Subset-optimized BLS Multi-signature with Key Aggregation

Foteini Baldimtsi[1,2], Konstantinos Kryptos Chalkias[1], François Garillot[5⋆], Jonas Lindstrøm[1], Ben Riva[1], Arnab Roy[1], Alberto Sonnino[1,3], Pun Waiwitlikhit[1,4⋆], and Joy Wang[1]

[1] Mysten Labs Research, `research@mystenlabs.com`
[2] George Mason University
[3] University College London
[4] Stanford University
[5] Lurk Labs

Abstract. We propose a variant of the original Boneh, Drijvers, and Neven (Asiacrypt '18) BLS multi-signature aggregation scheme best suited to applications where the full set of potential signers is fixed and known and any subset I of this group can create a multi-signature over a message m. This setup is very common in proof-of-stake blockchains where a $2f + 1$ majority of $3f$ validators sign transactions and/or blocks and is secure against *rogue-key* attacks without requiring a proof of key possession mechanism.
In our scheme, instead of randomizing the aggregated signatures, we have a one-time randomization phase of the public keys: each public key is replaced by a sticky randomized version (for which each participant can still compute the derived private key). The main benefit compared to the original Boneh at al. approach is that since our randomization process happens only once and not per signature we can have significant savings during aggregation and verification. Specifically, for a subset I of t signers, we save t exponentiations in $\mathbb{G}_2$ at aggregation and t exponentiations in $\mathbb{G}_1$ at verification or vice versa, depending on which BLS mode we prefer: *minPK* (public keys in $\mathbb{G}_1$) or *minSig* (signatures in $\mathbb{G}_1$).
Interestingly, our security proof requires a significant departure from the co-CDH based proof of Boneh at al. When n (size of the universal set of signers) is small, we prove our protocol secure in the Algebraic Group and Random Oracle models based on the hardness of the Discrete Log problem. For larger n, our proof also requires the Random Modular Subset Sum (RMSS) problem.

Keywords: BLS · multi-signatures · signature aggregation · blockchain

1 Introduction

A *multi-signature* scheme [Ita83] allows a set of n signers to generate a short signature σ, on the *same* message m (where the size of the signature should

⋆ Work done at Mysten Labs.

be independent of the number of signers). To verify the multi-signature one needs all the signers' public keys, m and σ. A useful property of many multi-signature schemes, is that they additionally support *public-key aggregation*; thus the verifier only needs a short aggregated public key instead of an explicit list of all n public keys[6].

Multi-signatures with public key aggregation play a fundamental role in the scalability of blockchain systems since they allow the compression of posted signatures and verification keys up to a factor of n. Of particular interest are multi-signature schemes the verification algorithm of which, is fully compatible with algorithms supported by blockchain systems such as Schnorr [Sch90] or BLS [BLS01].

BLS signatures in the Blockchain Space. Boneh–Lynn–Shacham (BLS) proposed an efficient signature scheme [BLS01] that uses pairing friendly elliptic curves. BLS supports multi-signing with signature/public-key aggregation [BDN18b] in a non-interactive, deterministic and non-malleable manner. More specifically, the scheme proposed in [BDN18b] has signature sizes of size $O(\lambda)$, where λ is the security parameter and its security is proven in the random oracle (RO) model under a generalization of the Computational Diffie-Hellman (CDH) assumption, called co-CDH [BLS01].

BLS signatures have recently seen an increased adoption in the blockchain space. Chia network adopted BLS[7] as its main signature scheme, primarily motivated by its non-interactiveness to generate threshold signatures. Instead of requiring multiple communication rounds and a dealer to ensure t-of-n participants had signed, a BLS aggregated signature can be incrementally aggregated. Celo developed a SNARK friendly BLS signature[8] on the BLS12-377 curve using a bespoke hash-to-curve function[9] to allow for efficient verification of multiple signatures. This benefited the Plumo ultralight client [VGS$^+$22] to be more efficient where the signers do not need to know in advance about the public keys of the other signers.

Dfinity is designed based on a Random Beacon that acts as a verifiable random function[10] to produce verifiable and deterministic randomness with the threshold version of BLS signatures [Gro21]. Each participant signs a message independently, and the aggregated signature itself serves as the deterministic random number that no party controls. Filecoin [BG18] uses BLS as one of the four signature schemes admissible for the blockchain's actors. A standardization attempt for BLS with IETF is ongoing since 2019 [11].

[6] We note that *aggregate signatures* are a more general primitive, which as opposed to multi-signatures, allow the aggregation of n signatures of *different* messages in a short single signature.

[7] https://github.com/Chia-Network/bls-signatures

[8] https://github.com/celo-org/celo-bls-snark-rs

[9] https://github.com/celo-org/celo-proposals/blob/master/CIPs/cip-0022.md

[10] https://dfinity.org/pdf-viewer/library/dfinity-consensus.pdf

[11] https://www.ietf.org/id/draft-irtf-cfrg-bls-signature-05.html

Although ECDSA signatures can be verified much faster individually, BLS signatures on the same message can be verified much faster in the aggregated form, therefore making it more practical for multiple validators attesting the same block or transaction. In particular, n aggregated signatures on the same message can be performed with just 2 pairings instead of $n+1$ [BDN18b]. Notably, the most recent, mainnet deployed, Ethereum Consensus client adopted BLS signatures for validators attesting block proposals [Eth,Edg] addressing the verification bottleneck [Dra18].

The BLS multi-signature [BDN18b]. When designing multi-signature schemes, a great challenge is to avoid *rogue-key* attacks: a forgery attack caused in schemes where the adversaries are allowed to choose their public keys arbitrarily. In a typical multi-signature rogue key attack, the adversary would attempt to create a public key that is a function of an honest user's key allowing possible forgeries. An easy defense against such attacks is to require the parties to present a proof of possession, i.e. a zero-knowledge proof of knowledge of their secret keys. However this complicates implementations and is not compatible with existing infrastructures.

The BLS multi-signature [BDN18b] avoids the need for proofs of possession by following the paradigm of [BN06] which allows the public keys to be aggregated without the need to check their validity through a series of signature and public key randomizations. At a high level the BLS multi-signature of [BDN18b] works as follows: let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear pairing in groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order q. Let g_1, g_2 be generators of $\mathbb{G}_1, \mathbb{G}_2$ respectively and let $\mathsf{H}_0 : \{0,1\}^* \to \mathbb{G}_1$ and $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{Z}_q$. A secret/public key pair is denoted by $(\mathsf{pk}, \mathsf{sk})$ where $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q^*$ and $\mathsf{pk} = g_2^{\mathsf{sk}} \in \mathbb{G}_2$. Let PK be the set of public keys of n signers. To compute a multi-signature for these signers each party computes $\sigma_i = \mathsf{H}_0(m)^{a_i \cdot \mathsf{sk}_i}$ where $a_i = \mathsf{H}_1(\mathsf{pk}_i \parallel \mathsf{PK})$ and a designated combiner computes the final aggregated signature to be $\sigma = \prod_{i=1}^{n} \sigma_i$. The aggregated public key across the n signers is $apk = \prod_{i=1}^{n} \mathsf{pk}_i^{\mathsf{H}_1(\mathsf{pk}_i \parallel \mathsf{PK})}$. The signature can be verified by checking $e(\sigma, g_2) = e(\mathsf{H}_0(m), apk)$ (which is identical to the single BLS verification process).

Our Results. Our results are inspired by the observation that the BLS multi-signature of [BDN18b] requires a total of n exponentiations in $\mathbb{G}_2$ during the aggregation of signature shares and n exponentiations in $\mathbb{G}_1$ during the computation of apk for signature verification in order to randomize the signature and keys. While this cost is needed if the set of the n signers is dynamic and constantly updated, it is not necessary in all settings. For example, in proof-of-stake settings it is common to have a static set of n signers per epoch during which multiple subsets of the n signers would be required to engage in multi-signing.

Our protocol takes advantage of that setting and moves the need of signature and apk randomization to a *one-time* public key randomization process which happens once the set of n signers is fixed. Thus, instead of randomizing every single multi-signature and the corresponding aggregated public key, we randomize each signer's public key *once* at the beginning of the protocol (or at the

beginning of each epoch for the consensus setting), and then for every signing subset I we simply aggregate signatures and the randomized public keys together via a cheap multiplication and without the need for any exponentiations. The advantage is that now the cost of a multi-signature is the same as a regular BLS signature. We first provide a new set of definitions that allow for *subset* multi-signing in Section 2.5, and then present our construction in Section 3.

Security. The security analysis of our scheme is technically interesting as it has to significantly depart from the analysis of the standard BLS multi-signature [BDN18b]. In particular, the rewinding approach of [BDN18b] would fail in our case unless the adversary was forced to declare the signer subset for which it would output its forgery (to explain the technicality of our proof we start with a security proof for this weaker adversary in Section 3.1). To overcome this limitation, we leverage the algebraic group model (AGM) [FKL18] and we prove our scheme secure under the discrete logarithm assumption in the combined AGM + RO models. When n is very large our proof additionally requires the Random Modular Subset Sum (RMSS) assumption. We show that this requirement is intrinsic, as there is an attack on our scheme if RMSS is easy when n is large. Our AGM proof can be found in Section 3.2.

Implementation. Finally, in Section 4, we provide an implementation of our construction and a baseline comparison with [BDN18b]. Notably, for $n = 500$ signers our signature aggregation saves between 150 ms - 180 ms; and our signature verification saves between 40ms - 75 ms (depending on whether signatures are in $\mathbb{G}_2$ and keys in $\mathbb{G}_1$ or vice versa). Our performance benefit increases linearly with the number of signers.

Related Work. There exists a long line of multi-signature proposals relying in different assumptions such as RSA [Ita83,OO93], discrete logarithm and Schnorr signatures [NKDM03,BN06,BCJ08,NRSW20,NRS21], bilinear pairings [Bol02,LOS+06,BGOY07,BDN18b] and more recently lattice based assumptions [ES16,FH20].

2 Preliminaries

Notations. We denote the security parameter by λ and $\xleftarrow{\$}$ denotes sampling uniformly at random.

2.1 Bilinear pairing

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ denote groups of prime order q and let g_1, g_2 be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. Also, let $\mathbb{Z}_q$ be the field of order q.

A bilinear pairing is an *efficiently* computable map, $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, satisfying the following properties:

- **Bilinearity:** For all $P \in \mathbb{G}_1, Q \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_q$ we have

$$e(P^a, Q^b) = e(P^a, Q)^b = e(P, Q^b)^a = e(P, Q)^{ab}.$$

– **Non-degeneracy:** $e(g_1, g_2) \neq 1$.

Bilinear pairings can be of a few types depending on whether there is an efficient isomorphism from $\mathbb{G}_1$ to $\mathbb{G}_2$ in both directions (Type 1), only one direction (Type 2), or in neither direction (Type 3). In general, BLS and derived protocols work for all three types - so we ignore the distinctions in the following sections. Type 3 pairings are more efficient and commonly deployed.

