FROSTLASS: Flexible Ring-Oriented Schnorr-like Thresholdized Linkably Anonymous Signature Scheme

Scheme Formalization & Review of Rust Implementation

Joshua Babb* Brandon Goodell* Luke Parker Rigo Salazar*

Freeman Slaughter* Luke Szramowski*

August 13, 2025

1 Introduction

Over the past decades, especially since Shamir's secret sharing and Shoup's threshold signatures, ([10], [11]) research into threshold and multiparty cryptographic schemes of different flavors has become fashionable. In [2], for example, Bellare and Neven famously proposed a framework to formalize multisignatures and to prove them secure with the generalized forking lemma.

The general forking lemma, which goes back at least to [8], is useful in proving a wide variety of modern cryptographic schemes secure, including ring signatures preceding [12] and the bulletproofs zero-knowledge proving system proposed in [3].

Concise linkable spontaneous anonymous group (CLSAG) signatures, proposed in [5] and built from the (LSAG) signatures from [9], are Schnorr-like ring signatures used in the Monero cryptocurrency protocol. A naïve thresholdization of CLSAG signatures, called *thring signatures*, was proposed in [4], building off of the linkable spontaneous anonymous group (LSAG) signatures , which are used in the Monero cryptocurrency protocol. The FROST approach to thresholdizing Schnorr signatures, first described in [7], is sufficiently flexible to work for CLSAG signatures, and are superior to the thring signatures of [4].

An opinionated Rust implementation of every major component of the Monero protocol at [6], written by Luke Parker (kayabaNerve), contains an implementation of FROSTLASS. Herein, we formalize FROSTLASS, present a novel definition of linkability, and prove FROSTLASS strongly unforgeable up to the hardness of the κ-one-more discrete logarithm problem, and statistically linkable.

1.1 Change Log

This document may be updated occasionally, especially if security-sensitive results come to light. We summarize such changes here.

- 15 March 2025. Initial preprint.

*Cypher Stack

2 Notation and Background Definitions

2.1 Notation

Tuples are generally denoted with underlines, i.e. $\underline{x} = (x_1, \ldots, x_n)$, and we abuse set notation for these, e.g. $x_1 \in \underline{x}$. The set of all finite-length bitstrings is denoted with $\{0,1\}^*$. For $n \in \mathbb{N}$, denote the set $\{1, 2, \ldots, n\}$ with $[n]$. For sets X, Y with $X \subseteq Y$, denote the set $\{y \in Y \mid y \notin X\}$ with $\overline{X}$.

Denote a prime modulus with $q \in \mathbb{N}$, an abelian group of order q with $\mathbb{G}$, and a generator of $\mathbb{G}$ with $G \in \mathbb{G}$. We say the tuple $(q, \mathbb{G}, G)$ are *group parameters*. Given $\underline{x} = (x_1, \ldots, x_n) \in \mathbb{Z}_q^n$ and $\underline{G} = (G_1, \ldots, G_n) \in \mathbb{G}^n$, we denote the Shur product $\underline{x} \circ \underline{G} = (x_1 G_1, \ldots, x_n G_n)$.

Denote "big-oh" notation with O and denote random oracles with $\mathcal{O}$. Denote algorithm run times with $t \geq 0$ and success probabilities with $\epsilon \in [0,1]$. Denote the event that a PPT algorithm $\mathcal{A}$ inputs some $\texttt{in}$ and outputs some $\texttt{out}$ with $\texttt{out} \leftarrow \mathcal{A}(\texttt{in})$. We use the same notation for oracles, but we refer to the inputs as queries, say $\texttt{query}$, and outputs as responses, say $\texttt{resp}$.

2.2 Definitions

Definition 2.1 (κ Random Oracle Distinguishing)**.** Let $\kappa \geq 0$ be an integer, let S, T be sets, $\mathcal{O} : S \to T$ be a random oracle, and $\phi : S \to T$ a function. Any PPT (t, ϵ)-algorithm $\mathcal{A}$ which plays the following game is an (ϕ, κ)-distinguisher.

1. The challenger samples $b \xleftarrow{\$} \{0,1\}$ and grants $\mathcal{A}$ access to an oracle $\mathcal{O}'_b$, where
   (a) $\mathcal{O}'_0$ is a simple wrapper for $\mathcal{O}$, and
   (b) $\mathcal{O}'_1$ is a simple wrapper for ϕ.
2. Eventually, $\mathcal{A}$ outputs a bit b', succeeding if and only if $b' = b$ and $\mathcal{O}'_b$ was queried at most κ times.

Definition 2.2 (κ-OMDL: One-More-Than-κ Discrete Logarithms over $G \in \mathbb{G}$)**.** Let $\kappa \geq 0$ be an integer. Let $\Phi = \{(q_\lambda, \mathbb{G}_\lambda, G_\lambda)\}_{\lambda \in \mathbb{N}}$ be a parameterized family of group parameters. Let $t \geq 0$ and $\epsilon \in [0,1]$ be real numbers. We say any PPT algorithm $\mathcal{A}$ that can successfully play the following game in time at most t and with probability at least ϵ is a (t, ϵ)-player of the one-more-than-κ discrete logarithms game over $G_\lambda \in \mathbb{G}_\lambda$.

1. The challenger grants $\mathcal{A}$ access to a key generation oracle $\mathcal{O}_{\texttt{key}} : \{*\} \to \mathbb{G}_\lambda$ and a corruption oracle $\mathcal{O}_{\texttt{corrupt}} : \mathbb{G}_\lambda \to \mathbb{Z}_{q_\lambda}$ which work as follows.
   (a) A valid query made to $\mathcal{O}_{\texttt{key}}$ is a simple request for a new key, which we model with a dummy singleton domain $\{*\}$. The response is some point $\texttt{resp} = X \in \mathbb{G}_\lambda$. We say the response is a *challenge key*. Let $\mathcal{L}_{\texttt{key}} = \{X \in \mathbb{G}_\lambda \mid X \leftarrow \mathcal{O}_{\texttt{key}} \text{ occurred}\}$ denote the set of all responses from $\mathcal{O}_{\texttt{key}}$.
   (b) A valid query made to $\mathcal{O}_{\texttt{corrupt}}$ a challenge key, $X \in \mathcal{L}_{\texttt{key}}$. The response to a valid query X is a scalar $x \in \mathbb{Z}_{q_\lambda}$ such that $X = xG$, and the response to an invalid query is a distinct failure symbol. Let $\mathcal{L}_{\texttt{corrupt}} \subseteq \mathcal{L}_{\texttt{key}}$ be the subset of valid queries made to $\mathcal{O}_{\texttt{corrupt}}$ be the *corrupted keys* and let $\overline{\mathcal{L}_{\texttt{corrupt}}}$ be the subset of *uncorrupted challenge keys*.
2. Eventually, the event $\texttt{out}_{\mathcal{A}} \leftarrow \mathcal{A}$ occurs. We say $\mathcal{A}$ succeeds at the κ-OMDL game if and only if all the following hold in this event:

(a) $|\mathcal{L}_{\mathtt{key}}| \geq \kappa + 1$,

(b) $|\mathcal{L}_{\mathtt{corrupt}}| \leq \kappa$,

(c) $\mathtt{out}_{\mathcal{A}} \in \mathbb{Z}_{q_\lambda}^{\kappa+1}$, and

(d) $\{xG \mid x \in \mathtt{out}_{\mathcal{A}}\} \subseteq \mathcal{L}_{\mathtt{key}}$

Moreover, if $t \in O(\text{poly}(\lambda))$ implies $\epsilon \in \text{negl}(\lambda)$ for every (t, ϵ)-player, we say the κ-OMDL game is hard over Φ.

In the sequel, we leave λ implicit, suppressing it in our notation for clarity, except when there is case of confusion.

Note that the 0-OMDL game is simply the discrete logarithm game. Moreover, an adaptive variation of this game is natural, where κ is determined in each instance of the game by the number of corruption oracle queries made by the adversary.

Oracle $\mathcal{O}_{\mathtt{key}}(*)$
$x \xleftarrow{\$} \mathbb{Z}_q$
$X = xG$
$\mathcal{L}_{\mathtt{key}} = \mathcal{L}_{\mathtt{key}} \cup \{X\}$
return X

Oracle $\mathcal{O}_{\mathtt{corrupt}}(X)$
if $X \in \mathcal{L}_{\mathtt{key}}$
$\quad x = \log_G X$
$\quad \mathcal{L}_{\mathtt{corrupt}} = \mathcal{L}_{\mathtt{corrupt}} \cup \{X\}$
$\quad$ **return** x
else return $\perp$

Oracle 1: The key generation and corruption oracles for the κ-OMDL game.

Game κ-OMDL$(\underline{x})$
if $|\mathcal{L}_{\mathtt{key}}| \geq \kappa + 1$ **and** $|\mathcal{L}_{\mathtt{corrupt}}| \leq \kappa$
and $\underline{x} \in \mathbb{Z}_q^{\kappa+1}$ **and** $\underline{x} \circ \underline{G} \subseteq \mathcal{L}_{\mathtt{key}}$
$\quad$ **return** 1
else return 0

Game 1: Success condition for the κ-OMDL game.

Definition 2.3 (General Forking Algorithm)**.** Let X, Y, H be finite sets, $\kappa \geq 1$ an integer parameter, and let $\mathcal{A}$ be a PPT algorithm which uses a random tape $\tau \in \{0,1\}^*$, inputs some $(x, \underline{h}) \in X \times H^\kappa$, and outputs a pair $(i, y) \in [\kappa] \times Y$ or a distinct failure symbol. Then the algorithm specified below, $\mathtt{Fork}_{\mathcal{A}}$, is a PPT algorithm which inputs $x \in X$, outputs $(i, y, y') \in [\kappa] \times Y^2$ or a distinct failure symbol, and is called the *general forking algorithm* for $\mathcal{A}$.

1. Sample $\tau \xleftarrow{\$} \{0,1\}^*$ for $\mathcal{A}$ to use in both executions.
2. Sample $\underline{h}, \underline{h}' \xleftarrow{\$} H^\kappa$.
3. Compute $\mathtt{out} \leftarrow \mathcal{A}(x, \underline{h}; \tau)$.
4. If $\mathtt{out}$ is a failure symbol, output a distinct failure symbol and terminate. Otherwise, $\mathtt{out}$ is not failure symbol, so parse $(i, y) := \mathtt{out}$.
5. Set $\underline{h}^* = (h_1, \ldots, h_{i-1}, h'_i, h'_{i+1}, \ldots, h'_\kappa)$.
6. Compute $\mathtt{out}' \leftarrow \mathcal{A}(x, \underline{h}^*; \tau)$.

7. If $\mathtt{out}'$ is a failure symbol, output a distinct failure symbol and terminate. Otherwise, $\mathtt{out}$ is not a failure symbol, so parse $(i', y') := \mathtt{out}'$.

8. If $i \neq i'$ or $h_i = h^*_{i'}$, then output a distinct failure symbol and terminate.

9. Otherwise, output (i, y, y').

Lemma 2.4 (General Forking Lemma)**.** For any finite sets X, H, for any algorithm $\mathcal{A}$ as in Definition 2.3 which runs in time at most t and fails with probability at most ϵ, for any probability mass function F over X, the general forking algorithm $\mathtt{Fork}_{\mathcal{A}}$ has advantage satisfying the following

$$\mathtt{Adv}_{\mathtt{Fork}_{\mathcal{A}}} \geq \epsilon \left(\frac{\epsilon}{\kappa} - \frac{1}{|H|} \right)$$

where this probability is measured over F and all randomness used in sampling. □

Definition 2.5 (LTM: Linkable Thring Multisignatures)**.** A tuple of algorithms $(\mathtt{PGen}, \mathtt{KGen}, \mathtt{Sign}, \mathtt{Combine}, \mathtt{Vf}, \mathtt{Link})$ as follows.

1. $\mathtt{PGen}(\lambda) \to \mathtt{pars}_\lambda$. Input a security parameter $\lambda \in \mathbb{N}$, and output some public parameters $\mathtt{pars}_\lambda$, which includes the description of secret signing key shares $\mathcal{SK}$, public verification key shares $\mathcal{VK}$, total verification keys $\mathcal{TVK}$, messages $\mathcal{MSG}$, signatures challenges $\mathcal{CH}$, partial signature shares $\mathcal{PSIG}$, and signatures $\mathcal{SIG}$.

2. $\mathtt{KGen}(\mathtt{pars}_\lambda, n, r) \to (\mathtt{tvk}, \underline{\mathtt{vk}}, \underline{\mathtt{sk}})$. An interactive probabilistic algorithm executed by some *capacity* of $n \geq 1$ participants called *threshold keyholders*. Users share as common input the capacity n and threshold $r \in n$. Output *total verification key* $\mathtt{tvk} \in \mathcal{TVK}$, *public verification key shares* $\underline{\mathtt{vk}} = (\mathtt{vk}_i)_{i=1}^n \in \mathcal{VK}^n$, and *secret signing key shares* $\underline{\mathtt{sk}} = (\mathtt{sk}_i)_{i=1}^n \in \mathcal{SK}^n$.

3. $\mathtt{Sign}(\mathtt{pars}_\lambda, \mathtt{msg}, \mathtt{ring}, \underline{\mathtt{vk}}, \mathtt{sk}) \to \mathtt{psig}$. Non-interactive probabilistic algorithm executed by a threshold keyholder. Input a message $\mathtt{msg} \in \mathcal{MSG}$, a tuple of $m \geq 1$ total verification keys $\mathtt{ring} = (\mathtt{tvk}_j)_{j=1}^m \in \mathcal{TVK}^m$ called a *ring**, some $r \geq 1$ public verification key shares $\underline{\mathtt{vk}} = (\mathtt{vk}_i)_{i=1}^r \in \mathcal{VK}^r$ called *signers' coalition key shares*, and a secret key share $\mathtt{sk} \in \mathcal{SK}$. Output a ring signature share $\mathtt{psig} \in \mathcal{PSIG}$.

4. $\mathtt{Combine}(\mathtt{pars}_\lambda, \mathtt{msg}, \mathtt{ring}, \underline{\mathtt{vk}}, \underline{\mathtt{psig}}) \to \mathtt{sig}$. Non-interactive deterministic algorithm executed by a user called the *combiner*. Input a message $\mathtt{msg} \in \mathcal{MSG}$, a ring $\mathtt{ring} = (\mathtt{tvk}_j)_{j=1}^m \in \mathcal{TVK}^m$, a signers' coalition of key shares $\underline{\mathtt{vk}} = (\mathtt{vk}_i)_{i=1}^r \in \mathcal{VK}^r$, and ring signature shares $\underline{\mathtt{psig}} = (\mathtt{psig}_i)_{i=1}^r \in \mathcal{PSIG}^r$. Output a ring signature $\mathtt{sig} \in \mathcal{SIG}$.

