Re-Randomized FROST

Conrado P. L. Gouvêa, Chelsea Komlo*

Zcash Foundation

Abstract. We define a (small) augmentation to the FROST threshold signature scheme that additionally allows for re-randomizable public and secret keys. We build upon the notion of re-randomizable keys in the literature, but tailor this capability when the signing key is secret-shared among a set of mutually trusted parties. We do not make any changes to the plain FROST protocol, but instead define additional algorithms to allow for randomization of the threshold public key and participant's individual public and secret key shares.
We show the security of this re-randomized extension to FROST with respect to the algebraic one-more discrete logarithm (AOMDL) problem in the random oracle model, the same security assumptions underlying plain FROST.

1 Introduction

Threshold signatures are a class of multi-party signature schemes, where n possible signers hold shares of a secret key, such that $t < n$ signers are required to jointly produce a signature over a message. A threshold signature scheme assumes that up to $t-1$ number of parties can be adversarial while still preserving unforgeability.

Re-Randomized Signatures. In scenarios where privacy is important, regular signatures provide a challenge since different signatures can be trivially linked to a specific party if their public key is known (by simply verifiying the signatures). To address this issue, re-randomization techniques have been applied to single-party signature schemes [5]. These allow the randomization of the key pair in such way that different signatures can't be linked to the same signer, while still allowing proving the authorship of a signature with the help of zero-knowledge proofs.

Re-Randomized FROST. The goal of this work is to define a re-randomized threshold signature scheme that outputs Schnorr signatures. Our choice of threshold signature is the FROST threshold signature scheme [10, 3, 1]. We build upon the definition and notions of security of re-randomized signature schemes by Fleischhacker et al. [5], as well as the re-randomized Schnorr signature scheme presented in the same work.

Re-Randomized FROST applications. One use case for re-randomized FROST are privacy-focused cryptocurrencies like Zcash [9] and Penumbra [12]. These cryptocurrencies require re-randomized signatures in order to preserve privacy of transactions published on the public blockchain. Enabling the usage of FROST to the setting where signatures are unlinkable allows it to be used to manage wallets for such privacy-preserving cryptocurrencies. This has multiple benefits: by splitting a spending key into shares distributed to multiple parties, the risk of losing funds by either key loss (i.e. losing the device with the key) or by an attacker compromising the key is greatly mitigated.

As an example, for each transaction in Zcash [9], the spending key is re-randomized and the re-randomized public key is included with the transaction along with a signature of the transaction contents generated with the re-randomized key. A zero-knowledge proof is then included that proves that the re-randomized public key is a re-randomization of the user's spending public key with respect to some randomizer, while not disclosing neither the randomizer nor the spending public key. The link between the spending public key and the user's address is also included in the proof, again without revealing the spending public key, binding everything together. Re-randomized FROST could then be used to generate the transaction signature, without requiring any change in the Zcash protocol.

* This work was not part of my University of Waterloo duties.

Our Contributions. In this work, we present the following.

- We define the notion of a re-randomizable threshold signature scheme, building on the notion of a single-party signature scheme by Fleischhacker et al. [5].
- We give definitions of correctness and security of a threshold signature scheme that can be re-randomized.
- We present a (small) natural extension to the FROST threshold signature scheme, allowing it to be securely employed in a re-randomizable setting. We refer to this re-randomizable extension as Rerandomized-FROST.
- We prove Rerandomized-FROST secure using our new notions of security.

2 Preliminaries

2.1 General Notation

Let $\lambda \in \mathbb{N}$ denote the security parameter and 1^λ its unary representation. A function $f : \mathbb{N} \to \mathbb{R}$ is called *negligible* if for all $c \in \mathbb{R}, c > 0$, there exists $k_0 \in \mathbb{N}$ such that $|f(k)| < \frac{1}{k^c}$ for all $k \in \mathbb{N}, k \geq k_0$. For a non-empty set S, let $x \xleftarrow{\$} S$ denote sampling an element of S uniformly at random and assigning it to x. We use $[n]$ to represent the set $\{1, \ldots, n\}$ and $[0..n]$ to represent the set $\{0, \ldots, n\}$. We represent vectors as $\vec{a} = (a_1, \ldots, a_n)$.

Let PPT denote probabilistic polynomial time. Algorithms are randomized unless explicitly noted otherwise. Let $y \leftarrow A(x; \omega)$ denote running algorithm A on input x and randomness ω and assigning its output to y. Let $y \xleftarrow{\$} A(x)$ denote $y \leftarrow A(x; \omega)$ for a uniformly random ω. The set of values that have non-zero probability of being output by A on input x is denoted by $[A(x)]$.

Group Generation. Let GrGen be a polynomial-time algorithm that takes as input a security parameter 1^λ and outputs a group description $(\mathbb{G}, p, g)$ consisting of a group $\mathbb{G}$ of order p, where p is a λ-bit prime, and a generator g of $\mathbb{G}$.