2.2 Computational Assumptions

For our security proofs of BLS subset multi-signatures, we recall the assumptions of Discrete Log (DL), a generalization of the Computational Diffie-Hellman (CDH) assumption, called co-CDH [BLS01], and the Random Modular Subset Sum (RMSS) assumption [IN96,Lyu05].

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

$$Pr\left[z' = z \ : \ z \xleftarrow{\$} \mathbb{Z}_q,\ Z \leftarrow g^z,\ z' \leftarrow \mathcal{A}(g, Z)\right]$$

where the probability is taken over the random choices of the adversary $\mathcal{A}$ and the random selection of z. DL is (τ, ϵ)-hard if there is no adversary $\mathcal{A}$ that can break the DL problem in time τ and with advantage $\mathsf{Adv}^{DL}_{\mathbb{G}}(\mathcal{A}) > \epsilon$.

Following [FKL18], we show that our construction is secure in AGM assuming the Discrete Log problem.

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

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

where the probability is taken over the random choices of the adversary $\mathcal{A}$ and the random selection of (α, β). co-CDH is (τ, ϵ)-hard if there is no adversary $\mathcal{A}$ that can break the co-CDH problem in time τ and with advantage $\mathsf{Adv}^{co\text{-}DH}_{\mathbb{G}_1,\mathbb{G}_2}(\mathcal{A}) > \epsilon$.

For symmetric pairing groups, co-CDH reduces to standard CDH.

Definition 3 (RMSS Problem). *For a prime number q, we define the advantage $\mathsf{Adv}^{RMSS}_{n,q}(\mathcal{A})$ of an adversary $\mathcal{A}$ as:*

$$Pr\left[\sum_{i \in I} s_i = t \ : \ S = \{s_i\}_{i=1}^n \xleftarrow{\$} \mathbb{Z}_q^n,\ t \leftarrow \mathbb{Z}_q,\ S \supseteq I \leftarrow \mathcal{A}(S, t)\right]$$

where the probability is taken over the random choices of the adversary $\mathcal{A}$ and the random selection of (S, t). RMSS is (τ, ϵ)-hard if there is no adversary that can break the RMSS problem in time τ and with advantage $\mathsf{Adv}^{RMSS}_{n,q}(\mathcal{A}) > \epsilon$.

Impagliazzo and Naor [IN96] argued that the hardest instances of RMSS are characterized by $n = c\log(q)$, where c is a constant factor. Although RMSS is poly-time solvable [LO85,Fri86] through a reduction to lattice SVP problems for $n = O(\sqrt{\log(q)})$, this is of lesser consequence to us, as the probability of the existence of a solution is low.

2.3 Forking Lemma

Pointcheval and Stern [PS00] first formalized the Forking Lemma which is used for bounding the success probability of reductions employing rewinding and re-programming random oracles. In our proofs, we will adopt the General Forking Lemma, as formalized by Bellare and Neven [BN06].

Lemma 1 (General Forking Lemma). *Fix an integer $q \geq 1$ and a set H of size $h \geq 2$. Let $\mathcal{A}$ be a randomized algorithm that on input $x, h_1, ..., h_q$ returns a pair (idx, σ), the first element of which is an integer in the range $0, ..., q$ and the second element of which we refer to as a side output. Let IG be a randomized algorithm that we call the input parameter. The accepting probability of $\mathcal{A}$, denoted acc, is defined as the probability that $idx \geq 1$ in the experiment*

$$x \xleftarrow{\$} IG;\ h_1, ..., h_q \xleftarrow{\$} H;\ (idx, \sigma) \xleftarrow{\$} \mathcal{A}(x, h_1, ..., h_q)$$

The forking algorithm $F_{\mathcal{A}}$ associated to A is the randomized algorithm that takes input x and proceeds as follows: Let

Algorithm $F_{\mathcal{A}}(x)$

- Pick coins ρ for $\mathcal{A}$ at random
- $h_1, ..., h_q \xleftarrow{\$} H$
- $(idx, \sigma) \xleftarrow{\$} \mathcal{A}(x, h_1, ..., h_q; \rho)$
- If $(idx = 0)$ then return $(0, \epsilon, \epsilon)$
- $h'_I, ..., h'_q \xleftarrow{\$} H$
- $(idx', \sigma') \xleftarrow{\$} \mathcal{A}(x, h_1, ..., h_{idx-1}, h'_{idx}, ..., h'_q; \rho)$
- If $(idx = idx'$ and $h_{idx} \neq h'_{idx})$ then return $(1, \sigma, \sigma')$
- Else return $(0, \epsilon, \epsilon)$

$$frk = Pr[b = 1 : x \xleftarrow{\$} IG; (b, \sigma, \sigma') \xleftarrow{\$} F_{\mathcal{A}}(x)]$$

Then

$$frk \geq acc \cdot \left(\frac{acc}{q} - \frac{1}{h}\right)$$

Alternatively,

$$acc \leq \frac{q}{h} + \sqrt{q \cdot frk}$$

When we apply the forking lemma in our security proofs, we will assume an adversary breaking the unforgeability property of our signature and build from it an algorithm $\mathcal{A}$ that works under the assumptions of the forking lemma. The intuition is that $h_1, ..., h_q$ can be seen as the set of replies to the random oracle queries made by the original adversary. The forking adversary implements the rewinding and the two executions of $\mathcal{A}$ performed by $F_{\mathcal{A}}$ use the same random coins ρ.

2.4 Algebraic Group Model

The algebraic group model (AGM) introduced in [FKL18], is a model for security proofs that lies between the generic group model (GGM) and the standard model. In AGM the adversary is considered *algebraic*: whenever it outputs a group element, it also outputs a representation of that group element relative to all of the other input group elements the algorithm has received up to that point.

Definition 4 (Algebraic Algorithm [FKL18]). *An algorithm $\mathcal{A}$ is called algebraic (over a group $\mathbb{G}$) if for all group elements $\zeta \in \mathbb{G}$ that $\mathcal{A}$ outputs, it additionally outputs a vector $z = (z_0, \ldots, z_m)$ of integers such that $\zeta = \prod_i g_i^{z_i}$ where $(g_0, \ldots, g_m)$ is the list of group elements $\mathcal{A}$ has received so far (w.l.o.g. $g_0 = g$).*

The AGM model was used before to tighten the security reduction of the standard BLS signature scheme [FKL18]. While previous reductions non-tightly reduced from the CDH problem with a tightness loss linear in the number of signing queries, [FKL18] provided a tight reduction in the AGM+RO model under the hardness of discrete log.

2.5 Definitions of (Subset-optimized) Multi-signatures

Informally, a multi-signature scheme (MS) allows multiple signers with public keys $\mathsf{PK} = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ to sign the same message with a signature size independent to the number of signers. The set of signers' public keys is aggregated into a single key *apk*.

We recall the definitions for multi-signatures by roughly following Bellare and Neven [BN06] and Drijvers et al. [DEF+19]. A multi-signature scheme with key aggregation consists of the following algorithms:

- **MS.Setup**(1^λ): On input the security parameter, it outputs the scheme's parameters par.
- **MS.KeyGen**(par): Given the parameters par, outputs a pair of public/secret keys $(\mathsf{pk}, \mathsf{sk})$.
- **for all** $i \in \{1, \ldots, n\}$**: MS.Sign**$(par, \mathsf{PK}, \mathsf{sk}_i, m)$: On input the set of public keys PK, a signing secret key sk_i and a message m, the signer outputs the signature σ_i. A designated combiner outputs the combined signature σ. (Instead of a designates combiner, this algorithm could be interactive.)

- $\textsf{MS.KeyAggr}(par, \textsf{PK})$: Output a single aggregated key apk for all the input pubic keys $\textsf{PK} = \{\textsf{pk}_1, \ldots, \textsf{pk}_n\}$.
- $\textsf{MS.Verify}(par, apk, \sigma, m)$: Output 1 if the signature σ verifies for message m under apk and 0 otherwise.

Subset multi-signatures (SMS). Compared to the above definition, the main difference in our scheme is that we assume a fixed set of signers with public keys $\textsf{PK} = \{\textsf{pk}_1, \ldots, \textsf{pk}_n\}$ and we allow different subsets I of them to compute signatures. However, during signing, the signer does not have to be aware of who are the rest of the subset members as long as it knows PK. We additionally separate the signing process from the signature aggregation. The updated definition is as follows:

- $\textsf{SMS.Setup}(1^\lambda)$: Outputs the scheme's parameters par.
- $\textsf{SMS.KeyGen}(par)$: Given the parameters par, output a pair of public/secret keys $(\textsf{pk}, \textsf{sk})$.
- $\textsf{SMS.Sign}(par, \textsf{PK}, \textsf{sk}_i, m)$: On input the set of public keys PK, a signing secret key $\textsf{sk}_i$ and a message m, the signer outputs the signature share σ_i.
- $\textsf{SMS.SigAggr}(par, \{\sigma_i : i \in I\}, m)$: On input $|I|$ signatures on message m, it outputs an aggregated signature σ_I for the subset of users I. (It could potentially also take PK as input.)
- $\textsf{SMS.SubsetKeyAggr}(par, \{\textsf{pk}_i : i \in I\})$ Output a subset aggregated key apk_I for all the input public keys of the subset I.
- $\textsf{SMS.Verify}(par, apk_I, \sigma_I, m)$: Output 1 if the signature σ_I verifies for message m under apk_I and 0 otherwise.

Security. A subset multi-signature scheme should satisfy the properties of completeness and unforgeability.

Definition 5 (Completeness). *A subset multi-signature scheme (SMS) satisfies completeness, if for every $n > 1$, $m \in \{0,1\}^*$, and subset $I \subseteq \{1, \ldots, n\}$ of signers, we compute*

- $(sk_i, pk_i) \leftarrow$ ***SMS.KeyGen***(par)*, for all* $i \in [1, n]$
- $\sigma_i \leftarrow$ ***SMS.Sign***$(par, PK, \textsf{sk}_i, m)$*, for all* $i \in I$
- $\sigma_I \leftarrow$ ***SMS.SigAggr***$(par, \{\sigma_i : i \in I\}, m)$
- $apk_I \leftarrow$ ***SMS.SubsetKeyAggr***$(par, \{\textsf{pk}_i : i \in I\})$.

