, (apk^*, S^*, m^*)\}$ and $apk^* \leftarrow \text{KAg}(\mathcal{PK}^*)$. If so, then let j_f be the index of that query and let b^* be the response to that query, and let \mathcal{A} return $(\{j_f\}, \{(\mathcal{PK}^*, S^*, m^*, PK^*, s, b^*)\})$. Otherwise, \mathcal{A} returns (\emptyset, \emptyset) . The success probability of \mathcal{A} is $\epsilon_{\mathcal{A}} \geq \epsilon/l$, while its running time is $\tau_{\mathcal{A}} = \tau + O(lq_H)$.

For the forgery to be valid, it must hold that

$$\begin{aligned} s &= H_0(apk^*, m^*)^{b^* \log PK^*} \cdot \prod_{j \in S^*} H_2(apk^*, j)^{b^* \log apk^*} \\ &\quad \cdot \prod_{i=1}^n \left(H_0(apk_i, m_i)^{\log PK_i} \cdot \prod_{j \in S_i} H_2(apk_i, j)^{\log apk_i} \right)^{b_i}, \end{aligned} \tag{4}$$

where $b_i = H_3((apk_i, S_i, m_i, PK_i), \mathcal{APK}^*)$.

Now consider the forger \mathcal{G} against \mathcal{ASM} that runs $\mathcal{GF}_{\mathcal{A}}$ to obtain two outputs $(\mathcal{PK}^*, S^*, m^*, PK^*, s, b^*)$ and $(\mathcal{PK}^*, S^*, m^*, PK^*, s', b'^*)$. From the way \mathcal{A}

simulated \mathcal{F} 's oracle queries, one can see that all variables and random-oracle responses in Eq. (4) are the same in both executions of \mathcal{A} , except that $s \neq s'$ and $b^* \neq b^{* \prime}$. By dividing both equations, we have that

$$\frac{s}{s'} = \mathsf{H}_0(apk^*, m^*)^{(b^* - b^{* \prime}) \log PK^*} \cdot \prod_{j \in S^*} \mathsf{H}_2(apk^*, j)^{(b^* - b^{* \prime}) \log apk^*},$$

so that \mathcal{G} can output $\mathcal{PK}^*, S^*, m^*, \sigma = (PK^*, (s/s')^{1/(b^* - b^{* \prime})})$ as its forgery against the \mathcal{ASM} scheme. The bounds stated by the theorem follow from Lemma 1.

5 A Scheme from Discrete Logarithms

The basic key aggregation technique of our pairing-based schemes is due to Maxwell et al. [36], 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 [22]. 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. [37] 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 $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2 : \{0, 1\}^* \rightarrow \mathbb{Z}_q$, 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 κ -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. $\mathsf{KAg}(\{pk_1, \dots, pk_n\})$ outputs

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

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

Round 1. Choose $r_i \xleftarrow{\$} \mathbb{Z}_q$ and compute $R_i \leftarrow g^{r_i}$. Let $t_i \leftarrow H_2(R_i)$. Send t_i to all other signers corresponding to pk_1, \dots, 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, \dots, pk_n and wait to receive R_j from all other signers $j \neq i$. Check that $t_j = H_2(R_j)$ for all $j = 1, \dots, n$.

Round 3. Compute $apk \leftarrow \text{KAg}(\{pk_1, \dots, pk_n\})$ and let $a_i \leftarrow H_1(pk_i, \{pk_1, \dots, pk_n\})$. 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 $c \leftarrow H_0(\bar{R}, apk, m)$. 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. $\text{Vf}(par, apk, m, \sigma)$ parses σ as $(\bar{R}, s) \in \mathbb{G} \times \mathbb{Z}_q$, computes $c \leftarrow H_0(\bar{R}, apk, m)$ 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 $c_i \leftarrow H_1(\bar{R}_i, apk_i, m_i)$, pick $\alpha_i \xleftarrow{\$} \mathbb{Z}_q$ for $i = 1, \dots, n$, 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 [36] by applying the forking lemma twice: once by forking on a random-oracle query $H_0(\bar{R}, apk, m)$ to obtain two forgeries from which the discrete logarithm w of apk can be extracted, and then once again by forking on a query $H_1(pk_i, \{pk_1, \dots, pk_n\})$ to obtain two such pairs (apk, w) and (apk', w') from which the discrete logarithm of the target public key can be extracted.

