Thring Signatures and their Applications to Spender-Ambiguous Digital Currencies

Brandon Goodell and Sarang Noether

Monero Research Lab {surae,sarang}@getmonero.org

Abstract. We present threshold ring multi-signatures (*thring signatures*) for collaborative computation of ring signatures, discuss a game of existential forgery for thring signatures, and discuss the uses of thring signatures in digital currencies, including spender-ambiguous cross-chain atomic swaps for confidential amounts without a trusted set-up. We present an implementation of thring signatures inspired by the works of [13], [20], [14], [1], [18], [15] we call linkable spontaneous threshold anonymous group (LSTAG) signatures, and we prove the implementation existentially unforgeable.

Keywords: ring signatures, confidential transactions, spender ambiguity, threshold multi-signatures, thring signatures, atomic swaps, public-key cryptography, implementation, applications

1 Introduction

Cryptocurrencies and e-cash schemes critically rely upon unforgeable digital signatures, and many use cases exist for threshold signatures in these settings, ranging from multi-factor authentication in transaction approval to cross-chain atomic swaps. The idea is simple: a group of collaborating users decide upon a threshold and some members of the group can collaborate to compute signatures just so long as more group members agree to a computation than the threshold. Upon inception, Bitcoin used ECDSA on secp256k1 for signatures, but this is by no means the only option available to cryptocurrency engineers. More recently, Schnorr signatures [22] have gained popularity and functionality, including threshold multi-signatures as presented in [15], and Okamoto signatures [19] have also been investigated for use in multi-signatures in cryptocurrencies [10]. Other options, like ring signatures [13] or zerocoin-style accumulators [16] allow for signer-ambiguous message authentication, where verifiers can check that a member of an anonymity set authenticates a message, and yet have a negligible advantage in determining which member.

In the cryptocurrency setting, signer-ambiguous message authentication is necessary for *spender ambiguity*. We interpret this ambiguity as a property regarding plausible deniability in the history of transactions: financial systems employing these options allow for publicly verifiable ledgers yet allow users the ability to at least plausibly deny their involvement in any one transaction (or some small set of them). It is therefore natural to seek threshold extensions to ambiguous message authentication like ring signatures. These ring signatures can also be exploited to construct ring confidential transactions in a digital currency. Multi-signature functionality in a digital currency also allows, for example, cross-chain atomic swaps. Hence, a thresholdized ring signature scheme can allow for spender-ambiguous cross-chain atomic swaps for confidential amounts without a trusted set-up.

We refer to thresholdized ring signatures as *thring signatures*. Previous proposals for thring signatures (e.g. [9], [12], [23]) are leaky (They reveal properties of the signing

coalition such as the number of signers, or even publish the list of signing public keys with signatures), or use key computations akin to those presented in [3], which are vulnerable to rogue key attacks. For these reasons, we find these proposals to be inappropriate for use in privacy-focused digital currencies.

We construct a thring signature implementation using a Musig-style approach applied to the linkable spontaneous anonymous group (LSAG) signatures of [13], taking into account the modifications presented in [1] and [18] intended for use in digital currency applications. The Musig approach from [15] aggregates keys with an approach first proposed in [20] that is resistant to rogue-key attacks. As is usual for multisignature schemes, we replace the random data for use in signing with sums of data selected by participants, which is revealed in a commit-and-reveal phase. Lastly, like the Musig approach, we include signing keys in signature challenge computation which prevents oracle queries in the proof of existential unforgeability from taking place in a bad order.

1.1 Our contribution

We present an LSTAG signature scheme, which resembles the Musig multi-signatures from [15], and is a thresholdized version of LSAG signatures. We present a definition of existential unforgeability in the ring signature setting and prove the scheme secure under this definition. We also make remarks on the difference between our scheme and ostensible applications for cryptocurrency purposes.

The scheme we present is n-of-n but can be extended to m-of-n threshold schemes using standard techniques, and can be extended to a multi-layered scheme for use in ring confidential transactions. The scheme is aggregate-keyed, in the sense that signature verification does not require knowledge of the signing threshold. Signature sizes are not dependent upon the number of colluding signers. The security proof is under the plain public key model. We use the same three-step process of Musig, and verification of the signatures proceeds identically to usual LSAG signatures.

In Section 2, we explain our notation, assumptions, and other pre-requisites. In Section 3, we loosely explain how LSAG signatures presently work in Monero, how to apply the Musig thresholdizing heuristic to them to obtain LSTAGs, and applications of LSTAGs for use in ring confidential transactions. In Section 4, we define LSTAG signatures, present an implementation example. In Section A, we define an unforgeability game for LSTAG signatures, prove the implementation from Section 4 is secure under this definition. We also discuss linkability, exculpability, signer ambiguity, and key aggregation indistinguishability.

1.2 Related works and challenges

In [13], LSAG signatures were first described; modifications were suggested in [1]. In [14], confidential transactions for use in digital currencies were first described, and in [18], ring signature extensions of confidential transactions using a key-vector extension of LSAG signatures called multi-layered linkable spontaneous anonymous group (MLSAG) signatures were first described. We generally use the approach from [3] for computing multi-signatures, with the Musig-style key aggregation from [20] and the commit-and-reveal approach of [15]. Our unforgeability proof, like that of [15], uses a double application of the rewind-on-success forking method (ostensibly first presented in [13]).

Other multi-signature techniques, especially in the pairings- and learning-with-errors based worlds, are available. See, for example, [8], a novel and quite general fully homomorphic thresholdizing set-up using only one round of communication is described in the learning-with-errors environment, leading to fully homomorphic threshold signatures and encryption. More recently, [7] uses a pairings-based setting to provide compact multi-signatures with extremely valuable properties.

Note that the thresholdizing heuristic described in [3] presented a general multi-signature scheme in the plain public key model, but also presented a detailed discussion of the knowledge-of-secret-key (KOSK) assumption. It is worth noting that there are some problems with practically and securely implementing schemes that are proven secure under the KOSK setting using proofs of possession of secret keys unless special care is taken (see, for example, [21]). For this reason, we avoid the KOSK assumption, although the security proofs for schemes such as ours are dramatically simpler in the KOSK setting with proofs of possession replacing the certificate authority. In [20], the KOSK-free method of aggregating a shared multi-signature public key we use here was presented which is resistant to rogue key attacks (albeit in a pairings-based setting). An early version of Musig used a single round of communication, which was later modified to include a commit-and-reveal stage to signing. Indeed, in [10] it is demonstrated that it is very unlikely that a single round of communication in a Musig-like signature scheme can be proven secure under the discrete logarithm hardness assumption.

For use in digital currency applications in ring confidential transactions, these thring signatures must be further extended. For example, in applications to digital currencies employing (some variant of) the Cryptonote protocol such as Monero, implementation must account for keypair vectors including a view key in addition to a spend/signing key, and must account for one-time key computations. Proving the security of our scheme under these extensions is beyond the scope of this document. For more detailed formalizations of ring confidential transactions and thring signatures, see for example [11].

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. We also wish to extend particular thanks to Andrew Poelstra and Yannick Seurin, and many others, for some extremely helpful comments.

2 Notation and assumptions

We generally use calligraphic font to denote PPT algorithms and oracles: $\mathcal{A}, \mathcal{A}', \mathcal{A}'', \mathcal{B}$ are PPT algorithms, $\mathcal{H}$ is a random oracle, $\mathcal{SO}$ is a signing oracle. We often grant an algorithm oracle access, which we denote with a superscript: $\mathcal{A}^{\mathcal{SO}}$ means $\mathcal{A}$ has oracle access to $\mathcal{SO}$. We shall often leave the superscripts implicit for clarity, unless there is risk of confusion. We use `teletype font` to describe the inputs and outputs of algorithms, or as the names of special algorithms from a cryptographic scheme: $\texttt{inp}_{\mathcal{A}}$ is the input for $\mathcal{A}$, $\texttt{out}_{\mathcal{A}}$ is the output, `Sign` is the signing algorithm in our implementation, and so on. Algorithms often come with a distinguished failure symbol (or a set of them) which we denote $\perp$: $\perp_{\mathcal{A}}$ is the failure symbol of $\mathcal{A}$, $\perp_{\texttt{Sign}}$ is the failure symbol of `Sign`, and so on.

We use miniscule english letters and greek letters for integers: $n, r, q, \ell, i, j, k, \alpha, \eta$ are all in $\mathbb{N}$. For any $r \in \mathbb{N} = \{1, 2, \ldots\}$, we denote the set of r elements $\{1, 2, \ldots, r\}$ with $[r]$. We use underlines to denote vectors, ordered lists, sequences, and sets indexed by well-ordered indexing sets. For example, for an unordered list of independent random oracles $\underline{H}$ indexed by some arbitrary finite set Λ with $n = |\Lambda|$ elements, we can harmlessly re-index and assume $\Lambda = [n]$ and write $\underline{\mathcal{H}} = \{\mathcal{H}_i\}_{i \in [n]}$.

For any $n > 1$, we denote $\mathbb{Z}/n\mathbb{Z}$ with $\mathbb{Z}_n$. We assume a set-up phase is executed with a security parameter $\eta > 1$ before beginning, resulting in some $(\mathfrak{p}, \mathbb{G}, G, \underline{\mathcal{H}}, \phi, \Phi)$, namely a cyclic group $\mathbb{G}$ of order $\mathfrak{p}$ with generator G such that elements from $\mathbb{Z}_{\mathfrak{p}}$ and $\mathbb{G}$ admit η-bit representations, a sequence $\underline{\mathcal{H}}$ of cryptographic hash functions, and two key aggregation

functions ϕ, Φ. We say an element of $\mathbb{G}$ is a *public key*, which we denote these throughout this document with majuscule english letters. For example, A, B, C, T, P, X are public keys, and $\underline{P} = \{P_\ell\}_{\ell \in [r]}$ denotes a sequence of r public keys. We say such a sequence is a *ring* if it is to be used as input for a ring signature. For computing ring signatures with $r \in \mathbb{N}, r > 1$ ring members, we allow indices from $[r]$ to "wrap around" by identifying the ring index $r+1$ with the ring index 1. We only use this convention for ring member indices, not for other indices.

We say members of $\mathbb{Z}_\mathfrak{p}$ are *private keys*. Since these are equivalence classes of integers, we also use miniscule english letters to denote these private keys, but we tend to use different sets of letters than for integers: a, b, t, p, and x are private keys from $\mathbb{Z}_\mathfrak{p}$. We say that some $X \in \mathbb{G}$ is a public key *associated with* some private key $x \in \mathbb{Z}_\mathfrak{p}$ if $X = xG$ for the generator G specified above, and we match letters: the public key associated with the private key a is $A = aG$, the public key associated with the private key b is $B = bG$, and so on. For a list of private keys, say $\underline{x} = (x_1, \ldots, x_n)$, we write the list of corresponding public keys as $\underline{X} = (X_1, \ldots, X_n)$ where each $X_i = x_i G$. We denote this $\underline{X} = \underline{x}G$ at the risk of abusing notation.

We use the convention that $\phi(x, \{X\}) = x$ and $\Phi(\{X\}) = X$. Here, Φ takes as input a non-empty multiset $\underline{X} = (X_1, \ldots, X_n)$ of public keys and produces as output a shared public key $X_{\text{sh}} \leftarrow \Phi(\underline{X})$, and ϕ takes as input a private key x_i with a non-empty multiset $\underline{X}$ of public keys (such that $X_i = x_i G$ in $\underline{X}$ at least once) and produces as output a coefficient $\beta_i = \phi(x_i, \underline{X})$. Users set their private share as $x_i^* := \beta_i x_i = \phi(x_i, \underline{X}) x_i$. We say X_{sh} is *related to* or a *child of* the keys in $\underline{X}$ and we use

$$\Phi(\underline{X}) := \sum_{i \in [n]} \beta_i X_i = \sum_i x_i^* G = X_{\text{sh}}.$$

We specify $\underline{\mathcal{H}}$, six arbitrary-length-input, η-bit-output cryptographic hash functions with independent outputs

$$\begin{array}{lll} \mathcal{H}_{\text{ki}} : \{0,1\}^* \to \mathbb{G} & \mathcal{H}_{\text{com}} : \{0,1\}^* \to \{0,1\}^\eta & \mathcal{H}_{\text{agg}} : \{0,1\}^* \to \mathbb{Z}_\mathfrak{p} \\ \mathcal{H}_{\text{sig}} : \{0,1\}^* \to \mathbb{Z}_\mathfrak{p} & \mathcal{H}_{\text{msg}} : \{0,1\}^* \to \mathbb{Z}_\mathfrak{p} & \mathcal{H}_{\text{sess}} : \{0,1\}^* \to \mathbb{Z}_\mathfrak{p} \end{array}$$

under the random oracle model. We denote concatenation of bitstrings with the symbol $||$. Since elements of $\mathbb{Z}_\mathfrak{p}$ can be described with η bits, we may as well take $\mathcal{H}_{\text{agg}}$, $\mathcal{H}_{\text{sig}}$, $\mathcal{H}_{\text{com}}$, $\mathcal{H}_{\text{msg}}$, and $\mathcal{H}_{\text{sess}}$ to all five have the same codomain. This way, for implementation purposes, these hash functions can be realized with a single hash function $\mathcal{H}_{\text{sc}} : \{0,1\}^* \to \mathbb{Z}_\mathfrak{p}$ using domain separation:

$$\begin{aligned} \mathcal{H}_{\text{com}}(x) &:= \mathcal{H}_{\text{sc}}(000 \,||\, x) \\ \mathcal{H}_{\text{agg}}(x) &:= \mathcal{H}_{\text{sc}}(001 \,||\, x) \\ \mathcal{H}_{\text{sig}}(x) &:= \mathcal{H}_{\text{sc}}(010 \,||\, x) \\ \mathcal{H}_{\text{msg}}(x) &:= \mathcal{H}_{\text{sc}}(011 \,||\, x) \\ \mathcal{H}_{\text{sess}}(x) &:= \mathcal{H}_{\text{sc}}(111 \,||\, x) \end{aligned}$$

This allows us to only require two hash functions for implementation, $\mathcal{H}_{\text{ki}}$ and $\mathcal{H}_{\text{sc}}$. Also, we only use $\mathcal{H}_{\text{sess}}$ in an early example; our implementation could use just the first four $\mathcal{H}_{sc}$ variants and use only two bits of prefixes. Note that although these functions will each have η-bit outputs, their effective entropy is actually somewhat lower due to this domain separation.

Of course, if $\mathcal{H}_{\text{ki}}$ can be factored into some $\mathcal{H}_{\text{ki}} = \mu_G \cdot \mathcal{H}_{\text{ki}}^*$ for some $\mathcal{H}_{\text{ki}}^* : \{0,1\}^* \to \mathbb{Z}_\mathfrak{p}$ (where $\mu_G : \mathbb{Z}_\mathfrak{p} \to \mathbb{G}$ is the canonical hard-to-invert homomorphism defined by mapping $x \mapsto xG$), then whoever knows this factorization of $\mathcal{H}_{\text{ki}}$ can compute the discrete logarithm of outputs of $\mathcal{H}_{\text{ki}}$, which is a highly undesirable property. We assume no such factorization is easily computable.

3 Motivation of scheme

In this section, we describe the motivation behind our thring signature scheme. We begin by describing, loosely, how Back-style LSAG signatures are computed using the particular CryptoNote-style key spaces. We then describe a thresholdizing heuristic for use in constructing our implementation, and then we make some brief comments on the application of these signatures in Monero.

3.1 Back-style LSAG signatures

In this section we provide a brief look at how signatures work in Cryptonote-styled digital currencies like Monero. The signatures in [13] use key images that are dependent upon the ring of signers, making them inappropriate for linkability in this setting. In [1], a modification was presented, which we summarize here. Our description makes use of four cryptographic hash functions modeled as random oracles, $\mathcal{H}_{\text{sess}}$, $\mathcal{H}_{\text{sig}}$, $\mathcal{H}_{\text{ki}}$, and $\mathcal{H}_{\text{msg}}$.