Then we have ***SMS.Verify***$(par, apk_I, \sigma_I, m) = 1$ *with overwhelming probability.*

A multi-signature is unforgeable if an adversarial user cannot forge a signature that verifies under apk_I for a set of signers where at least one signer is honest. In other words, assuming n signers, even if an adversary has corrupted all but one signer with public key $\textsf{pk}_0$, the user should still not be able to forge a signature that verifies under apk_I that includes $\textsf{pk}_0$. We recall the formal definition given in [BDN18b] as a three stage game (slightly adapted to capture our subset scenario):

- **Setup:** The challenger generates the parameters par and the key pair $(\mathsf{sk}_0, \mathsf{pk}_0)$ of the honest signer. It runs the adversary $\mathcal{A}(par, \mathsf{pk}_0)$. The adversary sets up a fixed set of public keys $\mathsf{PK}_A = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ and sends PK_A to the challenger. We denote $\mathsf{PK} = \mathsf{PK}_A \cup \{pk_0\}$.
- **Queries:** The adversary is allowed to perform a series of signing queries, where $\mathcal{A}$ picks m and queries a signing oracle $\mathcal{O}_{sign}^{(par,\mathsf{PK},\mathsf{sk}_0,\cdot)}$ which will simulate the honest user and return its signature share for message m. The oracle queries can be repeated for different inputs m. $\mathcal{A}$ can make any number of the above defined queries concurrently.
- **Output:** $\mathcal{A}$ outputs a multi-signature forgery (σ^*, m^*, I^*) and wins if no signature queries were made on m^* and $\textsf{\textbf{SMS.Verify}}(par, apk^*, \sigma^*, m^*) = 1$ for the aggregated subset public key $apk^* = \textsf{\textbf{SMS.SubsetKeyAggr}}(par, \{\mathsf{pk}_i : i \in I^* \cup \{0\}\})$.

Definition 6 (Unforgeability). *We say that $\mathcal{A}$ is a $(\tau, q_S, q_H, \epsilon)$ forger if it wins the above game with probability at least ϵ after running for time τ, and making q_S and q_H signing queries and random oracle queries respectively. An **SMS** scheme is $(\tau, q_S, q_H, \epsilon)$-unforgeable if no $(\tau, q_S, q_H, \epsilon)$ adversary exists.*

2.6 BLS Multi-Signatures

We start by recalling the standard BLS Signature scheme. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear pairing as defined above. Let g_1, g_2 be generators of $\mathbb{G}_1, \mathbb{G}_2$ respectively and let $\mathsf{H}_0 : \{0,1\}^* \to \mathbb{G}_1$. BLS can be instantiated either as *minSig* where signatures are in $\mathbb{G}_1$ and public keys in $\mathbb{G}_2$, or as *minPK* where signatures are in $\mathbb{G}_2$ and keys are in $\mathbb{G}_1$. Below we take the *minSig* approach. The BLS signature scheme consists of the following algorithms:

- $\textsf{\textbf{BLS.Setup}}(1^\lambda)$: Setup and output a bilinear group $par = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$.
- $\textsf{\textbf{BLS.KeyGen}}(par)$: Given the parameters par, output a pair of public/secret keys $(\mathsf{pk}, \mathsf{sk})$ where $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q^*$ and $\mathsf{pk} = g_2^{\mathsf{sk}} \in \mathbb{G}_2$.
- $\textsf{\textbf{BLS.Sign}}(par, \mathsf{sk}, m)$: Given a message $m \in \{0,1\}^*$, output a signature $\sigma = \mathsf{H}_0(m)^{\mathsf{sk}} \in \mathbb{G}_1$.
- $\textsf{\textbf{BLS.Verify}}(par, \mathsf{pk}, \sigma, m)$: Given a public key $\mathsf{pk} \in \mathbb{G}_2$, a signature $\sigma \in \mathbb{G}_1$ and a message $m \in \{0,1\}^*$, output 1 if $e(\sigma, g_2) = e(\mathsf{H}_0(m), \mathsf{pk})$ and 0 otherwise.

BLS signatures can support multi-signing with public-key aggregation. Bellow, we recall the MS scheme as proposed in [BDN18b]. We note that there exist two descriptions of the scheme: one in the full and proceedings version of the paper [BDN18b], and a slightly modified version of the scheme described by the authors in a blog-post [BDN18a]. We first recall the scheme from the full/proceedings version [BDN18b] and then discuss the differences with the blog-post version [BDN18a].

We assume the same setup as in BLS signatures and an additional hash function $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{Z}_q$.

- $\mathsf{MS.Setup}(1^\lambda)$: Output $\mathsf{BLS.Setup}(1^\lambda)$.
- $\mathsf{MS.KeyGen}(par)$: Output $\mathsf{BLS.KeyGen}(par)$.
- $\mathsf{MS.Sign}(par, \mathsf{PK}, \mathsf{sk}_i, m)$: On input the set of public keys PK, a signing secret key sk_i and a message m, compute $\sigma_i = \mathsf{H}_0(m)^{a_i \cdot \mathsf{sk}_i}$ where $a_i = \mathsf{H}_1(\mathsf{pk}_i \parallel \mathsf{PK})$. Send the signature to a designated combiner who computes the final signature to be $\sigma = \prod_{i=1}^{n} \sigma_i$. (The designated combiner can be one of the signers or an external party.)
- $\mathsf{MS.KeyAggr}(par, \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\})$: let $\mathsf{PK} = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$, then output

$$apk = \prod_{i=1}^{n} \mathsf{pk}_i^{\mathsf{H}_1(\mathsf{pk}_i \parallel \mathsf{PK})}$$

- $\mathsf{MS.Verify}(par, apk, \sigma, m)$: Output $\mathsf{BLS.Verify}(par, apk, \sigma, m)$.

The main difference between the scheme above and its blog-post version [BDN18a], is that the later scheme makes the signature aggregation process distinct while at the same time includes all the randomizations. Users compute their signatures as regular, individual BLS signatures on message m, completely oblivious of who else is signing the message. Then, an aggregator, given the set of public keys for the signers PK, and all individual signatures σ_i, computes the aggregated multi-sig. As the scheme is described in the blog-post, the aggregator has to pay the cost of n exponentiations in $\mathbb{G}_1$, instead of amortizing this cost across signers.

3 Scheme Description

Our scheme *SMSKR* (Subset Multi-Signature with Key Randomization) is a variant of the original Boneh at al. pairing-based BLS multi-signature scheme [BDN18b]. Our algorithms are defined in the same way as for subset multi-signatures (definition given in Section 2.5) but in order to showcase our optimizations for the specific case of BLS, we divide $\mathsf{MS.Sign}(par, \mathsf{PK}, \mathsf{sk}_i, m)$ into two modules:

- $\mathsf{SMSKR.KeyRand}(par, \mathsf{sk}_i, \mathsf{PK})$: this algorithm creates randomized secret and public keys $(\mathsf{sk}_i^*, \mathsf{pk}_i^*)$ for user i and for the set of signers captured in PK.
- $\mathsf{SMSKR.Sign}(par, \mathsf{sk}_i^*, m)$: this is the sign algorithm but using the already randomized secret key (and thus there is no need to include PK again.)

Let $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ be bilinear groups of prime order q as defined in Section 2.1. Let g_1 and g_2 be generators for $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively and let $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ be a bilinear pairing. Let $\mathsf{H}_0 : \{0,1\}^* \rightarrow \mathbb{G}_1$ and $\mathsf{H}_1 : \{0,1\}^* \rightarrow \mathbb{Z}_q^*$ be random-output hash functions and let n denote the number of parties. In our construction and definitions we will also assume *minSig*, as in Section 2.6, although *MSKR* can work with both modes. We assume access to the functions from a **BLS** signature scheme (as described in Section 2.6) over the same groups using H_0 as the hash function and define our **SMSKR** construction as follows:

- **SMSKR.Setup**(1^λ): Output **BLS.Setup**(1^λ).
- **SMSKR.KeyGen**(par): Output **BLS.KeyGen**(par).
- **SMSKR.KeyRand**$(par, \mathsf{sk}_i, \mathsf{PK})$: Given a set of public keys $\mathsf{PK} = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$, output the randomized public and secret keys $\mathsf{pk}_i^* = \mathsf{pk}_i^{\mathsf{H}_1(\mathsf{pk}_i || \mathsf{PK})}$ and $\mathsf{sk}_i^* = \mathsf{sk}_i \cdot \mathsf{H}_1(\mathsf{pk}_i \parallel \mathsf{PK}) \in \mathbb{Z}_q^*$ respectively[12].
- **SMSKR.Sign**$(par, \mathsf{sk}_i^*, m)$: Output the signature share $\sigma_i =$ **BLS.Sign**$(par, \mathsf{sk}_i^*, m)$.
- **SMSKR.SigAggr**$(par, I, \{\sigma_j : j \in I\})$: Given a subset of parties $I \subseteq \{1, \ldots, n\}$ and a set of signatures from the corresponding parties, output the aggregated multi-signature $\sigma_I = \prod_{j \in I} \sigma_j$.
- **SMSKR.SubsetKeyAggr**$(par, I, \{\mathsf{pk}_j^* : j \in I\})$: Given a subset of the parties denoted by their indices $I \subseteq \{1, \ldots, n\}$ and their randomized public keys, output $apk_I^* = \prod_{j \in I} \mathsf{pk}_j^*$.
- **SMSKR.Verify**$(par, apk_I^*, \sigma_I, m)$: Given an aggregated public key $apk_I^* \in \mathbb{G}_2$, the set of public keys under which pk_i^*'s where computed, an aggregated signature $\sigma_I \in \mathbb{G}_1$ and a message $m \in \{0,1\}^*$, output **BLS.Verify**$(par, apk_I^*, \sigma_I, m)$.

The separation of key randomization and signing, is the key point of our construction that allows for efficient implementations in the blockchain settings. Assuming a known set of eligible signers $\mathsf{PK} = \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$ at the beginning of a blockchain epoch, all entities can appropriately randomize their keys *once at the beginning of the epoch* and then participate in multiple BLS multi-signatures for any subset of $[n]$. The advantage is that now the cost of a multi-signature is the same as a regular BLS signature (plus the cost of a one-time key randomization), while the users do need to know who participates in the multi-signing amongst the eligible signers. This is different than the original BLS multi-sig constructions [BDN18b] where the secret keys of each user were repeatedly randomized during each signing session for the specific set of signers that participated in each multi-signature.

Discussion. We note that in blockchain settings, it is very reasonable to assume that applications already have access to full set of randomized public keys PK^* and they only require a bitmap that defines the indexes of the subset of entities that signed the message in **SMSKR.SubsetKeyAggr**. To optimize even further, applications could cache common subsets of I and their corresponding aggregated keys.

We also note that in certain blockchain applications, a party i might not have direct access to the private key sk_i, but only to a BLS signing oracle over the original private key. Thus, it could request a signature over m, receive $s_i = \mathsf{H}_0(m)^{\mathsf{sk}_i}$ and then randomize the signature by computing $\sigma_i^* = s_i^{\mathsf{H}_1(\mathsf{pk}_i, \mathsf{PK})}$. This approach is more expensive because signature randomization requires an operation in $\mathbb{G}_1$, while SMSKR's default approach randomizes the private key, which is a fast operation in $\mathbb{Z}_q^*$.

[12] We note that the randomization of the public key can happen by any third party since no secret is required.

3.1 Security Analysis (for a weaker Adversary)

We start by noting that our suggested change is not a simple modification of [BDN18b] as it requires a drastically different security proof. In [BDN18b] it is critical that the key randomization process happens at the same time as key aggregation as this allows the security reduction to handle all these hash queries as one which in turn allows to fix a specific set of public keys for the adversary's forgery even after rewinding.

To showcase the complication of the security analysis for our scheme, we first discuss the security of our scheme for a weaker unforgeability adversary which in the security game of Definition 6, defines a target subset I for which it will output its forgery before starting its queries. For this weaker case, our security analysis follows [BDN18b].