5. $\mathtt{Vf}(\mathtt{pars}_\lambda, \mathtt{msg}, \mathtt{ring}, \mathtt{sig}) \to b$. Non-interactive deterministic algorithm executed by a user called the *verifier*. Input message $\mathtt{msg} \in \mathcal{MSG}$, a ring $\mathtt{ring} = (\mathtt{tvk}_j)_{j=1}^m \in \mathcal{TVK}^m$, and a ring signature $\mathtt{sig} \in \mathcal{SIG}$. Output a bit.

6. $\mathtt{Link}(\mathtt{pars}_\lambda, \mathtt{sig}, \mathtt{sig}') \to b$. A non-interactive deterministic algorithm executed by a user called the *linker*. Input ring signatures $\mathtt{sig}, \mathtt{sig}' \in \mathcal{SIG}$, and output a bit.

Definition 2.5 extends naturally to a *verifiable* scheme by allowing the verification of signature shares with the following additional algorithm. Adding this additional level of verifiability requires modifying Definition 2.6 below in the natural way.

*A better term would be *anonymity tuple*, but we keep with tradition.

- $\mathtt{VfSh}(\mathtt{pars}_\lambda, \mathtt{msg}, \mathtt{ring}, \underline{\mathtt{vk}}, \mathtt{psig}) \to b$. Non-interactive deterministic executed by a user called a *share verifier*. Input message $\mathtt{msg} \in \mathcal{MSG}$, a ring $\mathtt{ring} = (\mathtt{tvk}_j)_{j=1}^m \in \mathcal{TVK}^m$, a signers' coalition of key shares $\underline{\mathtt{vk}} = (\mathtt{vk}_i)_{i=1}^r$, and a ring signature share $\mathtt{psig} \in \mathcal{PSIG}$. Outputs a bit.

Any of the algorithms in Definition 2.5 may input or output auxiliary data $\mathtt{aux}$, which we only include in notation when relevant. Following our convention for group parameter notation, we leave $\mathtt{pars}_\lambda$ implicit in our notation, as all algorithms require it.

Definition 2.6. Let Π be an LTM scheme. We define correctness using the following events.

1. Let E_1 be the event in which some signers' coalitions of key shares $\underline{\mathtt{vk}}', \underline{\mathtt{vk}}'' \subseteq \underline{\mathtt{vk}}$ is used to compute ring signature shares $\mathtt{psig}_i'$ and $\mathtt{psig}_i''$ semi-honestly. That is to say, the following holds.
   (a) For some $\mathtt{msg}', \mathtt{msg}'' \in \mathcal{MSG}$,
   (b) for some $n, r \in \mathbb{N}$ such that $r \in [n]$ and some event $(\mathtt{tvk}, \underline{\mathtt{vk}}, \underline{\mathtt{sk}}) \leftarrow \mathtt{KGen}(n, r)$ occurs,
   (c) for some $\underline{\mathtt{vk}}', \underline{\mathtt{vk}}''$ such that $\underline{\mathtt{vk}}', \underline{\mathtt{vk}}'' \subseteq \underline{\mathtt{vk}}$, $r' = |\underline{\mathtt{vk}}'|$, $r'' = |\underline{\mathtt{vk}}''|$, and $r \leq \min\{r', r''\}$,
   (d) for some $\sigma' \in \mathcal{S}_n$ such that, for every $i \in [r']$, $\mathtt{vk}_i' = \mathtt{vk}_{\sigma'(i)}$ and $\mathtt{sk}_i' = \mathtt{sk}_{\sigma'(i)}$,
   (e) for some $\sigma'' \in \mathcal{S}_n$ such that, for every $i \in [r'']$, $\mathtt{vk}_i'' = \mathtt{vk}_{\sigma''(i)}$ and $\mathtt{sk}_i'' = \mathtt{sk}_{\sigma''(i)}$,
   (f) for some $\mathtt{ring}', \mathtt{ring}'' \subseteq \mathcal{TVK}$ such that $\mathtt{tvk} \in \mathtt{ring}' \cap \mathtt{ring}''$,
   (g) for each $i \in [r']$, $\mathtt{psig}_i' \leftarrow \mathtt{Sign}(\mathtt{msg}', \mathtt{ring}', \underline{\mathtt{vk}}', \mathtt{sk}_i')$, and
   (h) for each $i \in [r'']$, $\mathtt{psig}_i'' \leftarrow \mathtt{Sign}(\mathtt{msg}'', \mathtt{ring}'', \underline{\mathtt{vk}}'', \mathtt{sk}_i'')$.
2. Let E_1^* be a similar event in which some $\underline{\mathtt{vk}}^{**} \subseteq \underline{\mathtt{vk}}^*$ compute ring signature shares $\mathtt{psig}_i^*$ semi-honestly from a different $\mathtt{tvk}^*$, i.e. all the following hold.
   (a) For some $\mathtt{msg}^* \in \mathcal{MSG}$,
   (b) for some $n^*, r^* \in \mathbb{N}$ such that $r^* \in [n^*]$ and some event $(\mathtt{tvk}^*, \underline{\mathtt{vk}}^*, \underline{\mathtt{sk}}^*) \leftarrow \mathtt{KGen}(n, r)$ occurs such that $(\mathtt{tvk}^*, \underline{\mathtt{vk}}^*, \underline{\mathtt{sk}}^*) \neq (\mathtt{tvk}', \underline{\mathtt{vk}}', \underline{\mathtt{sk}}')$ and $(\mathtt{tvk}^*, \underline{\mathtt{vk}}^*, \underline{\mathtt{sk}}^*) \neq (\mathtt{tvk}'', \underline{\mathtt{vk}}'', \underline{\mathtt{sk}}'')$,
   (c) for some $\underline{\mathtt{vk}}^{**}$ such that $\underline{\mathtt{vk}}^{**} \subseteq \underline{\mathtt{vk}}^*$, $r^{**} = |\underline{\mathtt{vk}}^*|$, and $r^* \leq r^{**}$,
   (d) for some $\sigma^* \in \mathcal{S}_n$ such that, for every $i \in [r^{**}]$, $\mathtt{vk}_i^* = \mathtt{vk}_{\sigma^*(i)}$ and $\mathtt{sk}_i^* = \mathtt{sk}_{\sigma^*(i)}$,
   (e) for some $\mathtt{ring}^* \subseteq \mathcal{TVK}$ such that $\mathtt{tvk}^* \in \mathtt{ring}^*$,
   (f) for each $i \in [r^{**}]$, $\mathtt{psig}_i^{**} \leftarrow \mathtt{Sign}(\mathtt{msg}^*, \mathtt{ring}^*, \underline{\mathtt{vk}}^*, \mathtt{sk}_i^*)$.
3. Let $E_2 \subseteq E_1^* \cap E_1$ be the event that the ring signature shares $\mathtt{psig}_i'$, $\mathtt{psig}_i''$, and $\mathtt{psig}_i^*$ are combined semi-honestly, i.e. all of the following hold.
   (a) $\mathtt{sig}' \leftarrow \mathtt{Combine}(\mathtt{msg}', \mathtt{ring}', \underline{\mathtt{psig}}')$,
   (b) $\mathtt{sig}'' \leftarrow \mathtt{Combine}(\mathtt{msg}'', \mathtt{ring}'', \underline{\mathtt{psig}}'')$,
   (c) $\mathtt{sig}^* \leftarrow \mathtt{Combine}(\mathtt{msg}^*, \mathtt{ring}^*, \underline{\mathtt{psig}}^*)$.
4. Let $E_3 \subseteq E_2$ be the event that the combined signatures are valid, i.e. all the following hold.
   (a) $\mathtt{Vf}(\mathtt{msg}', \mathtt{ring}', \mathtt{sig}') = 1$,
   (b) $\mathtt{Vf}(\mathtt{msg}'', \mathtt{ring}'', \mathtt{sig}'') = 1$, and
   (c) $\mathtt{Vf}(\mathtt{msg}'', \mathtt{ring}'', \mathtt{sig}'') = 1$.

5. Let $E_4 \subseteq E_3$ be the event that $\texttt{Link}$ is commutative, i.e. all the following hold.
   (a) $\texttt{Link}(\texttt{sig}', \texttt{sig}'') = \texttt{Link}(\texttt{sig}'', \texttt{sig}')$,
   (b) $\texttt{Link}(\texttt{sig}', \texttt{sig}^*) = \texttt{Link}(\texttt{sig}^*, \texttt{sig}')$, and
   (c) $\texttt{Link}(\texttt{sig}^*, \texttt{sig}'') = \texttt{Link}(\texttt{sig}'', \texttt{sig}^*)$
6. Let $E_5 \subseteq E_2$ that $\texttt{Link}(\texttt{sig}', \texttt{sig}'') = 1$.
7. Let $E_6 \subseteq E_3$ that $\texttt{Link}(\texttt{sig}', \texttt{sig}^*) = \texttt{Link}(\texttt{sig}^*, \texttt{sig}'') = 0$.

We say Π has *correct ring signature share verification* if $\mathbb{P}[E_2] = 1$, has *correct ring signature verification* if $\mathbb{P}[E_4] = 1$, has *commutative linking* if $\mathbb{P}[E_5] = 1$, has *correct positive linkability* if $\mathbb{P}[E_6] = 1$, and has *correct negative linkability* if $\mathbb{P}[E_7] = 1$, where these probabilities are computed over all choices of $n, r, n^*, r^*, \texttt{msg}', \texttt{msg}'', \texttt{msg}^*$, all executions of $\texttt{KGen}$, all choices of $\underline{\texttt{vk}}', \underline{\texttt{vk}}'', \underline{\texttt{vk}}^*$, and all randomness used by all algorithms. If Π satisfies all four notions of correctness, we simply say Π is a *correct* LTM.

For convenience, we present the following common setup useful for linkability and unforgebality.

Definition 2.7 (Common Setup with Key Generation, Corruption, and Signing Oracles)**.** Let Π be an LTM scheme. Let $\mathcal{A}$ be any PPT algorithm which runs in time at most $t > 0$, and successfully plays the following game with probability at least $\epsilon \in [0, 1]$.

1. $\mathcal{A}$ is granted to oracles $\mathcal{O}_{\texttt{key}}$, $\mathcal{O}_{\texttt{corrupt}}$, and $\mathcal{O}_{\texttt{Sign}}$ as follows.
   (a) $(\texttt{tvk}, \underline{\texttt{vk}}) \leftarrow \mathcal{O}_{\texttt{key}}(n, r)$. A valid query made to $\mathcal{O}_{\texttt{key}}$ is a simple request for r-of-n keys, which we model with the pair (n, r) such that $r \in [n]$. The response to a valid query is some $\texttt{resp} = (\texttt{tvk}, \underline{\texttt{vk}}) \in \mathcal{TVK} \times \mathcal{VK}^n$, and the response to an invalid query is a distinct failure symbol. Let $\mathcal{L}_{\texttt{key}} = \{(\texttt{tvk}, \underline{\texttt{vk}}) \mid \exists (n, r), (\texttt{tvk}, \underline{\texttt{vk}}) \leftarrow \mathcal{O}_{\texttt{key}}(n, r)\}$ denote the set of all responses from $\mathcal{O}_{\texttt{key}}$.

Oracle $\mathcal{O}_{\texttt{key}}(n, r)$
if $r \in [n]$
sample $(\texttt{tvk}, \underline{\texttt{vk}})$
$\mathcal{L}_{\texttt{key}} = \mathcal{L}_{\texttt{key}} \cup \{(\texttt{tvk}, \underline{\texttt{vk}})\}$
return $(\texttt{tvk}, \underline{\texttt{vk}})$
else return $\perp$

Oracle 2: The key generation oracle in the game of common setup.

   (b) $\texttt{sk} \leftarrow \mathcal{O}_{\texttt{corrupt}}(i, \texttt{tvk}, \underline{\texttt{vk}})$. A valid query made to $\mathcal{O}_{\texttt{corrupt}}$ is some $\texttt{query} = (i, \texttt{tvk}, \underline{\texttt{vk}})$ where $(\texttt{tvk}, \underline{\texttt{vk}}) \in \mathcal{L}_{\texttt{key}}$ is associated with some $(n, r) \in \mathbb{N}^2$ such that $r \in [n]$ and $(\texttt{tvk}, \underline{\texttt{vk}}) \leftarrow \mathcal{O}_{\texttt{key}}(n, r)$ occurred, and i is an index in $[n]$. The response to a valid query is a secret signing key $\texttt{sk}_i$ corresponding to $\texttt{vk}_i \in \underline{\texttt{vk}}$, and the response to an invalid query is a distinct failure symbol.
   Upon success, we say the verification key share $\texttt{vk}_i$ has been corrupted, and if r or more key shares have been corrupted associated with $\texttt{tvk}$, then we say $\texttt{tvk}$ has been totally corrupted. Let $\mathcal{L}^{\texttt{sh}}_{\texttt{corrupt}} = \{\texttt{vk}_i \mid \texttt{sk}_i \leftarrow \mathcal{O}_{\texttt{corrupt}}(i, \texttt{tvk}, \underline{\texttt{vk}}) \text{ occurred}\}$ be the set of corrupted key shares and let $\mathcal{L}^{\texttt{tot}}_{\texttt{corrupt}}$ be the set of totally corrupted keys.

Oracle $\mathcal{O}_{\texttt{corrupt}}(i, \texttt{tvk}, \underline{\texttt{vk}})$
if $(\texttt{tvk}, \underline{\texttt{vk}}) \leftarrow \mathcal{O}_{\texttt{key}}(n, r)$ **and** $i \in [n]$ **then**
$\quad \mathcal{L}^{\texttt{sh}}_{\texttt{corrupt}} = \mathcal{L}^{\texttt{sh}}_{\texttt{corrupt}} \cup \{\texttt{vk}_i\}$
$\quad$ **if** $\lvert \underline{\texttt{vk}} \cap \mathcal{L}^{\texttt{sh}}_{\texttt{corrupt}} \rvert \geq r$ **then**
$\quad\quad \mathcal{L}^{\texttt{tot}}_{\texttt{corrupt}} \leftarrow \mathcal{L}^{\texttt{tot}}_{\texttt{corrupt}} \cup \{\texttt{tvk}\}$
$\quad$ **return** $\texttt{sk}_i$
else return $\perp$

Oracle 3: The corruption oracle in the game of common setup.

(c) $\texttt{sig} \leftarrow \mathcal{O}_{\texttt{Sign}}(\texttt{msg}, \texttt{ring}, \texttt{tvk}, \underline{\texttt{vk}}, i)$. A valid query to $\mathcal{O}_{\texttt{Sign}}$ is a tuple $(\texttt{msg}, \texttt{ring}, \texttt{tvk}, \underline{\texttt{vk}}, i)$, where $\texttt{msg} \in \{0,1\}^*$, $\texttt{ring} \in \mathcal{TVK}^m$ for some $m \in \mathbb{N}$, $\texttt{tvk} \in \mathcal{TVK}$, $\underline{\texttt{vk}} \in \underline{\texttt{vk}}^r$ for some $r \in \mathbb{N}$, and $i \in \mathbb{N}$, such that all the following hold.

i. $\texttt{tvk} \in \texttt{ring}$,
ii. there exists some n, $\underline{\texttt{vk}}'$ such that $r \in [n]$ and $(\texttt{tvk}, \underline{\texttt{vk}}') \leftarrow \mathcal{O}_{\texttt{key}}(n, r)$ occurred,
iii. the query $\underline{\texttt{vk}}$ is a subset $\underline{\texttt{vk}} \subseteq \underline{\texttt{vk}}'$ such that $\lvert \underline{\texttt{vk}} \rvert \geq r$, and
iv. $i \in [r]$.