Consider first the CryptoNote-styled key spaces of Monero. Users have *user keypairs* (consisting of a *view key* and a *spend key*) and they sign with *signing keypairs* (consisting of a *transaction key* and a *session key* [alternately, a *one-time key* or *stealth key*]). Session keys are computed from user keypairs and transaction keys; ring signatures are computed using the session keys. A generic keypair space is made available, $\mathfrak{K} = \mathbb{Z}_{\mathfrak{p}}^2 \times \mathbb{G}^2$ and both sorts of keypairs come from $\mathfrak{K}$.

A user keypair is denoted $((a,b),(A,B)) \in \mathfrak{K}$ and 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*. Honest users select their private keys uniformly at random. A signing keypair is some $((t,p),(T,P)) \in \mathfrak{K}$ and we say t is the *private transaction key*, p is the *private session key*, T is the *public transaction key*, and P is the *public session key*. Honest users receive the public part of their transaction key T from a sender. In Monero, we say a public signing key pair (T,P) with index i in some transaction is *addressed to* a public user key pair (A,B) if $P = \mathcal{H}_{\text{sess}}(aT,i)G + B$. Given T,a,B,i or t,A,B,i, ownership of a session key can be easily tested: merely check if $P - \mathcal{H}_{\text{sess}}(aT,i) = B$. Given T,a,b,i or given t,A,b,i, the private session key can be easily computed: $p = \mathcal{H}_{\text{sess}}(aT,i) + b$. In the sequel we "forget" the transaction index i for clarity in our notation.

If Alice has a user key $((a,b),(A,B)) \in \mathfrak{K}$, has previously received a message containing some public signing keys (T,P) addressed to (A,B), and Alice wishes to address some keys to Bob (who has public user key (A',B')). Alice computes p, picks a new private transaction key $t' \leftarrow \mathbb{Z}_{\mathfrak{p}}$, and computes a new public session key $P' := \mathcal{H}_{\text{sess}}(t'A')G + B'$ for Bob. Alice then sends (T',P') to Bob in a message $\mathfrak{m}$, and produces a ring signature σ on a modified message $\mathfrak{m}^*$ that contains $\mathfrak{m}$ and some ring $\underline{P}$ such that $P \in \underline{P}$. Alice computes the ring signature in the following way.

Alice selects a message $\mathfrak{m} \in \{0,1\}^*$, computes her one-time key image $J = p\mathcal{H}_{\text{ki}}(P)$, and selects a ring of public signing keys $\underline{P} = \{P_1, \dots P_r\}$ such that, for a secret distinguished index π, $P_\pi = P$. Alice assembles a modified message $\mathfrak{m}^* = (\mathfrak{m}, \underline{P}, J, T', P')$ and computes $M = \mathcal{H}(\mathfrak{m}^*)$. For each $\ell = 1, \dots, r$, the signer computes an elliptic curve point from the ℓ^{th} ring member $H_\ell := \mathcal{H}_{\text{ki}}(P_\ell)$. The signer selects a random secret scalar $u \xleftarrow{\$} \mathbb{Z}_{\mathfrak{p}}$ and computes an initial temporary pair of points $L_\pi := uG$, $R_\pi := u\mathcal{H}_\pi$. The signer computes an initial commitment $c_{\pi+1} := \mathcal{H}_{\text{sig}}(M, L_\pi, R_\pi)$. We shall later use the *key prefixed variant* of this commitment, $c_{\pi+1} := \mathcal{H}_{\text{sig}}(M, P_\pi, L_\pi, R_\pi)$

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_ℓ, computing the next pair of points $L_\ell := s_\ell G + c_\ell P_\ell$ and $R_\ell := s_\ell H_\ell + c_\ell J$, and computes the next commitment $c_{\ell+1} := \mathcal{H}_{\text{sig}}(M, L_\ell, R_\ell)$ (or the key-prefixed variant $c_{\ell+1} := \mathcal{H}_{\text{sig}}(M, P_\ell, L_\ell, R_\ell)$). Once all commitments c_ℓ have been computed, the signer then computes $s_\pi := u - c_\pi p$ for the distinguished index π. $\sigma = (c_1, \underline{s})$

is the signature on $\mathfrak{m}$. Alice sends $(\mathfrak{m}^*, \sigma)$ to Bob. We say the set of equations

$$\begin{aligned} c_2 =& \mathcal{H}_{\text{sig}}(M, s_1 G + c_1 P_1, s_1 H_1 + c_1 J) = \mathcal{H}_{\text{sig}}(M, L_1, R_1) \\ c_3 =& \mathcal{H}_{\text{sig}}(M, L_2, R_2) \\ & \vdots \\ c_r =& \mathcal{H}_{\text{sig}}(M, L_{r-1}, R_{r-1}) \\ c_1 =& \mathcal{H}_{\text{sig}}(M, L_r, R_r) \end{aligned}$$

(or their key-prefixed variants) are the *verification equations*.

Upon receiving $(\mathfrak{m}^*, \sigma)$, Bob parses $\mathfrak{m}^* = (\mathfrak{m}, \underline{P}, J, (T', P'))$ and checks that $M = \mathcal{H}(\mathfrak{m}^*)$. Bob can easily test if $P' - \mathcal{H}_{\text{sess}}(a'T')G = B'$ to see if he is the addressee for the keys. Bob can also extract the private signing key by computing $p' = \mathcal{H}_{\text{sess}}(a'T') + b'$. This way, Bob can perform the same procedure Alice does above to pass new signing keys to other users. However, Bob may not be convinced the signature is genuine, so he verifies the signature in the following way.

Given $(\mathfrak{m}^*, \sigma)$, the verifier parses $\sigma = (c_1, \underline{s})$ and computes $M = \mathcal{H}(\mathfrak{m}^*)$. The verifier computes each $H_\ell = \mathcal{H}_{\text{ki}}(P_\ell)$. For each $1 \leq \ell \leq r$, the verifier finds $L'_\ell = s_\ell G + c_\ell P_\ell$, $R'_\ell = s_\ell H_\ell + c_\ell J$. The verifier uses these to compute the $(\ell+1)^{th}$ commitment $c_{\ell+1} = \mathcal{H}_{\text{sig}}(M, L'_\ell, R'_\ell)$. After computing $\underline{c} = (c_2, c_3, \ldots, c_r, c_{r+1})$, the verifier identifies index $r+1$ with index 1 and checks that $c_{r+1} = c_1$. If so, the verifier is convinced the signature is genuine. A verifier can check whether two signatures are signed by the same key by simply comparing key images.

3.2 Thresholdizing Back LSAG signatures with Musig-style aggregation

In this section, Alice and Bob wish to collaborate in constructing a 2-of-2 threshold version of an LSAG signature to send a key to Charlene. Signatures are verified exactly as before without any knowledge that they were constructed by a coalition rather than a single user. We proceed similarly to how we did before, making changes according to the following heuristics.

Keys are sums: Replace spend keys with linear combinations of spend key shares.

Signing data are sums: Replace the (random) signing data with sums.

Commit and reveal: Insert a commitment step before revealing the signing data.

Key-prefixing: Insert the ring member into each signature challenge.

In Musig, the key-aggregation function is $\phi(b_i, (\underline{A}, \underline{B})) := \mathcal{H}_{\text{agg}}(B_i, (\underline{A}, \underline{B}))$. Note that computing $\phi(b_i, (\underline{A}, \underline{B}))$ does not require the secret b_i. Alice picks a new private user key (a_1, b_1), computes the public key $A_1 = a_1 G$, $B_1 = b_1 G$, and sends (a_1, B_1) to Bob by secure side channel. Bob does the same by selecting (a_2, b_2) and sending (a_2, B_2) to Alice. They compute a private shared view key $a_{\text{sh}} = a_1 + a_2$ and a public shared spend key

$$B_{\text{sh}} = \mathcal{H}_{\text{agg}}(B_1, (\underline{A}, \underline{B}))B_1 + \mathcal{H}_{\text{agg}}(B_2, (\underline{A}, \underline{B}))B_2 = \beta_1 B_1 + \beta_2 B_2.$$

Alice and Bob could, alternatively, compute the private shared view key a_{sh} using any number of methods of computing a shared secret.

Alice and Bob receive a message $\mathfrak{m}$ containing some public signing keys (T, P) addressed to $(A_{\text{sh}}, B_{\text{sh}})$ so that $P = \mathcal{H}_{\text{sess}}(a_{\text{sh}} T)G + B_{\text{sh}}$. Alice and Bob wish to pass this along to Charlene, who has public user key (A', B'), by ring multi-signing some message $\mathfrak{m}'$. Alice

and Bob decide by side channel upon some ring $\underline{P} = (P_1, \dots, P_r)$ and a secret index π such that with $P_\pi = P$. Alice and Bob compute by side channel the key image

$$J = \left(\underbrace{\mathcal{H}_{\text{sess}}(a_{\text{sh}} T)}_{\text{view}} + \underbrace{b_1 \mathcal{H}_{\text{agg}}(B_1, (\underline{A}, \underline{B}))}_{\text{first participant}} + \underbrace{b_2 \mathcal{H}_{\text{agg}}(B_2, (\underline{A}, \underline{B}))}_{\text{second participant}} \right) \mathcal{H}_{\text{ki}}(P).$$

Alice and Bob also pick a new private transaction key $t' \leftarrow \mathbb{Z}_{\mathfrak{p}}$ (some member unilaterally selects or it is decided upon collaboratively somehow, similarly to the shared view key). Alice and Bob compute a new public session key for Charlene $P' = \mathcal{H}_{\text{sess}}(t'A')G + B'$. Alice and Bob compute a basepoint for each ring member, $H_\ell = \mathcal{H}_{\text{ki}}(P_\ell)$. Then Alice and Bob execute the ring signing algorithm in the following steps.

Commit: Alice selects $u_1 \xleftarrow{\$} \mathbb{Z}_{\mathfrak{p}}$ and Bob selects $u_2 \xleftarrow{\$} \mathbb{Z}_{\mathfrak{p}}$. Alice computes temporary pair of points $(L_{1,\pi}, R_{1,\pi}) = (u_1 G, u_1 H_\pi)$ and Bob computes temporary pair of points $(L_{2,\pi}, R_{2,\pi}) = (u_2 G, u_2 H_\pi)$. Alice selects random secret scalars $\underline{s}^{(1)} = \{s_{1,\ell}\}_{\ell \neq \pi}$ and Bob selects random secret scalars $\underline{s}^{(2)} = \{s_{2,\ell}\}_{\ell \neq \pi}$. Alice computes her partial key image $J_1 = b_1 \mathcal{H}_{\text{agg}}(B_1, (\underline{A}, \underline{B}))\mathcal{H}_{\text{ki}}(P)$ and Bob computes his partial key image $J_2 = b_2 \mathcal{H}_{\text{agg}}(B_2, (\underline{A}, \underline{B}))\mathcal{H}_{\text{ki}}(P)$. Alice computes the commitment $\texttt{com}_1 = \mathcal{H}_{\text{com}}(L_{1,\pi}, R_{1,\pi}, \underline{s}^{(1)})$ and Bob computes the commitment $\texttt{com}_2 = \mathcal{H}_{\text{com}}(L_{2,\pi}, R_{2,\pi}, \underline{s}^{(2)})$. Alice sends Bob $(J_1, \texttt{com}_1)$ and Bob sends Alice $(J_2, \texttt{com}_2)$.

Reveal: After receiving $(J_2, \texttt{com}_2)$, Alice sends $(L_{1,\pi}, R_{1,\pi}, \underline{s}^{(1)})$ to Bob; after receiving $(J_1, \texttt{com}_1)$, Bob sends $(L_{2,\pi}, R_{2,\pi}, \underline{s}^{(2)})$ to Alice. After receiving $(L_{2,\pi}, R_{2,\pi}, \underline{s}^{(2)})$, Alice checks that $\texttt{com}_2 = \mathcal{H}_{\text{com}}(L_{2,\pi}, R_{2,\pi}, \underline{s}^{(2)})$. If not, Alice outputs $\perp$ and terminates. After receiving $(L_{1,\pi}, R_{1,\pi}, \underline{s}^{(1)})$, Bob checks that $\texttt{com}_1 = \mathcal{H}_{\text{com}}(L_{1,\pi}, R_{1,\pi}, \underline{s}^{(1)})$. If not, Bob outputs $\perp$ and terminates.

Pre-compute signature: Alice and Bob compute the total key image $J = J_1 + J_2 + \mathcal{H}_{\text{sess}}(a_{\text{sh}} T)$, assemble a modified message $\mathfrak{m}^* = (\mathfrak{m}, \underline{P}, J, (T', P'))$, compute $M = \mathcal{H}(\mathfrak{m}^*)$, compute the sums $L_\pi = L_{1,\pi} + L_{2,\pi}$, $R_\pi = R_{1,\pi} + R_{2,\pi}$, and $s_\ell = s_{1,\ell} + s_{2,\ell}$ for each $\ell \neq \pi$. Alice and Bob can then compute the sequential commitments $c_{\ell+1} := \mathcal{H}_{\text{sig}}(M, P_\ell, L_\ell, R_\ell)$, proceeding through indices $\ell = \pi + 1, \pi + 2, \dots, \pi - 1$ with $L_\ell = s_\ell G + c_\ell P_\ell$ and $R_\ell = s_\ell H_\ell + c_\ell J$. The partial signature $\widehat{\sigma} = (c_1, \{s_\ell\}_{\ell \in [r] \setminus \pi})$ can be stored until later.

Complete signature: Alice computes $s_{1,\pi} = u_1 - c_\pi \beta_1 b_1$. Bob computes $s_{2,\pi} = u_2 - c_\pi \beta_2 b_2$. Alice sends $s_{1,\pi}$ to Bob and Bob sends $s_{2,\pi}$ to Alice. Either can compute $s_\pi = s_{1,\pi} + s_{2,\pi}$ and publish the completed signature $\sigma = (c_1, \underline{s})$ with the modified message $\mathfrak{m}^*$.

Observe that signature challenges have the ring member P_ℓ in their pre-image (compare to Section 3.1); this forces oracle queries to occur in a safe order in our security proofs.

4 Linkable thring signatures and an implementation

Definition 4.0.1. A *Linkable Thring Signature* is a quadruple of collaboratively computed polynomial-time algorithms $(\texttt{KeyGen}, \texttt{Sign}, \texttt{Ver}, \texttt{Link})$. We neglect notation for the common input η, the security parameter in our description:

1. $\texttt{KeyGen}$ produces as output a new random $x \xleftarrow{\$} \mathbb{Z}_{\mathfrak{p}}$, computes $X = xG$, and outputs $\texttt{out}_{\texttt{KeyGen}} = (x, X)$.

2. `Sign` is a multi-party algorithm executed by participants with private keys $\underline{x} = \{x_i\}_{i=1}^n$. Each participant uses their key x_i as private input and all participants use some shared input $\texttt{inp}_{\texttt{Sign}} = (\mathfrak{m}, \underline{P}, \pi)$ where $\mathfrak{m} \in \{0,1\}^*$, $\underline{P} = \{P_i\}_{i=1}^r$ is a ring of public keys, and π is a secret index satisfying $1 \leq \pi \leq r$. `Sign` outputs either a distinguished failure symbol $\texttt{out}_{\texttt{Sign}} = \perp_{\texttt{Sign}}$ to each participant or some $\texttt{out}_{\texttt{Sign}} = (\mathfrak{m}^*, \sigma)$ to each participant where σ is a ring signature and $\mathfrak{m}^* = (\mathfrak{m}, \underline{P}, J, \texttt{aux}_{\texttt{Sign}})$ for a linkability tag J and some auxiliary data $\texttt{aux}_{\texttt{Sign}}$.

3. Ver takes as input some $\texttt{inp}_{\texttt{Ver}} = (\mathfrak{m}^*, \sigma)$ and outputs a bit $\texttt{out}_{\texttt{Ver}} = b \in \{0,1\}$.