Theorem 1. *SMSKR is an unforgeable multi-signature scheme (for the weaker variation of the Definition 6 explained above) under the computational co-Diffie-Hellman problem in the random-oracle model.*

Proof. Following the proof of [BDN18b], let $\mathcal{F}$ be a $(\tau, q_S, q_H, \epsilon)$ forger that breaks the unforgeability of SMSKR as defined in Definition 6.

Let *par* denote the bilinear group parameters and assume it is given as input everywhere below. Let IG be an algorithm that generates instances for the co-CDH problem, i.e. it outputs $(A, B_1, B_2) = (g_1^\alpha, g_1^\beta, g_2^\beta)$ for $\alpha, \beta \xleftarrow{\$} \mathbb{Z}_q$.

We build an algorithm $\mathcal{A}$ that on input (A, B_1, B_2) proceeds as follows. $\mathcal{A}$ picks an index $k \xleftarrow{\$} \{1, \ldots, q_H\}$ and runs the forger $\mathcal{F}$ on input the honest public key $\mathsf{pk}_0 \leftarrow B_2$ with random tape ρ. $\mathcal{F}$ defines PK and its target forgery subset I.

H_0 **queries.** When $\mathcal{F}$ makes an H_0 query, $\mathcal{A}$ picks $r_i \xleftarrow{\$} \mathbb{Z}_q$ and returns $g_1^{r_i}$ for all i except the k'th query for which it returns A. We assume w.l.o.g. that $\mathcal{F}$ makes no repeated H_0 queries.

H_1 **queries.** When $\mathcal{F}$ queries on $(\mathsf{pk}_i, \mathsf{PK})$, $\mathcal{A}$ just returns a random value in $h_i \in \mathbb{Z}_q$.

Signing queries. The aggregated subset key apk^* is computed as in the actual protocol. When $\mathcal{F}$ makes a signing query on message m, $\mathcal{A}$ looks up $\mathsf{H}_0(m)$. If the lookup returns the value A, then $\mathcal{A}$ aborts. Else, the value must be of the form g_1^r, and $\mathcal{A}$ can simulate the honest signer by computing and returning $\sigma_i^* = B_1^r$. When $\mathcal{F}$ fails to output a successful forgery, then $\mathcal{A}$ aborts. If $\mathcal{F}$ successfully outputs a forgery for a message m for which $\mathsf{H}_0(m) \neq A$ then $\mathcal{A}$ also aborts. Otherwise, $\mathcal{F}$ outputs a forgery (σ^*, m^*, I) such that:

$$e(\sigma^*, g_2) = e(A, \textsf{\textbf{SMSKR.SubsetKeyAggr}}(par, \{\mathsf{pk}_j : j \in I \cup \{0\}\}))$$

Let j_f be the forgery index, i.e. the index for which $\mathcal{F}$ queried $\mathsf{H}_1(\mathsf{pk}_0, \mathsf{PK}) = h_{j_f}$. Let $a_j = \mathsf{H}_1(\mathsf{pk}_j, \mathsf{PK})$ for $\mathsf{PK} = \{\mathsf{pk}_0, \mathsf{pk}_1, \ldots, \mathsf{pk}_n\}$.
Then $\mathcal{A}$ outputs $(j_f, \{\sigma, \mathsf{PK}, I, apk_I, a_1, \ldots, a_n\})$.

To complete the proof, we construct 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 $(\mathbb{G}_1, \mathbb{G}_2)$. $\mathcal{B}$ will invoke the generalized forking algorithm GF_A (as defined in 1) on input (A, B_1, B_2) with the algorithm $\mathcal{A}$ running as described above. (Note that the co-CDH instance is distributed identically to the output of the IG). If GF_A aborts, then $\mathcal{B}$ outputs fail. If GF_A outputs (j_f, out^1, out^2), then $\mathcal{B}$ proceeds as follows: $\mathcal{B}$ parses the two outputs as: $out^1 = \{\sigma^1, \mathsf{PK}^1, I^1, apk_I^1, a_1^1, \ldots, a_n^1\}$ and $out^2 = \{\sigma^2, \mathsf{PK}^2, I^2, apk_I^2, a_1^2, \ldots, a_n^2\}$. By the forking lemma, we know that those two executions were identical up to the j_f'th H_1 query. In particular, this means that the arguments of the j_f'th query are identical, i.e., $\mathsf{PK}^1 = \mathsf{PK}^2$, $I^1 = I^2$, and $n^1 = n^2$. Let $h_{j_f^1} = a_i^1$ and $h_{j_f^2} = a_i^2$, then $a_i^1 \neq a_i^2$. Given that the two subsets are the same, we have $apk_I^1 / apk_I^2 = \mathsf{pk_0}^{a_i^1 - a_i^2}$. Thus, $(\sigma^1/\sigma^2)^{1/(a_i^1 - a_i^2)}$ is a solution to the co-CDH instance. The probability of success can be bound with the forking lemma.

As noted above, this proof would not go through if the adversary had not fixed the subset I for its forgery ahead of time (and before forking). If the adversary was allowed to output forgeries for different subsets before and after forking, the security proof would require exponential time.

3.2 Security Analysis in AGM

We now present a proof of our signature scheme in the Algebraic Group Model. Using AGM allows us to withstand the subset mismatch problem we had before. In particular, we have a reduction strategy even if the target subset is different after a rewind. Although we still have a 2^n security loss in the reduction, we show that this loss is intrinsic and can only be avoided by additionally assuming hardness of the RMSS problem.

We follow the security proof of BLS in [FKL18] up to a certain point - in particular they provide reduction strategies to address 2 cases that may arise. In addition to those 2 cases, we have an important 3rd case where our analysis highlights a fundamental distinguishing characteristic of the SMSKR construction. Specifically, there is a subset-sum attack possible in SMSKR which can either be negligibly low probability statistically, or be able to be argued to be computationally hard based on the RMSS assumption. We describe the proof in both scenarios and analyze how we can leverage both the DL and RMSS assumptions depending on concrete subset and universal set sizes.

As in [FKL18], we assume symmetric bilinear groups in the proof. We also note that the proof is extensible to asymmetric groups by assuming Discrete Log hardness of both groups. In the reduction the unforgeability challenger can receive DL challenges in both groups and choose which challenge to embed depending on the context.

Theorem 2. *SMSKR is an unforgeable multi-signature scheme (as defined in Definition 6) under the Discrete Logarithm problem in the random-oracle and AGM models, given $n = \log(o(q))$.*

Proof. Let $\mathcal{A}_{alg}$ be an algebraic adversary for the multi-signature unforgeability game defined in Definition 6. We build a Discrete Log adversary $\mathcal{A}_{DL}$ which uses $\mathcal{A}_{alg}$. At the beginning, $\mathcal{A}_{DL}$ receives a Discrete Log challenge $(g, Z = g^z)$, where z is the desired discrete logarithm output. The challenger $\mathcal{A}_{DL}$ samples pk_0 uniformly from $\mathbb{G}$ in a couple of different ways as we will outline below. Briefly, we will describe two algorithms $\mathcal{B}$ and $\mathcal{C}$ that $\mathcal{A}_{DL}$ will call with probability $1/2$ each. Algorithm $\mathcal{B}$ will embed the challenge Z in the target public key pk_0, while algorithm $\mathcal{C}$ will embed Z in the hash $\mathsf{H}_0(m_i)$ query responses. After sampling pk_0 either way, $\mathcal{A}_{DL}$ sends it to $\mathcal{A}_{alg}$ as the public key of the target party. Regardless of which algorithm is executed, define $x \in \mathbb{Z}_q$ implicitly such that $\mathsf{pk}_0 = g^x$.

H_0 and H_1 queries. The challenger $\mathcal{A}_{DL}$ also simulates the random oracles H_0 and H_1. Let $H_i = \mathsf{H}_0(m_i) = g^{r_i}$ denote the responses to the q_H hash H_0 queries. These are also sampled uniformly from $\mathbb{G}$ in different ways by algorithms $\mathcal{B}$ and $\mathcal{C}$: $\mathcal{B}$ samples r_i directly, while $\mathcal{C}$ embed Z in the responses. The random oracle H_1 is simulated by returning random elements from $\mathbb{Z}_q$.

At some point, $\mathcal{A}_{alg}$ returns a set of keys $\mathsf{PK}_A = \{\mathsf{pk}_1, ..., \mathsf{pk}_n\}$. Let $\mathsf{PK} = \mathsf{PK}_A \cup \{\mathsf{pk}_0\}$. As it's an algebraic adversary, it also returns representations $\{(u_i, v_i, \vec{w}_i)\}_i$ such that $\mathsf{pk}_i = g^{u_i}\mathsf{pk}_0^{v_i}\prod_{j=1}^{q_H} H_j^{w_{ij}}$ for all $i \in \{1, \ldots, n\}$. It's possible that some of the H_0 queries are sent after outputting PK_A - for those $\mathcal{A}_{alg}$ can set the w_{ij} exponents to 0.

The challenger $\mathcal{A}_{DL}$ also simulates signature queries in the following way. If the query is m_j, it first simulates computation of $\mathsf{H}_0(m_j)$ and then simulates and returns signature $\Sigma_j = \mathsf{H}_0(m_j)^x$. While x is explicitly known to algorithm $\mathcal{C}$, it can be implicitly simulated by algorithm $\mathcal{B}$, as we will describe below. Wlog, we also assume that $\mathcal{A}_{alg}$ queries H_0 with the target message m^*.

Finally, $\mathcal{A}_{alg}$ returns a forgery Σ^* on a message $m^* \notin Q$ and a set of indices $I \subseteq [1, n]$ together with a representation $\vec{a} = (\hat{a}, a', \bar{a}_1, ..., \bar{a}_{q_H}, \tilde{a}_1, ..., \tilde{a}_{q_S})$, consisting of $\mathbb{Z}_q$ elements, such that:

$$\Sigma^* = \mathsf{H}_0(m^*)^{x \cdot \mathsf{H}_1(\mathsf{pk}_0||\mathsf{PK}) + \sum_{i \in I} sk_i \cdot \mathsf{H}_1(\mathsf{pk}_i||\mathsf{PK})} = g^{\hat{a}}\mathsf{pk}_0^{a'}\prod_{i=1}^{q_H} H_i^{\bar{a}_i}\prod_{j=1}^{q_S}\Sigma_j^{\tilde{a}_j}$$

Here g is the generator of the group, pk_0 is the public key of the target party, $H_i = \mathsf{H}_0(m_i) = g^{r_i}$ are the responses to the q_H hash H_0 queries and Σ_j are the signatures $\mathsf{H}_0(m_j)^x = g^{xr_j}$ returned by the signing oracle. Let $\mathsf{H}_0(m^*) = g^{r^*}$. Let $h_i = \mathsf{H}_1(\mathsf{pk}_i||\mathsf{PK})$, for $i \in [0, n]$. Implicitly, $sk_i = u_i + \sum_{j=1}^{q_H} r_j w_{ij} + v_i x$, for all $i \in [1, n]$. This equation is equivalent to:

$$\left[xh_0 + \sum_{i \in I}\left(u_i + \sum_{j=1}^{q_H} r_j w_{ij} + v_i x\right)h_i\right]r^* = x\left(a' + \sum_{i=1}^{q_S}\tilde{a}_i r_i\right) + \left(\hat{a} + \sum_{i=1}^{q_H}\bar{a}_i r_i\right)$$

Therefore,

$$x = \frac{(\hat{a} + \sum_{i=1}^{q_H}\bar{a}_i r_i) - \sum_{i \in I}(u_i + \sum_{j=1}^{q_H} r_j w_{ij})h_i r^*}{(h_0 + \sum_{i \in I} v_i h_i)r^* - (a' + \sum_{i=1}^{q_S}\tilde{a}_i r_i)} \tag{1}$$

Define $H = h_0 + \sum_{i \in I} v_i h_i$. We define events E and F, which will let different strategies succeed for the reduction. Let E be the event that $H = 0$, and let F be the event that $H \cdot r^* - (a' + \sum_{i=1}^{q_S} \tilde{a}_i r_i) = 0$. The challenger $\mathcal{A}_{DL}$ randomly chooses one of two algorithms $\mathcal{B}$ or $\mathcal{C}$ with probability $1/2$ and executes the chosen one.