Oracle $\mathcal{O}_{\texttt{Sign}}(\texttt{msg}, \texttt{ring}, \texttt{tvk}, \underline{\texttt{vk}}, i)$
parse $r := \texttt{len}(\underline{\texttt{vk}})$
if $\texttt{tvk} \in \texttt{ring}$
and $\exists n, r', \underline{\texttt{vk}}'$ s.t. $(\texttt{tvk}, \underline{\texttt{vk}}') \leftarrow \mathcal{O}_{\texttt{key}}(n, r')$
and $\underline{\texttt{vk}} \subseteq \underline{\texttt{vk}}'$ **and** $r \geq r'$ **and** $i \in [r]$
$\quad$ **return** $\texttt{psig}$
else $\perp$

Oracle 4: The signing oracle

The response to an invalid query is a distinct failure symbol, and the response to a valid query is a valid partial signature $\texttt{psig}$ which is combinable with other valid signatures, and links to a challenge key, as follows.

- **Well-Formed Queries Combine to Valid Signatures.** If oracle response events $\texttt{psig}_{\texttt{vk}^*} \leftarrow \mathcal{O}_{\texttt{Sign}}(\texttt{msg}, \texttt{ring}, \texttt{tvk}, \underline{\texttt{vk}}, \texttt{vk}^*)$ occur for each $\texttt{vk}^* \in \underline{\texttt{vk}}$, the combination of these responses $\texttt{sig} = \texttt{Combine}(\texttt{msg}, \texttt{ring}, \underline{\texttt{vk}}, \underline{\texttt{psig}})$ is valid, $\texttt{Vf}(\texttt{msg}, \texttt{ring}, \texttt{sig}) = 1$.
- **Links to Challenge Key.** If $\texttt{msg}'$ is a message, $\underline{\texttt{vk}}'' \subseteq \underline{\texttt{vk}}'$ such that $\lvert \underline{\texttt{vk}}'' \rvert \geq r$, $\texttt{ring}'$ is a ring with $\texttt{tvk} \in \texttt{ring} \cap \texttt{ring}'$, and there exist signing oracle responses for each $\texttt{vk}^{**} \in \underline{\texttt{vk}}''$, $\texttt{psig}'_{\texttt{vk}^{**}} \leftarrow \mathcal{O}_{\texttt{Sign}}(\texttt{msg}', \texttt{ring}', \texttt{tvk}, \underline{\texttt{vk}}'', \texttt{vk}^{**})$, then $\texttt{Link}(\texttt{pars}_\lambda, \texttt{sig}, \texttt{sig}') = 1$ where $\texttt{sig}' = \texttt{Combine}(\texttt{msg}', \texttt{ring}', \underline{\texttt{vk}}'', \underline{\texttt{psig}}')$ is the combined signature.
- **Verifiability.** If Π is verifiable, then $\texttt{VfSh}(\texttt{msg}, \texttt{ring}, \underline{\texttt{vk}}, \texttt{psig}) = 1$

2. Eventually, $\mathcal{A}$ outputs some $\texttt{out}_{\mathcal{A}}$ which includes one or more message-ring-signature triples $(\texttt{msg}, \texttt{ring}, \texttt{sig})$ such that all the signatures are valid (in which case we say $\mathcal{A}$ succeeds) or a distinct failure symbol (in which case we say $\mathcal{A}$ fails).

Further assume that, if $\mathcal{A}$ requires more oracle queries allowed than the following bounds, or if $\mathcal{A}$ is about to make an oracle query which will cause the oracle to fail, then $\mathcal{A}$ outputs a distinct failure symbol and terminates.

1. There exists $\kappa_{\mathtt{key}}$, $\kappa_{\mathtt{corrupt}}$, $\kappa_{\mathtt{Sign}} \geq 0$ such that, in every successful transcript, $\mathcal{A}$ makes at most $\kappa_{\mathtt{key}}$ respective queries to $\mathcal{O}_{\mathtt{key}}$, at most $\kappa_{\mathtt{corrupt}}$ queries to $\mathcal{O}_{\mathtt{corrupt}}$, and at most $\kappa_{\mathtt{Sign}}$ queries to $\mathcal{O}_{\mathtt{Sign}}$.

2. For each random oracle, say H, to which $\mathcal{A}$ has access, there exists as similar integer κ_H such that $\mathcal{A}$ makes at most κ_H queries to the corresponding random oracle H in every successful transcript.

3. There exists an integer $n_{\mathtt{key}}$ such that all queries $\mathtt{query} = (n, r)$ made to $\mathcal{O}_{\mathtt{key}}$ satisfies $n \leq n_{\mathtt{key}}$ in every successful transcript.

If $\mathcal{A}$ succeeds with probability at least $\epsilon \in [0, 1]$ is (t, ϵ)-*player of the game with common setup* with key generation oracle access, corruption oracle access, and signing oracle access.

Winning this game is trivial, so the notion of security against players of this game is vacuous. However, players of our unforgeability and linkability games in Definition 2.8 and Definition 2.9 are also players of the game of common setup, just with nontrivial success conditions.

Definition 2.8 (LTM-SUF-1: Strong Unforgeability)**.** Let Π be an LTM scheme. Let $\mathcal{A}$ be any (t, ϵ)-player of the game with common setup with key generation, corruption, and signing oracle access such that every successful output of $\mathcal{A}$ has some $(\mathtt{msg}, \mathtt{ring}, \mathtt{sig}) \in \mathtt{out}_{\mathcal{A}}$ satisfying all the following.

1. The signature is valid, $\mathtt{Vf}(\mathtt{msg}, \mathtt{ring}, \mathtt{sig}) = 1$.

2. All ring members are challenge keys, $\mathtt{ring} \subseteq \mathcal{L}_{\mathtt{key}}$.

3. If all the following hold, then $\mathtt{Link}(\mathtt{sig}, \mathtt{sig}') = 1$:

   (a) there exists some $(j, \mathtt{tvk}_j) \in [m] \times \mathtt{ring}$, capacity $n \in \mathbb{N}$, and a threshold $r \in [n]$ such that a key generation oracle query event $(\mathtt{tvk}_j, \underline{\mathtt{vk}}) \leftarrow \mathcal{O}_{\mathtt{key}}(n, r)$ occurs,

   (b) there exists some $\mathtt{msg}' \in \{0, 1\}^*$, some $\mathtt{ring}' \in \mathcal{P}(\mathcal{TVK})$ such that $\mathtt{tvk}_j \in \mathtt{ring} \cap \mathtt{ring}' \setminus \mathcal{L}^{\mathtt{tot}}_{\mathtt{corrupt}}$, some signers' coalition of public verification keys $\underline{\mathtt{vk}}' \subseteq \underline{\mathtt{vk}}$ such that $|\underline{\mathtt{vk}}'| \geq r$, and events $\mathtt{psig}_{\mathtt{vk}'} \leftarrow \mathcal{O}_{\mathtt{Sign}}(\mathtt{msg}', \mathtt{ring}', \mathtt{tvk}_j, \underline{\mathtt{vk}}', \mathtt{vk}')$ for each $\mathtt{vk}' \in \underline{\mathtt{vk}}'$ which occurred, and

   (c) $\mathtt{sig}' = \mathtt{Combine}(\mathtt{msg}', \mathtt{ring}', \underline{\mathtt{vk}}', \underline{\mathtt{psig}})$.

4. For any $\mathtt{msg}'$, for any $\mathtt{ring}'$, for any $\underline{\mathtt{vk}}'$, if every event $(\mathtt{psig}_i \leftarrow \mathcal{O}_{\mathtt{Sign}}(\mathtt{msg}', \mathtt{ring}', \underline{\mathtt{vk}}', \mathtt{vk}_i)$ occurs for each $\mathtt{vk}_i \in \underline{\mathtt{vk}}$, then $\mathtt{sig} \neq \mathtt{Combine}(\mathtt{msg}', \mathtt{ring}', \underline{\mathtt{vk}}', \underline{\mathtt{psig}}')$.

Then we say $\mathcal{A}$ is an *LTM strong forger* for Π. Moreover, if $t \in O(\mathrm{poly}(\lambda))$ implies $\epsilon \in \mathrm{negl}(\lambda)$ for every LTM strong forger, then we say Π is *strongly unforgeable*.

We connect this definition to the definition of strong unforgeability for threshold digital signature schemes, TS-SUF-4 from [1]. Indeed, we include as "trivial" all ring signatures which are a superthreshold combinations of oracle-generated signature shares which all use a common query. This way, if an attacker can combine seemingly unrelated ring signature shares to obtain a valid signature, we count this as a forgery.

However, ring signatures have their own hierarchy of security definitions, and some of these depend on how many adversarially-selected ring members are allowable in a forgery. We call the previous definition LTM-SUF-1, because an unforgeable scheme under this definition stops forgers

from generating ostensibly valid signatures, but only when all ring members are challenge keys. Natural extensions may be a fruitful area of further research.

The following is, as far as the authors are aware, a novel definition of linkability for ring signatures.

Definition 2.9 (κ-Linkability)**.** Let Π be a LTM signature scheme and $\mathcal{A}$ be a (t, ϵ)-player of the game of common setup with key generation, corruption, and signing oracle access such that every successful output of $\mathcal{A}$ has some $\{(\mathtt{msg}_u, \mathtt{ring}_u, \mathtt{sig}_u)\}_{u\in[\kappa+1]}$ satisfying all the following properties.

1. For each $u \in [\kappa + 1]$, $\mathtt{Vf}(\mathtt{msg}_u, \mathtt{ring}_u, \mathtt{sig}_u) = 1$.
2. For each $u, v \in [\kappa + 1]$, $\mathtt{Link}(\mathtt{sig}_u, \mathtt{sig}_v) = \delta_{u,v}$, the Kronecker delta function.
3. At most κ keys can be under adversarial control, $\left|\cup_u \mathtt{ring}_u \setminus \overline{\mathcal{L}^{\mathtt{tot}}_{\mathtt{corrupt}}}\right| \leq \kappa$

Then we say $\mathcal{A}$ is a *κ-linkability breaker* for Π. Moreover, $t \in O(\mathrm{poly}(\lambda))$ implies $\epsilon \in \mathrm{negl}(\lambda)$ for every κ-linkability breaker, we say Π is *κ-linkable.*

Note that if we remove oracle access from Definition 2.9, we recover the notion of pigeonhole linkability. On the other hand, if we retain oracle access but set $\kappa = 1$, we recover the notion of ACST linkability.

3 FROSTLASS Construction

We now provide a formal description of FROSTLASS. We note that the definition provided here varies from the Rust implementation at [6] in several ways to improve readability, in a few security-neutral ways. We discuss these in section 3.3.

Definition 3.1. Let $F_{\mathtt{PRNG}}$ be a seedable pseudorandom number generator. FROSTLASS consists of the following algorithms.

1. $\mathtt{PGen}(\lambda) \to (q, \mathbb{G}, G, d, \underline{H})$ where $q \geq 1$ is a prime modulus, $\mathbb{G}$ is an abelian group of order q, $G \in \mathbb{G}$ is a generator, $d \in \mathbb{N}$ is a key dimension, and $\underline{H}$ are the following random oracles.
   (a) $H_{\mathtt{base}} : \{0,1\}^* \to \mathbb{G}$,
   (b) $H_{\mathtt{seed}} : \{0,1\}^* \to \{0,1\}^\lambda$,
   (c) $H_{\mathtt{FROST},i} : \{0,1\}^* \to \mathbb{Z}_q$ for each $i \in \mathbb{N}$,
   (d) $H^*_{\mathtt{lt}} : \{0,1\}^* \to \mathbb{Z}_q$.
   (e) $H_{\mathtt{lt},k} : \{0,1\}^* \to \mathbb{Z}_q$ for each $1 \leq k \leq d-1$,
   (f) $H_{\mathtt{ch}} : \{0,1\}^* \to \mathbb{Z}_q$.
2. $\mathtt{KGen}(n, r, \underline{z}) \to (\mathtt{tvk}, \underline{\mathtt{vk}}, \underline{\mathtt{lt}}, \underline{\mathtt{sk}})$. An interactive PPT algorithm which requires $n \geq 2$ participants. Participants decide upon a threshold $1 \leq r \leq n$, and scalars $\underline{z} = z_1, \ldots, z_{d-1} \in \mathbb{Z}_q$ via secure side channel; they share these data as common input. Participants do the following.
   (a) Participants use FROST key generation such that, for each $1 \leq i \leq n$, the i^{th} participant obtains the total verification FROST key Y, secret signing key share[†] y_i, and public verification key share Y_i.

[†]The secret share y_i is denoted s_i in the original FROST paper; however, we use s_i for signature data to maintain consistency with previous ring signature publications.

(b) For each $i \in [n]$, the i^{th} participant computes $Z_k = z_k G$ for each $k \in [d-1]$. These are called the *auxilliary keys*.

(c) Compute the *main linking tag share* $\mathfrak{T}_i = y_i H_{\mathtt{base}}(Y)$ and the *auxilliary linking tags* $\mathfrak{D}_k = z_k \cdot H_{\mathtt{base}}(Y)$ for $k \in [d-1]$.

(d) Set the following.

$$\begin{aligned}\mathtt{sk}_i &= (y_i, z_1, \ldots, z_{d-1}),\\ \mathtt{vk}_i &= (Y_i, Z_1, \ldots, Z_{d-1}),\\ \mathtt{tvk} &= (Y, Z_1, \ldots, Z_{d-1}), \text{ and}\\ \mathtt{lt}_i &= (\mathfrak{T}_i, \mathfrak{D}_1, \ldots, \mathfrak{D}_{d-1})\end{aligned}$$

The key $Y \in \mathtt{tvk}$ is called the *linking key*.

At the end of this process, each participant has learned their total verification key $\mathtt{tvk}$ for the group, secret key shares $\mathtt{sk}_i$, public key shares $\mathtt{vk}_i$, and linking tag share $(\mathtt{tvk}, \mathtt{vk}_i, \mathtt{lt}_i, \mathtt{sk}_i)$. We call these participants *signers*.