4. Link takes as input some $\texttt{inp}_{\texttt{Link}} = (\mathfrak{m}_1^*, \sigma_1)$, $(\mathfrak{m}_2^*, \sigma_2)$ and outputs a bit $\texttt{out}_{\texttt{Link}} = b \in \{0,1\}$.

We include the auxiliary data in the modified message to allow, for example, packing of a recipients' keys into $\mathfrak{m}^*$, although we do not make use of `aux` directly.

Definition 4.0.2 (Correctness)**.** For any $\mathfrak{m}^* \in \{0,1\}^*$ and any $\sigma \in \mathbb{Z}_{\mathfrak{p}}^*$, denote the event that $\texttt{Ver}(\mathfrak{m}^*, \sigma) = 1$ as $V(\mathfrak{m}^*, \sigma)$. For any $((\mathfrak{m}, \underline{P}, \pi, \underline{x}), (\mathfrak{m}^*, \sigma))$, denote the event that $\texttt{Sign}(\mathfrak{m}, \underline{P}, \pi, \underline{x}) = (\mathfrak{m}^*, \sigma)$ and $P_\pi = \Phi(\underline{X})$ with $S\left((\mathfrak{m}, \underline{P}, \pi, \underline{x}), (\mathfrak{m}^*, \sigma)\right)$. We say a linkable thring signature scheme is *correct* when, by measuring the probability over all input coins and all choices of hash functions, $\mathbb{P}\left[V(\mathfrak{m}^*, \sigma) \mid S\left((\mathfrak{m}, \underline{P}, \pi, \underline{x}), (\mathfrak{m}^*, \sigma)\right)\right] = 1$ for any $(\mathfrak{m}, \underline{P}, \pi, \underline{x})$.

Definition 4.0.3 (Linkability)**.** For any pair of signatures, $(\mathfrak{m}_1^*, \sigma_1)$, and $(\mathfrak{m}_2^*, \sigma_2)$, denote the event that $\texttt{Link}(\mathfrak{m}_1^*, \sigma_1, \mathfrak{m}_2^*, \sigma_2) = 1$ as $L(\mathfrak{m}_1^*, \sigma_1, \mathfrak{m}_2^*, \sigma_2)$. We define the subevent $S'(\mathfrak{m}_1^*, \sigma_1, \mathfrak{m}_2^*, \sigma_2) \subseteq V(\mathfrak{m}_1^*, \sigma_1) \cap V(\mathfrak{m}_2^*, \sigma_2)$ as the event that there exists some $\underline{x}$, some messages $\mathfrak{m}_1$, $\mathfrak{m}_2$, some rings $\underline{P}_1, \underline{P}_2$, and some indices π_1, π_2 satisfying both $\texttt{Sign}(\mathfrak{m}_1, \underline{P}_1, \pi_1, \underline{x}) = (\mathfrak{m}_1^*, \sigma_1)$ and $\texttt{Sign}(\mathfrak{m}_2, \underline{P}_2, \pi_2, \underline{x}) = (\mathfrak{m}_2^*, \sigma_2)$. We say a linkable thring signature scheme is *linkable* when, for any $(\mathfrak{m}_1^*, \sigma_1, \mathfrak{m}_2^*, \sigma_2)$,

$$\mathbb{P}\left[L(\mathfrak{m}_1^*, \sigma_1, \mathfrak{m}_2^*, \sigma_2) \mid S'(\mathfrak{m}_1^*, \sigma_1, \mathfrak{m}_2^*, \sigma_2)\right] = 1,$$

where this probability is measured over all participants' coins and all choices of hash functions.

Example 4.0.4 (LSTAGs)**.** We present a linkable thring signature scheme inspired by the LSAG signatures of [13]; we say this scheme is a *linkable spontaneous threshold anonymous group* signature, or an LSTAG signature.

We aggregate keys using the Musig approach by setting $\Phi(\underline{X}) = \sum_{i \in [n]} \beta_i X_i$ where $\beta_i = \phi(x_i, \underline{X}) = \mathcal{H}_{\text{agg}}(X_i, \underline{X})$. To ensure each participant computes keys in a consistent way, we assume users have, during some set-up phase, decided upon a canonical linear ordering of keys in $\underline{X}$ such as a bit-by-bit little endian lexicographic ordering. The i^{th} member has a share of the private key $x_i^* = \beta_i x_i = \phi(x_i, \underline{X}) x_i$. The key image is computed as is usual in Monero: for any private key $x \in \mathbb{Z}_{\mathfrak{p}}$, the key image is $x\mathcal{H}_{\text{ki}}(xG)$, so the key image of X_{sh} is exactly $\sum_i \beta_i x_i \mathcal{H}_{\text{ki}}(X_{\text{sh}})$.

1. `KeyGen` selects some $x \in \mathbb{Z}_{\mathfrak{p}}$ at random, computes $X := xG$, and outputs (x, X).

2. `Sign` is initiated by side channel when the group agrees upon a message $\mathfrak{m}$, decides upon a ring $\underline{P}$ and a secret index π, pre-computes the key image basepoints for each ring member, $H_\ell := \mathcal{H}_{\text{ki}}(P_\ell)$, computes the key image $J = \sum_j J_j = \sum_j x_j^* H_\pi$, assembles the modified message $\mathfrak{m}^* = (\mathfrak{m}, \underline{P}, J, \texttt{aux})$, and computes $M = \mathcal{H}_{\text{msg}}(\mathfrak{m}^*)$. The remainder is run collaboratively:

   Commit: Each signer, say with index j such that $1 \leq j \leq n$, does the following:

(a) Selects a random scalar u_j. Compute the points $U_j = u_j G$ and $V_j = u_j H_\pi$ and select random scalars $s_{\pi+1,j}, s_{\pi+2,j}, \ldots, s_{\pi-1,j}$.
(b) Set $\mathtt{dat}_j := (U_j, V_j, \{s_{\ell,j}\}_{\ell \neq \pi})$.
(c) Compute commitment $\mathtt{com}_j = \mathcal{H}_{\text{com}}(\mathtt{dat}_j)$.
(d) Send $\mathtt{com}_j$ to all other signers.

Reveal: After receiving each $\mathtt{com}_j$ from the rest of the coalition, each signer indexed as before does the following:

(a) Send $\mathtt{dat}_j$ to all other signers.
(b) After all $\mathtt{dat}_{j'}$ have been received, verify that $\mathcal{H}_{\text{com}}(\mathtt{dat}_{j'}) = \mathtt{com}_{j'}$ for each $j \neq j'$. If not all commitments open appropriately, output $\perp_{\mathtt{Sign}}$ and terminate.

Offline signature pre-processing Each signer indexed as before does the following:

(a) Compute $L_\pi = \sum_j U_j$, $R_\pi = \sum_j V_j$.
(b) Compute each $s_\ell = \sum_j s_{\ell,j}$ for each $1 \leq \ell \leq r$ such that $\ell \neq \pi$.
(c) For each $\ell = \pi, \pi+1, \ldots, \pi-1$ (where we re-assign overflow indices by mapping $r+1 \mapsto 1$, $r+2 \mapsto 2$, and so on), compute the following.

$$\begin{aligned} c_{\ell+1} =& \mathcal{H}_{\text{sig}}(M, P_\ell, L_\ell, R_\ell) \\ L_{\ell+1} =& s_{\ell+1} G + c_{\ell+1} P_{\ell+1} \\ R_{\ell+1} =& s_{\ell+1} H_{\ell+1} + c_{\ell+1} J \end{aligned}$$

(d) Store the pre-processed signature $(u_j, \mathfrak{m}^*, c_1, c_\pi, \{s_\ell\}_{\ell \neq \pi})$ for later.

Signature completion To finish signing with $(u_j, \mathfrak{m}^*, c_1, c_\pi, \{s_\ell\}_{\ell \neq \pi})$, each signer indexed as before does the following:

(a) Compute $s_{\pi,j} = u_j - c_\pi x_j^*$.
(b) Send $s_{\pi,j}$ to the other signers.
(c) After receiving all $\{s_{\pi,j'}\}_{j \neq j'}$, compute $s_\pi = \sum_j s_{\pi,j}$.
(d) Output $(\mathfrak{m}^*, \sigma)$ where $\sigma = (c_1, \underline{s})$ where $\underline{s} = (s_1, s_2, \ldots, s_r)$.

3. `Ver` takes as input some $(\mathfrak{m}^*, \sigma)$.

(a) Parse $\mathfrak{m}^* = (\mathfrak{m}, \underline{P}, J, \mathtt{aux})$ and $\sigma = (c_1, \underline{s})$.
(b) For $\ell = 1, 2, \ldots, r-1$, compute $L_\ell = s_\ell G + c_\ell P_\ell$, $R_\ell = s_\ell H_\ell + c_\ell J$, and $c_{\ell+1} = \mathcal{H}_{\text{sig}}(M, P_\ell, L_\ell, R_\ell)$.
(c) Compute $c_1' = \mathcal{H}_{\text{sig}}(M, P_r, L_r, R_r)$.
(d) Output 1 if $c_1' = c_1$ and 0 otherwise.

4. `Link` operates just like a Back LSAG signature: check if key images J match.

5 Unforgeability and thring signatures

5.1 Defining a forgery

What, exactly, does it mean to be a forger, or to present a forgery? A forgery should be some $(\mathfrak{m}^*, \sigma)$ such that σ is not an output in the transcript of queries made by $\mathcal{A}$ to $\mathcal{SO}$ and $\mathtt{Ver}(\mathfrak{m}^*, \sigma) = 1$. However, this is not enough. Indeed, if none of the keys in $\underline{P}$ are aggregated, a forgery of our scheme reduces to forgery of the underlying LSAG scheme; without loss of generality, a successful forgery should have at least one aggregate key in $\underline{P}$. Additionally, even if some key is aggregate, the forger could simply place their own key

in $\underline{P}$ along with the aggregate key. So, if the forger knows the discrete logarithm of any public key in some ring $\underline{P}$, then the forger can simply produce an honest signature with $\underline{P}$, which cannot count as a forgery. Without loss of generality, all ring members must either be an honest key (i.e. a discrete log challenge) or be a child of an honest key. Since the most powerful adversary has corrupted all keys except one, we assume only one target honest key X_h.

Hence, we modify the usual unforgeability game: given X_h and ring $\underline{P}$, a forgery is only successful if it comes equipped with evidence that every ring member is either X_h itself or a child of X_h, and that $\underline{P}$ has least one child of X_h. The forger can simply produce evidence of these relationships by presenting the aggregating sets $\left\{\underline{X}^{(\ell)}\right\}_{\ell\in[r]}$ such that $X_h \in \underline{X}^{(\ell)}$ and $P_\ell = \Phi(\underline{X}^{(\ell)})$ for each $1 \le \ell \le r$. In the following, a forger is a PPT algorithm $\mathcal{A}$ that takes as input some X_h and produces as output a distinguished failure symbol $\perp_{\mathcal{A}}$ or a successful forgery $\mathtt{out}_{\mathcal{A}} = \mathtt{forg} = (\mathfrak{m}^*, \sigma, \left\{\underline{X}^{(\ell)}\right\}_{\ell\in[r]})$.

Definition 5.1.1 (Existential Unforgeability for LSTAGs)**.** We say a PPT algorithm $\mathcal{A}$ is a (t, ϵ, q, n)-forger if, within time at most t and with at most q oracle queries, $\mathcal{A}$ can succeed at the following game with probability at least ϵ.

1. The challenger picks an honest key pair $(x_h, X_h) \leftarrow \mathtt{KeyGen}$ and sends the public key X_h to $\mathcal{A}$.

2. $\mathcal{A}$ can generate keys, can aggregate any group elements with X_h, and is granted access to a signing oracle $\mathcal{SO}$ and the random oracles $\mathcal{H}_{\text{agg}}$, $\mathcal{H}_{\text{sig}}$, $\mathcal{H}_{\text{com}}$, $\mathcal{H}_{\text{msg}}$, and $\mathcal{H}_{\text{ki}}$. $\mathcal{A}$ can perform any of these in any order, adaptively responding to previous results.

3. $\mathcal{A}$ outputs some $(\mathfrak{m}^*, \sigma, \left\{\underline{X}^{(\ell)}\right\}_{\ell=1}^{r})$.

4. $\mathcal{A}$ wins if all the following conditions are satisfied.

   Correct: For each ℓ, $P_\ell = \Phi(\underline{X}^{(\ell)})$.

   Bounded: For each ℓ, $1 \le \left|\underline{X}^{(\ell)}\right| \le n$.

   Honest parent: For each ℓ, $X_h \in \underline{X}^{(\ell)}$.

   Aggregated: For some ℓ, $\left|\underline{X}^{(\ell)}\right| \ge 2$ (so at least one key is aggregated).

   Non-trivial: σ is not an output in the transcript between $\mathcal{A}$ and $\mathcal{SO}$; and of course

   Valid: $\mathtt{Ver}(\mathfrak{m}^*, \sigma) = 1$.

While it seems not realistic for the adversary to present evidence of their forgery, we note that if a forger is placed in a black box by some master algorithm, the key aggregation queries made by the forger must be simulated by or also made by the master algorithm. Hence, evidence of these relationships are extractable from the transcript resulting any successful forgery.

5.2 Strategy for proving unforgeability

Suppose $\mathcal{B}$ is a meta-reduction of $\mathcal{A}$ that produces, for some fixed message $\mathfrak{m}$, four forged signatures σ, σ', σ'', σ''' with rings $\underline{P}$, $\underline{P}'$, $\underline{P}''$, $\underline{P}'''$ with family histories $\underline{X} = \left\{\underline{X}^{(\ell)}\right\}_{\ell\in[r]}$, $\underline{X}' = \left\{(\underline{X}^{(\ell)})'\right\}_{\ell\in[r']}$, $\underline{X}'' = \left\{(\underline{X}^{(\ell)})''\right\}_{\ell\in[r'']}$, and $\underline{X}''' = \left\{(\underline{X}^{(\ell)})'''\right\}_{\ell\in[r''']}$. Suppose

furthermore that $\mathcal{B}$ can extract from the oracles made by $\mathcal{A}$ a distinguished index ℓ such that

$$L_\ell = L'_\ell, L''_\ell = L'''_\ell, P_\ell = P'_\ell, P''_\ell = P'''_\ell.$$

Then $\mathcal{B}$ can compute the discrete logarithm of P_ℓ as $(c_\ell - c'_\ell)^{-1}(s'_\ell - s_\ell)$ and the discrete logarithm of P''_ℓ as $(c''_\ell - c'''_\ell)^{-1}(s'''_\ell - s''_\ell)$. However, the keys $P_\ell = P'_\ell$ are aggregated from $\underline{X}$ and $\underline{X}'$, respectively. Due to this, we can write $P_\ell = P'_\ell = \alpha X_h + Z$ for some α and for some Z using the key aggregation function Φ. Similarly $P''_\ell = P'''_\ell$ and can be written as $\alpha' X_h + Z'$ for some α', Z'.

If $\mathcal{B}$ can ensure that $Z = Z'$ and $\alpha \neq \alpha'$, then $P_\ell - P''_\ell = (\alpha - \alpha')X_h$ and so $\mathcal{B}$ can obtain the discrete logarithm of X_h:

$$x_h = (\alpha - \alpha')^{-1} \left(\frac{s'_\ell - s_\ell}{c_\ell - c'_\ell} - \frac{s'''_\ell - s''_\ell}{c''_\ell - c'''_\ell}. \right)$$

Hence, to demonstrate the absurdity of the existence of a forger $\mathcal{A}$, it is sufficient to demonstrate the existence of a meta-reduction that can produce four transcripts such that the following is extractable from those transcripts: an index $1 \leq \ell$, some public keys $(P_\ell, P'_\ell, P''_\ell, P'''_\ell, L_\ell, L'_\ell, L''_\ell, L'''_\ell, Z, Z')$, and some scalars $(s_\ell, s'_\ell, s''_\ell, s'''_\ell, c_\ell, c'_\ell, c''_\ell, c'''_\ell, \alpha, \alpha')$ such that

$$\begin{aligned}
L_\ell &= s_\ell G + c_\ell P_\ell & L'_\ell &= s'_\ell G + c'_\ell P'_\ell \\
L''_\ell &= s''_\ell G + c''_\ell P''_\ell & L'''_\ell &= s'''_\ell G + c'''_\ell P'''_\ell \\
L_\ell &= L'_\ell & L''_\ell &= L'''_\ell \\
P_\ell &= P'_\ell & P''_\ell &= P'''_\ell \\
P_\ell &= \alpha X_h + Z & P''_\ell &= \alpha' X_h + Z'
\end{aligned}$$

and such that $c''_\ell \neq c'''_\ell$, $c_\ell \neq c'_\ell$, and $\alpha \neq \alpha'$. We say this is the *system of forgery-to-discrete-log equations and inequalities*.