Algorithm $\mathcal{B}$: The algorithm $\mathcal{B}$ sets $\mathsf{pk}_0 = Z$, the Discrete Log challenge. It can simulate a signature on a message m_i by setting $\Sigma_i = Z^{r_i}$, such that $\mathsf{H}_0(m_i) = g^{r_i}$. If event F occurs, then $\mathcal{B}$ aborts. If event $\neg F$ occurs, then it can compute $z = x$, by Equation 1, as the denominator is not 0. Therefore, $Adv_{DL}(B) = Adv_{\mathsf{SMSKR}}(\mathcal{A}_{alg}) \cdot Pr[\neg F]$.

Algorithm $\mathcal{C}$: The algorithm $\mathcal{C}$ generates $\mathsf{pk}_0 = g^x$ by sampling x itself. It also generates the H_0 responses by sampling b_i and $\hat{r}_i$ and setting $\mathsf{H}_0(m_i) = g^{r_i} = Z^{b_i} g^{\hat{r}_i}$. If event $E \vee \neg F$ occurs, then $\mathcal{C}$ aborts. Otherwise, assume event $\neg E \wedge F$ occurs.

Given F, we get:

$$H \cdot (zb^* + \hat{r}^*) = H \cdot r^* = a' + \sum_{i=1}^{q_S} \tilde{a}_i r_i = a' + \sum_{i=1}^{q_S} \tilde{a}_i (\hat{r}_i + zb_i)$$

Hence,

$$z = \frac{(a' + \sum_{i=1}^{q_S} \tilde{a}_i \hat{r}_i) - H \cdot \hat{r}^*}{H \cdot b^* - \sum_{i=1}^{q_S} \tilde{a}_i b_i}$$

Note that the value of b^* is information-theoretically hidden from $\mathcal{A}_{alg}$ and is also absent from the sum $\sum_{i=1}^{q_S} \tilde{a}_i b_i$, as the forgery message mustn't have been queried to the signing oracle. Although the b_i's are also hidden to $\mathcal{A}_{alg}$, note that it could set all the $\tilde{a}_i$'s to 0 and hence force the sum $\sum_{i=1}^{q_S} \tilde{a}_i b_i$ to be 0. Given $\neg E$, H is non-zero, thus the denominator is whp $\neq 0$. Therefore, $Adv_{DL}(C) = Adv_{\mathsf{SMSKR}}(\mathcal{A}_{alg}) \cdot Pr[\neg E \wedge F]$.

Event E: We show that the probability of this event is negligible given Discrete Log hardness. Let $\mathcal{A}_E$ be an adversary which wins if it makes event E happen. We construct a challenger AE_{DL} which rewinds $\mathcal{A}_E$ and applies General Forking Lemma (Lemma 1) to break Discrete Log hardness.

We now describe the algorithm $\mathcal{A}_E$. It runs like algorithm $\mathcal{B}$ as described above in simulating the target public key, H_0, H_1, and signature queries, to the adversary $\mathcal{A}_{alg}$. Let $\mathcal{A}_{alg}$ return a set of public keys and representations $\mathsf{PK}_A, \{(u_i, v_i, \vec{w}_i)\}_{i=1}^n$ and produce a forgery $(\Sigma^*, m^*, I, \vec{a})$ as described before. If event E didn't happen, then $\mathcal{A}_E$ returns $(0, \epsilon)$. Otherwise, let u be the index of the first query of the form $(\mathsf{pk}_{i_u} || \mathsf{PK})$ to H_1, where $\mathsf{PK} = \mathsf{PK}_A \cup \{\mathsf{pk}_0\}$ and $\mathsf{pk}_{i_u} \in \mathsf{PK}$. Let $h_i = \mathsf{H}_1(\mathsf{pk}_i || \mathsf{PK})$ for $i \in [0, n]$, and r_i be such that $\mathsf{H}_0(m_i) = g^{r_i}$ for $i \in [1, q_H]$. Then $\mathcal{A}_E$ returns $(u, (\mathsf{PK}, \{(u_i, v_i, \vec{w}_i)\}_{i=1}^n, \{r_i\}_{i=1}^{q_H}, \{h_i\}_{i=0}^n, I))$. Based on this the General Forking Lemma algorithm $GF_{\mathcal{A}_E}$ returns:

$$1, (\mathsf{PK}, \{(u_i, v_i, \vec{w}_i)\}_{i=1}^n, \{r_i\}_{i=1}^{q_H}, \{h_i\}_{i=0}^n, I),$$

$$(\mathsf{PK}', \{(u'_i, v'_i, \vec{w}'_i)\}_{i=1}^n, \{r'_i\}_{i=1}^{q_H}, \{h'_i\}_{i=0}^n, I')$$

Since the u-th query is identical for the two executions, we must have $\mathsf{PK} = \mathsf{PK}'$. Also, by construction the sets $\{h_i\}_{i=0}^n$ and $\{h'_i\}_{i=0}^n$ in the two executions are independently random.

We first show that the probability that the vectors $(v_1, ..., v_n)$ and $(v'_1, ..., v'_n)$ are equal is negligible if $n = \log o(q)$.

Lemma 2. *For a given vector $\vec{v} = (v_1, ..., v_n) \in \mathbb{Z}_q^n$, the probability that for some $I \subseteq [1, n]$, $h_0 + \sum_{i \in I} h_i v_i = 0$ with $h_0, h_1, ..., h_n \leftarrow \mathbb{Z}_q$, is $< 2^n/q$.*

Proof. Let E_H denote the event that $\exists I \subseteq [1, n] : h_0 + \sum_{i \in I} h_i v_i = 0$. For any fixed I, the probability of E_H is $1/q$. Therefore, if we union bound the probabilities over all possible $I \subseteq [1, n]$, then the probability of E_H is at most $2^n/q$.

Since $(h'_0, ..., h'_1)$ are chosen independent of $(v_1, ..., v_n)$, we must have whp $(v_1, ..., v_n) \neq (v'_1, ..., v'_n)$, by the above lemma. In that case there is an index k, such that $v'_k \neq v_k$. The Discrete Log challenger AE_{DL} then calculates the discrete log of pk_0 as $(u_k + \sum_{j=1}^{q_H} r_j w_{kj} - u'_k - \sum_{j=1}^{q_H} r'_j w'_{kj})/(v'_k - v_k)$.

Using Generalized Forking Lemma, we get:

$$Adv(\mathcal{A}_E) \leq q_H/q + \sqrt{q_H \cdot Adv(AE_{DL})/(1 - 2^n/q)}$$

Since the sample space of $\mathcal{A}_E$ matches that of $\mathcal{A}_{DL}$, therefore $Pr[E] \leq q_H/q + \sqrt{q_H \cdot Adv(AE_{DL})/(1 - 2^n/q)}$

Putting everything together, we get that

$$Adv_{DL}(\mathcal{A}_{DL}) = 1/2(Adv_{DL}(\mathcal{B}) + Adv_{DL}(\mathcal{C}))$$

$$= 1/2Adv_{\mathsf{SMSKR}}(\mathcal{A}_{alg})(Pr[\neg F] + Pr[\neg E \wedge F]) = 1/2Adv_{\mathsf{SMSKR}}(\mathcal{A}_{alg})(1 - Pr[E \wedge F])$$
$$\geq 1/2Adv_{\mathsf{SMSKR}}(\mathcal{A}_{alg})(1 - Pr[E])$$

Therefore,

$$Adv_{\mathsf{SMSKR}}(\mathcal{A}_{alg}) \leq 2Adv_{DL}(\mathcal{A}_{DL})(1 - Pr[E])^{-1} \leq 2Adv_{DL}(\mathcal{A}_{DL})(1 + 2Pr[E])$$
$$\leq 2Adv_{DL}(\mathcal{A}_{DL})(1 + 2q_H/q + 2\sqrt{q_H \cdot Adv(AE_{DL})/(1 - 2^n/q)})$$

Is the 2^n security loss intrinsic? We argue that the 2^n security loss incurred in the above reduction is intrinsic, unless we resort to a hardness assumption related to random modular subset sums (RMSS), such as Definition 3. An adversary which can efficiently solve RMSS problems can break the security of the system as we show here.

Once this adversary $\mathcal{A}$ receives a target $\mathsf{pk}_0 = g^x$, it chooses $\{(u_i, v_i)\}_{i=1}^n$ randomly as $\mathbb{Z}_q$ elements and sends $\mathsf{PK}_A = \{\mathsf{pk}_i = g^{u_i} \mathsf{pk}_0^{v_i}\}_{i=1}^n$ to the challenger. Let $h_i = \mathsf{H}_1(\mathsf{pk}_i || \mathsf{PK})$ for all $i \in [0, n]$, where $\mathsf{PK} = \mathsf{PK}_A \cup \{\mathsf{pk}_0\}$. The adversary solves the following RMSS instance:

- Target sum: $-h_0 \mod q$
- Set: $\{h_i \cdot v_i\}_{i=1}^n$

If $n = \Omega(\log q)$, whp there is a solution. Let's say the solution is $I \subseteq [1, n]$. This means $h_0 + \sum_{i \in I} h_i v_i = 0$. A SMSKR signature on a message m^* with subset $I \cup \{0\}$ is thus $\mathsf{H}_0(m^*)^{h_0 x + \sum_{i \in I} h_i (u_i + v_i x)} = \mathsf{H}_0(m^*)^{\sum_{i \in I} h_i u_i}$. This quantity can be readily computed by the adversary as it is independent of x.

Is the [BDN18b] multi-sig immune from this attack? In the multi-sig scheme of [BDN18b] the hash is computed on the exact subset which is signing the multi-sig. Thus the exact subset is committed in the random oracle exponent multipliers. There is no room to apply it to different subsets as is the case with SMSKR. Hence the above attack does not apply to [BDN18b].