Theorem 5. \mathcal{MSDL} is an unforgeable multisignature scheme (as defined in Definition 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 $((\tau + 4lq_T \cdot \tau_{\text{exp}} + O(lq_T)) \cdot 512q_T^2 / (\epsilon - \delta)) \cdot \ln^2(64 / (\epsilon - \delta))$, $(\epsilon - \delta)/64$ -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 τ_{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 $in = (y, h_{1,1}, \dots, h_{1,q_H})$ and randomness $f = (\rho, h_{0,1}, \dots, h_{0,q_H})$ runs \mathcal{F} on input $pk^* = y$ and random tape ρ , 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 $pk \in \mathcal{PK} \setminus \{pk^*\}$. 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 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 $\mathcal{PK} = \{pk_1, \dots, pk_n\}$, \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 $i = 1, \dots, n$. If j is the index such that $c = h_{0,j}$, then \mathcal{A} returns $(j, (\bar{R}, c, s, apk, \mathcal{PK}, a_1, \dots, a_n))$.

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 $(\tau, q_S, q_H, \epsilon)$ -forger, then \mathcal{A} succeeds with probability

$$\begin{aligned} \epsilon_{\mathcal{A}} &= \Pr[\mathcal{F} \text{ succeeds} \wedge \overline{\mathbf{bad}_1} \wedge \overline{\mathbf{bad}_2} \wedge \overline{\mathbf{bad}_3}] \\ &\geq \Pr[\mathcal{F} \text{ succeeds}] - \Pr[\mathbf{bad}_1] - \Pr[\mathbf{bad}_2] - \Pr[\mathbf{bad}_3] \\ &\geq \epsilon - \frac{q_H(q_H + q_S + 1)}{q} - \left(\frac{(q_H + q_S)^2}{2q} + \frac{lq_H q_S}{q} \right) - \frac{q_H(q_H + q_S + 1)}{q} \\ &\geq \epsilon - \frac{4lq_T^2}{q} = \epsilon - \delta \end{aligned}$$

where $q_T = q_H + q_S + 1$ and $\delta = 4l(q_H + q_S + 1)^2/q$. The running time of \mathcal{A} is $\tau_{\mathcal{A}} = \tau + 4lq_T \cdot \tau_{\text{exp}} + O(lq_T)$.

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 $f = (\rho, h_{1,1}, \dots, h_{1,q_H})$, runs $\mathcal{GF}_{\mathcal{A}}$ on input $in' = (y, h_{1,1}, \dots, h_{1,q_H})$ to obtain output

$$(j, (\bar{R}, c, s, apk, \mathcal{PK}, a_1, \dots, a_n), (\bar{R}', c', s', apk', \mathcal{PK}', a'_1, \dots, a'_n)).$$

In its two executions by $\mathcal{GF}_{\mathcal{A}}$, \mathcal{F} 's view is identical up to the j -th H_0 query $H_0(\bar{R}, apk, m)$, 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 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 $i = 1, \dots, n$. 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 $w \leftarrow (s - s')/(c - c')$ mod q is the discrete logarithm of apk . If i is the index such that $H_1(pk^*, \mathcal{PK}) = h_{1,i}$, then \mathcal{B} outputs $(i, (w, \mathcal{PK}, a_1, \dots, a_n))$. It does so whenever $\mathcal{GF}_{\mathcal{A}}$ is successful, which according to Lemma 1 occurs with probability $\epsilon_{\mathcal{B}}$ and running time $\tau_{\mathcal{B}}$:

$$\begin{aligned} \epsilon_{\mathcal{B}} &\geq \frac{\epsilon_{\mathcal{A}}}{8} \geq \frac{\epsilon - \delta}{8} \\ \tau_{\mathcal{B}} &= \tau_{\mathcal{A}} \cdot 8q_H/\epsilon_{\mathcal{A}} \cdot \ln(8/\epsilon_{\mathcal{A}}) \\ &\leq (\tau + 4lq_T \cdot \tau_{\text{exp}} + O(lq_T)) \cdot \frac{8q_T}{\epsilon - \delta} \cdot \ln \frac{8}{\epsilon - \delta}. \end{aligned}$$