3. $\mathtt{Sign}(\mathtt{msg}, \mathtt{ring}, \underline{\mathtt{vk}}, \mathtt{sk}, (\mathtt{tlt}, \mathtt{com}, d, e)) \to \mathtt{psig}$. A non-interactive PPT algorithm individually carried out by signers. Signers are expected to interactively decide upon a message $\mathtt{msg}$, a ring $\mathtt{ring}$, a signers' coalition of public verification key shares $\underline{\mathtt{vk}}'$, a total linking tag $\mathtt{tlt}$, and a hash table $\mathtt{com}$ by secure side channel with authentication in a preprocessing step before executing $\mathtt{Sign}$; see $\mathtt{PreProc}$ below in section 3.1. Input a message $\mathtt{msg}$, a ring $\mathtt{ring} = (\mathtt{tvk}_1, \ldots, \mathtt{tvk}_m)$, a signers' coalition of public verification key shares $\underline{\mathtt{vk}}' = (\mathtt{vk}'_i)_{i=1}^r$, a secret key $\mathtt{sk}$, and auxiliary data $(\mathtt{tlt}, \mathtt{com}, d, e)$, where $\mathtt{tlt}$ is a total linking tag, $\mathtt{com} = \{(\mathtt{vk}_i, (D_i, E_i, D'_i, E'_i))\}_{i=1}^r$ is a hash table with keys $\mathtt{vk}_i \in \underline{\mathtt{vk}}$ and values which are quadruples $(D_i, E_i, D'_i, E'_i) \in \mathbb{G}^4$, and $d, e \in \mathbb{Z}_q$ are secret scalars.

The signer does the following.

(a) Find the index $j^* \in [m]$ such that the ring member $\mathtt{tvk}_{j^*} \in \mathtt{ring}$ is the total verification key. If no such index exists, output a distinct failure symbol and terminate.

(b) Find the index $i^* \in [r]$ such that $\mathtt{sk}$ corresponds to $\mathtt{vk}_{i^*} \in \underline{\mathtt{vk}}$. If no such index exists, output a distinct failure symbol and terminate.

(c) Parse:

i. $(y_{i^*}, z_1, \ldots, z_{d-1}) := \mathtt{sk}$,
ii. $(Y_i, Z_{i,1}, \ldots, Z_{i,d-1}) := \mathtt{vk}_i$ for $i \in [r]$,
iii. $\{(\mathtt{vk}_i, (D_i, E_i, D'_i, E'_i))\}_{i=1}^r := \mathtt{com}$,
iv. $(Y'_j, Z'_{j,1}, \ldots, Z'_{j,d-1}) := \mathtt{tvk}_j$ for $j \in [m]$, and
v. $(\mathfrak{T}, \mathfrak{D}_1, \ldots, \mathfrak{D}_{d-1}) := \mathtt{tlt}$.

(d) If there exists any $j \in [m]$, $k \in [d-1]$ such that $Z_{j,k} \neq Z'_{j,k}$, output $\perp$ and terminate.

(e) Compute:

i. the point hash $\widehat{Y}_{j^*} = H_{\mathtt{base}}(Y_{j^*})$.
ii. a seed[‡] $\gamma \leftarrow H_{\mathtt{seed}}(\underline{\mathtt{vk}} \,||\, \widehat{Y}_{j^*} \,||\, \mathtt{ring} \,||\, \mathtt{tlt} \,||\, \mathtt{msg} \,||\, \mathtt{com})$,
iii. the Lagrange coefficients $\lambda_i = \prod_{i' \neq i} \frac{i}{i'-i} (\mathrm{mod}\ q)$ for each $i \in [r] = \{1, 2, \ldots, r\}$.

[‡]A functionally equivalent implementation uses an extendable output function $\mathtt{xof}$ to extract $\underline{s}$ directly, bringing efficiency gains and reducing the risk of implementation errors.

iv. FROST coefficients for $i \in [r]$, $\rho_i = H_{\texttt{FROST},i}(\texttt{msg} \,||\, \widehat{Y}_{j^*} \,||\, \texttt{ring} \,||\, \underline{\texttt{vk}} \,||\, \underline{\texttt{lt}} \,||\, \texttt{com})$,

v. FROST nonces $F_i = D_i + \rho_i E_i$ for $i \in [r]$,

vi. FROST-like nonces $F'_i = D'_i + \rho_i E'_i$ for $i \in [r]$,

vii. starting nonces $L_{j^*} = \sum_{i=1}^{r} F_i$ and $R_{j^*} = \sum_{i=1}^{r} F'_i$, and

viii. starting signature challenge

$$c_{j^*+1} = H_{\texttt{ch}}(\texttt{dst}_{j^*} \,||\, \texttt{ring} \,||\, \texttt{tlt} \,||\, \underline{\widehat{Y}} \,||\, \mathfrak{W} \,||\, \underline{W} \,||\, \underline{\mu} \,||\, L_{j^*} \,||\, R_{j^*} \,||\, \texttt{msg})$$

where we identify $c_{m+1} \equiv c_1$ in the case that $j^* = m$.

(f) Sample $(s_j)_{j \neq j^*} \leftarrow F_{\texttt{PRNG}}(\gamma)$.

(g) Compute:

i. the point hashes of ring members' leading keys $\widehat{Y}_j = H_{\texttt{base}}(Y'_j)$.

ii. the aggregation coefficients

$$\begin{aligned} \mu_Y =& H^*_{\texttt{lt}}(\texttt{ring} \,||\, \texttt{tlt} \,||\, \underline{\widehat{Y}}), \text{ and} \\ \mu_k =& H_{\texttt{lt},k}(\texttt{ring} \,||\, \texttt{tlt} \,||\, \underline{\widehat{Y}}) \text{ for each } k \in [d-1], \end{aligned}$$

iii. the aggregated linking tag $\mathfrak{W} = \mu_Y \mathfrak{T} + \sum_{k=1}^{d-1} \mu_k \mathfrak{D}_k$,

iv. for each $j \in [m]$, the j^{th} aggregated ring member $W_j = \mu_Y Y'_j + \sum_{k=1}^{d-1} \mu_k Z'_{j,k}$,

v. For $j^* < j \leq m$, compute the nonces and signature challenges:

$$\begin{aligned} L_j =& s_j G + c_j W_j \\ R_j =& s_j \widehat{Y}_j + c_j \mathfrak{W} \\ c_{j+1} =& H_{\texttt{ch}}(\texttt{dst}_j \,||\, \texttt{ring} \,||\, \texttt{tlt} \,||\, \underline{\widehat{Y}} \,||\, \mathfrak{W} \,||\, \underline{W} \,||\, \underline{\mu} \,||\, L_j \,||\, R_j \,||\, \texttt{msg}) \end{aligned}$$

vi. Set $c_1 = c_{m+1}$ and, for $1 \leq j < j^*$, compute the nonces and signature challenges as in the previous step.

(h) Set $s_{j^*,i^*} = d + \rho_{i^*} e + \lambda_{i^*} \cdot c_{j^*} \cdot w_{i^*}$.

(i) Output $\texttt{psig} = (i^*, \texttt{lt}_{i^*}, c_1, s_1, \ldots, s_{j^*-1}, s_{j^*,i^*}, s_{j^*+1}, \ldots s_m)$.

4. $\texttt{Combine}(\texttt{msg}, \texttt{ring}, \underline{\texttt{psig}}) \to \texttt{sig}$. Input a message $\texttt{msg}$, a ring $\texttt{ring}$, and signature shares $\texttt{psig}_1, \ldots, \texttt{psig}_r$, and output a ring signature $\texttt{sig}$. Do the following.

(a) Parse $(\texttt{vk}_1, \ldots, \texttt{vk}_r) := \underline{\texttt{vk}}$, $(Y_i, Z_{i,1}, \ldots, Z_{i,d-1}) := \texttt{vk}_i$ for $i \in [r]$, and $Y := \texttt{tvk}$. Parse $(\texttt{tvk}_1, \ldots, \texttt{tvk}_m) := \texttt{ring}$ and find the index $1 \leq j^* \leq m$ in $\texttt{ring}$ such that $Y \texttt{tvk}_{j^*}$. Otherwise, output a distinct failure symbol and terminate.

(b) Parse each $(i'_i, \texttt{lt}_i, c_{i,1}, (s_{i,j})_{j=1}^m) := \texttt{psig}_i$ for each $i \in [r]$.

(c) If there exists indices $i_1 \neq i_2$ such that $\texttt{lt}_{i_1} = \texttt{lt}_{i_2}$, output a distinct failure symbol and terminate.

(d) Otherwise, if there exists indices $i_1 \neq i_2$ any signature challenges mismatch, $c_{i_1,1} \neq c_{i_2,1}$, output a distinct failure symbol and terminate.

(e) Otherwise, for any $1 \leq i_1, i_2 \leq r$, $1 \leq j \leq m$, if $j \neq j^*$ and $s_{i_1,j} \neq s_{i_2,j}$, output a distinct failure symbol and terminate.

(f) Otherwise, set $c_1 = c_{1,1}$, set $\widehat{s}_j = s_j$ for each $j \neq j^*$, set $\widehat{s}_{j^*} = \sum_{i=1}^{r} s_{i,j^*}$, and output the signature $\texttt{sig} = (c_1, (\widehat{s})_{j=1}^m, \texttt{tlt})$

5. $\texttt{Vf}(\texttt{msg}, \texttt{ring}, \texttt{sig}) \to b$. Input a message $\texttt{msg}$, a ring $\texttt{ring} = (\texttt{tvk}_1, \ldots, \texttt{tvk}_m)$, and a signature $\texttt{sig}$. Output a bit. Works as follows.
   (a) Parse $(c_1, s_1, \ldots, s_m, \mathfrak{T}, \mathfrak{D}_1, \ldots, \mathfrak{D}_{d-1}) := \texttt{sig}$.
   (b) If $\mathfrak{T} = 0$ or any $\mathfrak{D}_{d-1} = 0$, output 0 and terminate.
   (c) Using $j^* = 1$, execute step item 3g in $\texttt{Sign}$.
   (d) If $c_1 = c_{m+1}$, output 1 and terminate; otherwise output 0 and terminate.
6. $\texttt{Link}(\texttt{sig}, \texttt{sig}') \to b$. Do the following.
   (a) Parse the following.
      i. $(c_1, s_1, \ldots, s_m, \texttt{tlt}) := \texttt{sig}$, $(c'_1, s'_1, \ldots, s'_m, \texttt{tlt}') := \texttt{sig}'$,
      ii. $(\mathfrak{T}, \mathfrak{D}_1, \ldots, \mathfrak{D}_{d-1}) := \texttt{lt}$, $(\mathfrak{T}', \mathfrak{D}'_1, \ldots, \mathfrak{D}'_{d-1}) := \texttt{lt}'$.
   (b) Output a bit indicating whether $\mathfrak{T} = \mathfrak{T}'$.

Beware that signature shares leak information, like the true signer's ring index. Executing $\texttt{VfSh}$ or $\texttt{Combine}$ can only be safely done with other signers.

3.1 Extensions and Additional Algorithms

FROSTLASS (Definition 3.1) can be made verifiable as described in section 2.2 with a $\texttt{VfSh}$ algorithm as follows.

- $\texttt{VfSh}(\texttt{msg}, \texttt{ring}, \underline{\texttt{vk}}, \texttt{psig}, \texttt{tlt}, \texttt{com}) \to b$ inputs some $\texttt{msg}$, $\texttt{ring} = (\texttt{tvk}_1, \ldots, \texttt{tvk}_m)$, signers' coalition of key shares $\underline{\texttt{vk}} = (\texttt{vk}_i)_{i=1}^r$, a signature share $\texttt{psig} = (i^*, \texttt{lt}, c_1, s_1, \ldots, s_m)$, a total linking tag $\texttt{tlt}$, and a hash table $\texttt{com}$, and outputs a bit. Do the following.
   1. Set $j^* = 1$.
   2. Carry out step c of $\texttt{Sign}$.
   3. Parse $(i^*, \texttt{lt}, c_1, s_1, \ldots, s_m) := \texttt{psig}$.
   4. Carry out step d of $\texttt{Sign}$.
   5. Carry out step e of $\texttt{Sign}$. If any s_j mismatch their corresponding element in $\texttt{psig}$, output 0 and terminate.
   6. Carry out step f of $\texttt{Sign}$, to obtain each c'_j.
   7. If $c_1 \neq c'_1$, or $s_{j^*,i^*} G \neq \lambda_{i^*} c_{i^*} Y_{i^*}$, or $s_{j^*,i^*} \widehat{Y}_{j^*} \neq \lambda_{i^*} c_{j^*} \mathfrak{W}$, output 0 and terminate.
   8. Otherwise, output 1 and terminate.

FROSTLASS also admits a pre-processing step wherein participants may commit to their auxiliary signing data ahead of time and compute their total linking tag $\texttt{tlt}$, which works as follows.

- $\texttt{PreProc}(\texttt{tvk}, \underline{\texttt{vk}}, \underline{\texttt{sk}}) \to (\texttt{lt}, \texttt{com})$. An interactive PPT algorithm required to execute $\texttt{Sign}$ and $\texttt{VfSh}$, and which requires some $r \geq 1$ participants and a digital signature scheme Π_{DSS} as a subroutine. Input total verification key $\texttt{tvk}$, signers' coalition of public verification key shares $\underline{\texttt{vk}} = (\texttt{vk}_i)_{i=1}^r$, and secret signing key shares $\underline{\texttt{sk}} = (\texttt{sk}_i)_{i=1}^r$. Output a hash table $\texttt{com}$. Participants do the following.
   1. Parse $(\mathfrak{T}_i, \underline{\mathfrak{D}}^{(i)}) := \texttt{lt}_i$. If any $\underline{\mathfrak{D}}^{(i)} \neq \underline{\mathfrak{D}}^{(i')}$, output $\perp$ and terminate.

2. Otherwise, carry out step d.ii and d.iii from `Sign` to compute the Lagrange coefficients λ_i and the linking tag $\mathfrak{T} = \sum_i \mathfrak{T}_i$.
3. Compute the point hash of the linking key $\widehat{Y} = H_{\mathtt{base}}(Y)$.
4. For each $i \in [r]$, the i^{th} participant samples $(d_i, e_i) \in \mathbb{Z}_q^2$ and computes the following:

$$D_i = d_i G, \qquad D_i' = d_i \widehat{Y},$$
$$E_i = e_i G, \text{ and} \qquad E_i' = e_i \widehat{Y}.$$

5. The i^{th} participant sends $(\mathtt{lt}_i, D_i, E_i, D_i', E_i')$ in an authenticated all-to-all broadcast to the other signers[§].
6. After using Π_{DSS} to verify this communication, participants set `com` to be a hash table with keys $\mathtt{vk}_i$ and values $\mathtt{com}[\mathtt{vk}_i] = (D_i, E_i, D_i', E_i')$.

FROSTLASS also only links ring signatures according to whether the linking tags $\mathfrak{T}$ match, where $\mathfrak{T}$ is a collision-resistant image of the link of the signing key. That is, linking is not bound to the message, the other keys Z_k of the signing key, or the ring. Variations of this scheme binding linking to more data can provide a hierarchical expansion of linkability; this may be a fruitful area of further research.