Proof with RMSS assumption. The above attack highlights the need to assume that random subset sum problems are hard to compute for an adversary. In fact, now we show that the RMSS (Definition 3) and Discrete Log Problems together suffice to prove security of the scheme without an exponential loss in reduction.

Theorem 3. *SMSKR is an unforgeable multi-signature scheme (as per Definition 6) under the Discrete Logarithm and RMSS problems in the random-oracle and AGM models.*

Proof. We only show that the probability of event E is bound by Discrete Log and RMSS hardness. The rest of the proof is same as the last one.

Let $\mathcal{A}_E$ be an adversary which wins if it makes event E happen. We construct a challenger AE_{DL} which selects randomly, with probability $1/2$ each, from two rewinding strategies $GF^{eq}_{\mathcal{A}_E}$ and $GF^{\neg eq}_{\mathcal{A}_E}$ and applies forking to break Discrete Log hardness.

We now describe the algorithm $\mathcal{A}_E$. It runs like algorithm $\mathcal{B}$ as described above in simulating the target public key, H_0, H_1, and signature queries, to the adversary $\mathcal{A}_{alg}$. Let $\mathcal{A}_{alg}$ return a set of public keys and representations $\mathsf{PK}_A, \{(u_i, v_i, \vec{w}_i)\}_{i=1}^n$ and produce a forgery $(\Sigma^*, m^*, I, \vec{a})$ as described before. If event E didn't happen, then $\mathcal{A}_E$ returns $(0, \epsilon)$. Otherwise, let u be the index of the first query of the form $(\mathsf{pk}_{i_u} || \mathsf{PK})$ to H_1, where $\mathsf{PK} = \mathsf{PK}_A \cup \{\mathsf{pk}_0\}$ and $\mathsf{pk}_{i_u} \in \mathsf{PK}$. Let $h_i = \mathsf{H}_1(\mathsf{pk}_i || \mathsf{PK})$ for $i \in [0, n]$. Then $\mathcal{A}_E$ returns $(u, (\mathsf{PK}, \{(u_i, v_i, \vec{w}_{ij})\}_{i=1}^n,\ \{r_i\}_{i=1}^{q_H}, \{h_i\}_{i=0}^n, I))$. Based on this, the algorithm $GF^{\neg eq}_{\mathcal{A}_E}$ returns:

$$1, (\mathsf{PK}, \{(u_i, v_i, \vec{w}_i)\}_{i=1}^n, \{r_i\}_{i=1}^{q_H}, \{h_i\}_{i=0}^n, I),$$
$$(\mathsf{PK}', \{(u'_i, v'_i, \vec{w}'_i)\}_{i=1}^n, \{r'_i\}_{i=1}^{q_H}, \{h'_i\}_{i=0}^n, I')$$

Since the u-th query is identical for the two executions, we must have $\mathsf{PK} = \mathsf{PK}'$. Let E_{eq} denote the event $\forall i \in [1, n] : v_i = v'_i$. If E_{eq} occurs, then AE_{DL} aborts. Otherwise, there is an index k, such that $v'_k \neq v_k$. The Discrete Log challenger AE_{DL} then calculates the discrete log of pk_0 as $(u_k + \sum_{j=1}^{q_H} r_j w_{kj} - u'_k - \sum_{j=1}^{q_H} r'_j w'_{kj})/(v'_k - v_k)$.

Algorithm $GF^{eq}_{\mathcal{A}_E}$ gets an RMSS challenge $(S = \{s_i\}_{i=1}^n, t)$. It sends $h_0, h_1, ..., h_n \leftarrow \mathbb{Z}_q$ as usual in the first execution and gets back $(v_1, ..., v_n)$. In the rewinded execution, it sends $h'_0 = -t, h'_1 = s_i v_i^{-1}, ..., h'_n = s_n v_n^{-1}$. Observe that this respects the distribution of the original game and is also independently random from $h_0, h_1, ..., h_n$. Now if event $\neg E_{eq}$ occurs, then $\mathcal{A}_{DL}$ aborts. Otherwise, we have $(v_1, ..., v_n) = (v'_1, ..., v'_n)$. Therefore, we have $-t + \sum_{i \in I'} v_i s_i v_i^{-1} = 0$. In other words, $t = \sum_{i \in I'} s_i$, and hence I' is a valid solution to the RMSS problem.

Summing up, we have:

$$Pr[\mathcal{A}_E \text{ wins}] = 1/2(Pr[\mathcal{A}_E \text{ wins } DL] + Pr[\mathcal{A}_E \text{ wins } RMSS])$$

$$= 1/2\ (Pr[GF^{\neg eq}_{\mathcal{A}_E} \text{ wins }] \cdot Pr[\neg E_{eq}] + Pr[GF^{eq}_{\mathcal{A}_E}] \cdot Pr[E_{eq}])$$

Now, observe that,

$$Pr[GF^{eq}_{\mathcal{A}_E} \text{ wins }] = Pr[GF^{\neg eq}_{\mathcal{A}_E} \text{ wins }] \geq Pr[\mathcal{A}_E \text{ wins }] \cdot (Pr[\mathcal{A}_E \text{ wins }]/q_H - 1/q)$$

Therefore,

$$2\ Pr[AE \text{ wins }] \geq Pr[GF^{eq}_{\mathcal{A}_E}] \geq Pr[\mathcal{A}_E \text{ wins }] \cdot (Pr[\mathcal{A}_E \text{ wins }]/q_H - 1/q)$$

Therefore, following [BN06], we get:

$$Pr[\mathcal{A}_E \text{ wins }] \leq q_H/q + \sqrt{q_H \cdot 2Pr[AE \text{ wins }]}$$

$$= q_H/q + \sqrt{q_H \cdot (Pr[AE \text{ wins } DL] + Pr[AE \text{ wins } RMSS])}$$

On the dependence of assumptions on subset size. Our scheme allows any number of signers to form a subset, of size k out of a universe of size n, to aggregate sign the message. Some applications do restrict k to be within some limits, for example PoS blockchains that require 2/3 of the validators to sign. We take a look here to see how our security assumptions depend on the relation between k, n, q and the security parameter λ. Let's say we allow k to range between 1 and an upper bound max_k. The case of the range $[n - max_k, n]$ is symmetric.

- When the number of possible subsets is negligibly smaller than q, then the probability of existence of a subset sum solution is negligible. In this case, no subset attacks are possible and just the Discrete Log assumption is enough. This case arises if $\sum_{k=1}^{max_k} \binom{n}{k}/q \leq 2^{-\lambda}$.
- In all other cases, we have to additionally assume RMSS, albeit it will also include the maximum subset size max_k as a parameter.

We do an analysis with some concrete numbers of practical relevance. We pick the group size q to be a 256-bit prime, and security parameter $\lambda = 128$. In Figure 1, we plot the upper bound on subset size where the number of possible

subsets is less than the tolerance level $q/2^\lambda \approx 2^{128}$. To be precise, we plot the value of `max_k_aggregated_sum` in the y-axis against n in the x-axis, such that:

$$\texttt{max_k_aggregated_sum} = \max \left\{ max_k \in [1, n] \;:\; \sum_{k=1}^{max_k} \binom{n}{k} \le 2^{128} \right\}$$

Observe that the plot climbs linearly till $n = 128$. This is expected as the size of the full set of subsets keeps below threshold up to that point. When $n = 129$, the curve drops abruptly to $k = 64$. This is because, now the threshold is half the size of the full set of subsets, which is 2^{129}, and hence is realized at half the subset size. After this drop, the curve gradually slides down, reaching subset size ≤ 11 for $n = 10,000$. Python code for reproducing these are available open source. [13]

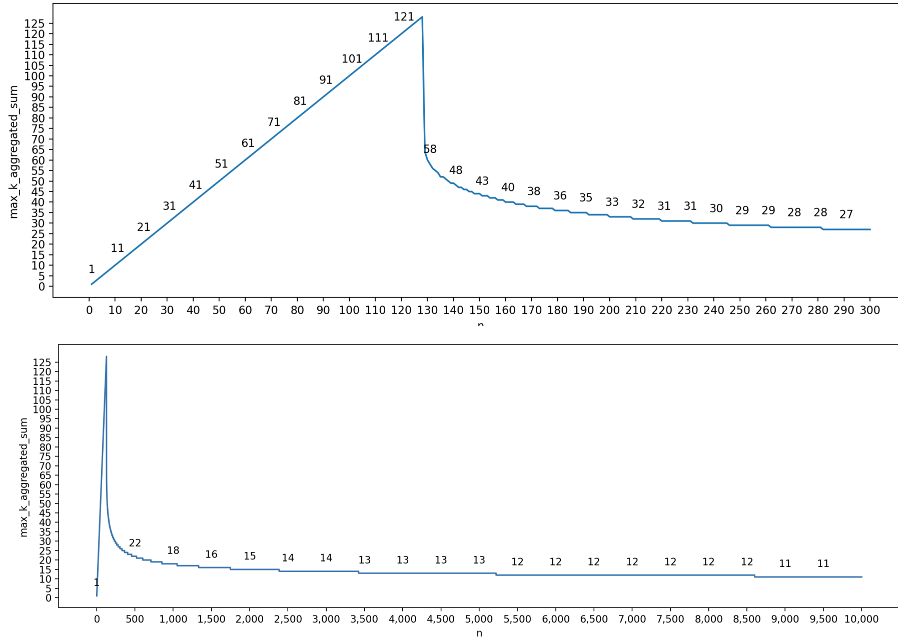

Fig. 1: Plots of upper bound on subset size where the number of possible subsets is less than the tolerance level. The top plot is for n going up to 300, and the bottom is for n going up to 10000.

Future Work. We concluded that the flexibility in choosing subsets, after committing to a superset of public keys, prevented us from using the [BDN18b]

[13] `https://github.com/MystenLabs/research/tree/main/cryptography/bls_aggregation_combinatorics`

template for security proof. This prompted us to explore the AGM model, led us to discover the subset-sum attack, and mitigate that using the RMSS assumption. In hindsight, we suspect that there may be further avenues to shore up the security proof. We plan to explore in the future if we can prove the scheme secure in the random oracle model assuming co-CDH and RMSS, or prove it in the Generic Group Model [Nec94,Sho97], instead of AGM. In addition, we are also exploring if it is possible to tweak the construction minimally to bolster security by binding to the subset, but at the same time retain its efficiency advantages.

4 Implementation and Evaluation

Implementation Details. We implement the scheme presented in Section 3 in Rust on top of the curve `bls12381`. We provide two production-ready implementations[14], one where the signature is a group element of $\mathbb{G}_1$ and the public key is a group elements of $\mathbb{G}_2$ (noted as *minSig*), and a second where the signature is a group element of $\mathbb{G}_2$ and the public key is group elements of $\mathbb{G}_1$ (noted as *minPk*). Our implementations are built on top of the library `blst` [Sup22] that provides base group operations over the curve `bls12381`. We implement the randomization components of the scheme as a self-contained Randomize trait allowing to augment existing BLS implementations to support SMSKR with minimal modifications. As a result, supporting our scheme only requires the addition of 50 LOC to define the Randomize trait and and extra 150 LOC to implement it. Furthermore, we provide an additional minimal prototype implementation[15] for didactic proposes (and that we do not benchmark). Its scope is to illustrate the implementation of our scheme with clarity without the numerous performance optimizations of our production-ready implementations. This minimal prototype is build on top of the library `bls12_381` [zkc22] providing base group operations over the curve `bls12381`.