Now consider the discrete-logarithm algorithm \mathcal{D} that, on input y , runs $\mathcal{GF}_{\mathcal{B}}$ on input y to obtain output $(i, (w, \mathcal{PK}, a_1, \dots, a_n), (w, \mathcal{PK}', a'_1, \dots, a'_n))$. Both executions of \mathcal{B} in $\mathcal{GF}_{\mathcal{B}}$ are identical up to the i -th H_1 query $H_1(pk, \mathcal{PK})$, so we have that $\mathcal{PK} = \mathcal{PK}'$. Because \mathcal{A} immediately assigns outputs of 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 $H_1(pk^*, \mathcal{PK})$, 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 $x \leftarrow (w - w')/(a_i - a'_i)$ mod 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 $\tau_{\mathcal{D}}$:

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

Acknowledgments. Boneh was supported by NSF, DARPA, a grant from ONR, the Simons Foundation, and a Google faculty fellowship. Drijvers and Neven were supported by the ERC under Grant PERCY #321310.

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, Chicago, Illinois, USA, 4–8 Oct 2010, pp. 473–484. ACM Press (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, Alexandria, Virginia, USA, 27–31 Oct 2008, pp. 449–458. ACM Press (2008)
4. Bagherzandi, A., Jarecki, S.: Multisignatures using proofs of secret key possession, as secure as the Diffie-Hellman problem. In: Ostrovsky, R., De Prisco, R., Visconti, I. (eds.) SCN 2008. LNCS, vol. 5229, pp. 218–235. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85855-3_15
5. El Bansarkhani, R., Sturm, J.: An efficient lattice-based multisignature scheme with applications to bitcoins. In: Foresti, S., Persiano, G. (eds.) CANS 2016. LNCS, vol. 10052, pp. 140–155. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48965-0_9
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. LNCS, vol. 3006, pp. 17–25. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24654-1_2
7. Bellare, M., Namprempre, C., Neven, G.: Unrestricted aggregate signatures. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 411–422. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73420-8_37
8. Bellare, M., Namprempre, C., Pointcheval, D., Semanko, M.: The one-more-RSA-inversion problems and the security of Chaum’s blind signature scheme. J. Cryptol. **16**(3), 185–215 (2003)
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, Alexandria, Virginia, USA, Oct 30–Nov 3 2006, pp. 390–399. ACM Press (2006)
10. Boldyreva, A.: Threshold signatures, multisignatures and blind signatures based on the Gap-Diffie-Hellman-Group signature scheme. In: Desmedt, Y.G. (ed.) PKC 2003. LNCS, vol. 2567, pp. 31–46. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36288-6_3
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, Alexandria, Virginia, USA, 28–31 Oct 2007, pp. 276–285. ACM Press (2007)

12. Boneh, D., Drijvers, M., Neven, G.: Compact multi-signatures for smaller blockchains. Cryptology ePrint Archive, Report 2018/483 (2018). <https://eprint.iacr.org/2018/483>
13. Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 416–432. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39200-9_26
14. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the Weil pairing. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 514–532. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45682-1_30
15. Brogk, K., Goldberg, S., Reyzin, L.: Sequential aggregate signatures with lazy verification from trapdoor permutations. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 644–662. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34961-4_39
16. 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>
17. Burmester, M., et al.: A Structured ElGamal-type multisignature scheme. In: Imai, H., Zheng, Y. (eds.) PKC 2000. LNCS, vol. 1751, pp. 466–483. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-46588-1_31
18. Castelluccia, C., Jarecki, S., Kim, J., Tsudik, G.: A robust multisignature scheme with applications to acknowledgement aggregation. In: Blundo, C., Cimato, S. (eds.) SCN 2004. LNCS, vol. 3352, pp. 193–207. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30598-9_14
19. Certicom Research: Sec 2: Recommended elliptic curve domain parameters. Technical report, Certicom Research (2010)
20. 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 1998. LNCS, vol. 1431, pp. 154–163. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0054022>
21. 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.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 392–397. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40061-5_25
22. Drijvers, M., Edalatnejad, K., Ford, B., Neven, G.: On the provable security of two-round multi-signatures. Cryptology ePrint Archive, Report 2018/417 (2018). <https://eprint.iacr.org/2018/417>
23. Fuentes-Castañeda, L., Knapp, E., Rodríguez-Henríquez, F.: Faster hashing to \mathbb{G}_2 . In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 412–430. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28496-0_25
24. 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. LNCS, vol. 10770, pp. 34–57. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-76581-5_2
25. Gentry, C., Ramzan, Z.: Identity-based aggregate signatures. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 257–273. Springer, Heidelberg (2006). https://doi.org/10.1007/11745853_17
26. Hardjono, T., Zheng, Y.: A practical digital multisignature scheme based on discrete logarithms (extended abstract). In: Seberry, J., Zheng, Y. (eds.) AUSCRYPT 1992. LNCS, vol. 718, pp. 122–132. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57220-1_56

