MONERO RESEARCH LAB

RESEARCH BULLETIN MRL-0007

# Linkable Spontaneous N-of-N and (N-1)-of-N Threshold Anonymous Group Signatures

Brandon Goodell* and Sarang Noether†
Monero Research Lab
March 17, 2018

**Abstract**

This research bulletin extends [10] by constructing $N$-of-$N$ and $(N-1)$-of-$N$ threshold linkable spontaneous anonymous group multisignatures without Shamir secret sharing in the random oracle model.

## 1 Introduction and Background

Cryptocurrencies are systems based on unforgeable message authentication. Although Bitcoin, the first and inarguably most important cryptocurrency, uses the usual ECDSA on secp256k1 for signatures, this is by no means the only option available to cryptocurrency engineers. Other methods allow for signer-ambiguous message authentication: a verifier may check that some member of an anonymity set authenticates the transaction, and yet the verifier has at best a negligible advantage in determining which member of the anonymity set computed the authentication. In the cryptocurrency setting, this translates to *spender ambiguity*. For example, ZCash uses ZK-SNARKs for message authentication, and Monero uses ring signatures.

Many proposals for scaling cryptocurrencies, such as cross-chain atomic swaps, involve threshold authentication schemes. These are also useful for multi-factor authentication. It is therefore natural to seek threshold extensions to message authentication: a sufficient number of private keys must collaborate to compute authentications. Threshold ring multisignatures have been previously proposed (see, e.g. [4], [5], [13], [1]). However, most of these previous proposals are leaky, revealing properties of the signing coalition such as the number of signers, or even publishing the list of signing public keys with signatures. For these reasons, we find these proposals to be inappropriate for use in a privacy-focused cryptocurrency like Monero.

### 1.1 Our Contribution

In [10], a threshold ring multisignature scheme based on the LSAG signatures from [6] was sketched. This approach seemed to be more private than those presented in [4], [5], [13], and [1]. We present this scheme formally, which uses the LSAG signature scheme as a sub-scheme, an encrypt-then-authenticate sub-scheme, a secure signature sub-scheme, and a Pedersen commitment sub-scheme. Our scheme allows cooperating sets of (possibly adversarial) users to aggregate keys with a key-merging algorithm and collaborate to compute multisignatures. We show our scheme has the property of *key aggregation* (first described in [8]) such that signature verification proceeds without the verifier necessarily learning that the underlying signature is a multisignature or that the signing key is an aggregated key. We strengthen this property: key aggregation is *indistinguishable* if adversaries cannot feasibly discern if a merged key is aggregated. The scheme presented

*surae.noether@protonmail.com
†sarang.noether@protonmail.com

in [8] is not indistinguishable, because it is trivial to check if a given public key has been aggregated from a specific set of public keys[‡].

The scheme also has the property of *privacy* such that users do not reveal their private keys to each other. The scheme has the property of *resistance to key cancellation* such that malicious players can either cancel keys or can participate in honest collaborative signing, but not both (except with negligible probability). We make some observations about *signer ambiguity*, *linkability*, and *exculpability*. We also consider *existential unforgeability* against chosen message attacks. Extending some approaches from [8], we present models of privacy, resistance to key cancellation, and unforgeability for threshold multisignatures; these models must take into account an adversary with *subthreshold insider access* to a coalition.

We endeavor to show which cryptographic hardness assumptions are required for each property. In [11], it is shown that the Schnorr signatures from [12] cannot reduce to the discrete logarithm problem under the standard model, and in [8], the one-more discrete logarithm problem is used to demonstrate the unforgeability of an $N$-of-$N$ aggregate-key Schnorr multisignature. Our unforgeability property relies upon a strengthening of the discrete logarithm hardness assumption to the one-more discrete logarithm hardness assumption.

### 1.2 Related Works

In [10], ring confidential transactions were first described for use in Monero and, in that paper, the thresholdizing heuristic used here was described briefly. In [6], a rewind-on-success forking lemma was first described to prove the unforgeability of a multisignature scheme; later, in [1] and [8] this proof technique is elaborated upon in detail. The recent paper [8] rigorously establishes the security of a rather elegant key aggregation multisignature scheme. That paper presents the formal and very general definition of the forking lemma we use herein, and proves the unforgeability of their signature scheme by forking upon success twice, rather than once. In [3], a novel and quite general fully homomorphic thresholdizing set-up using only one round of communication is described, leading to fully homomorphic threshold signatures and encryption.

### 1.3 Special Thanks

We want to specially thank the members of the Monero community who used the `GetMonero.org` Community Forum Funding System to support the Monero Research Lab. Readers may also regard this as a statement of conflict of interest, since our funding is denominated in Monero and provided directly by members of the Monero community by the Forum Funding System.

## 2 Notation and Prerequisites

Throughout, we assume the random oracle model and we make the discrete logarithm hardness assumption. For any set $\mathfrak{S}$, we define $\mathcal{P}(\mathfrak{S})$ as the power set of $\mathfrak{S}$. We let $\mathbb{G}$ be a group with prime order $\mathtt{q}$ and we let $G \in \mathbb{G}$ denote a commonly-known generator. We abuse notation by denoting the integers modulo $\mathtt{q}$ as $\mathbb{Z}_{\mathtt{q}}$. A key is a pair $(x, X) \in \mathbb{Z}_{\mathtt{q}} \times \mathbb{G}$ such that $X = xG$. We say $x$ is the private key and $X$ is the public key. Let $\mathbb{M} = \{0,1\}^*$ denote a message space, and let $H_p : \mathbb{M} \to \mathbb{G}$ (or "hash-to-point") and $H_s : \mathbb{M} \to \mathbb{Z}_{\mathtt{q}}$ (or "hash-to-scalar") be independent cryptographic hash functions under the random oracle model. The (additively homomorphic) Pedersen commitment scheme is a pair of PPT algorithms $(\mathtt{Com}, \mathtt{Open})$ such that $\mathtt{Com}(\alpha, r) := rG + \alpha H_p(G)$ and $\mathtt{Open}(C, \alpha, r)$ outputs a bit denoting whether $C - rG - \alpha H_p(G) = 0$. A commitment to zero is a usual public key.

[‡]Though, in practice, indistinguishability can simply be attained by not reusing keys, and this is recommended in implementation.

# 3 LSAG signatures and a thresholdizing heuristic

In this section, we briefly describe an application of LSAG signatures for use in cryptocurrencies. We use these as ring signatures with public one-time keys on specially constructed messages that pass new one-time keys to recipients. We also describe a key aggregation heuristic for thresholdizing the scheme. Observe that extending these approaches to the MLSAG signatures presented in [10] provides the framework for ring confidential transactions in Monero (and our thresholdizing heuristic applies). In future sections, we present a more usual signature scheme.

## 3.1 One-time LSAG signatures

We specify a user key space $\mathcal{K}_{\texttt{user}} = \mathbb{Z}_{\mathfrak{q}}^2 \times \mathbb{G}^2$ and a signing key space $\mathcal{K}_{\texttt{sig}} = \mathbb{Z}_{\mathfrak{q}}^2 \times \mathbb{G}^2$. For user key $((a,b),(A,B)) \in \mathcal{K}_{\texttt{user}}$, we say $a$ is the *private view key*, $b$ is the *private spend key*, $A$ is the *public view key*, and $B$ is the *public spend key*. For $((t,p),(T,P)) \in \mathcal{K}_{\texttt{sig}} = \mathbb{Z}_{\mathfrak{q}}^2 \times \mathbb{G}^2$, $t$ is the *private transaction key*, $p$ is the *private one-time key*, $T = tG$ is the *public transaction key*, and $P = pG$ is the *public one-time key*.

Alice has a user key $((a,b),(A,B)) \in \mathcal{K}_{\texttt{user}}$, and has previously received a message containing some public signing keys $(T,P)$ addressed to $(A,B)$. Moreover, Alice has computed $p$, the discrete logarithm of $P$ (how Alice did so will be clear by the end of this section). Alice wishes to address a message to Bob, who has public user key $(A',B')$. Alice picks a new private transaction key $t'$ and uses $t'$ and $(A',B')$ to compute a new public one-time key $P'$ addressed to $(A',B')$. Alice then sends $(T',P')$ to Bob in a message and authenticates the message with a ring signature with a ring including $P$.

To do so, Alice selects a random $t' \in \mathbb{Z}_{\mathfrak{q}}$, computes $P' := H_s(t'A')G + B'$, selects a message $M$, computes a key image $J = pH_p(P)$, and selects a ring of public signing keys $\mathcal{Q} = \{P_1, \dots P_R\}$ such that, for a secret distinguished index $\pi$, $P_\pi = P$. The signer assembles a modified message $M^* = (M, J, T', P', \mathcal{Q})$. For each $\ell = 1, \dots, R$, the signer computes an elliptic curve point from the $\ell^{th}$ ring member $H_\ell := H_p(P_\ell)$. The signer selects a random secret scalar $u \in \mathbb{Z}_{\mathfrak{q}}$ and computes an initial temporary pair of points $\mathcal{L}_\pi := uG$, $\mathcal{R}_\pi := uH_\pi$. The signer computes an initial commitment $c_{\pi+1} := H_s(M^*, \mathcal{L}_\pi, \mathcal{R}_\pi)$. The signer proceeds through indices $\ell = \pi+1, \pi+2, \dots, R-1, R, 1, 2, \dots, \pi-1$ by selecting a random scalar $s_\ell$, computing the next pair of points $\mathcal{L}_\ell := s_\ell G + c_\ell P_\ell$ and $\mathcal{R}_\ell := s_\ell H_\ell + c_\ell J$, and computes the next commitment $c_{\ell+1} := H_s(M^*, \mathcal{L}_\ell, \mathcal{R}_\ell)$. The signer continues proceeding through the set of keys until all commitments $c_\ell$ have been computed. The signer then computes $s_\pi := u - c_\pi p$ for the distinguished index $\pi$. Now $\sigma = (c_1, s_1, \dots, s_R)$ is the signature on $M^*$, which is sent to Bob.

A verifier checks the signature in the following way. Given a message $M^*$ and signature $\sigma$, the verifier parses $M^* = (M, J, T, P, \mathcal{Q})$ and $\sigma = (c_1, s_1, \dots, s_R)$. For each $1 \le \ell \le R$, the verifier finds $\mathcal{L}'_\ell = s^*_\ell G + c^*_\ell P_\ell$, $\mathcal{R}'_\ell = s^*_\ell H_\ell + c^*_\ell J^*$ and uses these to compute the $(\ell+1)^{th}$ commitment $c^*_{\ell+1} = H_s(M^*, \mathcal{L}'_\ell, \mathcal{R}'_\ell)$. After computing $c^*_2, c^*_3, \dots, c^*_R, c^*_1$, the verifier approves of the signature if and only if $c_1 = c^*_1$. A verifier can check against double spends by comparing the key images $J$ of two signature-tag pairs. Bob can parse $M^*$ to obtain the original message, the key image $J$, a public transaction key $T'$ and a public one-time key $P'$. Bob can easily test if $P' = H_s(a'T')G + B'$ and compute $p' = H_s(a'T') + b'$. This way, Bob can perform the same procedure Alice does above to pass new signing keys to other users.