We do this in Appendix A with careful construction of oracles and by forking twice: the first time upon the very first signature verification query of the form $\mathcal{H}(M, P_\ell, L_\ell, R_\ell)$, ensuring that $L_\ell = L'_\ell$ and $L''_\ell = L'''_\ell$, and the second time upon the computation of key aggregation coefficients, ensuring that $Z_\ell = Z'_\ell = Z$ and $\alpha \neq \alpha'$.

6 Use and abuse of applications and implementations

In this section, we discuss some implementation considerations for thring signature schemes, their extensions, and their applications in ring confidential transactions.

6.1 Danger in non-random or repeated signing

The protocol is dangerous if data is not randomly generated for each attempted signature or if more than one signature per key is ever published. If the same signature data $u_{j,\pi}$ is used by a participating member twice, they risk revealing their private keys: the equalities $s = u - cb^*$ and $s' = u - c'b^*$ can be used to compute $b^* = \frac{s-s'}{c'-c}$. Hence, any non-random method of selecting signing data should never be used. Similarly, two signatures with the same session key should never be provided, as the discrete logarithm of the signing key can be extracted.

6.2 Group property considerations

Also note that it is perfectly safe to use a group $\mathbb{G}$ with *composite order* instead of prime order. However, we must restrict our choices of public key to a specific prime-order subgroup

of $\mathbb{G}$. Using the so-called Ed25519 curve (which is a twisted Edwards curve presented in [6] that is birationally equivalent to the so-called Curve25519 curve presented in [5]), there are some implementation risks in selecting a public key outside of the prime-order subgroup.

Indeed, the prime-order subgroup has cofactor 8, introducing the possibility of malleability exploitations. To prevent these exploitations, all implementations with this curve require checking that group elements for use as public keys lie on the prime order subgroup by checking their order. Certainly any private key $x \in \mathbb{Z}_{\mathfrak{p}}$ will have a corresponding public key $X = xG$ on this prime-order subgroup since G is a generator of that subgroup. On the other hand, selecting a public key at random does not guarantee a discrete logarithm pre-image with respect to G. We must modify the hash-to-point function $\mathcal{H}_{\text{ki}}$ to have a codomain equal to the prime-order subgroup. This means multiplying public keys without corresponding private keys by the cofactor 8 when using the Ed25519 or Curve25519 curves.

6.3 View key extension and thring confidential transactions

Among other reasons, our thring signatures are not directly comparable to signatures employed in digital currencies like the MLSAG signatures used in protocols like Cryptonote. Indeed, Cryptonote user keys come in pairs with a view key and a spend key, and signatures are computed with one-time keys derived from these. The thresholdizing heuristic described in Section 3.2 extends naturally to a system with this one-time key extension, but our proofs of security properties do not immediately apply to these extensions without some further work.Establishing the unforgeability of an implementation of LSTAG thring signatures using Cryptonote-styled one-time keys with a view key extension remains an open task.

One model that may be helpful in proving the unforgeability of a view key extension of our implementation below is for collaborators to each receive a common shared secret $y = a_{\text{sh}}$ and use keys $\underline{x} = (x_1, \ldots, x_n)$ to compute a shared key of the form $Y + \Phi(\underline{X})$. Rogue-key or key cancellation attacks using this method may be mitigated by careful construction of y. For example, in Cryptonote protocols, tampering with Y requires tampering with the hash digest of a key from a Diffie-Hellman exchange.

Similarly, our heuristic also extends naturally to the MLSAG setting, but our security proofs again do not immediately apply without some further work. We briefly describe ring confidential transactions in Monero and a thresholdized extension to thring confidential transactions.

MLSAG signatures are constructed from vectors of keys of the form $(\mathcal{H}_{\text{sess}}(aT)G + B, \texttt{PedCom}(v, r))$ where `PedCom` is a Pedersen commitment scheme, v is a transaction amount, and r is a private amount-commitment key. MLSAG signatures, by construction, are multisignatures since they use multiple keys to sign a message but they are not 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 public keys $\underline{P} = (P_1, \ldots, P_n)$ interpreted as a vector, the signer (or signers) randomly selects similar vectors from the blockchain to construct a ring of such key vectors $\widetilde{\underline{P}}$, packing $\underline{P}$ into the π^{th} column for a secret π.

$$\widetilde{\underline{P}} = \begin{pmatrix} P_{1,1} & P_{1,2} & \cdots & P_{1,r} \\ P_{2,1} & P_{2,2} & \cdots & P_{2,r} \\ \vdots & & & \vdots \\ P_{n,1} & P_{n,2} & \cdots & P_{n,r} \end{pmatrix} = (\underline{P}_1, \ldots, \underline{P}_r)$$

where each $\underline{P}_\ell = \{P_{j,\ell}\}_{j \in [n]}$. For each $j = 1, 2, \ldots, n$ and $\ell = 1, 2, \ldots, r$, with the entry $P_{j,\ell}$ in $\widetilde{\underline{P}}$ we compute $H_{j,\ell} := \mathcal{H}_{\text{ki}}(P_{j,\ell})$. For each component $p_j \in \underline{p}$, the signer computes key image $J_j = p_j \mathcal{H}_{\text{ki}}(p_j G)$. The signer selects a vector of random scalars $\underline{u} = (u_1, \ldots, u_n) \in \mathbb{Z}_{\mathfrak{p}}^n$ for the π^{th} column and, for each $\ell \neq \pi$, the signer selects a vector

of random scalars $\underline{s}_\ell = (s_{1,\ell}, s_{2,\ell}, \dots, s_{n,\ell})$. Now, for each signing key $P_{j,\ell}$, the signer computes the pair of points and the commitment

$$L_{j,\ell} = s_{j,\ell}G + c_\ell P_{j,\ell}, \qquad R_{j,\ell} = s_{j,\ell}H_{j,\ell} + c_\ell J_j, \qquad c_{\ell+1} = \mathcal{H}_{\text{sig}}\left(M, \left\{(L_\ell^j, R_\ell^j)\right\}_{j=1}^{n}\right).$$

Once each commitment has been computed, the signer computes each $s_{j,\pi} = u_j - c_\pi p_j$ as usual, assembles $\underline{s}_\pi = (s_{j,\pi})_{j\in[n]}$, and the MLSAG signature is then $\sigma = (c_1, (\underline{s}_\ell)_{\ell\in[r]})$, which is verified similarly to LSAG signatures.

Initially, this scheme may seem to not be particularly ambiguous with respect to signer identification. 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. Rather than a drawback, this is how Monero links signing keys with transaction amounts. In Monero, the first row of keys in $\underline{\widetilde{P}}$ are signing keys, the second row consists of the differences between Pedersen commitments to transaction input amounts and output amounts. Signing with this matrix means both knowing the private signing key for the special index and being able to open the amount commitment at that index to zero.

To see how our thresholdizing heuristic extends naturally to the MLSAG setting, consider the following. For a collaborating coalition of signers, presume that each participating signer has a share of a key, which are aggregated in the Musig style, and each participant contributes some random scalars to be summed for u_j, $s_{j,\ell}$, etc, in a commit-and-reveal stage. Formally establishing the unforgeability of such an implementation of MLSTAG thring confidential transactions using Cryptonote-styled one-time keys with a view key extension also remains an open task. We leave the task of presenting more general formal definitions and implementations of thring confidential transactions and their security properties for future works, such as the upcoming work of [11].

6.3.1 Extending to m-of-n

Section 3.2 presents a simple 2-of-2 example, which extends naturally to n-of-n. With a Diffie-Hellman exchange, we may extend the above approach to an $(n-1)$-of-n threshold signature scheme in the following way: participants share their B_j with each other and compute pairwise shared secrets $z_{i,j} = \mathcal{H}_{\text{agg}}(b_i B_j)$. There are $\frac{n(n-1)}{2}$ 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. More general approaches for m-of-n are obviously available, and we go no further describing them here.

6.4 Cross-chain, confidential, spender-ambiguous atomic swaps

Simple cross-chain atomic swaps using thring signatures are possible. If all goes well, the swap goes like this: Alice sends x AliceCoins to the 2-of-2 key on the AliceCoin chain, Bob sends y BobCoins to the 2-of-2 key on the BobCoin chain, and once both parties are satisfied, they can collaborate to claim their funds. Of course, we cannot assume all goes well; as described in [2], refund transactions allow for semi-honest parties to halt the process in an adversarial environment. All that remains to complete a spender-ambiguous model of the cross-chain atomic swaps from [2] for a digital currency using confidential transactions is to formalize refund transaction capabilities for that currency (c.f. [17]).

References

[1] A. Back. Ring signature efficiency. https://bitcointalk.org/index.php?topic=%20972541.msg10619684#msg10619684, 2015. Accessed: 16-03-2018.

[2] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, 2014.

[3] 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.

[4] 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.

[5] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[6] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[7] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. Cryptology ePrint Archive, Report 1990/001, 2018. https://eprint.iacr.org/2018/483.

[8] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter MR Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. Cryptology ePrint Archive, Report 1990/001, 2017. https://eprint.iacr.org/2017/956.pdf.

[9] 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.

[10] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, and Gregory Neven. Okamoto beats schnorr: On the provable security of multi-signatures. Technical report, IACR Cryptology ePrint Archive, Report 2018/417, 2018. Available at http://eprint. iacr. org/2018/417, 2018.

[11] Russel WF Lai, Viktoria Ronge, Tim Ruffing, Dominique Schröder, Sri Aravinda Krishnan Thyagarajan, and Jiafan Wang. Foundations of ring confidential transactions. *NOT YET ANNOUNCED*, page NOT YET ANNOUNCED, 2018.

[12] 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.

[13] 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.

[14] Gregory Maxwell. Confidential transactions. *URL: https://people. xiph. org/˜ greg/confidential_values. txt (Accessed 08/24/2018)*, 2015.

[15] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. Cryptology ePrint Archive, Report 1990/001, 2018. https://eprint.iacr.org/2018/068.pdf.

[16] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 397–411. IEEE, 2013.

[17] Sarang Noether and BG Goodell. Dual-output ring signatures: Spender-ambiguous cross-chain confidential atomic swaps. *In prep.*, 2018.

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

[19] Tatsuaki Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. In *Annual International Cryptology Conference*, pages 31–53. Springer, 1992.

[20] Haifeng Qian and Shouhuai Xu. Non-interactive multisignatures in the plain public-key model with efficient verification. *Information Processing Letters*, 111(2):82–89, 2010.

[21] Thomas Ristenpart and Scott Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 228–245. Springer, 2007.

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

[23] 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.

A Security

In this Appendix, we prove the following theorem based on the game of Definition 5.1.1. In Appendix B, we discuss some other security properties of thring signatures and their LSTAG implementation. In A.1 we detail the signing oracle of Definition 5.1.1 and its simulation. In A.2, we review the rewind-on-success forking lemma. In A.3, we discuss the first reduction of the forging adversary and establish some probability bounds. In A.4, we explain our twice-forking approach, concluding the proof of the following theorem.

Theorem A.0.1. *Let $\mathcal{A}$ be a (t, ϵ, q, n)-forger for some non-negligible ϵ. There exists some $t' > 0$ and an algorithm $\mathcal{B}$ that is a $(t + t', \epsilon', q)$-solver of the discrete logarithm problem for some non-negligible ϵ'.*

A.1 The signing oracle

In this section, we explain the signing oracle for use in the unforgeability game of Definition 5.1.1.

The signing oracle in the unforgeability game captures the situation where an adversary $\mathcal{A}$ can persuade an honest party to sign some documents before attempting a forgery. In the multisignature case, we must also capture the implications of a malicious $\mathcal{A}$ persuading an honest party to collaborate to construct some multisignatures on similar documents. Due to this collaboration, $\mathcal{A}$ has some control over signing data so $\mathcal{SO}$ must be interactive. In an honest collaboration, $\mathcal{A}$ would also know which index π corresponds with the true signer, so we allow the signing oracle to be queried with the special signing index π.

Moreover, a reduction $\mathcal{A}'$ of $\mathcal{A}$ must simulate interactions between $\mathcal{A}$ and any oracles, in particular the signing oracle.

$\mathcal{SO}$ takes as input some $\mathtt{inp}_{\mathcal{SO}} = (\mathtt{m}, \underline{P}, \pi, \left\{\underline{X}^{(\ell)}\right\}_{\ell\in[r]})$ where $\mathtt{m} \in \{0,1\}^*$ is a message, $\underline{P}$ is a ring of public keys, π is a special index, and $\left\{\underline{X}^{(\ell)}\right\}_{\ell\in[r]}$ is a a multi-set of multi-sets of public keys. $\mathcal{SO}$ and $\mathcal{A}$ interact. $\mathcal{SO}$ outputs a distinguished failure symbol $\perp_{\mathcal{SO}}$ or $\mathcal{A}$ successfully simulates collaborating with an honest party to obtain a multi-signature.

The Oracle: $\mathcal{A}$ queries $\mathcal{SO}$ with some $\left(M, \underline{P}, \pi, \left\{\underline{X}^{(\ell)}\right\}_{\ell\in[r]}\right)$.

1. $\mathcal{SO}$ checks if, for each ℓ, $P_\ell = \Phi(\underline{X}^{(\ell)})$ and checks if $X_h \in \underline{X}^{(\ell)}$ and checks that at least one P_ℓ is aggregated. If not, $\mathcal{SO}$ outputs $\perp_{\mathcal{SO}}$ and terminates.
2. Otherwise, $\mathcal{SO}$ selects signing data $\mathtt{dat}_1 = (U_1, V_1, \{s_{1,\ell}\}_{\ell\neq\pi})$, computes the commitment $\mathtt{com}_1 \leftarrow \mathcal{H}_{\mathrm{com}}(\mathtt{dat}_1)$ and send $\mathtt{com}_1$ to $\mathcal{A}$.
3. After $\mathcal{A}$ responds with a set $\{\mathtt{com}_j\}_{j=2}^n$, $\mathcal{SO}$ sends $\mathtt{dat}_1$ to $\mathcal{A}$.
4. After $\mathcal{A}$ responds with $\{\mathtt{dat}_j\}_{j=2}^n$, $\mathcal{SO}$ checks that $\mathtt{com}_j = \mathcal{H}_{\mathrm{com}}(\mathtt{dat}_j)$ for each j. If not, $\mathcal{SO}$ outputs $\perp_{\mathcal{SO}}$ and terminates.
5. $\mathcal{SO}$ completes the off-line signature pre-processing stage, solves for $s_{\pi,j}$, and sends $s_{\pi,j}$ to $\mathcal{A}$.

The symbol $\perp_{\mathcal{SO}}$ only indicates a failed simulation of $\mathcal{SO}$. If $\mathcal{A}$ misbehaves but $\mathcal{SO}$ is successfully simulated, then it is possible that $\mathcal{SO}$ successfully simulates a bad execution of `Sign`. Since $\mathcal{SO}$ was successful, this does not result in $\perp_{\mathcal{SO}}$ but results in an invalid signature or the failure symbol $\perp_{\mathtt{Sign}}$ from `Sign`.