3.2 Concrete Instantiation

To concretely implement random oracles in practice, we employ hash functions $H_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$, $H_{\mathbb{Z}_q} : \{0,1\}^* \to \mathbb{Z}_q$, and $H_\lambda : \{0,1\}^* \to \{0,1\}^\lambda$ with distinct domain separating tags $\mathtt{dst}_{\mathtt{label}} \in \{0,1\}^*$. For example, we concretely instantiate $H_{\mathtt{base}}$ by mapping $x \mapsto H_{\mathbb{G}}(\mathtt{dst}_{\mathtt{base}} \mid\mid x)$. Note that we can avoid domain separating tags and associated implementation errors without losing efficiency by using an extendable output function `xof` instead of hash functions throughout, extracting $(\mu_Y, \mu_1, \ldots, \mu_{d-1}) \leftarrow \mathtt{xof}(\mathtt{vk}_{j^*})$ directly with one function call.

3.3 Variations From Older Versions of CLSAG and the Rust Implementation

The scheme Definition 3.1 varies from both older versions of CLSAG and the Rust implementation at [6] in a few ways.

- We try to strictly follow the "hash the complete transcript" paradigm to prevent malleability in Definition 3.1. This causes some significant variations from previous versions of CLSAG and the Rust implementation; much more data is included in our hash pre-images.
  - For example, we include the point hashes of the ring members $\underline{\widehat{Y}}$ when computing aggregation coefficients in g.ii of `Sign`. This prevents an adversary from attempting to pick some aggregation coefficients before selecting a ring.
  - Similarly, we also include the aggregated key image $\mathfrak{W}$ in the preimage of every signature challenge computation. This prevents an adversary from attempting to pick a signature challenge before deciding upon an aggregated key image.

[§]Equivalently, all users may send their commitment points to a single member, who may then broadcast `com` back to all other users after executing step 3 of `PreProc`

This binding prevents malleability. We see no obvious way to violate any of our security properties without taking such care, but we do so as a matter of good practice for the formal definition of the scheme. Practical implementations do not need to be quite so stringent. For example, although CLSAG signatures do not usually compute c_{j+1} with the total linking tag `tlt` included in the pre-image, the most notable application of CLSAG signatures (the Monero cryptocurrency) includes these data in the message being signed. By including these data in `msg`, those applications essentially enforce (some of) the "include all data in all hashes" paradigm.

- The order of our computations in Definition 3.1 is not necessarily faithful to the order of computations in [6]. Our description in Definition 3.1 is ordered in a way which makes cross-referencing within this document easier, improving the readability of our description of `Vf` and `VfSh` substantially. None of our variations from [6] cause security problems and are rather superficial.

- All hashing in [6] is deterministically computed from transcript data. The approach in this repo largely follows the "hash the complete transcript" paradigm. Indeed, the Rust implementation appends data to running transcripts constructed in a canonical way. However, this transcript is pruned whenever data can be deterministically and verifiably computed from the state of the transcript at some point. For example, computing $x = H(\texttt{msg})$, $y = H(0 \mid\mid x)$, and $z = H(1 \mid\mid x)$ is safe; we do not need to compute $z = H(1 \mid\mid y \mid\mid x)$. This approach is consistent with, e.g. approaches in IETF standards like RFC 9591.

The same can be said of all other hashes and randomness; this enforces a degree of semi-honesty among users of this Rust implementation.

4 FROSTLASS Security

Correctness and linkability depend on the following lemmata, wherein we intentionally conflate linking tags $\mathfrak{T}$ with a function mapping a linking key to its linking tag. Recalling $H_{\mathbb{G}}$ was a hash function modeled as a random oracle (see section 3.2), this lemma establishes that, as a corollary, $\mathfrak{T}$ is indistinguishable from a random oracle.

Lemma 4.1. Let $\texttt{dst} \in \{0,1\}^*$ be a domain separating tag, let $\theta : \mathbb{G} \to \mathbb{G}$ be any function, let $\phi : \mathbb{Z}_q \times \mathbb{G} \to \mathbb{G}$ be the function defined by mapping $(x, Y) \mapsto x \cdot \theta(Y)$, and let $t_{\texttt{scmul}}$ denote the time it takes to multiply a point in $\mathbb{G}$ by a scalar in $\mathbb{Z}_q$. If some PPT (t, ϵ) algorithm $\mathcal{A}$ (or $\mathcal{B}$, respectively) is a κ-distinguisher for ϕ (or θ, respectively) under definition Definition 2.1, then there exists a PPT (t', ϵ') algorithm $\mathcal{A}'$ (or $\mathcal{B}'$, respectively) which is a κ-distinguisher for θ (or ϕ, respectively). □

Proof. Assume the algorithm $\mathcal{A}$ can distinguish ϕ from a random oracle in Definition 2.1. We build a $\mathcal{A}'$ to distinguish θ as follows.

1. $\mathcal{A}'$ is granted oracle access to $\mathcal{O}'_b : \mathbb{G} \to \mathbb{G}$.

2. $\mathcal{A}'$ runs $\mathcal{A}$ as a subroutine, handling oracle queries as follows. When $\mathcal{A}$ sends some query $(x, Y) \in \mathbb{Z}_q \times \mathbb{G}$, $\mathcal{A}'$ computes $Y' \leftarrow \mathcal{O}'_b(Y)$, sets $Z = xY'$, and responds with Z.

3. When $\mathcal{A}$ outputs b', $\mathcal{A}'$ outputs b'.

It is clear that this $\mathcal{A}'$ correctly plays the κ-random oracle distinguisher game for θ, succeeds if and only if $\mathcal{A}$ succeeds at distinguishing ϕ, and takes only the additional time to compute xY' from x and Y', i.e. a single scalar multiplication.

Likewise, assume $\mathcal{B}$ can distinguish θ from a random oracle. We build $\mathcal{B}'$ similarly.

1. $\mathcal{B}'$ is granted oracle access to $\mathcal{O}'_b : \mathbb{Z}_q \times \mathbb{G} \to \mathbb{G}$.
2. $\mathcal{B}'$ runs $\mathcal{B}$ as a subroutine, handling oracle queries as follows. When $\mathcal{B}$ sends some query $Y \in \mathbb{G}$, $\mathcal{B}'$ samples $x \xleftarrow{\$} \mathbb{Z}_q$, computes $Y' \leftarrow \mathcal{O}'_b(x, Y)$, sets $Z = x^{-1}Y'$, and responds with Z.
3. When $\mathcal{B}$ outputs b', $\mathcal{B}'$ outputs b'.

It is also clear that this $\mathcal{B}'$ correctly plays the κ-random oracle distinguisher game for ϕ, succeeds if and only if $\mathcal{B}$ succeeds at distinguishing θ, and takes additional time for sampling, inverting an element from $\mathbb{Z}_q$, and multiplying a point by a scalar, i.e. extra time $t_{\texttt{sample}} + t_{\texttt{inv}} + t_{\texttt{scmul}}$. □

Corollary 4.2. Let $r \geq 1$. Each of the maps $\mathfrak{T} : \mathbb{Z}_q \to \mathbb{G}$ and $\mathfrak{T}_i : \mathbb{Z}_q^r \times \mathbb{G} \to \mathbb{G}$ defined by mapping $y \mapsto yH_{\texttt{base}}(yG)$ and $(\underline{y}, Y) \mapsto y_i H_{\texttt{base}}(Y)$ are indistinguishable from random oracles.

4.1 Correctness

Theorem 4.3. FROSTLASS is a correct LTM scheme under Definition 2.6.

Proof. In event E_2, consider the ring signature shares $\texttt{psig}'_i$, $\texttt{psig}''_i$, and $\texttt{psig}^*_i$. These have some corresponding indices j', j'', and j^*, respectively, and we write these ring signature shares as follows.

$$\begin{aligned}
\forall i \in [|\underline{\texttt{vk}}'|], \texttt{psig}'_i &= (i, c'_1, s'_1, \ldots, s'_{j^*-1}, s'_{j^*,i}, s'_{j^*+1}, \ldots, s'_m), \\
\forall i \in [|\underline{\texttt{vk}}''|], \texttt{psig}'_i &= (i, c'_1, s'_1, \ldots, s''_{j^*-1}, s''_{j^*,i}, s''_{j^*+1}, \ldots, s''_m), \\
\forall i \in [|\underline{\texttt{vk}}^*|], \texttt{psig}^*_i &= (i, c^*_1, s^*_1, \ldots, s^*_{j^*-1}, s^*_{j^*,i}, s^*_{j^*+1}, \ldots, s^*_m)
\end{aligned}$$

Each signer with index $i \in [|\underline{\texttt{vk}}'|]$, computes the same seed, say γ' in event E_1, so $(s'_j)_{j \neq j',i}$ is identical in each $\texttt{psig}'_i$, where j' is the ring index of the true signer for $\texttt{psig}'_i$. Similarly, each signer with index $i \in [|\underline{\texttt{vk}}''|]$ computes the same seed, say γ'', and $(s''_j)_{j \neq j'',i}$ is identical in each $\texttt{psig}''_i$, where j'' is the ring index of the true signer for $\texttt{psig}''_i$. In event E_1^*, each signer with index $i \in [|\underline{\texttt{vk}}^{**}|]$ compute the same seed γ^*, and $(s^*_j)_{j \neq j^*,i}$ is identical in each $\texttt{psig}^*_i$, where j^* is the index of the true signer of $\texttt{psig}^*_i$. Moreover, these ring signature shares were all output from honest executions of `Sign`.

To show $\mathbb{P}[E_2] = 1$, it is sufficient to demonstrate that each $\texttt{psig}'_i$ passes `VfSh` in E_1. Indeed, the ring signature shares $\texttt{psig}''_i$ and $\texttt{psig}^*_i$ are shown to be valid in a similar way, *mutatis mutandis*.

In E_1, the points $L'_{j'} = \sum_i F_i$ and $R'_{j'} = \sum_i F'_i$ are computed along with the starting signature challenge $c_{j'+1}$.

Then, for $j^* < j \leq m$, the following computations take place.

$$\begin{aligned}
L'_j &= s'_j G + c'_j W'_j \\
R'_j &= s'_j \widehat{Y}'_j + c'_j \mathfrak{W} \\
c_{j+1} &= H_{\texttt{ch}}(\texttt{dst}_j \,||\, \texttt{ring} \,||\, \texttt{tlt} \,||\, \underline{\widehat{Y}} \,||\, \mathfrak{W} \,||\, \underline{W} \,||\, \underline{\mu} \,||\, L_j \,||\, R_j \,||\, \texttt{msg}')
\end{aligned}$$

Then the value $c_1 = c_{m+1}$ is set and, for $1 \leq j \leq j^*$, the same computations for L'_j, R'_j, c'_j take place. Lastly, each $s'_{j',i} = d'_i + \rho'_i e'_i + \lambda'_i c'_{j'} y'_i$ for the random scalars d'_i, e'_i, the corresponding aggregation coefficient ρ'_i, Lagrange interpolation coefficient λ'_i, and secret signing key share y'_i. Thus, in E_1,

the verifier computes γ' the same as the signer, and so samples $(s'_j)_{j \neq j',i}$ identically to all the signers. Moreover, for each $1 \leq j \leq m'$, the signature nonce points satisfy $L'_j = s'_j G + c'_j W'_j$, $R'_j = s'_j \widehat{Y}'_j + c'_j \mathfrak{W}$, and the signature challenges satisfy the verification equations by construction, except $j = j'$.

In event E_3, we have the following combined signatures.

$$\begin{aligned}
\mathtt{sig}' &= (c'_1, s'_1, \ldots, s'_{j'-1}, \sum_i s'_{j',i}, s'_{j'+1}, \ldots, s'_m, \mathtt{lt}') \\
\mathtt{sig}'' &= (c''_1, s''_1, \ldots, s''_{j''-1}, \sum_i s''_{j'',i}, s''_{j''+1}, \ldots, s''_m, \mathtt{lt}'') \\
\mathtt{sig}^* &= (c^*_1, s^*_1, \ldots, s^*_{j^*-1}, \sum_i s^*_{j^*,i}, s^*_{j^*+1}, \ldots, s^*_m, \mathtt{lt}^*)
\end{aligned}$$

Moreover, the aggregation coefficients, aggregated ring members, point hashes of ring members' leading keys, and the seed are all computed exactly as in `Sign` and `VfSh`. So, by construction, $c_1 = c_{m+1}$ and the circle of hashes pass verification.

Lastly, consider events E_4, E_5, and E_6. `Link` merely compares the linking tags. Moreover, as `Link` is a check for equality of linking tags, it is necessarily commutative. In event E_5, $\mathtt{sig}'$ and $\mathtt{sig}''$ are both computed from superthreshold subsets of $\underline{\mathtt{vk}}$ with the same corresponding `lt`, and so have identical linking tags $\mathfrak{T}$.

In event E_6, on the other hand, the signatures are computed for distinct $\mathtt{tvk} \neq \mathtt{tvk}'$. By Corollary 4.2, $\mathfrak{T}(\mathtt{tvk}) \neq \mathfrak{T}(\mathtt{tvk}')$ except with negligible probability, so $\mathtt{Link}(\mathtt{sig}', \mathtt{sig}^*) = \mathtt{Link}(\mathtt{sig}^*, \mathtt{sig}'') = 0$ except with negligible probability. □

4.2 Strong Unforgeability

Theorem 4.4. Let $\kappa_{\mathtt{ch}}, \kappa_{\mathtt{key}}, n_{\mathtt{key}} \geq 1$ be integer parameters. For every PPT (t, ϵ)-forger $\mathcal{A}$ as described in Definition 2.8, there exists a PPT (t', ϵ')-player of the $(n_{\mathtt{key}}\kappa_{\mathtt{key}} - 1)$-OMDL game such that $t' \in O(2t)$ and $\epsilon' \in O(\frac{\epsilon^2}{\kappa_{\mathtt{ch}}})$.

Proof. We solve the κ-OMDL game by constructing a tower of algorithms $\mathcal{A}_4 \to \mathcal{A}_3 \to \mathcal{A}_2 \to \mathcal{A}_1$, where $\mathcal{A}_1 = \mathcal{A}$ is a forger, $\mathcal{A}_2$ is a simulator of the unforgeability challenger for $\mathcal{A}_1$, $\mathcal{A}_3 = \mathtt{Fork}_{\mathcal{A}_2}$ is the forking algorithm of Definition 2.3, and $\mathcal{A}_4 = \mathcal{A}'$ plays the κ-OMDL game. These arrows indicate $\mathcal{A}_4$ runs $\mathcal{A}_3$ as a subroutine, and so on. We discuss these in order beginning with $\mathcal{A}_1$, the forger.

The forger. Let $\mathcal{A}_1$ be a (t_1, ϵ_1)-algorithm which is an LTM strong forger of FROSTLASS as described in Definition 2.8 and runs with some random tape $\tau_{\mathcal{A}_1}$. $\mathcal{A}_1$ has access to the oracles $\mathcal{O}_{\mathtt{key}}$, $\mathcal{O}_{\mathtt{corrupt}}$, and $\mathcal{O}_{\mathtt{Sign}}$ from Definition 2.7 via Definition 2.8, and all the random oracles $H_{\mathtt{label}}$ with $\mathtt{label} \in \{\mathtt{base}, \mathtt{seed}, (\mathtt{FROST}, i), (\mathtt{kb}, k), \mathtt{ch}\}$ for $i \in \mathbb{N}$ and $k \in [d-1]$ from Definition 3.1.