## 3.2 A naive N-of-N key-aggregation threshold heuristic

We begin the task of aggregating keys with a heuristic: replace keys with sums of keys and replace random signing data with sums of random signing data. First, we combine user keys with sums to generate merged keys. Second, we execute signing algorithms similarly to as usual except any time a signing coalition (rather

than a signer) is tasked with selecting a random value for use in the LSAG signature, say $u$ or $s$, each member of the coalition first selects a random $u_j$ or $s_j$ and the coalition uses $u := \sum_j u_j$ or $s := \sum_j s_j$. This section mirrors Section 3.1, but Alice is now represented by a coalition of signers. We discuss drawbacks of this implementation later.

The coalition of signers computes the merged or shared key as $(A_{\mathtt{sh}}, B_{\mathtt{sh}}) = (a_{\mathtt{sh}}G, \sum_j B_j)$ where $a_{\mathtt{sh}}$ is a shared private view key. The coalition has previously received a message containing some public signing keys $(T, P)$ addressed to $(A_{\mathtt{sh}}, B_{\mathtt{sh}})$ such that $P = H_s(a_{\mathtt{sh}}T)G + B_{\mathtt{sh}}$. The coalition wishes to address a message $M$ to Bob, who has public user key $(A', B')$, using ring $\mathcal{Q}$.

To do so, each signing coalition member selects a share of a private transaction key $t'_j$, and communicates these values to all other coalition members. Each coalition member can then compute$t' = \sum_j t'_j$, $T' = t'G$, and $P' = H_s(t'A')G + B'$. To compute the key image of $P$, each coalition member computes $J_j = b_j H_p(P)$ and sends it to every other coalition member. Now each coalition member can compute $J = H_s(a_{\mathtt{sh}}T)H_p(P) + \sum_j b_j H_p(P) = pH_p(P)$. Now each coalition member can assemble the modified message $M^*$, select a random secret scalar $u_j \in \mathbb{Z}_{\mathtt{q}}$, compute the points $\mathcal{L}_{\pi,j} = u_j G$ and $\mathcal{R}_{\pi,j} = u_j H_\pi$, and send each $\mathcal{L}_{\pi,j}$, $\mathcal{R}_{\pi,j}$ to all other participants. The sums $\mathcal{L}_\pi = \sum_j \mathcal{L}_{\pi,j}$ and $\mathcal{R}_\pi = \sum_j \mathcal{R}_{\pi,j}$ can now be computed by every participant, who can now compute the commitment $c_{\pi+1} = H_s(M^*, \mathcal{L}_\pi, \mathcal{R}_\pi)$. The coalition proceeds through the indices $\ell = \pi+1, \ldots, \pi-1$ as before by selecting random scalars $s_{\ell,j}$ and sending them to all other coalition members. Each member can then compute $s_\ell = \sum_j s_{\ell,j}$, compute the next pair of points $\mathcal{L}_\ell := s_\ell G + c_\ell P_\ell$ and $\mathcal{R}_\ell := s_\ell H_\ell + c_\ell J$, and the commitment $c_{\ell+1} = H_s(M^*, \mathcal{L}_\ell, \mathcal{R}_\ell)$, proceeding this way until all commitments have been computed.

Each coalition member computes $s'_{\pi,j} = u_j - c_\pi b_j$ and sends these scalars to all other members. Each coalition member can now compute $s'_\pi := \sum_j s'_{\pi,j}$, $s_\pi := s'_\pi - c_\pi \left(H_s(a_{\mathtt{sh}}T)\right)$. Any member can now publish the signature-tag pair $(\sigma, J)$ where $\sigma = (c_1, s_1, \ldots, s_R)$ on the modified message $M^* = (M, J, T', P', \mathcal{Q})$.

As described, there are many rounds of communication, but the entire scheme can reduce to two stages of signing. The first stage is a pre-processing stage. Each participant computes their partial key image $J_j$ and selects the random signing data $t'_j$, $u_j$, and $s_{\ell,j}$. Each member computes the points $U_j = u_j G$ and $V_j = u_j H_\pi$ and sends $\left(t'_j, U_j, V_j, \{s_{\ell,j}\}_{\ell \neq \pi}\right)$ to all other coalition members. The sums $t' = \sum_j t'_j$, $U = \sum_j U_j$, $V = \sum_j V_j$, $s_\ell = \sum_j s_{\ell,j}$ are computed by every participant. This concludes the first stage. In the second stage, each coalition member computes $s_{\pi,j} := u_j - c_\pi b_j$ and sends it to all other coalition members. Anyone in the coalition can now compute $s'_\pi := \sum_j s_{\pi,j}$ and $s_\pi = s'_\pi - c_\pi H_s(a_{\mathtt{sh}}T)$. Any coalition member can then publish $\sigma = (c_1, s_1, \ldots, s_R)$ and $M^*$.

## 3.3 Observations

### 3.3.1 Benefits

No subthreshold coalition of malicious participants directly learn the private key $p$ or any other participants' private spend keys $b_j$. Indeed, coalition members share a point $U_j$ and a scalar $s_j$ such that $U + c_\pi B_j = s_j G$, but since the discrete logarithm $u_j$ is kept secret, an adversary $\mathcal{A}$ cannot gain information about $b_j$ without violating the discrete logarithm hardness assumption.

Another benefit is that, assuming at least one contributing user key was honestly generated from a uniform random scalar, an adversary from outside the coalition cannot determine the number of summands contributing to a key without brute-force guessing sums. Moreover, signatures computed by honest parties collaborating in Section 3.2 are statistically indistinguishable from signatures from Section 3.1 and are verified with the usual LSAG verification algorithm.

Furthermore, the thresholdizing heuristic works for coalitions containing coalitions: each time a member

of a coalition must make a decision in a signing protocol, the coalition may simply securely compute a sum. For example, when the $j^{th}$ coalition member computes the partial key image $J_j$, if this coalition member is some sub-coalition, then each member of the sub-coalition can compute their own partial key image $J_{j,k}$ and the sub-coalition can report the sum $J_j = \sum_k J_{j,k}$. In this manner, signatures involving nested coalitions may be executed recursively, allowing for detailed access structures to signing rights. We handle this in Section 4 by considering a directed acyclic graph on keys indicating which keys contribute to which shared or merged keys; we call this graph the *family history*.

The set-up extends naturally to $(N-1)$-of-$N$ using ECDH: each pair of users with indices $i$ and $j$ may construct a shared secret scalar $z_{i,j} = H_s(x_i X_j)$ with associated point $Z_{i,j} = z_{i,j}G$. Under the CDH hardness assumption, an observer without $x_i$ or $x_j$ cannot compute $x_i X_j$. Under the stronger DDH hardness assumption, an observer given some $P \in \mathbb{G}$ but without $x_i$ or $x_j$ cannot discern if satisfies $P = x_i X_j$ or not. Under the random oracle model, each $z_{i,j}$ is a uniform random choice from $\mathbb{Z}_{\mathtt{q}}$. This way, an adversary cannot feasibly discern whether some $z_{i,j}$ was computed by a given pair of public user keys. There are $\frac{N(N-1)}{2}$ such pairs and if any $N-1$ members get together, all of the associated shared secrets are known. Hence, we may simply take an $(N-1)$-of-$N$ threshold signature as an instantiation of an $\frac{N(N-1)}{2}$-of-$\frac{N(N-1)}{2}$ threshold signature. We describe this in some more detail in Section 6.1.

### 3.3.2 Drawbacks

This scheme as described is not particularly secure. Even if computation of the sum of user keys is done privately, using the simple sum allows distinguishability. Indeed, given public keys $X_1, X_2$, for a 2-of-2 shared key, an adversary $\mathcal{A}$ can use this public knowledge to trivially test for key aggregation $X_{\mathtt{shared}} = X_1 + X_2$. We require a method of aggregating keys that forbids an adversary from using strictly public information to test for key aggregation.

Another problem is key cancellation attacks, where an adversary collaborates with honest parties to compute a merged key with a smaller effective threshold. With a simple sum $X_{\mathtt{shared}} = X_1 + X_2$, setting $X_1 = aG - X_2$ for known scalar allows $\mathcal{A}$ to cancel the key $X_2$, yielding a "shared key" $aG$ to which $\mathcal{A}$ knows the discrete logarithm. This reduces an ostensible 2-of-2 key to a usual key.

One dangerous point of this protocol is that re-use of values of $u_j$ may accidentally reveal their $b_j$ values or the private key $p$. With a hash function resistant to second pre-image attacks, the commitments from two signature processes $c_\pi, c_\pi^*$ are unequal except with negligible probability, even if the other threshold members are colluding. Hence, re-using $u_j$ leads to the coalition learning from some coalition member $c_\pi$, $c_\pi^*$, $s$, and $s^*$ such that $s = u_j - c_\pi b_j$, $s^* = u_j - c_\pi^* b_j$, and such that $c_\pi \neq c_\pi^*$ except with negligible probability. This leaks the private spend key $b_j$:

$$\begin{aligned} s + c_\pi b_j =& s^* + c_\pi^* b_j \\ b_j =& \frac{s - s^*}{c_\pi^* - c_\pi} \end{aligned}$$

## 3.4 Modifications

In Section 4, we present a model and implementation that takes these observations into account by making the following modifications.

### 3.4.1 Indistinguishable key aggregation

A malicious party inside a coalition always can tell they have participated in constructing a key, so we only consider an adversary outside the coalition. The $j^{th}$ coalition members with user key $(A_j, B_j)$ and private

user keys $(a_j, b_j)$ selects a pair of random masks $\mu_j^a$, $\mu_j^b$ and computes $a_j^* := H_s(a_j, \mu_j^a)$, $b_j^* := H_s(b_j \mu_j^b)$. Now $(a_j^*, b_j^*)$ is an essentially unique commitment to $(a_j, b_j)$ and can be used as a private derived user key or *private masked user key*. This way, the owner of $(a_j, b_j)$ can make it difficult for adversaries to discern if a public key $(A_j, B_j)$ was involved in an aggregated key without merely generating new keys. The corresponding points $A_j^*$, $B_j^*$ are shared only with other participants via an encrypt-then-authenticate protocol. Without insider corruption access to the participating coalition, the secrecy of each $X_j^*$ within the coalition reduces to the security of the encrypt-then-authenticate protocol and key selection.

Note that if any participant in the coalition is honest then each $X_{\mathtt{shared}}$ is drawn from a uniform distribution. This is true even if this "honest" user is not fully honest and misbehaves by selecting a new random pair of keys $((a^*, b^*), (A^*, B^*))$ instead of hashing their old keys. Indeed, the sum of a uniform random variable on a finite cyclic group with any other random variable is a uniform random variable on that group.

#### 3.4.2 Resistance to key cancellation

We mitigate the problem of key cancellation, also. In [9], this attack is avoided with a sophisticated interactive key generation protocol with two rounds of communication between all participants that requires commitments to the random values used in key computation. In [8], key cancellation is avoided by computing keys as $X_{\mathtt{shared}} = \sum_j H_s(X_j, \{X_1, X_2, \ldots, X_N\})X_j$, but this key can be easily computed from strictly public information, requiring new keys for new multisignature addresses. Our scheme has a single round of communication between all players to merge keys. Our scheme is *key cancellation resistant* such that an attacker can successfully cancel keys with a set of honest players or successfully collaborate with that set to construct multisignatures, but not both.

Explained in Section 5.2.1, this is accomplished by requiring usual (Schnorr-like) signatures from each participant using their summand key shares; cancelling other keys and computing signatures requires knowledge of the discrete logarithm of the cancelled keys.