27. Harn, L.: Group-oriented (t, n) threshold digital signature scheme and digital multisignature. IEE Proc.-Comput. Digit. Tech. **141**(5), 307–313 (1994)
28. Horster, P., Michels, M., Petersen, H.: Meta-multisignature schemes based on the discrete logarithm problem. Information Security — the Next Decade. IFIP AICT, pp. 128–142. Springer, Boston (1995). https://doi.org/10.1007/978-0-387-34873-5_11
29. Itakura, K., Nakamura, K.: A public-key cryptosystem suitable for digital multisignatures. Technical report, NEC Research and Development (1983)
30. Komano, Y., Ohta, K., Shimbo, A., Kawamura, S.: Formal security model of multisignatures. In: Katsikas, S.K., López, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) ISC 2006. LNCS, vol. 4176, pp. 146–160. Springer, Heidelberg (2006). https://doi.org/10.1007/11836810_11
31. 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. LNCS, vol. 5671, pp. 35–51. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03298-1_3
32. Li, C.-M., Hwang, T., Lee, N.-Y.: Threshold-multisignature schemes where suspected forgery implies traceability of adversarial shareholders. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 194–204. Springer, Heidelberg (1995). <https://doi.org/10.1007/BFb0053435>
33. Lu, S., Ostrovsky, R., Sahai, A., Shacham, H., Waters, B.: Sequential aggregate signatures and multisignatures without random oracles. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 465–485. Springer, Heidelberg (2006). https://doi.org/10.1007/11761679_28
34. Lysyanskaya, A., Micali, S., Reyzin, L., Shacham, H.: Sequential aggregate signatures from trapdoor permutations. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 74–90. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_5
35. Ma, C., Weng, J., Li, Y., Deng, R.: Efficient discrete logarithm based multi-signature scheme in the plain public key model. Des. Codes Cryptogr. **54**(2), 121–133 (2010)
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/2018/068/20180118:124757>
37. 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>
38. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988). https://doi.org/10.1007/3-540-48184-2_32
39. Micali, S., Ohta, K., Reyzin, L.: Accountable-subgroup multisignatures: extended abstract. In: ACM CCS 01: 8th Conference on Computer and Communications Security, Philadelphia, PA, USA, 5–8 Nov 2001, pp. 245–254. ACM Press (2001)
40. Michels, M., Horster, P.: On the risk of disruption in several multiparty signature schemes. In: Kim, K., Matsumoto, T. (eds.) ASIACRYPT 1996. LNCS, vol. 1163, pp. 334–345. Springer, Heidelberg (1996). <https://doi.org/10.1007/BFb0034859>
41. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008). <http://bitcoin.org/bitcoin.pdf>
42. Neven, G.: Efficient sequential aggregate signed data. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 52–69. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78967-3_4

43. Ohta, K., Okamoto, T.: A digital multisignature scheme based on the Fiat-Shamir scheme. In: Imai, H., Rivest, R.L., Matssumoto, T. (eds.) ASIACRYPT 1991. LNCS, vol. 739, pp. 139–148. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57332-1_11
44. Ohta, K., Okamoto, T.: Multi-signature schemes secure against active insider attacks. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **82**(1), 21–31 (1999)
45. Park, S., Park, S., Kim, K., Won, D.: Two efficient RSA multisignature schemes. In: Han, Y., Okamoto, T., Qing, S. (eds.) ICICS 1997. LNCS, vol. 1334, pp. 217–222. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0028477>
46. Pointcheval, D., Stern, J.: Security arguments for digital signatures and blind signatures. J. Cryptol. **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.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 228–245. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72540-4_13
48. Schnorr, C.P.: Efficient signature generation by smart cards. J. Cryptol. **4**(3), 161–174 (1991)
49. Scott, M., Benger, N., Charlemagne, M., Dominguez Perez, L.J., Kachisa, E.J.: Fast hashing to g_2 on pairing-friendly curves. In: Shacham, H., Waters, B. (eds.) Pairing 2009. LNCS, vol. 5671, pp. 102–113. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03298-1_8