A.2 Rewind-on-success forking lemma

In this section, we explain the double forking technique and present the general forking lemma. Recall that, to prove unforgeability of our scheme, it is sufficient to prove that a forger $\mathcal{A}$ with non-negligible advantage at the existential unforgeability game can be reduced to some $\mathcal{B}$ that produces four forgeries as described above. As usual, we use reductions and meta-reductions of $\mathcal{A}$ with a general forking lemma to complete our task. We denote a reduction of $\mathcal{A}$ with black-box access to $\mathcal{A}$ as $(\mathcal{A}')^{\mathcal{A}}$, and we denote the process of taking reductions with the notation $\mathcal{A} \rightsquigarrow \mathcal{A}'$. Our proof strategy, roughly following the strategy from [15], can be visualized this way:

$$\mathcal{A} \rightsquigarrow \mathcal{A}' \rightsquigarrow \mathtt{fork}^{\mathcal{A}'} \rightsquigarrow \mathcal{A}'' \rightsquigarrow \mathtt{fork}^{\mathcal{A}''} \rightsquigarrow \mathcal{B}.$$

That is to say: our strategy is to prove that if a forger $\mathcal{A}$ exists, then there exists a reduction $\mathcal{A}'$ of $\mathcal{A}$ that satisfies the hypotheses of Lemma A.2.0.1, which can be forked. The forking algorithm can be put in a wrapper $\mathcal{A}''$, which can be again forked. If $\mathcal{A}'$ and $\mathcal{A}''$ have non-negligible acceptance probability, so does $\mathtt{fork}^{\mathcal{A}''}$, which we reduce to some discrete logarithm solver $\mathcal{B}$.

Lemma A.2.0.1 (General Forking Lemma)**.** *Let $q, \eta \geq 1$. Let $\mathcal{P}$ be any PPT algorithm which takes as input some $\mathtt{inp}_{\mathcal{P}} = (\mathtt{inp}, \underline{h})$ where $\underline{h} = (h_1, \ldots, h_q)$ is a sequence of oracle query responses (η-bit strings) and returns as output $\mathtt{out}_{\mathcal{P}}$ either a distinguished failure symbol $\perp$ or a pair $(i, \mathtt{out})$ where $i \in [q]$ and $\mathtt{out}$ is some output. Let $\mathtt{acc}_{\mathcal{P}}$ denote the probability that $\mathcal{P}$ does not output $\perp$ (where this probability is taken over all random coins of $\mathcal{P}$, the distribution of $\mathtt{inp}$, all choices $\underline{h}$).*

There exists an algorithm, $\mathtt{fork}^{\mathcal{P}}$ that takes as input some $\mathtt{inp}_{\mathtt{fork}^{\mathcal{P}}} = \mathtt{inp}_{\mathcal{P}}$ and produces as output $\mathtt{out}_{\mathtt{fork}^{\mathcal{P}}}$ either a distinguished failure symbol $\perp$ or a pair of pairs

$((i, \mathtt{out}), (i', \mathtt{out}'))$, *where* $(i, \mathtt{out})$ *and* $(i', \mathtt{out}')$ *are outputs from* $\mathcal{P}$ *such that* $i = i'$. *Furthermore, the accepting probability of* $\mathtt{fork}^{\mathcal{P}}$ *is bounded from below such that*

$$\mathtt{acc}_{\mathtt{fork}^{\mathcal{P}}} \geq \mathtt{acc}_{\mathcal{P}} \left(\frac{\mathtt{acc}_{\mathcal{P}}}{q} - \frac{1}{2^{\eta}} \right).$$

We refer the reader to [3] for a proof.

Algorithm $\mathtt{fork}^{\mathcal{P}}$: Let $\eta > 1$ be a security parameter, $q > 1$ be a polynomial function of η. Let $\mathcal{P}$ any PPT algorithm satisfying the hypotheses of Lemma A.2.0.1.

The algorithm $\mathtt{fork}^{\mathcal{P}}$ produces as output a distinguished failure symbol $\mathtt{out}_{\mathtt{fork}^{\mathcal{P}}} = \perp$ or some $\mathtt{out}_{\mathtt{fork}^{\mathcal{P}}} = (i', \mathtt{out}')$ where j is an index in $[q]$ and $\mathtt{out}' = (\mathtt{out}, \mathtt{out}'', \mathtt{aux})$ such that $\mathtt{out}$, $\mathtt{out}''$ are two outputs from $\mathcal{P}$.

1. Pick random coins $\rho = \rho_{\mathcal{P}}$ for $\mathcal{P}$ and select $\underline{h} \leftarrow (\{0,1\}^{\eta})^q$.
2. Execute $\mathtt{out}_{\mathcal{P}} \leftarrow \mathcal{P}(\mathtt{inp}_{\mathcal{P}}, \underline{h}; \rho)$.
3. If $\mathtt{out}_{\mathcal{P}} = \perp_{\mathcal{P}}$, output $\perp_{\mathtt{fork}^{\mathcal{P}}}$ and terminate. Otherwise, $\mathtt{out}_{\mathcal{P}} = (i, \mathtt{out})$ for some $\mathtt{out}$.
4. Pick new $\underline{h}' \leftarrow (\{0,1\}^{\eta})^q$.
5. Glue the oracle query sequences together: $\underline{h}^* := (h_1, \ldots, h_{i-1}, h'_i, \ldots, h'_q)$.
6. Execute $\mathtt{out}''_{\mathcal{P}} \leftarrow \mathcal{P}(\mathtt{inp}_{\mathcal{P}}, \underline{h}^*; \rho)$.
7. If $\mathtt{out}''_{\mathcal{P}} = \perp_{\mathcal{P}}$, output $\perp_{\mathtt{fork}^{\mathcal{P}}}$ and terminate. Otherwise, $\mathtt{out}''_{\mathcal{P}} = (i'', \mathtt{out}'')$.
8. If $i \neq i''$ or $i = i''$ but $h_i = h'_i$, output $\perp_{\mathtt{fork}^{\mathcal{P}}}$ and terminate.
9. Otherwise, select some auxiliary data $\mathtt{aux}$, assemble $\mathtt{out}' = (\mathtt{out}, \mathtt{out}'', \mathtt{aux})$, output $(i, \mathtt{out}')$, and terminate.

If $\mathcal{P}$ takes as input several sequences of oracle queries together, we can parse these as to be included in $\mathtt{inp} \in \mathtt{inp}_{\mathcal{P}}$. If $\mathtt{inp}_{\mathcal{P}}$ has some other sequence $\underline{h}^*$ of oracle queries packed into it like this, then $\mathtt{fork}^{\mathcal{P}}$ can be forked again in a chain. In the next section, we describe each algorithm in the chain of reductions comprising our proof strategy $\mathcal{A} \rightsquigarrow \mathcal{A}' \rightsquigarrow \mathtt{fork}^{\mathcal{A}'} \rightsquigarrow \mathcal{A}'' \rightsquigarrow \mathtt{fork}^{\mathcal{A}''} \rightsquigarrow \mathcal{B}$, proving their existence and the non-negligibility of their accepting probability as we go.

A.3 Reducing $\mathcal{A}$

In this section, we begin our forking journey by constructing the first reduction $\mathcal{A}'$ of $\mathcal{A}$ in the chain of reductions. We explain how $\mathcal{A}'$ simulates each oracle, and we establish some lemmata regarding the accepting probability of $\mathcal{A}'$.

We construct a reduction $\mathcal{A}'$ of $\mathcal{A}$ compatible by Lemma A.2.0.1, which applies to an algorithm $\mathcal{P}$ that takes as input some $\mathtt{inp}_{\mathcal{P}} = (\mathtt{inp}, \underline{h})$. $\mathcal{A}'$ takes as input $\mathtt{inp}_{\mathcal{A}'} = (\mathtt{inp}, \underline{h}_{\mathrm{sig}})$. However, $\mathcal{A}'$ must simulate both $\mathcal{H}_{\mathrm{sig}}$ and $\mathcal{H}_{\mathrm{agg}}$ queries, so we define $\mathtt{inp} := (\mathtt{inp}_{\mathcal{A}}, \underline{h}_{\mathrm{agg}})$. That is to say, $\mathcal{A}'$ takes as input $\mathtt{inp}_{\mathcal{A}'} = (\mathtt{inp}_{\mathcal{P}}, \underline{h}_{\mathrm{agg}}) = \big((X_h, \underline{h}_{\mathrm{agg}}), \underline{h}_{\mathrm{sig}}\big)$.

$\mathcal{A}'$ responds to oracle queries with $\underline{h}_{\mathrm{agg}}$ and $\underline{h}_{\mathrm{sig}}$ as described below. $\mathcal{A}'$ augments the output of $\mathcal{A}$, producing as output $\perp$ when $\mathcal{A}$ fails. On the other hand, if $\mathcal{A}$ produces a non-$\perp$ output, say $\mathtt{forg} = \left(\mathtt{m}^*, \sigma, \left\{\underline{X}^{(\ell)}\right\}_{\ell}\right)$, $\mathcal{A}'$ repeats this in its output but augments this with some transcript information, $\mathtt{out}_{\mathcal{A}'} = (i_{\mathrm{sig}}, \mathtt{out}^*)$ where $\mathtt{out}^* = (\mathtt{forg}, i_{\mathrm{sig}}, h_{\mathrm{sig},i_{\mathrm{sig}}}, \ell_{\mathrm{sig}}, \pi_{\mathrm{sig}}, i_{\mathrm{agg}}, h_{\mathrm{agg},i_{\mathrm{agg}}}, \underline{a})$ for some signature and aggregation query indices $i_{\mathrm{sig}}, i_{\mathrm{agg}}$ and responses $h_{\mathrm{sig},i_{\mathrm{sig}}}, h_{\mathrm{agg},i_{\mathrm{agg}}}$, some ring indices $\ell_{\mathrm{sig}}, \pi_{\mathrm{sig}}$, and a list of coefficients $\underline{a}$ extracted from the transcript in the following way:

1. $(i_{\text{sig}}, \ell_{\text{sig}})$ are defined such that the response to first query made by $\mathcal{A}$ to $\mathcal{H}_{\text{sig}}$ to compute any verification equation used in the forgery is the i_{sig}^{th} such query and takes place for the ℓ_{sig}^{th} ring member, i.e. $h_{\text{sig},i_{\text{sig}}} = c_{\ell_{\text{sig}}+1} = \mathcal{H}_{\text{sig}}(M, P_{\ell_{\text{sig}}}, L_{\ell_{\text{sig}}}, R_{\ell_{\text{sig}}})$ for some index ℓ; and

2. π_{sig} is defined such that the response to the final query made by $\mathcal{A}$ to $\mathcal{H}_{\text{sig}}$ to compute any verification equation used in the forgery is used in the π_{sig}^{th} signature challenge, i.e. the final signature challenge to be computed in the transcript is $c_{\pi_{\text{sig}}} = \mathcal{H}_{\text{sig}}(M, P_{\pi_{\text{sig}}-1}, L_{\pi_{\text{sig}}-1}, R_{\pi_{\text{sig}}-1})$; and

3. i_{agg} is the index of the first aggregation query made for any member of $\underline{X}^{(\ell_{\text{sig}})}$, i.e. the aggregation coefficient on X_h in $\underline{X}^{(\ell)}$ is $h_{\text{agg},i_{\text{agg}}}$; and

4. $\underline{a}$ contains the aggregation coefficients of all adversarially selected keys in $\underline{X}^{(\ell_{\text{sig}})}$ (arranged in some canonical manner).

Lemma A.3.0.1. *Assume $\mathcal{A}$ makes at most q random oracle queries and does not output $\perp_{\mathcal{A}}$. Every query made to $\mathcal{H}_{sig}$ for the verification equations is made by $\mathcal{A}$ before terminating except with a probability bounded above by $1-(1-(\mathfrak{p}-q)^{-1})^r$ (conditioned upon the event that $\mathcal{A}$ does not output $\perp_{\mathcal{A}}$).*

Proof. The event that $\mathcal{A}$ does not query $\mathcal{H}_{\text{sig}}$ for some verification query requires that $\mathcal{A}$ guess the output of $\mathcal{H}_{\text{sig}}$ for some query by flipping coins. This occurs with probability at most $(\mathfrak{p}-q)^{-1}$ (the result must avoid the queries already made). The probability of successfully doing this even once for any of the r verification queries is therefore $1-(1-(\mathfrak{p}-q)^{-1})^r$. This is the upper bound on the probability that $\mathcal{A}$ can get away with flipping coins instead of querying for a verification equation. □

As a corollary to this lemma, $\mathcal{A}'$ can find i_{sig}, $h_{\text{sig},i_{\text{sig}}}$, ℓ_{sig}, and π_{sig} easily.

Lemma A.3.0.2. *Assume $\mathcal{A}$ makes at most q random oracle queries and does not output $\perp$. Every query made to $\mathcal{H}_{agg}$ for every aggregation coefficient for X_h is made by $\mathcal{A}$ before terminating, except with a probability bounded above by $1-(1-(\mathfrak{p}-q)^{-1})^{nr}$ (conditioned upon the event that $\mathcal{A}$ does not output $\perp_{\mathcal{A}}$).*

Proof. The event that $\mathcal{A}$ does not make one of these queries is requires that $\mathcal{A}$ guess the output of $\mathcal{H}_{\text{agg}}$. With r ring members, each with some contributing keys $\underline{X}^{(\ell)}$ such that each $\left|\underline{X}^{(\ell)}\right| \leq n$ has at most nr such queries, each with a successful guess probability of $(\mathfrak{p}-q)^{-1}$. The probability of doing this successfully even once in a trial of nr attempts is therefore $1-(1-(\mathfrak{p}-q)^{-1})^{nr}$. □

As a corollary, $\mathcal{A}'$ can extract i_{agg} and $h_{\text{agg},i_{\text{agg}}}$ easily.

Lemma A.3.0.3. *Assume $\mathcal{A}$ is a (t,ϵ,n,q)-forger. Let E be the event that $\mathcal{A}$ does not output $\perp$. In E, for each $1 \leq \ell \leq r$, the query made to $\mathcal{H}_{sig}$ for the verification equation challenge $c_{\ell+1}$ is made after the query to $\mathcal{H}_{agg}$ for the aggregation coefficient on X_h in the associated $\underline{X}^{(\ell)}$, except with probability bounded above by $1-(1-(\mathfrak{p}-q)^{-1})^r$.*

Proof. Since $\Phi(\underline{X}^{(\ell)}) = P_\ell$ is part of the pre-image for $c_{\ell+1}$, the probability that $\mathcal{A}$ can compute $c_{\ell+1}$ before querying to compute P_ℓ (and assuming all queries are made except this final one) is at most $(\mathfrak{p}-q)^{-1}$, and there are at most r such queries relevant to the forgery. □

These lemmata and corollaries demonstrate that all verification queries appear, and each of them appears after the aggregation coefficients have been computed. With these lemmata at hand, we can describe $\mathcal{A}'$. $\mathcal{A}'$ places $\mathcal{A}$ in a black box, simulating all oracle

queries made by $\mathcal{A}$. $\mathcal{A}'$ keeps internal tables denoted with $\mathbb{T}_{\text{sig}}$, $\mathbb{T}_{\texttt{agg}}$ and a counter denoted $\texttt{ctr}$ to keep track of queries made to maintain internal consistency and track oracle query indices.

Algorithm $\mathcal{A}'$: $\mathcal{A}'$ takes as input $\texttt{inp}_{\mathcal{A}'} = ((X_h, \underline{h}_{\text{agg}}), \underline{h}_{\text{sig}})$. $\mathcal{A}'$ has black-box access to $\mathcal{A}$, simulating oracle queries as described below and produces as output either a distinguished failure symbol $\perp$ or some $(\underline{i}, \texttt{out})$.