#### 3.4.3 Privacy

In order to pass secret information around the coalition securely, it should be done with an encrypt-then-authenticate protocol. We use a secure encryption sub-scheme and a usual (Schnorr-like) signature sub-scheme to accomplish this. For example, in order to compute $J$, each player sends their share $J_j$ to each other player first by encrypting and then by authenticating with a signature. This way, the key image can be kept secret before publication, preventing transaction censorship.

#### 3.4.4 Randomization

The only rectification for the problem of re-using $u_j$ is to repeat often and strongly: *always use new values of* $u_j$. One ostensibly common choice for generating pseudorandom numbers is to follow the standard RFC6979, but this is deterministic and can lead to accidental re-use of random data.

## 4 LSTAG signatures

In this section we define a simple $N$-of-$N$ linkable spontaneous threshold anonymous group (LSTAG) signature scheme. Let $(\mathtt{KeyGen}^*, \mathtt{Sign}^*, \mathtt{Ver}^*, \mathtt{Link}^*)$ be a linkable ring signature sub-scheme. Let $(\mathtt{KeyGen}', \mathtt{Enc}, \mathtt{Dec})$ be a secure symmetric-key encryption sub-scheme, and let $(\mathtt{KeyGen}'', \mathtt{Sign}'', \mathtt{Ver}'')$ be an existentially unforgeable signature of knowledge sub-scheme. We assume all key spaces are $\mathcal{K} = \mathbb{Z}_{\mathfrak{q}} \times \mathbb{G}$ and $\mathtt{KeyGen}^* = \mathtt{KeyGen}' = \mathtt{KeyGen}''$.

**Definition 4.1.** An $N$-of-$N$ linkable aggregate-key ring multisignature scheme is a tuple of PPT algorithms $(\texttt{KeyGen}, \texttt{Merge}, \texttt{Sign}, \texttt{Ver}, \texttt{Link})$ satisfying the following.

1. $\texttt{KeyGen}$ is identical to $\texttt{KeyGen}^* = \texttt{KeyGen}' = \texttt{KeyGen}''$, producing a key pair $(x, X) \in \mathcal{K}$ chosen uniformly at random.

2. $\texttt{Merge}$ is run collaboratively and takes as input a subset $\{X_1, \ldots, X_N\} \subset \mathbb{G}$, and a set of pairs of scalars $(x_i, m_i) \in \mathbb{Z}_{\mathtt{q}}^2$ where each $x_i$ is the private key for the corresponding $X_i$ and $m_i$ is a random scalar. $\texttt{Merge}$ outputs an $N$-of-$N$ shared public key $X_{\mathtt{sh}}$ to all participants, and locally outputs a secret share $x_i^* \in \mathbb{Z}_{\mathtt{q}}$ to each participant such that $X_{\mathtt{sh}} = \sum_i x_i^* G$.

3. If $N = 1$, $\texttt{Sign}$ is identical to $\texttt{Sign}^*$. Otherwise, $\texttt{Sign}$ is run collaboratively, takes as input a message $M \in \{0,1\}^*$, a ring $\mathcal{Q} = \{X_1, \ldots, X_R\}$ of public keys, a secret index $1 \leq \pi \leq R$, and a set of secret shares $\left\{x_j^*\right\}_{j=1}^{N}$ such that $X_\pi = \sum_j x_j^* G$. $\texttt{Sign}$ outputs a signature $\sigma$ on a modified message $M^* = (M, J, \mathcal{Q})$.

4. $\texttt{Ver}$ is identical to $\texttt{Ver}^*$, taking as input a signature $\sigma$ with a modified message $M^* = (M, J, \mathcal{Q})$ and producing as output a bit indicating whether $\sigma$ is a valid ring signature on $M^*$ with ring $\mathcal{Q}$.

5. $\texttt{Link}$ is identical to $\texttt{Link}^*$, taking as input two ring signatures and modified messages $(\sigma_1, M_1^*)$ and $(\sigma_2, M_2^*)$ and outputting a bit indicating whether $\sigma_1$ is a valid ring signature on $M_1^*$ and $\sigma_2$ is a valid ring signature on $M_2^*$ and $\sigma_1$, $\sigma_2$ have been computed by the same key.

**Definition 4.2.** We say an implementation of Definition 4.1 is *correct* if both of the following hold:

(i) *correctness as a signature*: for any message $M$, for any ring $\mathcal{Q}$, for any index $\pi$, and for any set of private keys $\left\{x_j^*\right\}_j \subseteq \mathbb{Z}_{\mathtt{q}}$ such that $\sum_j x_j^* G = X_\pi \in \mathcal{Q}$, we have that

$$\mathbb{P}\left[\texttt{Ver}(\texttt{Sign}(M, \mathcal{Q}, i, \left\{x_j^*\right\}_j), M, \mathcal{Q}) = 1\right] = 1$$

(ii) *correctness in linkability and exculpability*: for any pair of messages $M_1, M_2$, any rings $\mathcal{Q}_1$, $\mathcal{Q}_2$, any pair of indices $\pi_1$, $\pi_2$, and any pair of sets of private keys $\left\{x_{1,j}^*\right\}_{j=1}^{N_1}$ and $\left\{x_{2,j}^*\right\}_{j=1}^{N_2}$ such that $\sum_j x_{1,j}^* = X_{1,\pi_1} \in \mathcal{Q}_1$ and $\sum_j x_{2,j}^* = X_{2,\pi_2} \in \mathcal{Q}_2$, if

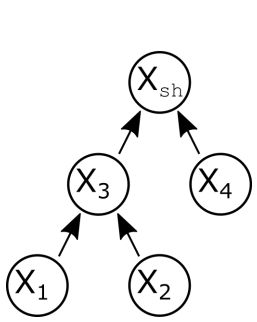
$$(\sigma_1, M_1^*) \leftarrow \texttt{Sign}(M_1^*, \mathcal{Q}_1, \pi_1, \left\{x_{1,j}^*\right\}_j)$$
$$(\sigma_2, M_2^*) \leftarrow \texttt{Sign}(M_2^*, \mathcal{Q}_2, \pi_2, \left\{x_{2,j}^*\right\}_j)$$

then

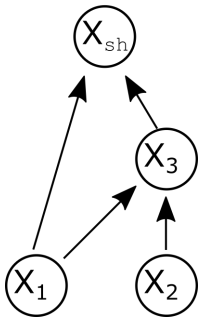
$$\mathbb{P}\left[\texttt{Link}((\sigma_1, M_1^*), (\sigma_2, M_2^*)) = 1 \mid X_{1,\pi_1} = X_{2,\pi_2}\right] = 1$$
$$\mathbb{P}\left[\texttt{Link}((\sigma_1, M_1^*), (\sigma_2, M_2^*)) = 1 \mid X_{1,\pi_1} \neq X_{2,\pi_2}\right] < \epsilon$$

for some $\epsilon$ negligible in a security parameter.

Note that if a scheme under this definition is correct then $(\texttt{KeyGen}^*, \texttt{Sign}^*, \texttt{Ver}^*, \texttt{Link}^*)$ is correct as a ring signature scheme (by setting $N = 1$). We present Example 4.5, an implementation in the random oracle model, after we discuss how the family history of keys is used to comptue signatures.




(a) Simple family history.




(b) A more complicated family history.

Figure 1: Two family history graphs of keys. Random signing data from a simple family history is computed by a simple sum taken over the root keys. Random signing data from more complicated family history must be summed over the paths from $X_{\mathtt{sh}}$ to root keys.

## 4.1 Family history of keys

We refer to the output from `Merge` as a *child* key and the input keys for `Merge` as *parent* keys. If one key is a parent of a parent, and so on, we say it is an *ancestor* key. If an ancestor key has no parents, we say it is a *root ancestor*. We say a key together with all of its ancestors constitutes a *family history*. Note that the parent-child relation seems to induce a natural directed acyclic graph on the family history of a key, where nodes are keys and directed edges are defined by child keys pointing to their parent keys. However, this construction is rather illusory: since keys are members of a finite group, if too many keys are in use, eventually the pigeon-hole principle kicks in and a cycle is created. In practice, we control the size of the key space with a security parameter to prevent this except with negligible probability as usual. We use the family history graph extensively, noting that this graph can be extracted from the transcripts of participants merging keys.

**Example 4.3.** Alice and Bob already have a 2-of-2 key and want to use it to make a 3-of-3 key with Charlene. Presume that Alice and Bob used keys $X_1, X_2$ to construct $X_3 = X_1^* + X_2^*$ for masked keys $X_1^*$, $X_2^*$. Charlene wants to use $X_4$ to obtain the shared key $X_{\mathtt{sh}} = X_4^* + X_3^*$ for some masked keys $X_3^*$, $X_4^*$. This leads to the family history directed acyclic graph depicted in Figure 1a. We apply the thresholdizing heuristic when the top key computes a signature. To decide on a random scalar $s$, each of $X_3$ and $X_4$ contribute some random data $s_3$ and $s_4$ such that $s = s_3 + s_4$. But $s_3$ comes from a merged key, so it, too, is a sum: keys $X_1$ and $X_2$ contribute $s_1, s_2$ such that $s_3 = s_1 + s_2$. Thus, $s = s_1 + s_2 + s_4$. That is to say, this family graph leads us to take a simple sum of random values for each root key.

This example, while somewhat illuminating, is deceptive: in general, keys can be re-used in merging processes, leading to many paths from $X_{\mathtt{sh}}$ to any root key. We provide Example 4.4.

**Example 4.4.** Alice and Bob wish to merge keys to create a 3-of-3 key where Alice must contribute twice and Bob must contribute once in the following way. Alice and Bob have keys $X_1, X_2$, merge keys to get a 2-of-2 key $X_3 = X_1^* + X_2^*$ for some random masked keys $X_1^*$ and $X_2^*$. Alice then merges $X_1$ with $X_3$ again by picking another masked key $X_1^{**}$ and, for $X_3$, collaborating with Bob to pick a random masked key $X_3^*$. The shared key $X_{\mathtt{sh}} = X_1^{**} + X_3^*$. We display this family history in Figure 1b.

Now to compute some $s$ for a signature, $X_1^{**}$ contributes some $s_1$ and $X_3^*$ contributes some $s_3$. But $X_3^*$ is a merged key so $s_3$ is a sum, $s_3 = s_1' + s_2$, where $s_1$ is selected by Alice for $X_1^*$ and $s_2$ is selected by Bob

for $X_2^*$. Then $s = s_1 + s_3 = s_1 + s_1' + s_2$.

Example 4.4 demonstrates that a single piece of random information selected by each root key is not sufficient to contribute to a signature in general: for each root key, for each path from the shared key to the root key, a piece of random information must be contributed, but we still take the simple sum.

## 4.2 Implementation

**Example 4.5.** Let $(\texttt{KeyGen}^*, \texttt{Sign}^*, \texttt{Ver}^*, \texttt{Link}^*)$ be the Back LSAG ring signature as described in Section 3.1, let $(\texttt{KeyGen}', \texttt{Enc}', \texttt{Dec}')$ be a symmetric-key encryption scheme, and let $(\texttt{KeyGen}'', \texttt{Sign}'', \texttt{Ver}'')$ be a signature scheme. Assume all three have the key space $\mathcal{K} = \mathbb{Z}_{\mathfrak{q}} \times G$. Let $(\texttt{Com}, \texttt{Open})$ be the Pedersen commitment scheme previously described. Consider the pentuple $(\texttt{KeyGen}, \texttt{Merge}, \texttt{Sign}, \texttt{Ver}, \texttt{Link})$ with the following definitions.