Wrap the forger. We first wrap $\mathcal{A}_1$ in an algorithm $\mathcal{A}_2$. This $\mathcal{A}_2$ simulates the forgery challenger for $\mathcal{A}_1$ and is compatible with Definition 2.3, and is a helper algorithm for playing Definition 2.2 which requires oracle access; we denote the oracles of Definition 2.2 with $\mathcal{O}^*_{\mathtt{key}}$ and $\mathcal{O}^*_{\mathtt{corrupt}}$ to prevent confusion. $\mathcal{A}_2$ works as follows.

1. Initialize empty tables $T_{\mathtt{label}}$ for $\mathtt{label} \in \{\mathtt{base}, \mathtt{seed}, (\mathtt{FROST}, i), (\mathtt{lt}, k), \mathtt{ch}, \mathtt{DL}\}$.

2. Run $\mathcal{A}_1$ as a subroutine. As a simulator of the game of Definition 2.8, $\mathcal{A}_2$ handles all oracle queries made by $\mathcal{A}_1$ as follows.

(a) For $\texttt{label} \in \{\texttt{seed}, (\texttt{FROST}, i), (\texttt{lt}, y), (\texttt{lt}, k) \mid i \in [n_{\texttt{key}}], k \in [d-1]\}$, when $\mathcal{A}_1$ queries $H_{\texttt{label}}$, $\mathcal{A}_3$ simulates responses using its own internal random tape, resampling in the event of a collision, and storing query-response pairs as key-value pairs in hash tables $T_{\texttt{label}}$ to maintain consistency with later queries. We assume handling these queries requires no other oracle queries, takes negligible time, and certainly succeeds. These simulations are indistinguishable from real oracles, as they are directly simulated from the random tape of $\mathcal{A}_2$.

(b) When $\mathcal{A}_1$ makes some $\texttt{query}$ to $H_{\texttt{ch}}$ and $\texttt{query} \notin T_{\texttt{ch}}$, $\mathcal{A}_2$ computes $i = |T_{\texttt{ch}}| + 1$, finds $h_i \in \underline{h}$, stores $T_{\texttt{ch}}[\texttt{query}] = (i, h_i)$, and responds with h_i. We assume handling these queries requires no other oracle queries, takes negligible time, and succeeds with certainty.

(c) When $\mathcal{A}_1$ makes some $\texttt{query}$ to $H_{\texttt{base}}$, $\mathcal{A}_2$ checks if $\texttt{query} \notin T_{\texttt{base}}$. If so, $\mathcal{A}_2$ samples $\alpha \leftarrow \mathbb{Z}_q$, resampling in the case of a collision, and sets $T_{\texttt{base}}[\texttt{query}] = \alpha$. The response is computed in two cases.

i. If $\texttt{query} \notin \mathbb{G}$, then $\mathcal{A}_2$ samples $Y \leftarrow \mathbb{G}$ and responds with αY.

ii. Otherwise, $\mathcal{A}_2$ parses $Y \leftarrow \texttt{query}$, and responds with αY.

This query requires no further oracle access, takes the time to sample $\alpha \xleftarrow{\$} \mathbb{Z}_q$, possibly the time it takes to sample $Y \in \mathbb{G}$, and the time it takes to compute αY, a scalar multiplication of a point. Thus, this query takes time at most $t_{\texttt{base}} \approx t_q + t_{\mathbb{G}} + t_{\texttt{scmul}}$, where t_q is the time it takes to sample α, $t_{\mathbb{G}}$ is the time it takes to sample $Y \in \mathbb{G}$, and $t_{\texttt{scmul}}$ is the time it takes to compute αY. This query certainly succeeds.

(d) When $\mathcal{A}_1$ queries $\mathcal{O}_{\texttt{key}}$ with some pair (n, r), $\mathcal{A}_2$ does the following.

i. If $r \notin [n]$, output a distinct failure symbol and terminate.

ii. Otherwise, for each $i \in [n]$, make a query $Y_i \leftarrow \mathcal{O}^*_{\texttt{key}}(*)$ from Definition 2.2.

iii. Compute $Y = \sum_{i \in [n]} Y_i$.

iv. Sample $(z_1, \ldots, z_{d-1}) \leftarrow \mathbb{Z}_q^{d-1}$.

v. Compute each $Z_k = z_k G$ and store $T_{\texttt{DL}}[Z_k] = z_k$.

vi. Simulate a query made to $H_{\texttt{base}}$ by $\mathcal{A}_1$, $\widehat{Y} = H_{\texttt{base}}(Y)$.

vii. Retrieve $\alpha = T_{\texttt{base}}[Y]$; this table entry is non-empty with certainty due to the previous step.

viii. Set $\mathfrak{T} = \alpha Y$, $\mathfrak{T}_i = \alpha Y_i$, and each $\mathfrak{D}_k = \alpha Z_k$ for each $i \in [n]$ and each $k \in [d-1]$.

ix. Set $\texttt{tvk} = (Y, Z_1, \ldots, Z_{d-1})$, $\texttt{lt} = (\mathfrak{T}, \mathfrak{D}_1, \ldots, \mathfrak{D}_{d-1})$, each $\texttt{vk}_i = (Y_i, Z_1, \ldots, Z_{d-1})$, and each $\texttt{lt}_i = (\mathfrak{T}_i, \mathfrak{D}_1, \ldots, \mathfrak{D}_{d-1})$.

x. Store $T_{\texttt{lt}}[\texttt{vk}_i] = \texttt{lt}_i$ and $T_{\texttt{tlt}}[\texttt{vk}] = \texttt{tlt}$.

xi. Respond with $(\texttt{tvk}, \underline{\texttt{vk}})$.

Handling one $\mathcal{O}_{\texttt{key}}$ query takes n queries to $\mathcal{O}^*_{\texttt{key}}$ and one query to $H_{\texttt{base}}$, n^2 sums of points from $\mathbb{G}$, $(d-1)$ samples from $\mathbb{Z}_q$, and $2d+1$ scalar multiplications against points. This query takes time at most $t_{\texttt{key}} \approx n t^*_{\texttt{key}} + n^2 t_+ + (d-1) t_q + (1+2d) t_{\texttt{scmul}} + t_{\texttt{base}}$, where $t^*_{\texttt{key}}$ is the time it takes to query $\mathcal{O}^*_{\texttt{key}}$, t_+ is the time it takes to sum two arbitrary group elements, and $t_{\texttt{base}}$ is the time it takes to simulate a query $H_{\texttt{base}}$. This query succeeds with certainty.

(e) When $\mathcal{A}_1$ queries $\mathcal{O}_{\texttt{corrupt}}$ with some $(i, (\texttt{tvk}, \underline{\texttt{vk}}))$, $\mathcal{A}_2$ parses $(Y_i, Z_1, \ldots, Z_{d-1}) := \texttt{vk}_i$. If this is not possible, then $\mathcal{O}_{\texttt{corrupt}}$ responds with a distinct failure symbol. Otherwise, $\mathcal{A}_2$ looks up $z_k \leftarrow T_{\texttt{DL}}[Z_k]$ for each k. If this is not possible, then $\mathcal{O}_{\texttt{corrupt}}$ responds with

a distinct failure symbol. Otherwise, $\mathcal{A}_2$ queries $y_i \leftarrow \mathcal{O}^*_{\texttt{corrupt}}(Y_i)$, sets $T_{\texttt{corrupt}}[Y_i] = y_i$, and responds with $(y_i, z_1, \ldots, z_{d-1})$.

This query takes one query to $\mathcal{O}^*_{\texttt{corrupt}}$ and $d-1$ retrievals from $T_{\texttt{DL}}$. As this is a hash table, lookups are constant-time, so this query takes time at most $t_{\texttt{corrupt}} = t^*_{\texttt{corrupt}} + (d-1)O(1)$. Moreover, this query fails if and only if $\mathcal{A}_1$ makes a query that is not a challenge key. By assumption, $\mathcal{A}_1$ prefers to fail than to make a failed corruption oracle query, so without loss of generality, this corruption oracle certainly succeeds.

This is also the only oracle which may fail if queried incorrectly, say with non-challenge data.

(f) When $\mathcal{A}_1$ queries $\mathcal{O}_{\texttt{Sign}}$ with some $(\texttt{msg}, \texttt{ring}, \texttt{tvk}, \underline{\texttt{vk}}, \texttt{vk})$, $\mathcal{A}_2$ does the following to back-patch an ostensibly valid signature.

i. If $(\texttt{tvk}, \underline{\texttt{vk}}')$ does not appear as a $\mathcal{O}_{\texttt{key}}$ response to $\mathcal{A}_1$ for any $\underline{\texttt{vk}} \subseteq \underline{\texttt{vk}}'$, or $\texttt{tvk} \notin \texttt{ring}$, output a distinct failure symbol and terminate.

ii. Otherwise, there is some query made by $\mathcal{A}_1$ to $\mathcal{O}_{\texttt{key}}$ with keys matching this $\mathcal{O}_{\texttt{Sign}}$ query, say $(\texttt{tvk}, \underline{\texttt{vk}}') \leftarrow \mathcal{O}_{\texttt{key}}(n, r)$ occurred for some $\underline{\texttt{vk}} \subseteq \underline{\texttt{vk}}'$. If $|\underline{\texttt{vk}}| < r$, or $|\underline{\texttt{vk}}'| \neq n$, output a distinct failure symbol and terminate.

iii. Otherwise, there is a superthreshold number of signers in the coalition and the correct total number of keyholders. Parse $(Y, Z_1, \ldots, Z_{d-1}) := \texttt{tvk}$.

iv. Retrieve $\alpha = T_{\texttt{base}}[Y]$, then compute $\mathfrak{T} = \alpha Y$ and each $\mathfrak{D}_k = \alpha Z_k$. Set $\texttt{tlt} = (\mathfrak{T}, \mathfrak{D}_1, \ldots, \mathfrak{D}_{d-1})$.

v. Simulate a query made to $H_{\texttt{base}}$ by $\mathcal{A}_1$, say $\widehat{Y}_j = H_{\texttt{base}}(Y_j)$, for each ring members' linking keys Y_j (i.e. for each $j \in [m]$).

vi. Simulate a query made to $H^*_{\texttt{lt}}$ and $H_{\texttt{lt},k}$ from $\mathcal{A}_1$ for each $k \in [d-1]$ to obtain $\mu_Y = H^*_{\texttt{lt}}(\texttt{ring} \,||\, \texttt{tlt} \,||\, \underline{\widehat{Y}})$ and $\mu_k = H_{\texttt{lt},k}(\texttt{ring} \,||\, \texttt{tlt} \,||\, \underline{\widehat{Y}})$ for each $k \in [d-1]$.

vii. Compute $W_j = \mu_Y Y_j + \sum_k \mu_k Z_{j,k}$ for each ring member $\texttt{tvk}_j = (Y_j, Z_{j,1}, \ldots, Z_{j,d-1})$.

viii. Compute $\mathfrak{W} = \mu_Y \mathfrak{T} + \sum_k \mu_k \mathfrak{D}_k$.

ix. Sample $s_1, \ldots, s_m \leftarrow \mathbb{Z}_q$.

x. Retrieve $i^* = |T_{\texttt{ch}}| + 1$ and the challenges $h_{i^*}, h_{i^*+1}, \ldots, h_{i^*+m-1} \leftarrow \underline{h}$.

xi. Find $j^* \in [m]$ such that $\texttt{tvk}_{j^*} = \texttt{tvk}$.

xii. Set the following:

$$
\begin{aligned}
c_{j^*+1} &= h_{i^*} & \qquad c_m &= h_{i^*+(m-j^*-1)} \\
c_{j^*+2} &= h_{i^*+1} & \qquad c_1 &= h_{i^*+(m-j^*)} \\
c_{j^*+3} &= h_{i^*+2} & \qquad c_2 &= h_{i^*+(m-j^*)+1} \\
&\vdots & \qquad &\vdots \\
c_{m-2} &= h_{i^*+(m-j^*-3)} & \qquad c_{j^*-1} &= h_{i^*+(m-2)} \\
c_{m-1} &= h_{i^*+(m-j^*-2)} & \qquad c_{j^*} &= h_{i^*+(m-1)}
\end{aligned}
$$

xiii. If any $\texttt{query} = (\texttt{dst}_j \,||\, \texttt{ring} \,||\, \texttt{tlt} \,||\, \underline{\widehat{Y}} \,||\, \mathfrak{W} \,||\, \underline{W} \,||\, \underline{\mu} \,||\, L_j \,||\, R_j \,||\, \texttt{msg}) \in T_{\texttt{ch}}$, then output a distinct failure symbol and terminate.

xiv. For $j \in [m]$, $T_{\texttt{ch}}(\texttt{dst}_j \,||\, \texttt{ring} \,||\, \texttt{tlt} \,||\, \underline{\widehat{Y}} \,||\, \mathfrak{W} \,||\, \underline{W} \,||\, \underline{\mu} \,||\, L_j \,||\, R_j \,||\, \texttt{msg}) := c_{j+1}$.

xv. Set $\texttt{sig} = (c_1, s_1, \ldots, s_m, \mathfrak{T}, \mathfrak{D}_1, \ldots, \mathfrak{D}_{d-1})$.

xvi. Respond with $\texttt{sig}$.

This query requires one simulated query to $H^*_{\mathtt{lt}}$, one query to $H_{\mathtt{lt},k}$ for each $k \in [d-1]$, and one query to $H_{\mathtt{base}}$ for each ring member. This query requires a lookup in $T_{\mathtt{key}}$, a lookup in $T_{\mathtt{key}}$, a lookup in $T_{\mathtt{base}}$, and m lookups in $\underline{h}$. This query takes time $t_{\mathtt{Sign}} \approx t_{\mathtt{lt},y} + (d-1)t_{\mathtt{lt},k} + t_{\mathtt{base}} + (2+m)O(1)$. The only way this algorithm fails is if $\mathcal{A}_1$ makes a poorly-formed query. By assumption, $\mathcal{A}_1$ prefers to output a distinct failure symbol than do so. That is, this simulation of $\mathcal{O}_{\mathtt{Sign}}$ succeeds with certainty.

3. If $\mathcal{A}_1$ outputs a distinct failure symbol and terminates, then $\mathcal{A}_2$ outputs a distinct failure symbol and terminates.

4. Otherwise, $\mathtt{out}_1 \leftarrow \mathcal{A}_1$. In this event, $\mathcal{A}_2$ parses $(\mathtt{msg}, \mathtt{ring}, \mathtt{sig}) \leftarrow \mathtt{out}_1$, sets $m = |\mathtt{ring}|$, and then does the following.