1. $\mathcal{A}'$ selects random coins $\rho = \rho_{\mathcal{A}}$, sets $\texttt{ctr} := 0$, and sets $\texttt{inp}_{\mathcal{A}} := \{X_h\}$.
2. $\mathcal{A}'$ executes $\mathcal{A}(\texttt{inp}_{\mathcal{A}}; \rho_{\mathcal{A}})$, answering oracle queries made by $\mathcal{A}$ as described below.
3. If $\mathcal{A}$ outputs $\perp_{\mathcal{A}}$, $\mathcal{A}'$ outputs $\perp_{\mathcal{A}'}$ and terminates.
4. Otherwise, $\mathcal{A}$ outputs a forgery $\texttt{forg} = (\mathfrak{m}^*, \sigma, \left\{\underline{X}^{(\ell)}\right\}_\ell)$.
5. $\mathcal{A}'$ finds all verification queries in the transcript and finds the following:
   (a) the query index i_{sig} of the first verification equation used in the forgery, and
   (b) the response $h_{\text{sig},i_{\text{sig}}}$ to that first verification query, and
   (c) the ring index ℓ_{sig} of the input to that first verification query (satisfying the verification equation $c_{\ell_{\text{sig}}+1} = \mathcal{H}(M, P_{\ell_{\text{sig}}}, L_{\ell_{\text{sig}}}, R_{\ell_{\text{sig}}}) = h_{\text{sig},i_{\text{sig}}}$), and
   (d) the response $h_{\text{sig},i'_{\text{sig}}}$ to the final verification query, and
   (e) the ring index $\pi_{\text{sig}} - 1$ of the input to this query, which satisfies a similar verification equation $c_{\pi_{\text{sig}}} = \mathcal{H}(M, P_{\pi_{\text{sig}}-1}, L_{\pi_{\text{sig}}-1}, R_{\pi_{\text{sig}}-1})$, and
   (f) the query index i_{agg} of the first aggregation query for any member of $\underline{X}^{(\ell_{\text{sig}})}$, and
   (g) the response $h_{\text{agg},i_{\text{agg}}}$ to that query, and lastly
   (h) $\mathcal{A}'$ finds all other aggregation coefficients for members of $\underline{X}^{(\ell_{\text{sig}})}$ in $\mathbb{T}_{\texttt{agg}}$, say $\underline{a}$. See description of $\mathcal{H}_{\text{agg}}$ oracle for more information.
6. $\mathcal{A}'$ outputs $\texttt{out}_{\mathcal{A}'} = (i_{\text{sig}}, (\texttt{forg}, i_{\text{sig}}, h_{\text{sig},i_{\text{sig}}}, \ell_{\text{sig}}, \pi_{\text{sig}}, i_{\text{agg}}, h_{\text{agg},i_{\text{agg}}}, \underline{a}))$.

Simulating $\mathcal{H}_{\texttt{com}}$: To simulate queries of the form $\mathcal{H}_{\text{com}}(\texttt{inp})$, $\mathcal{A}'$ keeps track of an internal table $\mathbb{T}_{\text{com}}$. $\mathcal{A}'$ checks if $\mathbb{T}_{\text{com}}[\texttt{inp}]$ is empty. If so, a random $\texttt{out} \xleftarrow{\$} \{0,1\}^\eta$ is selected and stored $\mathbb{T}_{\text{com}}[\texttt{inp}] \leftarrow \texttt{out}$. In either case, $\mathbb{T}_{\text{com}}[\texttt{inp}]$ is sent to $\mathcal{A}$.

Simulating $\mathcal{H}_{\texttt{ki}}$: To simulate queries made by $\mathcal{A}$ of the form $\mathcal{H}_{\text{ki}}(\texttt{inp})$, $\mathcal{A}'$ keeps track of an internal table $\mathbb{T}_{\text{ki}}$. $\mathcal{A}'$ checks if $\mathbb{T}_{\text{ki}}[\texttt{inp}]$ is empty. If so, a random point $Y' \in \mathbb{G}$ is selected and stored $\mathbb{T}_{\text{ki}}[\texttt{inp}] \leftarrow Y'$. In either case, $\mathbb{T}_{\text{ki}}[\texttt{inp}]$ is sent to $\mathcal{A}$.

Simulating $\mathcal{H}_{\texttt{agg}}$: $\mathcal{A}'$ tracks aggregation queries carefully and always ensures that the aggregation coefficient for X_h is selected after all other coefficients.

1. $\mathcal{A}$ queries $\mathcal{H}_{\text{agg}}$ with $\texttt{inp}$.
2. $\mathcal{A}'$ checks if $\mathbb{T}_{\text{agg}}[\texttt{inp}]$ is undefined. If so, $\mathcal{A}$ does the following:
   (a) $\mathcal{A}'$ checks if $\texttt{inp}$ can be parsed as some $(Y, \underline{X})$. If not, $\mathcal{A}'$ picks a random entry $\mathbb{T}_{\texttt{agg}}[\texttt{inp}] \xleftarrow{\$} \mathbb{Z}_{\mathfrak{p}}$.
   (b) Otherwise, $\mathcal{A}'$ parses $\texttt{inp} = (Y, \underline{X})$ and checks if $X_h \in \underline{X}$. If not, then for each $Y' \in \underline{X}$, $\mathcal{A}'$ picks a random entry $\mathbb{T}_{\texttt{agg}}[Y', \underline{X}] \xleftarrow{\$} \mathbb{Z}_{\mathfrak{p}}$.
   (c) Otherwise, $\texttt{inp} = (Y, \underline{X})$ for some $Y \in \underline{X}$ such that $X_h \in \underline{X}$ and yet $\mathbb{T}_{\texttt{agg}}[\texttt{inp}]$ is undefined. $\mathcal{A}'$ checks if $\mathbb{T}_{\texttt{agg}}[X_h, \underline{X}]$ is defined. If so, $\mathcal{A}'$ outputs $\perp_{\text{agg}}$ and terminates

(d) Otherwise, for each $Y' \in \underline{X} \setminus \{X_h\}$ such that $\mathbb{T}_{\mathrm{agg}}[Y', \underline{X}]$ is undefined, $\mathcal{A}'$ selects a random entry $\mathbb{T}_{\mathtt{agg}}[Y', \underline{X}] \xleftarrow{\$} \mathbb{Z}_{\mathtt{p}}$.

(e) After all entries of $\underline{X}$ except X_h have an entry in $\mathbb{T}_{\mathrm{agg}}$, $\mathcal{A}'$ increments $\mathtt{ctr}_{\mathtt{agg}}$ and stores $\mathbb{T}_{\mathtt{agg}}[X_h, \underline{X}] \leftarrow h_{\mathrm{agg},\mathtt{ctr}_{\mathtt{agg}}}$.

3. $\mathcal{A}'$ outputs $\mathbb{T}_{\mathrm{agg}}[\mathtt{inp}]$.

Note that $\mathcal{A}'$ responds with $\perp_{\mathrm{agg}}$ if and only if $\mathbb{T}_{\mathrm{agg}}[X_h, \underline{X}]$ is defined but some $Y \in \underline{X}$ has $\mathbb{T}_{\mathrm{agg}}[Y, \underline{X}]$ defined. If $\mathcal{A}'$ follows the protocol as described, this never happens. In all other cases, the aggregation coefficient for X_h is decided after all other aggregation coefficients have been decided. This is important to our proof of unforgeability.

Simulating $\mathcal{H}_{\mathtt{sig}}$: To simulate queries made by $\mathcal{A}$ for signing, $\mathcal{H}_{\mathrm{sig}}(\mathtt{inp})$, $\mathcal{A}'$ checks if $\mathbb{T}_{\mathtt{sig}}[\mathtt{inp}]$ is defined. If not, $\mathcal{A}'$ checks if $\mathtt{inp}$ can be parsed as $\mathtt{inp} = (M, P, U, V)$ for some M and group points P, U, V. If not, $\mathbb{T}_{\mathtt{sig}}[\mathtt{inp}] \xleftarrow{\$} \{0,1\}^{\eta}$ is selected at random. Otherwise, $\mathtt{ctr}_{\mathtt{sig}}$ is incremented and $\mathbb{T}_{\mathtt{sig}}[M, P, U, V] \leftarrow h_{\mathrm{sig},\mathtt{ctr}_{\mathtt{sig}}}$ is stored. Either way, $\mathcal{A}'$ sends $\mathbb{T}_{\mathtt{sig}}[\mathtt{inp}]$ to $\mathcal{A}$.

Simulating $\mathcal{SO}$: For notational simplicity, $\mathcal{SO}$ uses notations assuming it is the first member of the coalition to begin signing (with index $j = 1$ in the coalition). $\mathcal{A}'$ simulates $\mathcal{SO}$ in the following way:

1. $\mathcal{A}$ queries $\mathcal{SO}$ with $\mathtt{inp}_{\mathcal{SO}}$.
2. After receiving the query, $\mathcal{A}'$ parses $\mathtt{inp}$ as $\left(M, \underline{P}, \pi, \left\{\underline{X}^{(\ell)}\right\}_{1 \le \ell \le r}\right)$ such that $\left|\underline{X}^{(\ell)}\right| \ge 2$ for some ℓ and, for each ℓ, $\left|\underline{X}^{(\ell)}\right| \le n$, $X_h \in \underline{X}^{(\ell)}$, and $P_\ell \in \underline{P}$ is derived from the corresponding $\underline{X}^{(\ell)}$. If $\mathcal{A}'$ cannot parse this way, $\mathcal{A}'$ sends $\perp_{\mathcal{SO}}$ to $\mathcal{A}$ indicate the query is rejected and halts simulating $\mathcal{SO}$.
3. Otherwise, $\mathcal{A}'$ selects signing data $\mathtt{dat}_1$ as in step 2 in Section A.1, computes the commitment $\mathtt{com}_1$, and send $\mathtt{com}_1$ to $\mathcal{A}$ in the following way:
   (a) $\mathcal{A}'$ increments $\mathtt{ctr}$ and picks the critical commitment $c_\pi \leftarrow h_{\mathrm{sig},\mathtt{ctr}_{\mathtt{sig}}}$.
   (b) $\mathcal{A}'$ selects a random set of signing data $\{s_{1,\ell}\}_{\ell \in [r]}$ (including the index $\ell = \pi$).
   (c) $\mathcal{A}'$ computes $L_{1,\pi} = s_{1,\pi}G + c_\pi P_\pi$ and $R_{1,\pi} = s_{1,\pi}H_\pi + c_\pi J$, assembles $\mathtt{dat}_1 := (L_{1,\pi}, R_{1,\pi}, \{s_{1,\ell}\}_{\ell \in [r], \ell \ne \pi})$, and selects a random $\mathtt{com}_1 \xleftarrow{\$} \{0,1\}^{\eta}$.
   (d) If $\mathbb{T}_{\mathtt{com}}[\mathtt{dat}_1]$ is defined, $\mathcal{A}'$ halts simulating $\mathcal{SO}$, outputs $\perp_1$ to indicate that some $\mathtt{dat}_1$ has already been used, and terminate.
   (e) Otherwise, $\mathcal{A}'$ simulates a query of $\mathcal{H}_{\mathrm{com}}$ by setting $\mathbb{T}_{\mathtt{com}}[\mathtt{dat}_1] \leftarrow \mathtt{com}_1$, sends $\mathtt{com}_1$ to $\mathcal{A}$.
4. After $\mathcal{A}$ responds with commitments $\underline{\mathtt{com}} = \{\mathtt{com}_j\}_{j \in [n], j \ne 1}$, $\mathcal{A}'$ does some back-patching and then sends $\mathtt{dat}_1$ to $\mathcal{A}$.
   (a) For each $j > 1$, $\mathcal{A}'$ searches $\mathbb{T}_{\mathtt{com}}$ for any $\mathtt{dat}$ such that $\mathbb{T}_{\mathtt{com}}[\mathtt{dat}] = \mathtt{com}_j$.
   (b) If, for any j, more than one $\mathtt{dat}$ is found, $\mathcal{A}'$ halts simulation of $\mathcal{SO}$, outputs $\perp_2$, and terminates.
   (c) If, for any j, no such $\mathtt{dat}$ is found, then $\mathcal{A}'$ sets $\mathtt{alert}_1 \leftarrow \mathtt{true}$.
   (d) Otherwise, exactly one $\mathtt{dat} = \mathtt{dat}_j$ is found in $\mathbb{T}_{\mathtt{com}}$ for each $\mathtt{com}_j$. If any $\mathtt{dat}_j$ cannot be parsed as $(U_j, V_j, \{s_{j,\ell}\}_{\ell \ne \pi})$, $\mathcal{A}'$ sets $\mathtt{alert}_2 \leftarrow \mathtt{true}$.
   (e) Otherwise, exactly one $\mathtt{dat}_j$ is found in $\mathbb{T}_{\mathtt{com}}$ for each $\mathtt{com}_j$ and can be parsed as $(U_j, V_j, \{s_{j,\ell}\})$. After parsing each $\mathtt{dat}_j$, $\mathcal{A}'$ does the following:

i. $\mathcal{A}'$ computes $U = \sum_j U_j$, $V = \sum_j V_j$, and $s_\ell = \sum_j s_{j,\ell}$ for each $\ell \neq \pi$.
ii. $\mathcal{A}'$ checks if (M, P_ℓ, U, V) appears in $\mathbb{T}_{\mathtt{sig}}$. If so, $\mathcal{A}'$ halts simulating $\mathcal{SO}$, outputs $\perp_3$, and terminates.
iii. Otherwise, $\mathcal{A}'$ increments $\mathtt{ctr}_{\mathtt{sig}}$, sets $c_{\pi+1} \leftarrow h_{\mathrm{sig},\mathtt{ctr}_{\mathtt{sig}}}$, and stores $\mathbb{T}_{\mathtt{sig}}[M, P_\ell, U, V] \leftarrow c_{\pi+1}$.
iv. For each $\ell = \pi+1, \pi+2, \ldots, \pi-2$, $\mathcal{A}'$ computes $L_\ell = s_\ell G + c_\ell P_\ell$ and $R_\ell = s_\ell H_\ell + c_\ell J$ and checks if $(M, P_\ell, L_\ell, R_\ell)$ appears in $\mathbb{T}_{\mathtt{sig}}$. If so, $\mathcal{A}'$ halts simulating $\mathcal{SO}$, outputs $\perp_3$, and terminates.
v. Otherwise, $\mathcal{A}'$ increments $\mathtt{ctr}_{\mathtt{sig}}$, sets $c_{\ell+1} \leftarrow h_{\mathrm{sig},\mathtt{ctr}_{\mathtt{sig}}}$, stores

$$\mathbb{T}_{\mathtt{sig}}[M, P_\ell, L_\ell, R_\ell] \leftarrow c_{\ell+1}$$

and then moves to the next ℓ.
vi. $\mathcal{A}'$ checks that $\mathbb{T}_{\mathtt{sig}}[M, P_{\pi-1}, L_{\pi-1}, R_{\pi-1}]$ is empty. If not, $\mathcal{A}'$ halts simulating $\mathcal{SO}$, outputs $\perp_3$, and terminates.
vii. Otherwise, $\mathbb{T}_{\mathtt{sig}}[M, s_{\pi-1}G + c_{\pi-1}P_{\pi-1}, s_{\pi-1}H_{\pi-1} + c_{\pi-1}J] \leftarrow c_\pi$.

(f) $\mathcal{A}'$ sends $\mathtt{dat}_1$ to $\mathcal{A}$.

5. After $\mathcal{A}$ responds with $\underline{\mathtt{dat}}' = \{\mathtt{dat}'_j\}_{j\neq 1}$, $\mathcal{A}'$ checks $\mathtt{com}_j = \mathbb{T}_{\mathtt{com}}[\mathtt{dat}'_j]$ for each j. If any do not match $\mathcal{A}'$ sends $\perp_{\mathcal{SO}}$ to $\mathcal{A}$ indicating a successful simulation of a failed signing ceremony.
6. Otherwise, if $\mathtt{alert}_1 = \mathtt{true}$ or $\mathtt{alert}_2 = \mathtt{true}$, $\mathcal{A}'$ $\perp_4$ and terminates.
7. Otherwise, $\mathcal{A}'$ sends $s_{1,\pi}$ to $\mathcal{A}$, which is sufficient information for $\mathcal{A}$ to compute the rest of the signature.