1. $\texttt{KeyGen}$ is identical to $\texttt{KeyGen}^*$, which outputs a key pair $(x, X) \in \mathcal{K}$ chosen uniformly at random.
2. $\texttt{Merge}$ takes as input the set of public keys $\{X_1, \ldots, X_N\}$ and takes as input a set of pairs $\{(x_j, m_j)\}_{j=1}^N$ such that $X_j = x_j G$ for each $1 \leq j \leq N$. The following is executed:
   (a) Each contributor $X_j$ computes $x_j^* := H_s(x_j, m_j)$ and $X_j^* := x_j^* G$. For each other contributor $X_{j'}$, $X_j$ also computes a shared secret encryption key $y_{j,j'} = H_s(x_j X_{j'})$, computes the ciphertext $\mathfrak{C}_{j,j'} \leftarrow \texttt{Enc}'(X_j^*, y_{j,j'})$ and the signature $\sigma_{j,j'} \leftarrow \texttt{Sign}''(\mathfrak{C}_{j,j'}, x_j)$, and sends $(\mathfrak{C}_{j,j'}, \sigma_{j,j'})$ to each $X_{j'}$.
   (b) Each contributor $X_j$ verifies they received $N$ ciphertexts and signatures, verifies each signature $\sigma_{j,j'}$ is a valid signature on each $C_{j,j'}$ with key $X_{j'}$. If any signature is invalid, $X_j$ outputs $\perp$ and terminates. Otherwise, $X_j$ uses $y_{j,j'}$ to decrypt $\mathfrak{C}_{j,j'}$ to obtain $X_j^*$. Each contributor $X_j$ now computes the shared public key $X_{\texttt{sh}} = \sum_j X_j^*$.
3. $\texttt{Sign}$ takes as input a message $M \in \{0,1\}^*$, a ring of public keys $\mathcal{Q} = \{Y_1, \ldots, Y_R\}$, a secret index $1 \leq \pi \leq R$, and a set of private keys $\{x_j^*\}$ (when executed honestly, $Y_\pi = \sum_j x_j^* G$). The following is executed:
   (a) Each honest contributor $X_j^*$ computes each $H_i := H_p(Y_i)$ for $1 \leq i \leq R$ and their share of the key image, $J_j := x_j^* H_\pi$.
   (b) Each honest contributor $X_j^*$ enumerates all paths from $Y_\pi$ that terminate at $X_j^*$ in the family history of $Y_\pi$; say there are $\eta_j$ of these. For each such path, $X_j^*$ selects a random scalar $\{u_{\pi,j,i}\}_{i=1}^{\eta_j}$ for the $\pi^{th}$ ring member, selects random scalars for the other ring members $\{s_{\ell,j,i}\}_{\ell=1,i=1}^{\ell=R,\eta_j}$, and computes the points $U_{\pi,j,i} := u_{\pi,j,i} G$ and $V_{\pi,j,i} := u_{\pi,j,i} H_\pi$.
   (c) Each contributor $X_j^*$ computes, for each other $X_{j'}^*$, the shared secret encryption key $y_{j,j'}^* = H_s(x_j^* X_{j'}^*)$, the ciphertext $\mathfrak{C}_j \leftarrow \texttt{Enc}'(\{J_j, U_{\pi,j,i}, V_{\pi,j,i}, s_{\pi+1,j,i}, \ldots, s_{\pi-1,j,i}, s_{\pi,j,i}\}_{i=1}^{\eta_j}, y_{j,j'}^*)$, and signs the ciphertext $\sigma_{1,j,j'} \leftarrow \texttt{Sign}''(\mathfrak{C}_j, x_j^*)$. For each other $X_{j'}^*$, each contributor $X_j^*$ sends each $(\mathfrak{C}_j, \sigma_{j,j'})$ to $X_{j'}^*$.
   (d) Each contributor $X_j^*$ verifies they received $N$ ciphertexts and signatures, and verifies each signature $\sigma_{j,j'}$ is a valid signature on each $\mathfrak{C}_{j'}$ with key $X_{j'}^*$. If any signatures are invalid, $X_j^*$ outputs $\perp$ and terminates. Otherwise, $X_j^*$ decrypts $\mathfrak{C}_{j'}$ with $y_{j,j'}$.

(e) Each contributor $X_j^*$ computes the following points

$$\begin{aligned} J &:= \sum_{j'} J_{j'} \\ U_\pi &:= \sum_{j'} \sum_i U_{\pi,j',i} \\ V_\pi &:= \sum_{j'} \sum_i V_{\pi,j',i}. \end{aligned}$$

(f) Each contributor $X_j^*$ can now assemble the modified message $M^* = (M, J, \mathcal{Q})$, and the following scalars

$$\begin{aligned} s_\ell &:= \sum_{j'} \sum_i s_{\ell,j',i} \text{ for } \ell \neq \pi \\ c_{\pi+1} &:= H_s(M^*, U_\pi, V_\pi) \\ c_{\pi+2} &:= H_s(M^*, s_{\pi+1}G + c_{\pi+1}Y_{\pi+1}, s_{\pi+1}H_{\pi+1} + c_{\pi+1}J) \\ &\vdots \\ c_{\pi-1} &:= H_s(M^*, s_{\pi-2}G + c_{\pi-2}Y_{\pi-2}, s_{\pi-2}H_{\pi-2} + c_{\pi-2}J) \\ c_\pi &:= H_s(M^*, s_{\pi-1}G + c_{\pi-1}Y_{\pi-1}, s_{\pi-1}H_{\pi-1} + c_{\pi-1}J) \end{aligned}$$

Now each contributor has the modified message $M^*$, each $c_\ell$ for $1 \leq \ell \leq R$, and each $s_\ell$ for $\ell \neq \pi$. This concludes the first stage of signing.

(g) Each contributor $X_j^*$ computes the value $s_{\pi,j,i} = u_{\pi,j,i} - c_\pi x_j^*$ for the $i^{th}$ path from $Y_\pi$ to $X_j^*$. Each contributor $X_j^*$ then computes $\mathfrak{D}_{j,j'} \leftarrow \texttt{Enc}'(\{s_{\pi,j,i}\}_i, y_{j,j'}^*)$, computes $\sigma_{2,j,j'} \leftarrow \texttt{Sign}'(\mathfrak{D}_{j,j'}, x_j^*)$, and sends $(\mathfrak{D}_{j,j'}, \sigma_{2,j,j'})$ to $X_{j'}^*$.

(h) Each contributor $X_j^*$ verifies they received $N$ ciphertexts and signatures, and verifies each signature $\sigma_{2,j,j'}$ is a valid signature on each $\mathfrak{D}_{j,j'}$ with key $X_{j'}^*$. If any signatures are invalid, $X_j^*$ outputs $\perp$ and terminates. Otherwise, $X_j^*$ decrypts $\mathfrak{D}_{j,j'}$ with $y_{j,j'}^*$. Each contributor may now compute $s_\pi = \sum_{j'} \sum_i s_{\pi,j',i}$ and publish the modified message $M^* = (M, J, \mathcal{Q})$ and signature $\sigma = (c_1, s_1, \ldots, s_R)$.

4. `Ver` is identical to a usual LSAG verification $\texttt{Ver}^*$, which takes as input a ring signature and modified message $(\sigma, M^*)$ where $\sigma = (c_1, s_1, \ldots, s_R)$ and $M^* = (M, J, \mathcal{Q})$. The verifier parses $M^*$, parses $\mathcal{Q}$, computes each $H_\ell = H_p(Y_\ell)$, and computes

$$\begin{aligned} c_2 &:= H_s(M^*, s_1G + c_1Y_1, s_1H_1 + c_1J) \\ c_3 &:= H_s(M^*, s_2G + c_2Y_2, s_2H_2 + c_2J) \\ &\vdots \\ c_R &:= H_s(M^*, s_{R-1}G + c_{R-1}Y_{R-1}, s_{R-1}H_{R-1} + c_{R-1}J) \\ c_1^* &:= H_s(M^*, s_RG + c_RY_R, s_RH_R + c_RJ) \end{aligned}$$

The verifier and outputs 1 if $c_1 = c_1^*$ and 0 otherwise.

5. `Link` is identical to $\texttt{Link}^*$, which takes as input two ring signatures and modified messages $(\sigma_1, M_1^*)$ and $(\sigma_2, M_2^*)$, and parses $M_i^*$ to get each key image $J_i$. `Link` outputs 1 if $J_1 = J_2$ and 0 otherwise.

# 5 Security of implementation

Desirable properties for $N$-of-$N$ aggregate-key ring multisignature schemes include *privacy* (`Merge` should not allow malicious participants to trick honest participants into revealing any information about their private keys $x_i$) and *indistinguishability* (it should be hard to determine if any given key $X$ is a merged key or not). A weaker property than indistinguishability is *hiding*, where it is hard for an adversary to tell if an arbitrary key $Y$ was used in the computation of another arbitrary key $X$; if the scheme is indistinguishable, then an adversary cannot discern even if a key is a shared key or not, let alone determine which keys contributed, so indistinguishability implies hiding. A common sticking point for multisignature scheme is *resistance to key cancellation*, where a malicious participant cannot both cancel participation of other contributors and successfully collaborate honestly. Of course, we are also interested in the properties of *unforgeability*, *signer ambiguity*, *linkability*, and *exculpability* from the underlying LSAG scheme.

## 5.1 Hardness assumptions

We use the random oracle model throughout. Many of our security properties reduce to a hardness problem on the following spectrum (from weak assumptions to strong ones): discrete logarithm hardness, one-more discrete logarithm hardness, computational Diffie-Hellman, decisional Diffie-Hellman.

**Definition 5.1** (OMDL Hardness)**.** Let $\mathcal{DL}(-)$ be an oracle taking as input a public key $X$ and responding with the discrete logarithm $x$. We say a PPT algorithm $\mathcal{A}$ $(q, t, \epsilon)$-solves the one-more discrete logarithm problem if $\mathcal{A}$ takes as input a set of $q+1$ public keys $X_1, \ldots, X_{q+1}$, runs in time at most $t$, queries $\mathcal{DL}$ at most $q$ times, and, with probability at least $\epsilon$ taken over all choices of $X_i$ and all random coins of $\mathcal{A}$, outputs a set of scalars $x_1, \ldots, x_{q+1}$ such that each $X_i = x_i G$.

**Definition 5.2** (CDH Hardness)**.** We say a PPT algorithm $\mathcal{A}$ $(t, \epsilon)$-solves the computational Diffie-Hellman problem if $\mathcal{A}$ takes as input a triple of points $(G, xG, yG)$, runs in time at most $t$ and, with probability at least $\epsilon$ taken over all choices of $(xG, yG)$ and all random coins of $\mathcal{A}$, outputs a point $zG$ such that $z = xy$.