   (a) Find all queries $\mathtt{query} \in T_{\mathtt{ch}}$, $\ell \in [\kappa_{\mathtt{ch}}]$, $j \in [m]$, $c \in \mathbb{Z}_q$ such that $T_{\mathtt{ch}}[\mathtt{query}] = (\ell, c)$, $\mathtt{tvk}_j \in \mathtt{ring}$, and $c = c_{j+1}$ is used to verify $\mathtt{sig}$; call this set of queries S.

   (b) If $S = \emptyset$, output a distinct failure symbol and terminate.

   (c) Otherwise, find the argument $\mathtt{query} \in S$ which minimizes $\ell \in T_{\mathtt{ch}}[\mathtt{query}]$; i.e. the index of the first $H_{\mathtt{ch}}$ query used during verification.

   (d) Set $\mathtt{tables}$ to consist of all the tables $\{T_{\mathtt{label}}\}$.

   (e) Set $\mathtt{out}_2 = (\ell, \mathtt{out}_1, \mathtt{tables})$ for this minimal ℓ.

5. Output $\mathtt{out}_2$.

The random oracles $H_{\mathtt{seed}}$, $H_{\mathtt{FROST},i}$, $H_{\mathtt{lt},k}$, and $H_{\mathtt{base}}$ are all clearly simulated correctly, in the standard way which is indistinguishable from random oracles. Moreover, $H_{\mathtt{ch}}$ is also simulated correctly, up to the quality of the randomness used for the input $\underline{h}$. $\mathcal{O}_{\mathtt{corrupt}}$ is also correct, as the simulator knows the table $T_{\mathtt{DL}}$ and has access to $\mathcal{O}^*_{\mathtt{corrupt}}$. Certainly $\mathcal{O}_{\mathtt{key}}$ is simulated correctly, as the computation of the Y_i and Y points correctly simulates FROST key generation, and the remainder of the response is computed honestly. Now, consider $\mathcal{O}_{\mathtt{Sign}}$. By construction, the output of $\mathcal{O}_{\mathtt{Sign}}$ is a ring signature which passes verification. Moreover, since $H_{\mathtt{lt},k}$ and $H_{\mathtt{base}}$ are simulated correctly, this simulation of $\mathcal{O}_{\mathtt{Sign}}$ is correct, also, at least up to randomness used to sample $\underline{h}$. Thus, $\mathcal{A}_2$ is a correct simulation of the unforgeability challenger for $\mathcal{A}_1$.

Now, consider the runtime of $\mathcal{A}_2$. If $t_{\mathtt{label}}$ denotes the time it takes to simulate each query made to $H_{\mathtt{label}}$ or $\mathcal{O}_{\mathtt{label}}$, and $\kappa_{\mathtt{label}}$ is the number of such queries, then $\mathcal{A}_2$ takes up to $\kappa_{\mathtt{label}} t_{\mathtt{label}}$ time for all these queries to $H_{\mathtt{label}}$. So, $\mathcal{A}_2$ takes time

$$\begin{aligned} t_2 &\approx t_1 + n_{\mathtt{key}}\kappa_{\mathtt{key}} t_{\mathtt{key}} + \kappa_{\mathtt{Sign}} t_{\mathtt{Sign}} + \kappa_{\mathtt{base}} t_{\mathtt{base}} + \sum_{\mathtt{label} \notin \{\mathtt{key},\mathtt{Sign},\mathtt{base}\}} \kappa_{\mathtt{label}} t_{\mathtt{label}} \\ &\approx t_1 + n_{\mathtt{key}}\kappa_{\mathtt{key}} t_{\mathtt{key}} + \kappa_{\mathtt{Sign}} t_{\mathtt{Sign}} + \kappa_{\mathtt{base}} t_{\mathtt{base}} \end{aligned}$$

where the times $t_{\mathtt{label}}$ in the sum are assumed to be negligible, and where $\mathcal{A}_1$ makes some $\kappa_{\mathtt{key}}$ queries to $\mathcal{O}_{\mathtt{key}}$, and each of these is handled with up to $n_{\mathtt{key}}$ queries to $\mathcal{O}^*_{\mathtt{key}}$. Although we assume querying $\mathcal{O}^*_{\mathtt{key}}$ takes negligible time, the wrapper used to simulate responses from $\mathcal{O}_{\mathtt{key}}$ requires sampling randomness and assembling the response.

Now consider the success probability. Certainly $\mathcal{A}_2$ succeeds at simulating all oracle queries, so $\mathcal{A}_2$ can only terminate with a distinct failure symbol if $\mathcal{A}_1$ fails, or if no index ℓ exists as described above. However, we claim that if if $\mathcal{A}_1$ succeeds, then $T_{\mathtt{ch}}$ contains a suitable pair (ℓ, c) except with negligible probability. Indeed, if $\mathcal{A}_1$ outputs a successful forgery with ring signature

$\mathtt{sig} = (c_1, \underline{s}, \mathtt{tlt})$ which passes verification, then $\mathcal{A}_1$ selected this signature to satisfy the verification equations

For this forgery to pass verification, these c_j must be consistent with responses from the random oracle $H_{\mathtt{ch}}$ when queried by a verifier. So, $\mathcal{A}_1$ guessed or queried $H_{\mathtt{ch}}$ for these c_j. Guessing one $\mathbb{Z}_q$ output of $H_{\mathtt{ch}}$ successfully from amongst some $\kappa \in \mathbb{N}$ queries succeeds with probability at most $\prod_{i\in[\kappa]}(q-i)^{-1}$, which is negligible in q. Since $q \in O(\mathrm{poly}(\lambda))$, this probability is negligible in λ. Thus, the probability that no index ℓ can be found is negligible, and $\epsilon_2 \approx \epsilon_1$.

Fork this simulator. Note that $\mathcal{A}_2$ is compatible with Definition 2.3, leading to the forking algorithm. Define $\mathcal{A}_3$ to be similar to $\mathtt{Fork}_{\mathcal{A}_2}$ as in Definition 2.3, except with oracle access to $\mathcal{O}^*_{\mathtt{key}}$ and $\mathcal{O}^*_{\mathtt{corrupt}}$ from Definition 2.2. Then Lemma 2.4 implies $\mathcal{A}_3$ is a PPT (t_3, ϵ_3)-algorithm where

$$t_3 = 2t_2 + t_2' \approx 2t_2 \in \mathrm{negl}(\lambda)$$
$$\epsilon_3 = \epsilon_2\left(\frac{\epsilon_2}{\kappa_{\mathtt{ch}}} - \frac{1}{q-1}\right) \in O\left(\frac{\epsilon_2^2}{\kappa_{\mathtt{ch}}}\right)$$

where $t_2' \in \mathrm{negl}(\lambda)$ is the additional time it takes to sample randomness in Definition 2.3.

Solve OMDL. Lastly, we build an algorithm $\mathcal{A}_4$ which has access to $\mathcal{O}^*_{\mathtt{key}}$ and $\mathcal{O}^*_{\mathtt{corrupt}}$ as defined in Definition 2.2 (and which we assume take negligible time to invoke), runs $\mathcal{A}_3$ as a subroutine, and plays the κ-OMDL game in time at most t_4 and succeeds with probability at least ϵ_4 as follows, where $\kappa = n_{\mathtt{key}}\kappa_{\mathtt{key}} - 1$.

Observe that $\mathcal{A}_2$ makes some $\mathtt{query}$ at the fork point. Moreover, this fork point is selected so that the response appears in verification. Thus, $\mathtt{query} = (\mathtt{dst}_j \,||\, \mathtt{ring} \,||\, \mathtt{tlt} \,||\, \widehat{\underline{Y}} \,||\, \mathfrak{W} \,||\, \underline{W} \,||\, \underline{\mu} \,||\, L_j \,||\, R_j \,||\, \mathtt{msg})$ for some $j \in [m]$ and for some points $L_j, R_j \in \mathbb{G}$, and $T_{\mathtt{ch}}[\mathtt{query}] = c_{j+1}$. All data available to $\mathcal{A}_2$, as well as its random tape, are identical until the fork point. Thus, the queries are identical on both sides of the fork with certainty, but by the definition of Definition 2.3, the responses vary in both transcripts except with negligible probability. In particular, the points L_j and R_j are common to both transcripts with certainty. That is to say, we see the same query with two responses appear as in the following diagram.

$$H_{\mathtt{ch}}(\mathtt{dst}_j \,||\, \mathtt{ring} \,||\, \mathtt{tlt} \,||\, \widehat{\underline{Y}} \,||\, \mathfrak{W} \,||\, \underline{W} \,||\, \underline{\mu} \,||\, L_j \,||\, R_j \,||\, \mathtt{msg})$$
$$\downarrow \quad \downarrow$$
$$c_{j+1} \neq c'_{j+1}$$

That is to say, on one side of the fork, the same query yields the response c_{j+1}, and on the other side of the fork, the different response $c_{j+1} \neq c'_{j+1}$. Since $\mathtt{lt} = (\mathfrak{T}, \mathfrak{D}_1, \ldots, \mathfrak{D}_{d-1})$ appears in this query, these points are certainly identical on both sides of the fork.

1. Run $\mathcal{A}_3$ as a subroutine, responding to $\mathcal{O}^*_{\mathtt{key}}$ and $\mathcal{O}^*_{\mathtt{corrupt}}$ queries by consulting these oracles and responding faithfully. If $\mathcal{A}_3$ fails, then $\mathcal{A}_4$ outputs a distinct failure symbol and terminates. This step takes time t_3 and succeeds (terminates without a failure symbol) with probability ϵ_3.

2. Otherwise, $\mathtt{out}_{\mathcal{A}_3} \leftarrow \mathcal{A}_3$. If $\mathcal{A}_4$ cannot parse $(\ell, (\mathtt{out}_{\mathcal{A}_1}, \mathtt{tables}), (\mathtt{out}'_{\mathcal{A}_1}, \mathtt{tables}')) := \mathtt{out}_{\mathcal{A}_3}$, then $\mathcal{A}_4$ outputs a distinct failure symbol and terminates. This takes negligible time. By construction of $\mathcal{A}_4$, this step, conditioned on the success of the previous step, certainly succeeds.

3. Otherwise, $\mathcal{A}_4$ attempts to parse the following.

$$(\mathtt{msg}, \mathtt{ring}, \mathtt{sig}) := \mathtt{out}_{\mathcal{A}_1} \qquad (\mathtt{msg}', \mathtt{ring}', \mathtt{sig}') := \mathtt{out}'_{\mathcal{A}_1}$$
$$(\mathtt{tvk}_1, \ldots, \mathtt{tvk}_m) := \mathtt{ring}, \qquad (\mathtt{tvk}'_1, \ldots, \mathtt{tvk}'_{m'}) := \mathtt{ring}',$$
$$(c_1, \underline{s}, \mathtt{tlt}) := \mathtt{sig} \qquad (c'_1, \underline{s}', \mathtt{tlt}') := \mathtt{sig}'$$
$$(\mathfrak{T}, \underline{\mathfrak{D}}) := \mathtt{tlt} \qquad (\mathfrak{T}', \underline{\mathfrak{D}}') := \mathtt{tlt}$$

   If $\mathcal{A}_4$ cannot parse this, then $\mathcal{A}_4$ outputs a distinct failure symbol and terminates. By construction of $\mathcal{A}_3$, this step, conditioned on the success of the previous step, certainly succeeds.

4. For each $j \in [m]$, $\mathcal{A}_4$ searches for each $\mathtt{query}_j$ such that $T_{\mathtt{ch}}[\mathtt{query}_j] = (\ell_j, c_{j+1})$. This ℓ_j is the $H_{\mathtt{ch}}$ query index whose response is c_{j+1} used during verification. For the signature challenge c_{j+1} used in verification, call this set S_1. Recalling that the forger makes all these queries to $H_{\mathtt{ch}}$ in every successful transcript except with negligible probability, then conditioned on the success of the previous steps, this step succeeds except with negligible probability. Moreover, this step takes time at most $O(\kappa_{\mathtt{ch}})$ in case we must touch every entry in $T_{\mathtt{ch}}$.

5. Find the $j^* \in [m]$ such that $\mathtt{query}_{j^*} \in S$ minimizes $\ell_{j^*} \in T_{\mathtt{ch}}[\mathtt{query}_{j^*}]$. This way, c_{j^*} is the first oracle response used during verification.

6. $\mathcal{A}_4$ parses $(\mathtt{dst}_{j^*} \,||\, \mathtt{ring} \,||\, \mathtt{tlt} \,||\, \underline{\widehat{Y}} \,||\, \mathfrak{W} \,||\, \underline{W} \,||\, \underline{\mu} \,||\, L_{j^*} \,||\, R_{j^*} \,||\, \mathtt{msg}) := \mathtt{query}_{j^*}$ for some points $L_{j^*}, R_{j^*} \in \mathbb{G}$. If $\mathcal{A}_4$ cannot do this, then $\mathcal{A}_4$ outputs a distinct failure symbol and terminates. This takes negligible time. Say this step succeeds with probability ϵ', and see below.

7. For this j^*, parse $(Y_{j^*}, Z_{j^*,1}, \ldots, Z_{j^*,d-1}) := \mathtt{tvk}_{j^*}$ (where $\mathtt{tvk}_{j^*} \in \mathtt{ring}$). Retrieve $\mu_y \leftarrow T^*_{\mathtt{lt}}[\mathtt{ring} \,||\, \mathtt{tlt}]$ and, for each $k \in [d-1]$, $\mu_k \leftarrow T_{\mathtt{lt},k}[\mathtt{ring} \,||\, \mathtt{tlt}]$.

8. $\mathcal{A}_4$ computes $\widetilde{w}_{j^*} = \frac{s_{j^*} - s'_{j^*}}{c'_{j^*} - c_{j^*}}$. This takes the time of two negations, two additions, an inversion, and a multiplication in $\mathbb{Z}_q$, say $2t_{\mathtt{neg}} + 2t_{+} + t_{\mathtt{inv}} + t_{\mathtt{mul}}$. By the construction of $\mathcal{A}_3$, and conditioned on the previous step's success, $c'_{j^*} \neq c_{j^*}$ with certainty, so this step succeeds with certainty.

9. $\mathcal{A}_4$ finds a response $(\mathtt{tvk}, \underline{\mathtt{vk}}) \leftarrow \mathcal{O}_{\mathtt{key}}(n, r)$ to a query made by $\mathcal{A}_1$ such that $\mathtt{tvk} = \mathtt{tvk}_{j^*}$, and parses $(\mathtt{vk}_1, \ldots, \mathtt{vk}_n) := \underline{\mathtt{vk}}$. This takes negligible time, and, conditioned on the success of the previous step, this step certainly succeeds.