Evaluation Results. We evaluate the performance of our production-ready SMSKR implementations described above. We perform our benchmarks on both a cheap Amazon Web Services (AWS) instance and a Macbook Pro equipped with a M1 processor. Our AWS experiments illustrates the performance of SMSKR on low-end devices. We select a *t3.medium* instance that comes with 2 virtual CPUs (1 physical core) on a 2.5GHz Intel Xeon Platinum 8259 and 4GB of RAM. Our experiments on the Macbook Pro illustrate the performance of our scheme on high-end devices with powerful CPUs. We select a Macbook Pro 14" equipped with a M1 Pro and 16GB of RAM. All our evaluations use Rust 1.65 and run with `cargo criterion` [bhe22]. We open-source our benchmarking scripts to enable reproducible results[16].

[14] `https://github.com/MystenLabs/fastcrypto`, (`6eb758ba78612e5e22a2748dd7a4b2c8b3724377`)

[15] `https://github.com/asonnino/mskr`, (`b212cb1ade13533ef330278dc8784d84641111e8`)

[16] `https://github.com/MystenLabs/fastcrypto/blob/mskr-bench/fastcrypto/src/mskr_bench.rs`, (`4d1bad60b6db5bfbb448d98d89a72cfaebab6e56`)

Function	AWS		M1	
	minSig	*minPk*	*minSig*	*minPk*
SMSKR.KeyGen	0.25	0.25	0.18	0.18
SMSKR.Sign	0.44	0.44	0.31	0.31
SMSKR.SigAggr	0.06	0.17	0.05	0.11
SMSKR.Verify	1.39	1.46	0.72	0.75

Table 1: Micro-benchmark of the main SMSKR functions on a a low-end *t3.medium* AWS instance and a high-end Macbook Pro equipped with a M1 CPU. Each data point represents the average time (over 100 runs) in milliseconds required to evaluate the function. All functions are evaluated for 100 public keys (except SMSKR.Sign that is independent of the number of public keys).

Our experiments aim to demonstrate the following claims. **(C1)** All functions of SMSKR are lightweight and performant even on low-end devices, **(C2)** SMSKR scales well when the number of signers increases, and **(C3)** SMSKR strictly outperforms the baseline scheme multi-signature of Boneh et.al. [BDN18b] (and the performance benefit increases with the number of signers).

4.1 Microbenchmarks

Table 1 illustrates the performance of both our implementations (*minSig* and *minPk*) on a single CPU core. The implementation of SMSKR.SubsetKeyAggr is deeply embedded into the function SMSKR.Verify. We thus report the performance of both functions together in the last row of the table. All functions are evaluated for 100 signers, except SMSKR.Sign which is independent of the number of signers. We compute the average time over 100 runs.

The table shows that key generation is cheap, taking respectively about 250 and 180 μs on our low-end AWS instance and on our high-end M1 Macbook Pro. Signing is also cheap and can be performed in less than 500 μs even on our low-end machine. Aggregating 100 signatures is the cheapest operation taking only a few microseconds on any machine. Finally, verifying 100 signatures takes 1.39 ms on our low-end machine and half that time (0.72 ms) on our high-end machine. The table shows there is little difference between our *minSig* and *minPk* implementations when operating with 100 signers. Section 4.2 illustrates that the performance difference between these implementations increases rapidly with the number of signers.

The results of Table 1 illustrates that even the most expensive function (SMSKR.Verify) running on a low-end machine takes less than 2 milliseconds. This demonstrates that SMSKR is lightweight and performant even on low-end devices, thus validating our claim **C1**.

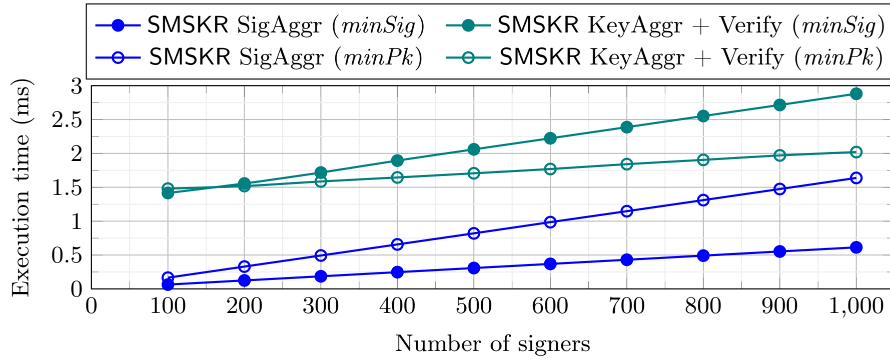

Fig. 2: Scalability of SMSKR on a low-end *t3.medium* AWS instance. Every data point on the graph is the average of 100 runs.

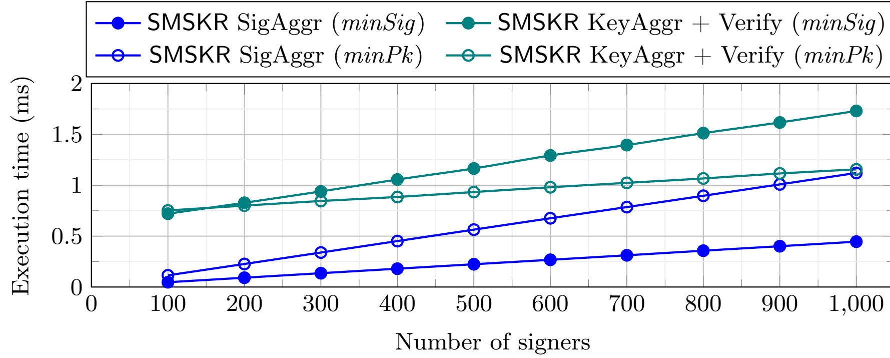

Fig. 3: Scalability of SMSKR on a high-end Macbook Pro equipped with a M1 processor. Every data point on the graph is the average of 100 runs.

4.2 Scalability

Figure 2 and Figure 3 illustrate the performance of SMSKR when the number of signers increases. We omit SMSKR.KeyGen and SMSKR.Sign from our analysis since the former function is only ran once at setup and the latter is independent of the number of signers. Each data point on the graphs below represent the average time computed over 100 runs. Similarly to Section 4.1 the verification process (green lines) includes both SMSKR.SubsetKeyAggr and SMSKR.Verify. The signature aggregation process (blue lines) is simply a call to SMSKR.SigAggr.

Figure 2 shows the time required to aggregate and verify signatures on our low-end machine. As expected, the figures show the time required to execute the aggregation and verification process increases linearly with the number of public keys. The signature and key aggregation processes require one EC addition for each signature (and public key) to aggregate. This cost quickly dominates the cost of any other operation (including the single pairing check required by

the verification process) as the number of signers increases. Both our *minSig* and *minPk* SMSKR implementations require less than 200 μs to aggregate signatures in a setting with less than 100 signers. SMSKR's signature aggregation scales well: our *minSig* and *minPk* implementations respectively require 300 μs and 800 μs to aggregate 500 signatures, and only 0.6 ms and 1.6 ms (respectively) to aggregate 1,000 signatures. We observe that our *minSig* signature aggregation implementation is faster than our *minPk* implementation. Indeed, our *minSig* (resp. *minPk*) implementation represents signatures in $\mathbb{G}_1$ (resp. $\mathbb{G}_2$) and EC additions are faster in $\mathbb{G}_1$ than in $\mathbb{G}_2$. The SMSKR verification process (SMSKR.SubsetKeyAggr and SMSKR.Verify) also scales well. Both our *minSig* and *minPk* implementations take about 1.5ms to run with 100 signers and respectively require 3 ms and 2 ms to run the verification process with 1,000 signers. Contrarily to signature aggregation, the verification process of our *minPk* implementation is faster than our *minSig* implementation. This is expected as our *minPk* implementation represents public keys in $\mathbb{G}_1$ (while *minSig* represents them in $\mathbb{G}_2$) where their aggregation is faster.

Figure 3 shows the time required to aggregate and verify signatures on our high-end machine. The graphs are similar to Figure 2 but display better performance. Signature aggregation (1,000 signatures) takes respectively 0.5 ms and 1.3 ms for our *minSig* and *minPk* implementations. The verification process (1,000 signers) takes respectively 1.7 ms and 1.3 ms for our *minSig* and *minPk* implementations.

Figure 2 and Figure 3 thus validate our scalability claim **C2**.

For completeness, Figure 4 and Figure 5 provide a 'zoomed' view on SMSKR's performance when the number of signers ranges from 10 to 100. We observe that signature aggregation is only affected by a few microseconds and the verification time does not visibly change.

4.3 Baseline Comparison

Figure 6 and Figure 7 compare the performance of SMSKR with the baseline scheme of Boneh et.al. [BDN18b] [17].

Figure 6 compares the time required to aggregate and verify SMSKR signatures with the baseline on our low-end machine. The figure shows that signatures aggregation of our SMSKR *minSig* and *minPk* implementations outperforms the baseline by two orders of magnitude, regardless of the number of signers. Our SMSKR *minSig* and *minPk* implementations respectively save 25 ms and 50 ms with respect to the baseline when aggregating 100 signatures, and a staggering 250 ms and 300 ms when aggregating 500 signatures. Similarly, the verification process of both our SMSKR *minSig* and *minPk* implementations outperforms the baseline by respectively 50x and 30x. The baseline scheme of Boneh et.al. [BDN18b] randomizes each signature before aggregation. Furthermore, it multiplies each signature and public key by a random exponent before their aggregation, thus

[17] Note that in both Figure 6 and Figure 7 the performance results for all SMSKR operations collapse to a single line.

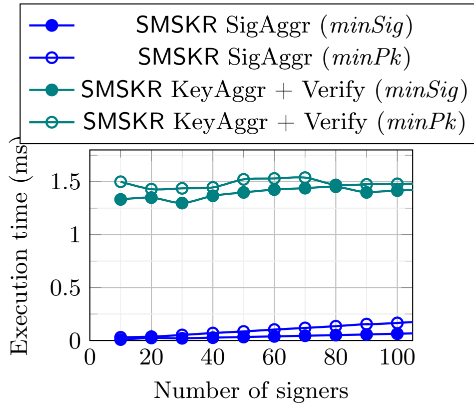

Fig. 4: Performance of SMSKR on a low-end *t3.medium* AWS instance. Every data point on the graph is the average of 100 runs.