**Definition 5.3** (DDH Hardness)**.** We say a PPT algorithm $\mathcal{A}$ $(t, \epsilon)$-solves the computational Diffie-Hellman problem if $\mathcal{A}$ takes as input a quadruple of points $(xG, yG, zG)$, runs in time at most $t$ and, with probability at least $\epsilon$ taken over all choices of $(xG, yG, zG)$ and all random coins of $\mathcal{A}$, outputs a bit $b \in \{0, 1\}$ indicating whether $z = xy$.

Obviously an adversary $\mathcal{A}$ who can $(t, \epsilon)$-solve the CDH problem can be used to $(t, \epsilon)$-solve the DDH problem by allowing a PPT algorithm $\mathcal{A}'$ to execute $\mathcal{A}$ as a subroutine. $\mathcal{A}'$ takes as input $(xG, yG, zG)$, sends $(G, xG, yG)$ to $\mathcal{A}$, and with probability at least $\epsilon$ gets in return $xyG$. It is then trivial to check if $zG = xyG$.

**Lemma 5.4.** The OMDL hardness assumption is weaker than the CDH hardness assumption.

*Proof.* Assume $\mathcal{A}$ can $(q, t, \epsilon)$-solve the OMDL problem. We build a PPT algorithm $\mathcal{A}'$ that executes $\mathcal{A}$ a subroutine and solves the CDH problem with probability at least $\epsilon$ in time at most. $\mathcal{A}'$ takes as input some $(G, aG, bG)$. $\mathcal{A}'$ constructs a list $\{X_1, \ldots, X_{q+1}\}$ such that $X_1 = aG$ and each $X_i$ is a random point from $\mathbb{G}$. Then with probability at least $\epsilon$, within time $t$, $\mathcal{A}$ outputs a list $\{x_1, \ldots, x_{q+1}\}$ such that each $X_i = x_i G$ and such that $x_1 G = aG$ does not appear in the $\mathcal{DL}$ transcript. Now $\mathcal{A}$ can compute $a \cdot (bG)$ with the cost of a single exponentiation, which is a constant $t'$. $\mathcal{A}'$ outputs $a \cdot (bG)$. Thus $\mathcal{A}'$ can $(t + t', \epsilon)$-solve the CDH problem, where $t + t' = O(t)$. □

## 5.2 Definitions and Properties

We first define a key cancellation game and show why adversaries have negligible advantage in this game. We next demonstrate unforgeability. Lastly, we discuss properties such as signer ambiguity, key aggregation with privacy and indistinguishability, linkability, and exculpability.

### 5.2.1 Key cancellation

We show that if discrete logarithms are hard to compute then no PPT adversary $\mathcal{A}$ can both cancel honest keys in `Merge` and know the discrete logarithm of the adversarial public keys used in `Merge`. Due to this, no PPT adversary $\mathcal{A}$ can both construct signatures to pass the first authentication stage of `Sign` and cancel keys.

Key cancellation means executing `Merge` with honest participants such that some of the honest participants could be excluded from signing. A universe of honest keys $\mathcal{U}_h$ is generated from `KeyGen` and the public keys are given to $\mathcal{A}$. $\mathcal{A}$ is granted $\mathcal{DL}$ access. The adversary wishes to find a chain of subsets from the universe of honest keys $\emptyset \neq K_1^h \subset K_2^h \subset \mathcal{U}_h$ and two sets of public keys $\mathbb{G}$, say $K_1^A$, $K_2^A \subset \mathbb{G}$, such that merging $K_1^A$ with $K_1^h$ provides the same result as merging $K_2^A$ with $K_2^h$ (and such that $K_1^h$ and $K_2^h$ do not appear in $\mathcal{T}_{A^{\mathcal{DL}}}$). Since $K_1^h \subset K_2^h$, this reduces the number of honest contributors required for honest signature collaboration. To cancel keys, then, $\mathcal{A}$ receives the masked honest keys for each $X \in K_i^h$, some $X^*$, and $\mathcal{A}$ selects masked keys for each $X \in K_i^A$, $\mathcal{A}$ picks some $X^*$ such that

$$X_{\mathtt{sh}} = \sum_{X \in K_1^h} X^* + \sum_{X \in K_1^A} X^* = \sum_{X \in K_2^h} X^* + \sum_{X \in K_2^A} X^*$$

The task of $\mathcal{A}$, then, is to select points such that

$$\sum_{X \in K_1^h} X^* - \sum_{X \in K_2^h} X^* = \sum_{X \in K_2^A} X^* - \sum_{X \in K_1^A} X^*$$

Note that since the masked keys on the left are honestly selected, the difference of these sums is a group element selected uniformly at random. Solutions are very easy for $\mathcal{A}$ to produce. For example, if $K_1^h = \{X_1\}$ and $K_2^h = \{X_2\}$, then $\mathcal{A}$ can select, for $X \in K_1^A$, $X^* := Z$ for any $Z$ just so long as, for $X \in K_2^A$, $X^* := X_2^* - X_1^* - Z$. However, if $\mathcal{A}$ can present a solution in time $t$ with probability at least $\epsilon$ such that $\mathcal{A}$ knows the discrete logarithms of each masked key $X^*$ for some $X \in K_1^A \cup K_2^A$, then $\mathcal{A}$ can easily compute the discrete logarithm of a sum of the honestly selected masked keys, violating the discrete logarithm hardness assumption.

### 5.2.2 Indistinguishably private key aggregation

Since an adversary inside a coalition will clearly know they participated in an aggregate key, we only consider adversaries outside of a coalition. Note that we use ECDH secret sharing to let coalition members compute shared encryption keys, the security of which relies on the CDH hardness assumption. If the CDH hardness assumption fails, an adversary can find $x_j X_{j'}$ for many pairs of public keys $X_j, X_{j'}$, compute the hashes $H_s(x_j X_{j'})$, and collect the encryption keys $y_{j,j'}$. This allows the adversary to overhear the public masked keys $X_j^*$ being shared in `Merge` and gain an advantage in discerning if a random key $X$ is shared or not.

The CDH hardness assumption together with the random oracle model are sufficient for our scheme to have indistinguishable key aggregation: CDH hardness ensures the summands $X_j^*$ are not leaked in `Merge` if the encryption sub-scheme is secure and each $X^* = H_s(x, m)G$ is statistically indistinguishable from a

uniformly random selection generated by `KeyGen`. Hence, when `Merge` is executed by honest parties under the CDH hardness assumption in the random oracle model, the distribution of the output is uniform.

The discrete logarithm hardness assumption together with the random oracle model are sufficient for our scheme to have private key aggregation: an adversary who breaks the CDH hardness assumption to compute $y_{j,j'}$ to overhear the masked keys $X_j^* = H_s(x_j, m_j)G$ is then tasked with computing the discrete logarithm of a point drawn from uniform distribution.

#### 5.2.3 Signer Ambiguity

In [2], signer ambiguity for ring signatures was considered with adversarially chosen keys. Since an adversary inside a coalition will clearly know they participated in a signature, we only consider adversaries outside all signing coalitions. Violating signer ambiguity also violates indistinguishability. Indeed, Example 4.5 reduces to LSAG signatures when $N = 1$, which is signer-ambiguous. So violating signer ambiguity allows an adversary to identify that the true signer must have been a merged key, giving the adversary an advantage in identifying merged keys, which violates indistinguishability.

#### 5.2.4 Linkability and Exculpability

Linkability and exculpability are simultaneously enforced by construction. Indeed, for a signature to be valid, each verification equation must be satisfied:

$$c_{\ell+1} = H_s(M, s_\ell G + c_\ell H_\ell, s_\ell H_\ell + c_\ell J)$$

where each $H_\ell = H_p(Y_\ell)$. That is to say, the key image $J$, except with negligible probability, must be computed with basepoint $H_\pi$ where $\pi$ is the (secret) true signing index, and therefore there is only one possible key image per ring member per valid signature.

#### 5.2.5 Unforgeability

For existential unforgeability, it is usual to model the adversary's ability to obtain ostensibly honest signatures before attempting a forgery by granting the adversary signing oracle access. In the same spirit, for the threshold multisignature setting, we model the adversary's ability to both merge keys with honest parties with a merge oracle $\mathcal{MO}$ and to collaborate with honest parties to compute multisignatures on arbitrary messages with an interactive signing oracle $\mathcal{SO}$. Since we are in the random oracle model, the oracles $H_s$ and $H_p$ are simulated as usual: these oracles use internal tables to maintain consistency between queries and flip coins randomly to determine digests for new messages.

However, since the adversary can query oracles in any order, it's possible that $\mathcal{A}$ has queried $H_s$ for some $c_{\ell+1} = H_s(M^*, s_\ell G + c_\ell Y_\ell, s_\ell H_\ell + c_\ell J)$ before interacting with the signing oracle. Hence, the signing oracle cannot always select all random signature data by flipping coins and "backpatching" or "programming" $H_s$ and $H_p$ appropriately in the usual method of simulation proofs. We will see that the signing oracle must be granted access to $\mathcal{DL}$ in order to perform its job.

**The Merge Oracle**. $\mathcal{MO}$ takes as input a pair of sets of public keys $K^A \subset \mathbb{G}$ and $K^h \subset \mathtt{Pub}(\mathcal{U}_h)$, and sends a set of masked public keys $\{Y^*\}$ to $\mathcal{A}$. $\mathcal{A}$ computes $X_{\mathtt{sh}}$ by selecting masked keys $\{X_i^*\}_{i=1}^k$ and computing the grand sum of all the masked keys $X_{\mathtt{sh}} = X_1^* + \cdots + X_k^* + Y_{k+1}^* + \cdots + Y_N^*$. $\mathcal{A}$ sends $X_{\mathtt{sh}}$ back to $\mathcal{MO}$. To maintain consistency with past queries, $\mathcal{MO}$ keeps two internal tables. In the first table, $T_M$, $\mathcal{MO}$ picks a random masked key $Y^*$ contributed by honest key $Y \in K^h$ in coalition $(K^A, K^h)$ by flipping coins and setting $T_M[(Y, K^A, K^h)] = Y^*$. In the second table, $T_S$, $\mathcal{MO}$ tracks coalitions $(K^A, K^h)$ for given shared keys by setting $T_S[X_{\mathtt{sh}}] = (K^A, K^h)$. Note that the family history is computable from $T_S$.

A query $\mathcal{MO}(K^A, K^h)$ with $K^A = (X_1, \ldots, X_k)$ and $K^h = (Y_{k+1}, \ldots, Y_N)$ in $\mathcal{T}_{A^{\mathcal{MO}}}$ looks to $\mathcal{A}$ like the following diagram.

$$\mathcal{A} \begin{pmatrix} (K^A, K^h) \\ \xrightarrow{\quad} \\ \{Y^*\} \\ \xleftarrow{\quad} \\ X_{\mathtt{sh}} \\ \xrightarrow{\quad} \end{pmatrix} \mathcal{MO}$$

Note that this oracle could be extended to merge all honest keys, $\mathcal{MO}(K^h)$, but this is statistically indistinguishable from selecting a new group element at random (and since signing will still have to be simulated by $\mathcal{SO}$ with $\mathcal{DL}$, it is unnecessary to consider this edge case).

Also note that we can take de-randomization into account this way: $\mathcal{A}$ could take the values $\{Y^*\}$ and run with them, computing many ostensibly legitimate shared signatures from the same masked keys (as if the honest user reused their random data). In these scenarios, $\mathcal{MO}$ has at most one $X_{\mathtt{sh}}$ recorded for each $(K^A, K^h)$ in $T_S$. Indeed, the entire family history can be computed from $T_S$; success conditions in the forgery game shall hinge on the existence of a family history for each ring member in $T_S$.