If the output is one of the symbols in $\{\perp_1, \perp_2, \perp_3, \perp_4\}$, then $\mathcal{A}'$ actually terminates: these are the failure symbols that indicate something strange is happening to the orders of oracle assignments while simulating $\mathcal{SO}$. This indicates failure of the signing oracle from a malformed query or some other bad ordering of events. We shall prove these only occur with negligible probability.

Moreover, $\perp_{\mathcal{SO}}$ only appears if $\mathcal{A}$ queries $\mathcal{SO}$ with something beyond the scope of the unforgeability game defined in Definition 5.1.1, or sent commitments that did not open correctly; $\mathcal{A}'$ does not terminate because these are successful simulations of a failed signing ceremony, not a failed simulation.

We investigate the acceptance probability of $\mathcal{A}'$.

Lemma A.3.0.4. *Let $\mathcal{A}$ be a (t, ϵ, q, n)-forger with $\mathcal{SO}$ and $\underline{\mathcal{H}}$ access, and let $\mathcal{A}'$ be any reduction of $\mathcal{A}$ that simulates the oracle queries as described above, let $t' > 0$, and let $c > 0$ be the amount of time required for $\mathcal{A}'$ to select a new $\mathbb{Z}_{\mathfrak{p}}$ or $\mathbb{G}$ element at random. Then in time at most $t + t'$ and with probability at most $\epsilon' = \epsilon_1 + \epsilon_2 + \epsilon_3 + \epsilon_4$, $\mathcal{A}'$ terminates without outputting any $\perp_i \in \{\perp_1, \perp_2, \perp_3, \perp_4, \perp_{agg}\}$ where*

$$\epsilon_1 = 1 - \prod_{k\in[q-1]} (1 - k\mathfrak{p}^{-r-1}) \qquad \epsilon_2 = 1 - \prod_{k\in[q-1]} (1 - k\mathfrak{p}^{-1})$$

$$\epsilon_3 = 1 - \prod_{k\in[q-1]} (1 - k\mathfrak{p}^{-4}) \qquad \epsilon_4 = 1 - exp(-\frac{-t'(t'-c)}{2(\mathfrak{p}-q)c^2}).$$

Proof. The failure symbols partition the event that $\mathcal{A}'$ outputs some $\perp$ and terminates, so the probability of any $\perp$ symbol appearing is exactly the sum of each individual one. That is to say, if E is the event that $\mathcal{A}'$ outputs some $\perp$ and terminates and E_i is the event that $\mathcal{A}'$ outputs $\perp_i$ for some $i \in \{1, 2, 3, 4, \mathrm{agg}\}$ and terminates, then we have $\epsilon_i = \mathbb{P}[E_i]$ and, by the law of total probability $\mathbb{P}[E] = \mathbb{P}[E \mid E_{\mathrm{agg}}]\mathbb{P}[E_{\mathrm{agg}}] + \sum_{i\in[4]} \mathbb{P}[E \mid E_i]\mathbb{P}[E_i]$.

Moreover, each conditional probability here is obviously 1 (the probability that $\mathcal{A}'$ outputs some $\perp_i$ given the fact that $\mathcal{A}'$ outputs $\perp_3$ is 1, for example) so we have the simple sum. Hence, bounding the acceptance probability from below is equivalent to bounding each of the probabilities of each of these failure events from above. We immediately simplify the analysis: $\mathbb{P}[E_{\text{agg}}] = 0$ by construction.

To bound ϵ_1: If $\mathcal{A}'$, while simulating a query to $\mathcal{SO}$ for $\mathcal{A}$, selects $\mathtt{dat}_1$ such that $\mathbb{T}_{\mathtt{com}}[\mathtt{dat}_1]$ is non-empty, then $\perp_1$ is output. Hence, this symbol occurs if and only if $\mathcal{A}'$ selects random signing data $\mathtt{dat}_1$ for which an image under $\mathcal{H}_{\text{com}}$ is already determined. $\mathcal{A}'$ never selects the same random data twice by specification of the $\mathcal{SO}$ simulation, so this implies that $\mathcal{A}$ queried $\mathcal{H}_{\text{com}}$ with $\mathtt{dat}_1$ at some point in the past and, moreover, $\mathcal{A}'$ randomly discovered its pre-image. There are $\mathfrak{p}^{r+1}$ choices of $(U_\ell, V_\ell, \{s_\ell\})$; this scenario is precisely the scenario of a usual birthday attack. Presuming that at most q such choices are used in $\mathcal{H}_{\text{com}}$, the probability that $\mathcal{A}'$ sees no collisions is exactly $\prod_{k\in[q-1]}(1 - k\mathfrak{p}^{-r-1})$. Hence we have

$$\epsilon_1 \leq 1 - \prod_{k\in[q-1]} (1 - k\mathfrak{p}^{-r-1}).$$

To bound ϵ_2: The failure symbol $\perp_2$ occurs if and only if at least two $(U, V, \{s_\ell\})$ are found with table entries $\mathbb{T}_{\mathtt{com}}[U, V, \{s_\ell\}] = \mathtt{com}_j$ for the same $\mathtt{com}_j$. This implies a collision of the simulated random oracle. We have

$$\epsilon_2 \leq 1 - \prod_{k\in[q-1]} (1 - k\mathfrak{p}^{-1}).$$

To bound ϵ_3: Let E be the event that $\mathcal{A}'$, in simulating a query made to $\mathcal{SO}$, checks $\mathbb{T}_{\mathtt{sig}}$ and finding that some query of the form $(M, P_\ell, U_\ell, V_\ell)$ has already been made, outputting $\perp_3$. This is also a birthday attack: since M is output from $\mathcal{H}_{\text{msg}}$, we have $\mathfrak{p}^4$ such possible choices, and we make up to q queries, so the probability of seeing no collisions is exactly $\prod_{k\in[q-1]}(1 - k\mathfrak{p}^{-4})$.

To bound ϵ_4: The failure symbol $\perp_4$ is output only in the case that $\mathtt{alert}_1 = \mathtt{true}$ and the simulation has almost come to an end. $\mathcal{A}'$ only gets to this point when $\mathcal{A}$ misbehaves and sends a commitment $\mathtt{com}_j$ in the commit-and-reveal stage that has not yet been associated with any query of the form $\mathcal{H}_{\text{com}}$, and yet still produced opening data $\{\mathtt{dat}_j\}_j$ that pass the reveal phase: $\mathcal{A}$ guessed $\mathtt{com}_j = \mathcal{H}_{\text{msg}}(\text{dat}_j)$ without querying $\mathcal{H}_{\text{msg}}$. This, too, is a birthday attack. The probability that an attacker requires more than k attempts at this before seeing the first collision is bounded from above by $\exp(\frac{-k(k-1)}{2(\mathfrak{p}-q)})$. Assuming each attempt takes constant time (say c units of time per attempt), since $\mathcal{A}'$ is granted $t' > 0$ time in addition to the runtime of $\mathcal{A}$, the probability that $\mathcal{A}'$ outputs $\perp_4$ is at most $1 - \exp(-\frac{-t'(t'-c)}{2(\mathfrak{p}-q)c^2})$.

□

Note that by rescaling time $(t', t) \mapsto (\frac{t'}{c}, \frac{t}{c})$ we can rewrite $\epsilon_4 = 1 - e^{-t'(t'-1)/(\alpha(\mathfrak{p}-q))}$ for some $\alpha > 0$. Also note that each ϵ_i is negligible in $\mathfrak{p}$ so the sum of these are negligible in $\mathfrak{p}$. Since $\mathcal{A}'$ is compatible with the hypotheses of Lemma A.2.0.1, we immediately obtain the following.

Corollary A.3.1. The algorithm $\mathtt{fork}^{\mathcal{A}'}$ and $\mathtt{fork}^{\mathcal{A}''}$ have acceptance probabilities bounded from below:

$$\begin{aligned}\mathtt{acc}_{\mathtt{fork}^{\mathcal{A}'}} \geq & \mathtt{acc}_{\mathcal{A}'}\left(\frac{\mathtt{acc}_{\mathcal{A}'}}{q} - \frac{1}{2^\eta}\right)\\ \mathtt{acc}_{\mathtt{fork}^{\mathcal{A}''}} \geq & \mathtt{acc}_{\mathtt{fork}^{\mathcal{A}'}}\left(\frac{\mathtt{acc}_{\mathtt{fork}^{\mathcal{A}'}}}{q} - \frac{1}{2^\eta}\right)\end{aligned}$$

In the next section we describe $\mathtt{fork}^{\mathcal{A}'}$ and $\mathtt{fork}^{\mathcal{A}''}$.

A.4 Forking twice

In this section we apply the general forking lemma twice and obtain the punchline, a discrete logarithm solver. We fork $\mathcal{A}'$ in two stages. We first construct $\mathtt{fork}^{\mathcal{A}'}$ to take as input some $\mathtt{inp}_{\mathtt{fork}^{\mathcal{A}'}} = (X_h, \underline{h}_{\text{agg}})$, selects some $\underline{h}_{\text{sig}}$ at random, and executes $\mathcal{A}'$ with input $\mathtt{inp}_{\mathcal{A}'} = (X_h, \underline{h}_{\text{sig}})$. If $\mathcal{A}'$ outputs any failure symbol $\perp$, then $\mathtt{fork}^{\mathcal{A}'}$ outputs $\perp_{\mathtt{fork}^{\mathcal{A}'}}$ and terminates.

Otherwise, $\mathcal{A}'$ outputs some $(i_{\text{sig}}, \mathtt{out})$ where $\mathtt{out} = (h_{\text{agg},i_{\text{agg}}}, h_{\text{sig},i_{\text{sig}}}, \underline{a}, \mathtt{forg})$. A second $\underline{h}^*_{\text{sig}}$ is selected at random, the sequences $\underline{h}_{\text{sig}}$ and $\underline{h}^*_{\text{sig}}$ are glued together as usual to get a sequence $\underline{h}'_{\text{sig}}$. $\mathcal{A}'$ is run again except with input $(X_h, \underline{h}'_{\text{sig}})$ in pursuit of a second success. If $\mathcal{A}'$ outputs any failure symbol $\perp$, then $\mathtt{fork}^{\mathcal{A}'}$ outputs $\perp_{\mathtt{fork}^{\mathcal{A}'}}$ and terminates.

Otherwise, $\mathcal{A}'$ comes through with a second success, say $(i^*_{\text{sig}}, \mathtt{out}^*)$. If $i_{\text{sig}} \neq i^*_{\text{sig}}$, $\mathtt{fork}^{\mathcal{A}'}$ outputs $\perp_{\mathtt{fork}^{\mathcal{A}'}}$ and terminates. Otherwise, $\mathtt{fork}^{\mathcal{A}'}$ outputs

$$\mathtt{out}_{\mathtt{fork}^{\mathcal{A}'}} = (i_{\text{sig}}, (i, \mathtt{out}), (i^*, \mathtt{out}^*)).$$

Algorithm $\mathtt{fork}^{\mathcal{A}'}$: Takes as input $\mathtt{inp}_{\mathtt{fork}^{\mathcal{A}'}} = (\mathtt{inp}, \underline{h}_{\text{agg}}) = (X_h, \underline{h}_{\text{agg}})$.

1. $\mathtt{fork}^{\mathcal{A}'}$ picks random coins for $\mathcal{A}'$, selects $\underline{h}_{\text{sig}} \leftarrow (\{0,1\}^\eta)^q$, assembles $\mathtt{inp}_{\mathcal{A}'} = (\mathtt{inp}, \underline{h}_{\text{agg}})$.
2. $\mathtt{fork}^{\mathcal{A}'}$ runs $\mathcal{A}'$ with $\mathtt{inp}_{\mathcal{A}'}$.
3. If $\mathcal{A}'$ outputs $\perp_{\mathcal{A}'}$, $\mathtt{fork}^{\mathcal{A}'}$ outputs $\perp_{\mathtt{fork}^{\mathcal{A}'}}$ and terminates.
4. Otherwise, $\mathcal{A}'$ outputs some $\mathtt{out}_{\mathcal{A}'} = (i_{\text{sig}}, \mathtt{out})$ where
$$\mathtt{out} = (i_{\text{sig}}, \ell_{\text{sig}}, \pi_{\text{sig}}, i_{\text{agg}}, h_{\text{sig},i_{\text{sig}}}, h_{\text{agg},i_{\text{agg}}}, \underline{a}, \mathtt{forg}).$$
5. $\mathtt{fork}^{\mathcal{A}'}$ selects $\underline{h}'_{\text{sig}} \leftarrow (\{0,1\}^\eta)^q$, sets the gluing index as $j = i_{\mathtt{sig}}$, and glues oracle query response sequences together as usual
$$\underline{h}''_{\text{sig}} = (h_{\text{sig},1}, h_{\text{sig},2}, \ldots, h_{\text{sig},j-1}, h'_{\mathtt{sig},j}, h'_{\mathtt{sig},j+1}, \ldots).$$
6. $\mathtt{fork}^{\mathcal{A}'}$ runs $\mathcal{A}'$ with $\mathtt{inp}'_{\mathcal{A}'} = (\mathtt{inp}, \underline{h}''_{\text{sig}})$.
7. If $\mathcal{A}'$ outputs $\perp$, then $\mathtt{fork}^{\mathcal{A}'}$ outputs $\perp$.
8. Otherwise, $\mathcal{A}'$ outputs some $\mathtt{out}^*_{\mathcal{A}'} = (i^*_{\text{sig}}, \mathtt{out}^*)$ where
$$\mathtt{out}^* = (i^*_{\text{sig}}, \ell^*_{\text{sig}}, \pi^*_{\text{sig}}, i^*_{\text{agg}}, h''_{\text{sig},i^*_{\text{sig}}}, h_{\text{agg},i^*_{\text{agg}}}, \underline{a}^*, \mathtt{forg}^*).$$
9. If $i_{\text{sig}} \neq i^*_{\text{sig}}$, output $\perp_{\mathtt{fork}^{\mathcal{A}'}}$ and terminate.
10. Otherwise, output $(i_{\text{sig}}, \mathtt{out}')$ where $\mathtt{out}' = (\mathtt{out}, \mathtt{out}^*)$.

The following lemma is obvious: aggregation coefficients for P_ℓ must be decided before the $\mathcal{H}_{\text{sig}}$ oracle is queried with $(M, P_\ell, L_\ell, R_\ell)$ except with negligible probability. Hence, the queries that determines $h_{\text{agg},i_{\text{agg}}}$ and $h^*_{\text{agg},i^*_{\text{agg}}}$ are made before the fork, so they must be the same queries.

Lemma A.4.0.1. *A successful output from* $\mathtt{fork}^{\mathcal{A}'}$ *has* $i_{agg} = i^*_{agg}$ *except with negligible probability.*

Proof. To guess the output of $\mathcal{H}_{\text{sig}}(M, P_\ell, L_\ell, R_\ell)$ before learning P_ℓ occurs with probability at most $(\mathfrak{p} - q)^{-1}$. This is sufficient, but not necessary: it's also possible that P_ℓ is learned without making aggregation coefficient queries.

Hence, except with some probability at most $(\mathfrak{p} - q)^{-1}$, P_ℓ is learned before this query is made. Conditioning upon the event that $\mathcal{A}'$ does learn P_ℓ, the probability that P_ℓ is guessed without computing any aggregation coefficients is also at most $(\mathfrak{p} - q)^{-1}$.