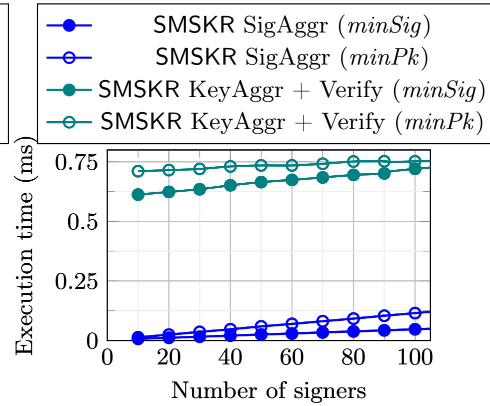

Fig. 5: Performance of SMSKR on a high-end Macbook Pro equipped with a M1 processor. Every data point on the graph is the average of 100 runs.

paying the cost of one EC addition and one scalar multiplication for every signature and public key. This accounts for the performance differences with SMSKR that randomizes secret keys (upon setup) rather than individual signatures and entirely forgoes any scalar multiplication during signature aggregation.

Figure 7 compares the time required to aggregate and verify SMSKR signatures with the baseline on our high-end machine. SMSKR greatly outperforms the baseline. The performance benefits are similar to the experiments on our low-end machine as the more powerful CPU scales performance roughly linearly. For 500 signers our SMSKR *minSig* and *minPk* signature aggregation respectively save 150 ms and 180 ms; and our SMSKR *minSig* and *minPk* signature verification implementations respectively save around 75 ms and 40 ms with respect to the baseline.

These figures validate our final claim **C3** by showing that SMSKR strictly outperforms the baseline and that the performance benefit increases linearly with the number of signers.

Acknowledgement

We thank the a16z crypto team for reviewing the paper, recommending improvements, and providing future extension ideas. In particular, we thank Dan Boneh (Stanford University) and Valeria Nikolaenko (a16z crypto research). We also thank Mahdi Sedaghat for reviewing this manuscript.

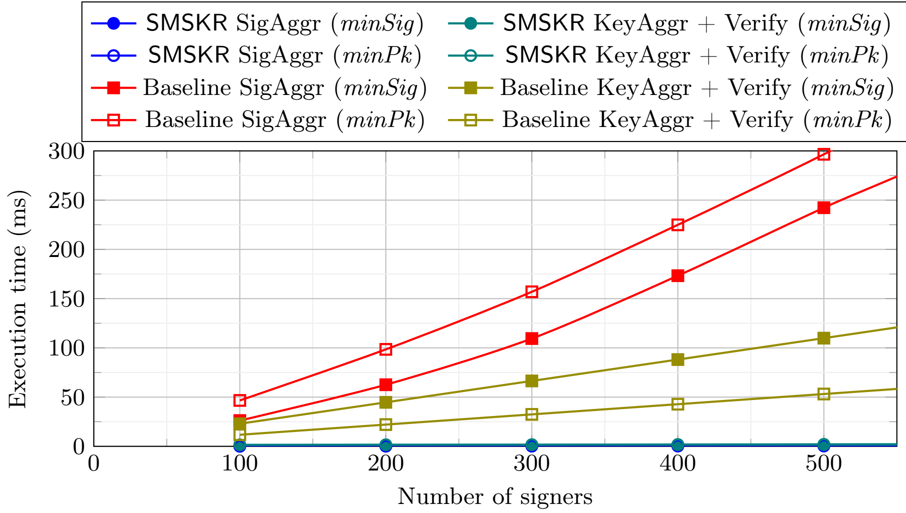

Fig. 6: Comparative performance of SMSKR with the baseline scheme of Boneh et.al. [BDN18b] on a low-end *t3.medium* AWS instance. Every data point on the graph is the average of 100 runs.

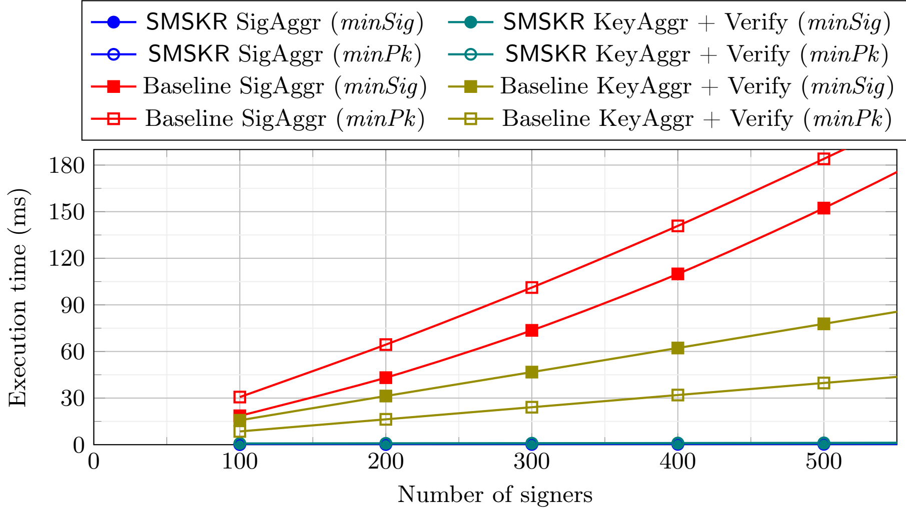

Fig. 7: Comparative performance of SMSKR with the baseline scheme of Boneh et.al. [BDN18b] on a high-end Macbook Pro equipped with a M1 processor. Every data point on the graph is the average of 100 runs.

References

BCJ08. Ali Bagherzandi, Jung Hee Cheon, and Stanislaw Jarecki. Multisignatures secure under the discrete logarithm assumption and a generalized fork-

ing lemma. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008: 15th Conference on Computer and Communications Security*, pages 449–458, Alexandria, Virginia, USA, October 27–31, 2008. ACM Press.

BDN18a. Dan Boneh, Manu Drijvers, and Gregory Neven. Bls multi-signatures with public-key aggregation. `https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html`, 2018.

BDN18b. Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 435–464, Brisbane, Queensland, Australia, December 2–6, 2018. Springer, Heidelberg, Germany.

BG18. J Benet and N Greco. Filecoin: A decentralized storage network. *Protocol Labs*, 2018.

BGOY07. Alexandra Boldyreva, Craig Gentry, Adam O'Neill, and Dae Hyun Yum. Ordered multisignatures and identity-based sequential aggregate signatures, with applications to secure routing. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007: 14th Conference on Computer and Communications Security*, pages 276–285, Alexandria, Virginia, USA, October 28–31, 2007. ACM Press.

bhe22. bheisler. cargo-criterion. `https://github.com/bheisler/cargo-criterion`, 2022.

BLS01. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532, Gold Coast, Australia, December 9–13, 2001. Springer, Heidelberg, Germany.

BN06. Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006: 13th Conference on Computer and Communications Security*, pages 390–399, Alexandria, Virginia, USA, October 30 – November 3, 2006. ACM Press.

Bol02. Alexandra Boldyreva. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In Yvo G. Desmedt, editor, *Public Key Cryptography — PKC 2003*, Lecture Notes in Computer Science, pages 31–46, Berlin, Heidelberg, 2002. Springer.

DEF+19. Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the security of two-round multi-signatures. In *2019 IEEE Symposium on Security and Privacy*, pages 1084–1101, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

Dra18. Justin Drake. Pragmatic signature aggregation with BLS - Sharding, May 2018.

Edg. Ben Edginton. Upgrading Ethereum.

ES16. Rachid El Bansarkhani and Jan Sturm. An efficient lattice-based multisignature scheme with applications to bitcoins. In Sara Foresti and Giuseppe Persiano, editors, *CANS 16: 15th International Conference on Cryptology and Network Security*, volume 10052 of *Lecture Notes in Computer Science*, pages 140–155, Milan, Italy, November 14–16, 2016. Springer, Heidelberg, Germany.

Eth. Ethereum Core developers. Ethereum Proof-of-Stake Consensus Specifications.

FH20. Masayuki Fukumitsu and Shingo Hasegawa. A lattice-based provably secure multisignature scheme in quantum random oracle model. In Khoa Nguyen, Wenling Wu, Kwok-Yan Lam, and Huaxiong Wang, editors, *ProvSec 2020: 14th International Conference on Provable Security*, volume 12505 of *Lecture Notes in Computer Science*, pages 45–64, Singapore, November 29 – December 1, 2020. Springer, Heidelberg, Germany.

FKL18. Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 33–62, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

Fri86. Alan M Frieze. On the lagarias-odlyzko algorithm for the subset sum problem. *SIAM Journal on Computing*, 15(2):536–539, 1986.

Gro21. Jens Groth. Non-interactive distributed key generation and key resharing, 2021. Report Number: 339.

IN96. Russell Impagliazzo and Moni Naor. Efficient cryptographic schemes provably as secure as subset sum. *Journal of Cryptology*, 9(4):199–216, September 1996.

Ita83. K. Itakura. A public-key cryptosystem suitable for digital multisignatures. 1983.

LO85. Jeffrey C Lagarias and Andrew M Odlyzko. Solving low-density subset sum problems. *Journal of the ACM (JACM)*, 32(1):229–246, 1985.

LOS+06. Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 465–485, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.

Lyu05. Vadim Lyubashevsky. On random high density subset sums. *Electron. Colloquium Comput. Complex.*, TR05, 2005.

Nec94. Vasilii Il'ich Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Matematicheskie Zametki*, 55(2):91–101, 1994.

NKDM03. Antonio Nicolosi, Maxwell N. Krohn, Yevgeniy Dodis, and David Mazières. Proactive two-party signatures for user authentication. In *ISOC Network and Distributed System Security Symposium – NDSS 2003*, San Diego, CA, USA, February 5–7, 2003. The Internet Society.

NRS21. Jonas Nick, Tim Ruffing, and Yannick Seurin. MuSig2: Simple two-round Schnorr multi-signatures. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part I*, volume 12825 of *Lecture Notes in Computer Science*, pages 189–221, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.

NRSW20. Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. MuSig-DN: Schnorr multi-signatures with verifiably deterministic nonces. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1717–1731, Virtual Event, USA, November 9–13, 2020. ACM Press.

OO93. Kazuo Ohta and Tatsuaki Okamoto. A digital multisignature scheme based on the Fiat-Shamir scheme. In Hideki Imai, Ronald L. Rivest, and Tsutomu

Matsumoto, editors, *Advances in Cryptology – ASIACRYPT'91*, volume 739 of *Lecture Notes in Computer Science*, pages 139–148, Fujiyoshida, Japan, November 11–14, 1993. Springer, Heidelberg, Germany.

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

Sch90. C. P. Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York.

Sho97. Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266, Konstanz, Germany, May 11–15, 1997. Springer, Heidelberg, Germany.

Sup22. Supranational. blst. `https://github.com/supranational/blst`, 2022.

VGS+22. Psi Vesely, Kobi Gurkan, Michael Straka, Ariel Gabizon, Philipp Jovanovic, Georgios Konstantopoulos, Asa Oines, Marek Olszewski, and Eran Tromer. Plumo: An Ultralight Blockchain Client. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 597–614, Berlin, Heidelberg, May 2022. Springer-Verlag.

zkc22. zkcrypto. bls12_381. `https://github.com/zkcrypto/bls12_381`, 2022.