**The Signing Oracle**. The signing oracle $\mathcal{SO}$ is queried with $(M, \mathcal{Q}, \pi)$ where $M$ is a message, $\mathcal{Q} = \{Y_1, \ldots, Y_R\}$, and $1 \leq \pi \leq R$ is an index. Depending on the characteristics of $Y_\pi$, $\mathcal{SO}$ interacts with $\mathcal{A}$ throughout the query. In the end, $\mathcal{SO}$ either outputs a signature and modified message $(\sigma, M^*)$ or outputs enough information for $\mathcal{A}$ to complete a multisignature-tag pair. The oracle outputs $\perp$ if no root ancestor of $Y_\pi$ is from $\mathcal{U}_h$. So we assume without loss of generality that $Y_\pi \in \mathcal{U}_h$ or $Y_\pi$ has at least one honest ancestor in $\mathcal{U}_h$.

If all root ancestors of $Y_\pi$ are from $\mathcal{U}_h$, $\mathcal{SO}$ queries $\mathcal{DL}$ to get $y_\pi \leftarrow \mathcal{DL}(Y_\pi)$ and stores the result in a table $T_D[Y_\pi] = y_\pi$. $\mathcal{SO}$ computes a usual LSAG ring signature with $\mathtt{Sign}^*$. Otherwise, if $Y_\pi$ has a family history in $T_S$, then $Y_\pi$ has at least one honest ancestor and at least one adversarial root ancestor. In this case, $\mathcal{SO}$ does the following:

1. $\mathcal{SO}$ queries $\mathcal{DL}$ with each honest root ancestor $Z \in \mathcal{U}_h$. Results are stored in a table $T_D[Z] \leftarrow \mathcal{DL}(z)$ to maintain consistency with later queries and to ensure $\mathcal{DL}$ is never queried with the same input twice.

2. For each honest root ancestor $Z$, $\mathcal{SO}$ computes the partial key image $zH_p(Y_\pi)$ and enumerates all paths from $Y_\pi$ to $Z$ in the family history of $Y_\pi$.

3. For the $j^th$ honest root ancestor $Z_j$, $i^{th}$ path from $Y_\pi$ to $Z_j$, $\mathcal{SO}$ selects a random scalar $u_{j,i}$, computes the points $U_{\pi,j,i} := u_{j,i}G$ and $V_{\pi,j,i} := u_{j,i}H_\pi$, selects random scalars $s_{\pi+1,j,i}, s_{\pi+2,j,i}, \ldots, s_{\pi-1,j,i}$, assembles the set
$$\Lambda_1 := \{J_j, (U_{\pi,j,i}, V_{\pi,j,i}, s_{\pi+1,j,i}, s_{\pi+2,j,i}, \ldots, s_{\pi-1,j,i})_i\}_j$$
and sends $\Lambda_1$ to $\mathcal{A}$.

4. For each adversarial root ancestor $Z$, $\mathcal{A}$ enumerates all paths from $Y_\pi$ to $Z$ in the family history of $Y_\pi$.

5. For the $j^{th}$ adversarial root ancestor $Z_j$, for the $i^{th}$ path from $Y_\pi$ to $Z_j$, $\mathcal{A}$ selects a scalar $u_{j,i}$, computes the points $U_{\pi,j,i} := u_{j,i}G$ and $V_{j,i} := u_{j,i}H_\pi$, computes the partial key image $J_j := z_jH_\pi$, and selects the scalars $s_{\pi+1,j,i}, s_{\pi+2,j,i}, \ldots, s_{\pi-1,j,i}$. $\mathcal{A}$ sends the set
$$\Lambda_2 := \{J_j, (U_{\pi,j,i}, V_{\pi,j,i}, s_{\pi+1,j,i}, s_{\pi+2,j,i}, \ldots, s_{\pi-1,j,i})_i\}_j$$
to $\mathcal{SO}$.

6. $\mathcal{SO}$ computes the sums $J = \sum J_j$, $U_\pi = \sum_{j,i} U_{\pi,j,i}$, $V_\pi = \sum_{j,i}$, and $s_\ell = \sum_{j,i} s_{\ell,j,i}$. $\mathcal{SO}$ then queries $H_s$ to compute $c_{\pi+1} := H_s(M, U_\pi, V_\pi)$ and each $c_{\ell+1} = H_s(M, s_\ell G + c_\ell Y_\ell, s_\ell H_\ell + c_\ell J)$ for $\ell \neq \pi$.

7. For the $j^{th}$ honest root ancestor $Z_j$, for the $i^{th}$ path from $Y_\pi$ to $Z_j$, $\mathcal{SO}$ computes each $s_{\ell,j,i} := u_{\pi,j,i} - c_\pi z_j$ and sends the set $\Lambda_3 := \{s_{\ell,j,i}\}_{j,i}$ to $\mathcal{A}$.

Now $\mathcal{A}$ has enough information to complete the signature by computing $J = \sum J_j$, $U_\pi = \sum_{j,i} U_{\pi,j,i}$, $V_\pi = \sum_{j,i}$, $s_\ell = \sum_{j,i} s_{\ell,j,i}$, each $c_{\ell+1}$, and each $s_\pi = \sum_{j,i} s_{\pi,j,i}$, completing the signature. A query in $\mathcal{T}_{A^{\mathcal{SO}}}$ looks something like this:

$$\mathcal{A}\begin{pmatrix} \xrightarrow{(M,\mathcal{Q},i)} \\ \xleftarrow{\Lambda_1} \\ \xrightarrow{\Lambda_2} \\ \xleftarrow{\Lambda_3} \end{pmatrix}\mathcal{SO}$$

We begin the forgery game as usual by giving $\mathcal{A}$ a set of public keys $\mathcal{U}_h$. We also grant the adversary corruption oracle access, $\mathcal{CO} : \texttt{Pub}(\mathcal{U}_h) \to \texttt{Priv}(\mathcal{U}_h)$. This oracle is given all the private keys of all keys from $\mathcal{U}_h$ at the beginning of the forgery game and does not require discrete logarithm oracle access. We grant the adversary merge oracle $\mathcal{MO}$ access, which also does not require discrete logarithm oracle access and allows the adversary to construct coalition over which it has subthreshold control. We grant the adversary signing oracle $\mathcal{SO}$ access, which does require $\mathcal{DL}$ access.

We task $\mathcal{A}$ with providing a valid message $M$, ring $\mathcal{Q}$, and valid signature-tag pair $(\sigma, J)$ such that (i) every key in $\mathcal{Q}$ has at least one honest root ancestor[§] from $\mathcal{U}_h$ and (ii) $\mathcal{SO}$ is not queried with $(M, \mathcal{Q})$ for any index, (iii) $\mathcal{CO}$ and $\mathcal{DL}$ are not queried with any key in $\mathcal{Q}$, and (iv) $\mathcal{CO}$ and $\mathcal{DL}$ are not queried with any honest root ancestor of any key in $\mathcal{Q}$.

**Definition 5.5** (Existential Unforgeability vs. Chosen-Message Attack with Insider Corruption and Adversarial Cooperation)**.** Let $g(-)$ be a polynomial. Consider the following game:

1. A set of keys is generated $\mathcal{U}_h \leftarrow \texttt{KeyGen}$ such that $|\mathcal{U}_h| \geq 2^{g(\lambda)}$. $\texttt{Pub}(\mathcal{U}_h)$ is given to $\mathcal{A}$.

2. $\mathcal{A}$ is granted $\mathcal{MO}$, $\mathcal{SO}$, and $\mathcal{CO}$ access.

3. $\mathcal{A}$ outputs a message $M$, a ring $\mathcal{Q}$, and a ring signature-tag pair $(\sigma, J)$. $\mathcal{A}$ succeeds if

   (a) $\texttt{Ver}(M, \mathcal{Q}, (\sigma, J)) = 1$, and

   (b) $(M, \mathcal{Q}, i)$ does not appear in $\mathcal{T}_{A^{\mathcal{SO}}}$ for any $1 \leq i \leq R = |\mathcal{Q}|$, and

   (c) for each $Y \in \mathcal{Q}$, $Y \in \mathcal{U}_h$ or $Y$ has a root ancestor from $\mathcal{U}_h$ in $\mathcal{T}_{A^{\mathcal{MO}}}$,

   (d) $\mathcal{DL}$ and $\mathcal{CO}$ have not been queried with any $Y \in \mathcal{Q}$, and

   (e) $\mathcal{DL}$ and $\mathcal{CO}$ have not been queried with any ancestor of any $Y \in \mathcal{Q}$.

If every PPT adversary has at most a negligible advantage at this game, we say the signature scheme is *existentially unforgeable against subthreshold corruption* or EUF-ST. We say an adversary that can succeed at this game in time $t$ with probability at least $\epsilon$ is a $(t, \epsilon)$-forger.

Note here that we do not allow the adversary to corrupt honest root keys, but this game does describe adversaries who control all but one honest key in the family history of every key in a ring. We call this subthreshold adversarial control.

[§]Hence every key in $\mathcal{Q}$ is in $\mathcal{U}_h$ or has an entry in $T_S$: no participants re-used random data while merging keys.

### 5.2.6 Rewind on success

To prove Example 4.5 is unforgeable under subthreshold corruption, we briefly describe the rewind-on success forking lemma first used in [6], made more general in [8], and based on the forking lemma presented in [1]. We let $\mathbb{S}$ be an arbitrary set of auxiliary inputs. Let $\mathcal{A}$ be a randomized algorithm that takes some input $\mathtt{inp}$, some random oracle queries $\underline{h} = \{h_i\}_i$, and some auxiliary inputs $\underline{\Gamma} = \{\Gamma_j\}_j \subseteq \mathbb{S}$, and outputs some $(i, j, \mathtt{out}) \leftarrow \mathcal{A}(\mathtt{inp}, \underline{h}, \underline{\Gamma})$ or a failure $\perp \leftarrow \mathcal{A}(\mathtt{inp}, \underline{h}, \underline{\Gamma})$. We denote the random coins of $\mathcal{A}$ as $\rho$ (we do not detail the distribution of $\rho$). We denote $\mathcal{ACC}(\mathcal{A})$ as the *acceptance probability* for $\mathcal{A}$:

$$\mathcal{ACC}(\mathcal{A}) = \mathbb{P}\left[\perp \not\leftarrow \mathcal{A}(\mathtt{inp}, \underline{h}, \underline{\Gamma})\right]$$

where this probability is computed over the joint distribution of $(\mathtt{inp}, \underline{h}, \underline{\Gamma}, \rho)$. We construct a second algorithm $\mathtt{Fork}^A$ from $\mathcal{A}$.