10. If $\mathtt{tvk}_{j^*} \in \mathcal{L}^{\mathtt{tot}}_{\mathtt{corrupt}}$, output a distinct failure symbol and terminate. Otherwise, for this $\underline{\mathtt{vk}}$, let $C = \underline{\mathtt{vk}} \cap \mathcal{L}_{\mathtt{corrupt}}$ be the set of corrupted public verification key shares corresponding to the total verification key ring member $\mathtt{tvk}_j$. This step takes negligible time. Moreover, conditioned on the success of the previous step, this step succeeds with certainty. Indeed, $\mathcal{A}_1$ would have failed (causing a cascade of failures) if too many key shares had been corrupted.

11. Otherwise, $|C| < r - 1$. In this event, $\mathcal{A}_4$ selects $r - 1 - |C|$ uncorrupted keys associated with $\mathtt{tvk}_{j^*}$ to parse, say $(Y_i, Z_1, \ldots, Z_{d-1}) := \mathtt{vk}_i$, and corrupts them by querying $y_i \leftarrow \mathcal{O}^*_{\mathtt{corrupt}}(Y_i)$, until $|C| = r - 1$ exactly. This step takes negligible time and certainly succeeds.

12. Compute the indices $I \subseteq [n]$ of keys in $\underline{\mathtt{vk}}$ corresponding to the elements in C, and compute the Lagrange interpolation coefficients λ_i for $I \in [n]$.

13. $\mathcal{A}_4$ picks an uncorrupted challenge key of $\underline{\mathtt{vk}}$, say $\mathtt{vk}_{i^*}$, and parses $(y_{i^*}, z_1, \ldots, z_{d-1}) := \mathtt{vk}_{i^*}$. This is the target challenge key. This step takes negligible time and succeeds with certainty.

14. $\mathcal{A}_4$ computes the Lagrange interpolation coefficients corresponding to the target challenge key $\mathtt{vk}_{i^*}$ and the corrupted keys $(Y_i, Z_1, \ldots, Z_{d-1})$ for each $i \in I$. Call these λ_{i^*} for $\mathtt{vk}_{i^*}$, and λ_i for each $\mathtt{vk}_i$ with $i \in I$. This takes $r(r-1)$ multiplications and certainly succeeds. Moreover, the Lagrange interpolation coefficients are all nonzero with certainty.

15. $\mathcal{A}_4$ computes the following.

$$\begin{aligned} z^* &= \sum_{k \in [d-1]} \mu_k z_k \\ \overline{y} &= \sum_{i \in I, i \neq i^*} \lambda_i y_i \\ y_{j^*,i^*} &= \lambda_{i^*}^{-1} \left(\mu_y^{-1} \left(\widetilde{w}_{j^*} - z^* \right) - \overline{y} \right) \end{aligned}$$

    Then $\mathcal{A}_4$ sets $\mathtt{out}_4 = \mathcal{L}_{\mathtt{corrupt}} \cup \{y_{j^*,i^*}\}$. Computing z^* takes time $(d-1)t_{\mathtt{mul}} + (d-1)t_+$, computing $\overline{y}$ takes time $(r-1)t_{\mathtt{mul}} + (r-1)t_+$, and computing y_{j^*,i^*} from these takes time $2t_{\mathtt{mul}} + 2t_+$. These all certainly succeed. Thus, this step takes $(r+d)(t_{\mathtt{mul}} + t_+)$ time.

Consider the correctness of the composition of all these algorithms $\mathcal{A}_4$ through $\mathcal{A}_1$ as a κ-OMDL player. Given the auxiliary keys $z_1, \ldots, z_{d-1}$ and the aggregation coefficients, the aggregated key w yields the linking key $y = \mu_Y^{-1}(w - \sum_k \mu_k z_k)$, and given the $r-1$ corrupted secret signing key shares y_i and the value y, Lagrange interpolation implies $y = \lambda_{i^*} y_{j^*,i^*} + \sum_{i \in I, i \neq i^*} \lambda_i y_i$. Moreover, since this is a successful forgery, all ring members are challenge keys. Solving these for y_{j^*,i^*} provides correctness.

Now consider the runtime. $\mathcal{A}_4$ takes time

$$t_4 \approx t_3 + (r+2)t_+ + 3t_{\mathtt{inv}} + 2t_{\mathtt{scmul}} + O(\kappa_{\mathtt{ch}}) + 2t_{\mathtt{neg}} + r^2 t_{\mathtt{mul}} + (r - 1 - |C|)t_{\mathtt{corrupt}}$$

obtained by summing the times of each step above. However, r is selected by $\mathcal{A}_4$ in the course of executing $\mathcal{A}_1$. Moreover, the strongest adversary corrupts no keys at all. So, we use $r \leq n \leq n_{\mathtt{ch}}$ and $r - 1 - |C| \leq n_{\mathtt{ch}} - 1$ to obtain

$$t_4 \approx t_3 + (n_{\mathtt{ch}} + 2)t_+ + 3t_{\mathtt{inv}} + 2t_{\mathtt{scmul}} + O(\kappa_{\mathtt{ch}}) + 2t_{\mathtt{neg}} + n_{\mathtt{ch}}^2 t_{\mathtt{mul}} + (n_{\mathtt{ch}} - 1)t_{\mathtt{corrupt}}.$$

However, $t_3 \approx 2t_2 \approx 2(t_1 + n_{\mathtt{key}}\kappa_{\mathtt{key}} t_{\mathtt{key}} + \kappa_{\mathtt{Sign}} t_{\mathtt{Sign}} + \kappa_{\mathtt{corrupt}} t_{\mathtt{corrupt}} + \kappa_{\mathtt{base}} t_{\mathtt{base}})$, so $t_4 \approx 2t_1 + \widetilde{t}$ where

$$\begin{aligned} \widetilde{t} =& 2n_{\mathtt{key}}\kappa_{\mathtt{key}} t_{\mathtt{key}} + 2\kappa_{\mathtt{Sign}} t_{\mathtt{Sign}} + 2\kappa_{\mathtt{corrupt}} t_{\mathtt{corrupt}} + 2\kappa_{\mathtt{base}} t_{\mathtt{base}} + \\ & (n_{\mathtt{ch}} + 2)t_+ + 3t_{\mathtt{inv}} + 2t_{\mathtt{scmul}} + O(\kappa_{\mathtt{ch}}) + 2t_{\mathtt{neg}} + n_{\mathtt{ch}}^2 t_{\mathtt{mul}} + (n_{\mathtt{ch}} - 1)t_{\mathtt{corrupt}} \end{aligned}$$

Now consider success probability. If $\mathcal{A}_3$ succeeds, $\mathcal{A}_4$ can fail if the $\mathtt{query}$ made to $H_{\mathtt{ch}}$ such that $T_{\mathtt{ch}}[\mathtt{query}] = (\ell, c)$ cannot be parsed as $(\mathtt{dst}_j \mid\mid \mathtt{ring} \mid\mid \mathtt{tlt} \mid\mid \underline{\widehat{Y}} \mid\mid \mathfrak{W} \mid\mid \underline{W} \mid\mid \underline{\mu} \mid\mid L_j \mid\mid R_j \mid\mid \mathtt{msg})$. We denoted above the probability that $\mathtt{query}$ cannot be parsed appropriately with ϵ'.

All that remains is to show ϵ' is negligible. Indeed, the $\mathtt{query}$ satisfies $T_{\mathtt{ch}}[\mathtt{query}] = (\ell, c)$ where $c = c_{j+1}$ is used in verification. If $\mathtt{query}$ cannot be parsed as $(\mathtt{dst}_j \mid\mid \mathtt{ring} \mid\mid \mathtt{tlt} \mid\mid \underline{\widehat{Y}} \mid\mid \mathfrak{W} \mid\mid \underline{W} \mid\mid \underline{\mu} \mid\mid L_j \mid\mid R_j \mid\mid \mathtt{msg})$, but c_{j+1} is used in verification, then there is some other $\mathtt{query}' \in T_{\mathtt{ch}}$ with this c. This is a second pre-image for $\mathtt{query}$ for the random oracle $H_{\mathtt{ch}}$, so this $\epsilon' \in \mathrm{negl}(\lambda)$.

□

4.3 Linkability

Theorem 4.5. FROSTLASS is linkable under Definition 2.9.

Proof. Let $\kappa \in \mathbb{N}$ and let $\mathcal{A}$ be a κ-linkability breaker.

We build a tower of algorithms, $\mathcal{A}_3 \to \mathcal{A}_2 \to \mathcal{A}_1$, where $\mathcal{A}_1 = \mathcal{A}$, similar to those in theorem 4.4, to extract the discrete logarithm of an aggregated ring member which is under adversarial control. We use this discrete logarithm to show the linkability tag is, except with negligible probability, the image of the linking key under a collision-resistant function.

First, recall $\mathcal{A}_2$ in theorem 4.4 was a five-step algorithm. We let $\mathcal{A}_2$ operate similarly to $\mathcal{A}_2$ from theorem 4.4, modifying the following steps:

4. Otherwise, $\texttt{out}_1 \leftarrow \mathcal{A}_1$. In this event, $\mathcal{A}_2$ parses the first message-ring-signature triple present, $(\texttt{msg}, \texttt{ring}, \texttt{sig}) \leftarrow \texttt{out}_1$, sets $m = |\texttt{ring}|$, and does the following.
   (a) Find all queries $\texttt{query} \in T_{\texttt{ch}}$, $\ell \in [\kappa_{\texttt{ch}}]$, $j \in [m]$, $c \in \mathbb{Z}_q$ such that $T_{\texttt{ch}}[\texttt{query}] = (\ell, c)$, $\texttt{tvk}_j \in \texttt{ring}$ is not an uncorrupted challenge key, and $c = c_{j+1}$ is used to verify $\texttt{sig}$; call this set of queries S.

All other steps are otherwise similar to theorem 4.4. That we fork only on queries related to uncorrupted challenge keys is critically important here, just as forking only on challenge keys was critical in theorem 4.4. As before, $\mathcal{A}_2$ is compatible with the forking algorithm, so let $\mathcal{A}_3 = \texttt{Fork}_{\mathcal{A}_2}$ as in theorem 4.4. Now consider a successful output of $\mathcal{A}_3$.

The forking query $(\texttt{dst}_{j^*} \mid\mid \texttt{ring} \mid\mid \texttt{tlt} \mid\mid \underline{\widehat{Y}} \mid\mid \mathfrak{W} \mid\mid \underline{W} \mid\mid \underline{\mu} \mid\mid L_{j^*} \mid\mid R_{j^*} \mid\mid \texttt{msg})$ is the same on both sides of the fork. Moreover, in one transcript we obtain $L_{j^*} = s_{j^*}G + c_{j^*}W_{j^*}$ where W_{j^*} is the aggregation of the ring member with index j^*, and in the other transcript the same L_{j^*} satisfies $L_{j^*} = s'_{j^*}G + c'_{j^*}W_{j^*}$. Thus, just as before, we obtain the discrete logarithm $W_{j^*} = \frac{s_{j^*} - s'_{j^*}}{c'_{j^*} - c_{j^*}}G$. However, this $W_{j^*} = \mu_Y Y_{j^*} + \sum_k \mu_k Z_{j^*,k}$. Thus, if y_{j^*} is the discrete logarithm of Y_{j^*} with respect to G and each $z_{j^*,k}$ is the discrete logarithm of $Z_{j^*,k}$ with respect to G, then $\frac{s_{j^*} - s'_{j^*}}{c'_{j^*} - c_{j^*}} = \mu_Y y_{j^*} + \sum_k \mu_k z_{j^*,k}$. Note this equation holds for these scalars, even though we do not (and perhaps even $\mathcal{A}$ may not) know them.

Also, we obtain in one transcript $R_{j^*} = s_{j^*}\widehat{Y}_{j^*} + c_{j^*}\mathfrak{W}$ where $\mathfrak{W}$ is the aggregation of the linking tag base points $\mathfrak{T}$ and $\mathfrak{D}_k$. In the other transcript, the same R_{j^*} satisfies $R_{j^*} = s'_{j^*}\widehat{Y}_{j^*} + c'_{j^*}\mathfrak{W}$. Thus, just as before, we obtain the discrete logarithm $\mathfrak{W} = \frac{s_{j^*} - s'_{j^*}}{c'_{j^*} - c_{j^*}}\widehat{Y}_{j^*}$. However, $\mathfrak{W} = \mu_Y\mathfrak{T} + \sum_k \mu_k \mathfrak{D}_k$, so $\frac{s_{j^*} - s'_{j^*}}{c'_{j^*} - c_{j^*}}\widehat{Y}_{j^*} = \mu_Y\mathfrak{T} + \sum_k \mu_k \mathfrak{D}_k$. Re-arranging, we have $\mu_Y\left(y_{j^*}\widehat{Y}_{j^*} - \mathfrak{T}\right) + \sum_k \mu_k \left(z_{j^*,k}\widehat{Y}_{j^*} - \mathfrak{D}_k\right) = 0$, which can only be satisfied if $\mathfrak{T} = \widehat{Y}_{j^*}$, except with negligible probability. Since $\mathfrak{T}$ is a collision-resistant function of y_{j^*}, we conclude that valid signatures, except with negligible probability, have linking tags which are collision-resistant functions of some ring member which is not an uncorrupted challenge key. This prevents more linking tags than linking keys. □

References

[1] Mihir Bellare, Elizabeth C Crites, Chelsea Komlo, Mary Maller, Stefano Tessaro, and Chenzhi Zhu, *Better than advertised security for non-interactive threshold signatures*, Advances in Cryptology-CRYPTO 2022 **4** (2022).

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

[3] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell, *Bulletproofs: Short proofs for confidential transactions and more*, 2018 IEEE symposium on security and privacy (SP), IEEE, 2018, pp. 315–334.

[4] Brandon Goodell and Sarang Noether, *Thring signatures and their applications to spender-ambiguous digital currencies*, 2018.

[5] Brandon Goodell, Sarang Noether, and Arthur Blue, *Concise linkable ring signatures and forgery against adversarial keys*, 2019.

[6] Luke Parker (KayabaNerve), *serai-dex/serai/*, 2024, GitHub repository.

[7] Chelsea Komlo and Ian Goldberg, *Frost: flexible round-optimized schnorr threshold signatures*, Selected Areas in Cryptography: 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers 27, Springer, 2021, pp. 34–65.

[8] X Li and Miroslaw Malek, *Analysis of speedup and communication/computation ratio in multiprocessor systems*, Proceedings. Real-Time Systems Symposium, IEEE Computer Society, 1988, pp. 282–283.

[9] Joseph K Liu, Victor K Wei, and Duncan S Wong, *Linkable spontaneous anonymous group signature for ad hoc groups*, Information Security and Privacy: 9th Australasian Conference, ACISP 2004, Sydney, Australia, July 13-15, 2004. Proceedings 9, Springer, 2004, pp. 325–335.

[10] Adi Shamir, *How to share a secret*, Communications of the ACM **22** (1979), no. 11, 612–613.

[11] Victor Shoup, *Practical threshold signatures*, Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19, Springer, 2000, pp. 207–220.

[12] Fangguo Zhang, Shengli Liu, and Kwangjo Kim, *Id-based one round authenticated tripartite key agreement protocol with pairings*, Cryptology ePrint Archive (2002).