The probability that this ordering does not hold is bounded from above by $(\mathfrak{p} - q)^{-1} + (1 - (\mathfrak{p} - q)^{-1})(\mathfrak{p} - q)^{-1} = (2 - (\mathfrak{p} - q)^{-1})(\mathfrak{p} - q)^{-1})$. So the probability that this ordering does hold is at least $(1 - (\mathfrak{p} - q)^{-1})^2$. □

The acceptance probability of $\mathtt{fork}^{\mathcal{A}'}$ is provided by the general forking lemma. Due to our choice of forking, the query to $\mathcal{H}_{\text{sig}}$ in the two resulting transcripts have the same input $(M, P_\ell, L_\ell, R_\ell)$ but different outputs. This is the condition that $c_\ell \neq c'_\ell$ from the *system of forgery-to-discrete-log equations and inequalities*. We wrap this algorithm with the following algorithm $\mathcal{A}''$ that rejects certain executions and reformats their outputs.

Algorithm $\mathcal{A}''$:
1. Take as input some $\mathtt{inp}_{\mathtt{fork}^{\mathcal{A}'}} = (X_h, \underline{h}_{\text{agg}})$.
2. Select some random coins $\rho = \rho_{\mathtt{fork}^{\mathcal{A}'}}$.
3. Execute $\mathtt{fork}^{\mathcal{A}'}$ with $\mathtt{inp}_{\mathtt{fork}^{\mathcal{A}'}}$ and these random coins.
4. If the result is $\perp_{\mathtt{fork}^{\mathcal{A}'}}$, output $\perp_{\mathcal{A}''}$ and terminate. Otherwise, assemble together $\mathtt{out}_{\mathtt{fork}^{\mathcal{A}'}} = (i_{\text{sig}}, \mathtt{out}')$ where $\mathtt{out}' = (\mathtt{out}, \mathtt{out}^*)$ and output $\mathtt{out}_{\mathtt{fork}^{\mathcal{A}'}}$.
5. Find $\ell_{\text{sig}}, \pi_{\text{sig}} \in \mathtt{out}$; find $\ell^*_{\text{sig}}, \pi^*_{\text{sig}} \in \mathtt{out}^*$.
6. If $\ell_{\text{sig}} \neq \ell^*_{\text{sig}}$ or $\pi_{\text{sig}} \neq \pi^*_{\text{sig}}$, output $\perp$ and terminate. Otherwise, assemble together $\mathtt{out}_{\mathcal{A}''} = (i_{\text{agg}}, \mathtt{out}')$ and output $\mathtt{out}_{\mathcal{A}''}$.

The proof of the non-negligibility of the acceptance probability of $\mathcal{A}''$ is very similar to a proof presented in [13] for the non-threshold case of LSAG signatures. We omit this, noting merely that the threshold property of our scheme makes no difference in determining the acceptance probability.

Now we fork on $i_{\mathtt{agg}}$. This way, all queries made before this point remain the same. Moreover, since the aggregation coefficients on the adversarially chosen keys are determined by random coins, not the hash query tape, and due to our structure of the $\mathcal{H}_{\mathtt{agg}}$ simulations, it is always the case that these random coins are selected before the output for the honest key. Hence, if some algorithm is making decisions adaptively based on previous input, the random coins chosen for the aggregation coefficients on the adversarially selected keys are identical between the two branches with probability 1.

Algorithm $\mathtt{fork}^{\mathcal{A}''}$: Takes as input some $\mathtt{inp}_{\mathtt{fork}^{\mathcal{A}''}} = X_h$.

1. Selects some random coins $\rho = \rho_{\mathcal{A}''}$.
2. $\mathtt{fork}^{\mathcal{A}''}$ selects $\underline{h}_{\text{agg}} \leftarrow (\{0,1\}^\eta)^q$ and sets $\mathtt{inp}_{\mathcal{A}''} = \mathtt{inp}_{\mathtt{fork}^{\mathcal{A}'}} = (X_h, \underline{h}_{\text{agg}})$.
3. $\mathtt{fork}^{\mathcal{A}''}$ runs $\mathcal{A}''$ with $\mathtt{inp}_{\mathcal{A}''}$.
4. If $\mathcal{A}''$ outputs $\perp_{\mathcal{A}''}$, then $\mathtt{fork}^{\mathcal{A}''}$ outputs $\perp_{\mathtt{fork}^{\mathcal{A}''}}$ and terminates. Otherwise, $\mathtt{fork}^{\mathcal{A}''}$ receives some $(i_{\text{agg}}, \mathtt{out}_{\mathcal{A}''})$.
5. $\mathtt{fork}^{\mathcal{A}''}$ selects $\underline{h}'_{\text{agg}} \leftarrow (\{0,1\}^\eta)^q$, sets the gluing index $j = i_{\mathtt{agg}}$, and glues oracle query responses together as usual

$$\underline{h}''_{\text{agg}} = (h_{\text{agg},1}, h_{\text{agg},2}, \ldots, h_{\text{agg},j-1}, h'_{\mathtt{agg},j}, h'_{\mathtt{agg},j+1}, \ldots),$$

and assembles $\mathtt{inp}'_{\mathcal{A}''} = (X_h, \underline{h}''_{\text{agg}})$

6. $\texttt{fork}^{\mathcal{A}''}$ runs $\mathcal{A}''$ with $\texttt{inp}'_{\texttt{fork}^{\mathcal{A}''}}$.
7. If $\mathcal{A}''$ outputs $\perp$, then $\texttt{fork}^{\mathcal{A}''}$ outputs $\perp$. Otherwise, $\texttt{fork}^{\mathcal{A}''}$ receives some $(i^*_{\text{agg}}, \texttt{out}^*_{\mathcal{A}''})$.
8. If $i_{\text{agg}} \neq i^*_{\text{agg}}$, output $\perp_{\texttt{fork}^{\mathcal{A}''}}$ and terminate.
9. Otherwise, output $(i_{\texttt{agg}}, \texttt{out}_{\texttt{fork}^{\mathcal{A}''}})$ where $\texttt{out}_{\texttt{fork}^{\mathcal{A}''}} = (\texttt{out}_{\mathcal{A}''}, \texttt{out}^*_{\mathcal{A}''})$

The acceptance probability of $\texttt{fork}^{\mathcal{A}''}$ is bounded from below by some non-negligible function, following the general forking lemma, bounded from below:

$$\texttt{acc}_{\texttt{fork}^{\mathcal{A}''}} \geq \texttt{acc}_{\mathcal{A}''}\left(\frac{\texttt{acc}_{\mathcal{A}''}}{q} - \frac{1}{2^\eta}\right).$$

Lastly we construct our discrete log solver. Note that the following algorithm succeeds if and only if $\texttt{fork}^{\mathcal{A}''}$ does, so the probability of success is identical, establishing our main theorem.

Algorithm $\mathcal{B}$: $\mathcal{B}$ has black-box access to $\texttt{fork}^{\mathcal{A}''}$. $\mathcal{B}$ takes as input an honest public key X_h and is granted random oracle access, and outputs the discrete logarithm x_h.

1. $\mathcal{B}$ takes as input some X_h.
2. $\mathcal{B}$ executes $\texttt{fork}^{\mathcal{A}''}$ with $\texttt{inp} = \{X_h\}$.
3. If $\texttt{fork}^{\mathcal{A}''}$ outputs $\perp_{\texttt{fork}^{\mathcal{A}''}}$, $\mathcal{B}$ outputs $\perp_{\mathcal{B}}$ and terminates. Otherwise, $\mathcal{B}$ receives $(i_{\text{agg}}, \texttt{out}_{\texttt{fork}^{\mathcal{A}''}})$.
4. $\mathcal{B}$ goes through $\texttt{out}_{\texttt{fork}^{\mathcal{A}''}}$ to extract the four signatures, $\sigma^{(j)}$ (for $j \in [4]$), the signing query indices $i^{(j)}_{\text{sig}}$, the signing query responses $h^{(j)}_{\text{sig},i^{(j)}_{\text{sig}}}$, the ring indices $\ell^{(j)}_{\text{sig}}$ and $\pi^{(j)}_{\text{sig}}$, the aggregation query indices i_{agg} and the aggregation query responses $h^{(j)}_{\text{agg},i^{(j)}_{\text{agg}}}$.
5. $\mathcal{B}$ verifies the following system of equalities and inequalities

$$\begin{aligned}
i^{(1)}_{\text{sig}} &= i^{(2)}_{\text{sig}} & \ell^{(1)}_{\text{sig}} &= \ell^{(2)}_{\text{sig}} & \pi^{(1)}_{\text{sig}} &= \pi^{(2)}_{\text{sig}} \\
i^{(3)}_{\text{sig}} &= i^{(4)}_{\text{sig}} & \ell^{(3)}_{\text{sig}} &= \ell^{(4)}_{\text{sig}} & \pi^{(3)}_{\text{sig}} &= \pi^{(4)}_{\text{sig}} \\
i^{(i)}_{\text{agg}} &= i^{(j)}_{\text{agg}} \text{ for each } i,j & h^{(1)}_{\text{agg},i_{\text{agg}}} &= h^{(2)}_{\text{agg},i_{\text{agg}}} & h^{(3)}_{\text{agg},i_{\text{agg}}} &= h^{(4)}_{\text{agg},i_{\text{agg}}} \\
h^{(1)}_{\text{sig},i^{(1)}_{\text{sig}}} &\neq h^{(2)}_{\text{sig},i^{(1)}_{\text{sig}}} & h^{(3)}_{\text{sig},i^{(1)}_{\text{sig}}} &\neq h^{(4)}_{\text{sig},i^{(1)}_{\text{sig}}} & h^{(1)}_{\text{agg},i_{\text{agg}}} &\neq h^{(3)}_{\text{agg},i_{\text{agg}}}
\end{aligned}$$

   outputting $\perp_{\mathcal{B}}$ if any fail.
6. From each $\texttt{forg}^{(j)}$, the random signature data $s^{(j)}_\ell$ can be extracted.
7. $\mathcal{B}$ outputs

$$\widehat{x}_h := (h^{(1)}_{\text{agg},i_{\text{agg}}} - h^{(3)}_{\text{agg},i_{\text{agg}}})^{-1}\left(\frac{s^{(2)}_\ell - s^{(1)}_\ell}{h^{(2)}_{\text{sig},i^{(2)}_{\text{sig}}} - h^{(1)}_{\text{sig},i^{(1)}_{\text{sig}}}} - \frac{s^{(4)}_\ell - s^{(3)}_\ell}{h^{(3)}_{\text{sig},i^{(3)}_{\text{sig}}} - h^{(4)}_{\text{sig},i^{(4)}_{\text{sig}}}}\right).$$

   Of course, $\mathcal{B}$ only fails if $\texttt{fork}^{\mathcal{A}''}$ fails or if the system is not verified, but this is a sub-event of the failure of $\texttt{fork}^{\mathcal{A}''}$. So the probability that $\mathcal{B}$ is bounded above by the probability that $\texttt{fork}^{\mathcal{A}''}$ fails.

B Properties other than unforgeability

We establish correctness and linkability and describe some security properties like signer ambiguity and its relation to key aggregation indistinguishability. Correctness and linkability take into account semi-honest adversaries, who carry out algorithms to specification (although may also take additional steps outside of specification). In this case, this is a group of "curious but honest" friends wishing to collaborate upon a signature.

Lemma B.0.0.1. *Example 4.0.4 is correct.*

Proof. In the event $S((\mathfrak{m}, \underline{P}, \pi, \underline{x}), (\mathfrak{m}^*, \sigma))$ where $\sigma = (c_1, \underline{s})$ and $\mathfrak{m}^* = (\mathfrak{m}, \underline{P}, J, \mathtt{aux})$, the semi-honest signer computes $c_{\pi+1}$, $c_{\pi+2}$, ..., c_r, and c_1 with usual/honest queries made to $\mathcal{H}_{\text{sig}}$ with probability 1. Moreover, the semi-honest verifier makes queries $c_2, \ldots, c_{\pi-1}, c_\pi$ by making usual/honest queries to $\mathcal{H}_{\text{sig}}$ with probabilty 1. The semi-honest signer also, by specification, ensured the verification equations are satisfied with use of the secret key p_π. Hence, a semi-honest verifier given $\mathfrak{m}^*$ and $\sigma = (c_1, (s_1, s_2, \ldots, s_r))$ who computes the following

$$\begin{aligned}
L_1' &:= c_1 G + s_1 P_1 & R_1' &:= c_1 H_1 + s_1 J & c_2' &:= \mathcal{H}_{\text{sig}}(M, P_1, R_1, L_1) \\
L_2' &:= c_2' G + s_2 P_2 & R_2' &:= c_2' H_2 + s_2 J & c_3' &:= \mathcal{H}_{\text{sig}}(M, P_2, R_2, L_2) \\
&\vdots & &\vdots & &\vdots \\
L_r' &:= c_r' G + s_r P_r & R_r' &:= c_r' H_r + s_r J & c_1' &:= \mathcal{H}_{\text{sig}}(M, P_r, R_r, L_r)
\end{aligned}$$

obtains $c_1' = c_1$ with probability 1. □

Lemma B.0.0.2. *Example 4.0.4 is linkable.*

Proof. In the event $S'(\mathfrak{m}_1^*, \sigma_1, \mathfrak{m}_2^*, \sigma_2)$, the semi-honest adversary used the same key in each signature. That is to say, for the rings $\underline{P}_1$, $\underline{P}_2$, there is a common key; that is, for some indices i_1, i_2, $(\underline{P}_1)_{i_1} = (\underline{P}_2)_{i_2} = P_{\text{common}}$. In both signatures, the key image is $J = P_{\text{common}} \mathcal{H}_{\text{ki}}(P_{\text{common}})$, and so linking occurs with probability 1. □

We modify the definition of anonymity with adversarially chosen ring members from [4] to take into account key aggregation; the stronger definition presented in [4] with *full key exposure* is not possible to satisfy in the linkable transaction setting of CryptoNote-styled ring signatures we see in Monero.

Definition B.0.1 (Threshold signer ambiguity with adversarially chosen ring members)**.** Let $f(-)$ be a positive polynomial. Consider the following game:

1. A set of private-public key pairs $\{(x_i, X_i)\}_{i \in [f(\lambda)]}$ is selected by the challenger with `KeyGen`. Denote $SK = \{x_i\}$ and $PK = \{X_i\}$.
2. The public keys PK are sent to $\mathcal{A}$, who is granted access to a signing oracle, $\mathcal{SO}$.
3. $\mathcal{A}$ outputs a message $\mathfrak{m}$, two non-empty aggregation multi-sets of public keys $\underline{X}^{(0)}$, $\underline{X}^{(1)} \subset PK$, and a ring $\underline{P}$ such that $P_{i_j} = \Phi(\underline{X}^{(j)})$ for $j = 0, 1$.
4. The challenger selects a random bit b, computes $(\mathfrak{m}^*, \sigma) \leftarrow \mathtt{Sign}(\mathfrak{m}, \underline{P}, \pi_b)$ where π_b denotes the index of P_{i_b} in $\underline{P}$, and sends $(\mathfrak{m}^*, \sigma)$ to $\mathcal{A}$.
5. $\mathcal{A}$ outputs a bit b'.

$\mathcal{A}$ wins the game if $b' = b$ and the key images for P_{i_0} and P_{i_1} do not appear in the output of any query made to $\mathcal{SO}$ by $\mathcal{A}$.

The following lemma, which establishes key aggregation indistinguishability, is obvious in the case with the semi-honest adversary, since the distribution of $(c_1, \underline{s})$ in both signature schemes is determined by the hash function $\mathcal{H}_{\text{sig}}$ and the choice made by the signer for $\underline{s}$. Under the random oracle model, c_1 is uniformly distributed over $\mathbb{Z}_{\mathfrak{p}}$. In the case with the semi-honest adversary, each $s_i \in \underline{s}$ is also uniformly distributed over $\mathbb{Z}_{\mathfrak{p}}$ and, moreover, all of these are independent from each other.

Lemma B.0.1.1. *Signatures produced by semi-honest adversaries using Example 4.0.4 are statistically indistinguishable from LSAG signatures.*

As a corollary, we immediately obtain that an adversary who can violate the signer ambiguity of Example 4.0.4 must likewise be able to violate the signer ambiguity of LSAG signatures.

Corollary B.0.2. Example 4.0.4 is signer ambiguous with adversarially chosen ring members.