**input :** $\mathtt{inp}$, $\Gamma^{(1)} = (\Gamma_1^{(1)}, \ldots, \Gamma_m^{(1)})$, $\Gamma' = (\Gamma'_1, \ldots, \Gamma'_m)$

$\rho \leftarrow \mathtt{Coins}(\mathcal{A})$
$\underline{h}^{(1)} \leftarrow (\{0,1\}^\ell)^q$
$\alpha \leftarrow \mathcal{A}(\mathtt{inp}, \underline{h}^{(1)}, \Gamma^{(1)}; \rho)$
**if** $\alpha \neq \perp$**:**
  $(i, j, \mathtt{out}) \leftarrow \mathtt{parse}(\alpha)$
  $\underline{h}' \leftarrow (\{0,1\}^\ell)^q$
  $\underline{h}^{(2)} := (h_1^{(1)}, \ldots, h_{i-1}^{(1)}, h'_i, h'_{i+1}, \ldots, h'_q)$
  $\Gamma^{(2)} := (\Gamma_1^{(1)}, \ldots, \Gamma_j^{(1)}, \Gamma'_{j+1}, \ldots, \Gamma'_m)$
  $\alpha' \leftarrow \mathcal{A}(\mathtt{inp}, \underline{h}^{(2)}, \Gamma^{(2)}; \rho)$
  **if** $\alpha' \neq \perp$**:**
    $(i', j', \mathtt{out}') \leftarrow \mathtt{parse}(\alpha')$
    **if** $i' = i$ *and* $h_i^{(1)} \neq h_i^{(2)}$**:** Return $(i, \mathtt{out}, \mathtt{out}')$;
    **else:** Return $\perp$;
  **else:**
    Return $\perp$
**else:**
  Return $\perp$

**Algorithm 1:** The algorithm $\mathtt{Fork}^{\mathcal{A}}$, a general rewind-on-sucess forking algorithm.

$\mathtt{Fork}^A$ takes as input $\mathtt{input}$ and two subsets $\underline{\Gamma}^{(1)}, \underline{\Gamma}' \subseteq S^m$. $\mathtt{Fork}^A$ chooses $\underline{h}^{(1)} \in (\{0,1\}^\ell)^q$ at random, runs $\mathcal{A}$ once as a sub-algorithm with $(\mathtt{input}, \underline{h}^{(1)}, \underline{\Gamma}^{(1)})$. If this produces $\perp$, $\mathtt{Fork}^A$ immediately outputs $\perp$ and terminates. Otherwise, $\mathtt{Fork}^A$ receives from $\mathcal{A}$ some $(i, j, \mathtt{out})$. $\mathtt{Fork}^A$ picks a new $\underline{h}' \in (\{0,1\}^\ell)^q$ at random, and assembles

$$\underline{h}^{(2)} := (h_1^{(1)}, \ldots, h_{i-1}^{(1)}, h'_i, h'_{i+1}, \ldots, h'_q)$$
$$\Gamma^{(2)} := (\Gamma_1^{(1)}, \ldots, \Gamma_j^{(1)}, \Gamma'_{j+1}, \ldots, \Gamma'_m)$$

That is to say, $\mathtt{Fork}^A$ rewinds upon an $(i, j, \mathtt{out})$-success, preserving the first $i-1$ random oracle queries made by $\mathcal{A}$, and preserving the first $j$ auxiliary inputs. $\mathtt{Fork}^A$ then executes $\mathcal{A}$ with $(\mathtt{inp}, \underline{h}^{(2)}, \underline{\Gamma}^{(2)})$. If the second instantiation of $\mathcal{A}$ returns $\perp$, $\mathtt{Fork}^A$ outputs $\perp$ and terminates. Otherwise, $\mathcal{A}$ outputs some $(i', j', \mathtt{out}')$ and $\mathtt{Fork}^A$ has two $\mathcal{A}$-transcripts with these outputs matching in their first $i-1$ random oracle queries and the first $j$ auxiliary inputs.

If $i \neq i'$, $\texttt{Fork}^A$ outputs $\perp$. $\texttt{Fork}^A$ checks that the $i^{th}$ random oracle queries $h_i^{(1)} \neq h_i^{(2)}$ differ in each transcript. If not, $\texttt{Fork}^A$ outputs $\perp$. Otherwise, $\texttt{Fork}^A$ outputs the rewind index $i$ and both outputs $\texttt{out}$ and $\texttt{out}'$. This is summarized in [8] (c.f. also Figure 1 from that paper). So defined, [8] also proves the following general forking lemma; we take the proofs of these lemmata for granted.

**Lemma 5.6.** Let $\mathcal{A}$, $p_A$, and $\texttt{Fork}^A$ be as described. If $p_F := \mathbb{P}\left[\perp \neq \texttt{Fork}^{\mathcal{A}}(\texttt{inp}, \underline{\Gamma}^{(1)}, \underline{\Gamma}')\right]$ then we have the lower bound $p_F \geq p_A\left(\frac{p_A}{q} - \frac{1}{2^\ell}\right)$.

**Corollary 5.7.** If $p_A$ is non-negligible, then $p_F$ is non-negligible.

We apply the above to a $(t, \epsilon)$-forger by setting $\texttt{inp} = \texttt{Pub}(\mathcal{U}_h)$, denoting the sequence of random oracle queries by $\mathcal{A}$ as $\underline{h}$, and using for the sequences of auxiliary inputs $\Gamma_i$ (i) the masked keys $Y^*$ resulting from queries in $\mathcal{MO}$, with (ii) the sets $\Lambda_1$ sent from $\mathcal{SO}$ to $\mathcal{A}$ in $\mathcal{T}_{A^{s\mathcal{O}}}$.

If $\mathcal{A}$ can succeed and the unforgeability game, then $\mathcal{A}$ outputs a message $M$, a ring $\mathcal{Q}$, and a signature-key image pair $(\sigma, J)$ where $\sigma$ is a valid ring signature on $M$ with ring $\mathcal{Q}$ and key image $J$. We need output of some form $(i, j, \texttt{out})$ for some indices $i, j$. We inspect a successful transcript for inspiration.

**Lemma 5.8.** In each successful transcript with a forged signature using Example 4.5, the $H_s$ oracle query for each verification equation of the form $c_{\ell+1} = H_s(M, s_\ell G + c_\ell Y_\ell, s_\ell H_\ell + c_\ell J)$ occurs except with negligible probability.

*Proof.* Let $q_H$ denote the total number of queries made to $H_s$ by $\mathcal{A}$ including queries made by other oracles. Let $E$ be the event that each query to $H_s$ for the verification equation for each $c_\ell$ occurs in the transcript of $\mathcal{A}$. Note that the law of total probability gives us

$$\mathbb{P}\left[\mathcal{A} \text{ forges}\right] = \mathbb{P}\left[\mathcal{A} \text{ forges} \mid E\right] + \mathbb{P}\left[\mathcal{A} \text{ forges} \mid \overline{E}\right].$$

In the event $\overline{E}$, $\mathcal{A}$ produced a successful forgery without computing some $c_\ell$ by querying $H_s$. Hence, in the event $\overline{E}$, $\mathcal{A}$ had to guess some $c_\ell$ by flipping coins; $\mathcal{A}$ can do this with probability of success $1/(\mathfrak{q} - q_H)$. Indeed, the group has $\mathfrak{q}$ elements and $q_H$ of them have already been used in previous queries to $H_s$. Hence, forgeries by $\mathcal{A}$ happen in event $E$ except with negligible probability since $\mathbb{P}\left[\mathcal{A} \text{ forges} \mid E\right] = \mathbb{P}\left[\mathcal{A} \text{ forges}\right] - \frac{1}{\mathfrak{q} - q_H}$. $\square$

So in a successful forging transcript with input $\texttt{inp} = \texttt{Pub}(\mathcal{U}_h)$, we can find all $R$ verification queries and their line numbers in the transcript, say $i_1 < i_2 < \cdots < i_R$. For each $i_k$, we can find an index $j_k$ such that $\Gamma_{j_k}$ is the auxiliary input for the $i_k^{th}$ query. Set $i' := i_1$ (the line number of the first oracle query) and $j' = j_1$ (the index of auxiliary information used in this query). For each verification query $h_{i_k}$, there also exists a ring index $\ell_k$ such that $h_{i_k}$ is the verification query for $c_{\ell_k}$. Set $\ell' := \ell_R$ so that $\ell' - 1$ is the ring index of the final verification query. We say this transcript results in a $((i', \ell'), j')$-forgery.

Since pairs of natural numbers of the form $(i, \ell)$ can be totally ordered with the lexicographic ordering, our general forking lemma applies: we parse the output of $\mathcal{A}$ into the form $(i, \ell, j, M, \mathcal{Q}, J, \sigma)$, corresponding to a $((i, \ell), j)$-forged ring signature $\sigma$ on modified message $M^* = (M, \mathcal{Q}, J)$ with ring $\mathcal{Q}$ and key image $J$.

**Theorem 5.9.** Example 4.5 is EUF-ST.

*Proof.* Assume $\mathcal{A}$ is a $(t, \epsilon)$-forger with non-negligible $\mathcal{ACC}(\mathcal{A}) \geq \epsilon$. The rewind-on-success forking lemma allows $\texttt{Fork}^{\mathcal{A}}$ to produce two successful transcripts with non-negligible probability, resulting in an $((i, \ell), j)$-forgery $(i, \ell, j, M, \mathcal{Q}, J, \sigma)$, and an $((i, \ell), j')$-forgery. Moreover, these transcripts random oracle queries match until the $(i, \ell)^{th}$ random oracle query in $\underline{h}^{(1)}, \underline{h}^{(2)}$, and their auxiliary data $\Gamma^{(1)}, \Gamma^{(2)}$ match until the $(j+1)^{th}$ entry. For convenience, denote the index of the first distinct random oracle query in $\underline{h}^{(1)}$, $\underline{h}^{(2)}$ as $i'$.

The first $i'-1$ queries to the random oracle are common between these two transcripts. Also, due to the way we constructed $\underline{h}^{(2)}$, the $(i')^{th}$ random oracle queries, which compute $c_\ell$, differ in these two transcripts. By definition, the query for $c_\ell$ uses data from $\Gamma^{(2)}_{j'}$:

$$c_\ell = H_s(M, uG, uH_p(Y_{\ell-1}))$$

From this point on, the transcripts differ in their oracle queries. By the end of the two transcripts, $\mathcal{A}$ has used different signature data $\{s_\ell\}, \{s^*_\ell\}$ and different oracle queries to compute two different chains of commitments $c_{\ell+1}, \ldots, c_\ell$ and $c^*_{\ell+1}, \ldots, c^*_\ell$ in both transcripts to obtain

$$uG = s_{\ell-1}G + c_{\ell-1}Y_{\ell-1} = s^*_{\ell-1}G + c^*_{\ell-1}Y_{\ell-1}$$

Now $\mathcal{A}$ can trivially solve for the discrete logarithm of $Y_{\ell-1}$ by computing $\frac{s_{\ell-1}-s^*_{\ell-1}}{c^*_{\ell-1}-c_{\ell-1}}$.

Recall $\mathcal{A}$ had access to the discrete logarithm oracle through the $\mathcal{SO}$ oracles. However, while $\mathcal{A}$ may used many public keys in queries to $\mathcal{DL}$ indirectly through $\mathcal{SO}$ or in queries to $\mathcal{CO}$ directly, the forgery produced by $\mathcal{A}$ is not valid if any summand of any signing key appears in a query to either oracle. Hence, computing the discrete logarithm of $Y_{\ell-1}$ violates the OMDL hardness assumption. □

# 6 Extensions

## 6.1 Smaller Thresholds

We may extend the above approach to an $(N-1)$-of-$N$ threshold signature scheme in the following way: upon receipt of each $X^*_j$ during `Merge`, the shared secrets $z^*_{i,j} = H_s(x^*_i X^*_j)$ are computed. Now there are $\frac{1}{2}N(N-1)$ distinct shared secrets split across $N$ parties such that any $N-1$ members can regain all of the secrets.

Hence, an $(N-1)$-of-$N$ scheme may be implemented as an $\frac{N(N-1)}{2}$-of-$\frac{N(N-1)}{2}$ scheme by augmenting `Merge` to share secrets. `Merge` takes as input a set of public keys $\{X_1, \ldots, X_N\}$, a threshold $t$, and a set of pairs $(x_i, m_i)$ such that $X_j = x_j G$ for each $j$. The following is executed.

(a) If $\max(2, N-1) \leq t \leq N$ does not hold, $\perp$ is output and `Merge` is terminated.

(b) If $t = N$, execute `Merge` as described in Example 4.5.

(c) If $t = N-1$, each contributor $X_i$ picks their masked key $X^*_i$ and shares it with the rest of the coalition. Each participant now computes each shared secret $z_{i,j} = H_s(x^*_i X^*_j)$ and the public point $Z_{i,j} = z_{i,j}G$. Now each pair of participants pick a new random scalars $m_{i,j}$ and `Merge` is executed as described in Example 4.5 with the secrets $\{(z_{i,j}, m_{i,j})\}_{i,j}$.

Note that construction of an $N-1$-of-$N$ key this way requires all $N$ participants. It should be clear that this generalization does not impact our security definitions; an LSTAG scheme with $(N-1)$-of-$N$ and $N$-of-$N$ thresholds as described will still be indistinguishably private, unforgeable, and signer ambiguous, and so on. Moreover, assuming the CDH hardness assumption is true, no adversary learns any $x^*_i X^*_j$ (let alone any $z_{i,j}$.

## 6.2 Monero-style keys

In Monero, the user key space is separated from the signing key space. Indeed, the private user key space is $\mathbb{Z}^2_{\mathfrak{q}}$ and the public user key space is $\mathbb{G}^2$. That is, user keys come in pairs $(a, b)$ and public user keys come in pairs, $(A, B)$. The signing key space is still $\mathbb{Z}_{\mathfrak{q}} \times \mathbb{G}$.

A Monero transaction is addressed to the public user key $(A, B)$ (with private key $(a, b)$) if it contains the one-time, session, or derived public signing key $P = H_s(tA)G + B$ where $(t, T)$ is the *private-public transaction key pair*. An observer who knows $(a, B)$ sees $T$ and $P$ from some passing transaction and can check if a the transaction belongs to $(A, B)$ if $T - H_s(aT)G = B$. For this reason, $\mathcal{A}$ is called the *view key*. An observer who knows $(a, b)$ can sign messages with the private key $p = H_s(aT) + b$, and we call $b$ the *spend key* as a consequence. As before, the key image is still $J = pH_p(P)$.

Implementing LSTAG signatures with Monero-style keys, then, requires a modification, where *signing* keys are not merged but in fact addressed to merged keys. Our way of merging keys must take into account this view key and key images.

Using the thresholdizing heuristic where we replace keys with sums of keys and random signature data with sums of random signature data, we have a shared key $(A_{\mathtt{sh}}, B_{\mathtt{sh}}) = (\sum_i A_i, \sum_i B_i)$. We allow every threshold member to learn the shared private view key $a_{\mathtt{sh}} = \sum_i a_i$ so collaboration for watch-only wallets is unnecessary. Now each member knows $a_{\mathtt{sh}}$ and a share $b_i$ such that $\sum_i b_i G = B_{\mathtt{sh}}$. The signing key $P = H_s(rA)G + B$ can now be written as $P = H_s(rA)G + \sum_i b_i G$. In order to solve for the missing signature data $s = u - cp$, the coalition members must then collaboratively compute $s = u - c(H_s(aR) + \sum_i b_i) = (u - c\sum_i b_i) - cH_s(aR)$.

Proofs of security in this situation require more sophisticated definitions than those we take into account herein; we leave these for future works.

### 6.3 MLSAG and MLSTAG signatures as ring confidential transactions

LSAG signatures are made into a multisignature with the MLSAG construction introduced in [10]; the MLSAG construction extends to the threshold situation, resulting in MLSTAGs (multi-layered spontaneous threshold anonymous group signatures). We briefly describe the MLSAG extension of LSAG signatures and how to use them in ring confidential transactions; we leave complete, formal definitions of ring confidential transactions, their security properties, and the thresholdized extension of those constructions for future works.

MLSAG signatures, by construction, are multisignatures since they use multiple keys to sign a message but they are collaboratively computed. They produce a signature size that is independent of the number of signers, but MLSAG signatures still reveal the number of signing keys. With a list of signing private keys $p_1, \ldots, p_N$ interpreted as a vector, the signer (or signers) randomly selects similar vectors from the blockchain to construct a ring of such key vectors $\widetilde{\mathcal{Q}}$, packing the public keys for $\underline{p}$ into the $\ell^{*th}$ column for a secret $\ell^*$.

$$\widetilde{\mathcal{Q}} = \begin{pmatrix} P_{1,1} & P_{1,2} & \cdots & P_{1,R} \\ P_{2,1} & P_{2,2} & \cdots & P_{2,R} \\ \vdots & & & \vdots \\ P_{N,1} & P_{N,2} & \cdots & P_{N,R} \end{pmatrix}.$$

For each $1 \leq k \leq N$ and $1 \leq \ell \leq R$, with the entry $P_{j,\ell}$ in $\widetilde{\mathcal{Q}}$ we compute $H_{j,\ell} := H_p(P_{j,\ell})$. For each component $p_j \in \underline{p}$, the signer computes key image $J_j = p_j H_p(p_j G)$. The signer selects a vector of random scalars $\underline{u} = (u_1, \ldots, u_N) \in \mathbb{Z}_{\mathfrak{q}}^N$ for the $\ell^{*th}$ column and, for each $\ell \neq \ell^*$, the signer selects a vector of random scalars $\underline{s}_\ell = (s_{1,\ell}, s_{2,\ell}, \ldots, s_{N,\ell})$. The signer can now compute commitments

$$c_{\ell+1} = H_s\left(M^*, \left\{(\mathcal{L}_\ell^j, \mathcal{R}_\ell^j)\right\}_{j=1}^N\right)$$

Once each commitment has been computed, the signer computes $s_{j,\ell^*} = u_j - c_\ell p_j$ as usual and the MLSAG signature is then $\sigma = (c_1, \{s_{j,\ell}\})$, which is verified similarly to LSAG signatures. Certainly we also may apply

the same thresholdizing heuristic: for a collaborating coalition of signers, presume that each participating signer has a share of a key and each participant must contribute some random scalars to be summed for $u_j$, $s_{j,\ell}$, etc.

Anyone can discern that one of these columns contains all the signing keys. That is to say, it is not possible that $p_{2,2}$ and $p_{1,1}$ may both be the true keys used in this ring signature. If we interpret all keys in $\widetilde{\mathcal{Q}}$ as signing keys, this reduces the signer ambiguity we so value in ring signatures. However, if we interpret some keys as associated with transaction amounts, we can construct confidential transactions. With our thresholdizing heuristic, we can construct threshold confidential multitransactions.

Introduced in [7], a confidential transaction may be loosely regarded as a multisignature fashioned using the usual private signing key but also the secret opening information for some corresponding homomorphic commitments to transaction amounts. To accomplish this, each private one-time signing key $p$ now comes paired with an amount $\alpha$ and mask $r$ (both from $\mathbb{Z}_{\mathtt{q}}$), and each public one-time key $P$ comes with a Pedersen commitment $C$. A sender with public key-commitment pair $(P, C)$ and private $(p, (\alpha, r))$ who desires to send amount $0 \leq \alpha' \leq \alpha$ to another user does the following.

The sender constructs two new Pedersen commitments by selecting two new random masks $r'$, $r''$ and computing the commitments $C' = \mathtt{Com}(\alpha', r')$ and $C'' = \mathtt{Com}(\alpha - \alpha', r'')$. These commitments must be paired with one-time signing keys so that the receivers can subsequently construct new transactions, so one-time keys $P'$ and $P''$ are selected; $P'$ is addressed to the receiver of the transaction and $P''$ is addressed back to the sender for chagne. The signer constructs an MLSAG signature with a signing matrix of the form

$$\mathcal{Q} = \begin{pmatrix} \widetilde{P}_1 & \widetilde{P}_2 & \cdots & \widetilde{P}_R \\ \widetilde{C}_1 - C' - C'' & \widetilde{C}_2 - C' - C'' & \cdots & \widetilde{C}_R - C' - C'' \end{pmatrix}$$

where each $(\widetilde{P}_\ell, \widetilde{C}_\ell)$ is chosen at random from the blockchain for $\ell \neq \pi$, and where $\widetilde{P}_\pi = P$ and $\widetilde{C}_\pi = C$. Now if the receiver of $C'$ learns the opening information for $C'$ and the private key for $P'$, they may subsequently construct new transactions. In Monero, these are accomplished via encryption with secrets generated by ECDH transfer as described herein.

This approach forces transactions to be *balanced* by construction (the input amount and output amount must match in order for the signer to know the discrete logarithm of the $(2, \ell^*)^{th}$ entry in $\widetilde{\mathcal{Q}}$) and still functions as a ring signature for message authentication. The sender should also include in $M$ a *range proof*, which verifies whether $\alpha'$ is a reasonable amount, say $0 \leq \alpha' < 2^\iota$ for some large $\iota$ (say $2^{64}$ as in Monero), so that integer-like arithmetic for these transactions are not spoiled by the modular arithmetic of the underlying (finite order) group.

Applying the thresholdizing heuristic we present here yields MLSTAG Ring Confidential Transactions.

# References


[1] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 390–399. ACM, 2006.

[2] Adam Bender, Jonathan Katz, and Ruggero Morselli. Ring signatures: Stronger definitions, and constructions without random oracles. In *TCC*, volume 6, pages 60–79. Springer, 2006.

[3] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter MR Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption.

[4] Emmanuel Bresson, Jacques Stern, and Michael Szydlo. Threshold ring signatures and applications to ad-hoc groups. In *Annual International Cryptology Conference*, pages 465–480. Springer, 2002.

[5] Joseph K Liu, Victor K Wei, and Duncan S Wong. A separable threshold ring signature scheme. In *International Conference on Information Security and Cryptology*, pages 12–26. Springer, 2003.

[6] Joseph K Liu, Victor K Wei, and Duncan S Wong. Linkable spontaneous anonymous group signature for ad hoc groups. In *ACISP*, volume 4, pages 325–335. Springer, 2004.

[7] Gregory Maxwell. Confidential transactions. *URL: https://people. xiph. org/˜ greg/confidential_values. txt (Accessed 09/05/2016)*, 2015.

[8] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. 2018.

[9] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 245–254. ACM, 2001.

[10] Shen Noether, Adam Mackenzie, et al. Ring confidential transactions. *Ledger*, 1:1–18, 2016.

[11] Pascal Paillier and Damien Vergnaud. Discrete-log-based signatures may not be equivalent to discrete log. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–20. Springer, 2005.

[12] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[13] Patrick P Tsang, Victor K Wei, Tony K Chan, Man Ho Au, Joseph K Liu, and Duncan S Wong. Separable linkable threshold ring signatures. In *Indocrypt*, volume 3348, pages 384–398. Springer, 2004.