Polynomial Interpolation. A polynomial $f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{t-1} x^{t-1}$ of degree $t-1$ over a field $\mathbb{F}$ can be interpolated by t points. Let $\eta \subseteq [n]$ be the list of t distinct indices corresponding to the x-coordinates $x_i \in \mathbb{F}, i \in \eta$ of these points. Then the Lagrange polynomial $L_i(x)$ has the form:

$$L_i(x) = \prod_{j \in \eta; j \neq i} \frac{x - x_j}{x_i - x_j} \tag{1}$$

Given a set of t points $(x_i, f(x_i))_{i \in [t]}$, any point $f(x_\ell)$ on the polynomial f can be determined by Lagrange interpolation as follows:

$$f(x_\ell) = \sum_{k \in \eta} f(x_k) \cdot L_k(x_\ell)$$

2.2 Definitions and Assumptions

Assumption 1 (Algebraic One-More Discrete Logarithm Assumption) [11] *Let the advantage of an adversary $\mathcal{A}$ playing the ℓ-algebraic one-more discrete logarithm game $\mathsf{Game}^{\ell\text{-aomdl}}$, as defined in Figure 1, be as follows:*

$$\mathsf{Adv}^{\ell\text{-}aomdl}_{\mathcal{A}}(\lambda) = \left|\Pr[\mathsf{Game}^{\ell\text{-aomdl}}_{\mathcal{A}}(\lambda) = 1]\right|$$

The algebraic one-more discrete logarithm assumption holds if for all PPT adversaries $\mathcal{A}$, $\mathsf{Adv}^{\ell\text{-}aomdl}_{\mathcal{A}}(\lambda)$ is negligible.

Definition 1 (Shamir secret sharing [13]). *Shamir secret sharing is an (n, t)-threshold secret sharing scheme $\mathcal{SS} = (\mathsf{IssueShares}, \mathsf{Recover})$ that consists of the following algorithms:*

MAIN $\mathsf{Game}_{\mathcal{A}}^{\ell\text{-aomdl}}(\lambda)$	$\mathcal{O}^{\mathsf{dl}}(X, \alpha, \{\beta_i\}_{i=1}^{\ell})$
$(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$	$x \leftarrow \alpha + \sum_{i=0}^{\ell} x_i \beta_i$
$Q \leftarrow \emptyset$	**return** x
$q \leftarrow 0$	
for $i \in [0..\ell]$ **do**	
$x_i \xleftarrow{\$} \mathbb{Z}_p$; $X_i \leftarrow g^{x_i}$	
$Q[X_i] = x_i$	
$\vec{x} \leftarrow (x_0, \ldots, x_\ell)$	
$\vec{X} \leftarrow (X_0, X_1, \ldots, X_\ell)$	
$\vec{x'} \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{dl}}}((\mathbb{G}, p, g), \vec{X})$	
if $\vec{x'} = \vec{x} \ \wedge \ q < \ell + 1$	
return 1	
return 0	

Fig. 1. The Algebraic One-More Discrete Logarithm (AOMDL) games. $\mathbb{G}$ is a cyclic group with prime order p and generator g. $\vec{a}$ is a vector whose elements are in $\mathbb{Z}_p$. In the AOMDL game, the adversary can query for linear combinations of elements which the environment has generated directly; in the OMDL game, the adversary can query its DL oracle with an arbitrary group element.

– $\mathsf{IssueShares}(x, n, t) \to \{(1, x_1), \ldots, (n, x_n)\}$*: On input a secret* x*, the number of participants* n*, and a threshold* t*, perform the following. First, define a polynomial* $f(Z) = x + a_1 + a_2 Z^2 + \cdots + a_{t-1} Z^{t-1}$ *by sampling* t *coefficients at random* $(a_1, \ldots, a_{t-1}) \xleftarrow{\$} \mathbb{Z}_p$*. Then, set each participant's share* $x_i, i \in [n]$*, to be the evaluation of* $f(i)$*:*

$$x_i \leftarrow x + \sum_{j \in [t-1]} a_j i^j$$

Output $\{(i, x_i)\}_{i \in [n]}$*.*

– $\mathsf{Recover}(t, \{(i, x_i)\}_{i \in \mathcal{S}}) \to \perp / x$*: On input a threshold* t *and a set of shares* $\{(i, x_i)\}_{i \in \mathcal{S}}$*, output* $\perp$ *if* $\mathcal{S} \not\subseteq [n]$ *or if* $|\mathcal{S}| < t$*. Otherwise, recover* x *as follows:*

$$x \leftarrow \sum_{i \in \mathcal{S}} \lambda_i x_i$$

where the Lagrange coefficient for the set $\mathcal{S}$ *is defined by*

$$\lambda_i = \prod_{j \in \mathcal{S}, j \neq i} \frac{-j}{i - j}$$

2.3 Threshold Signature Schemes

A threshold signature scheme TS whose signing protocol consists of two rounds is a tuple $\mathsf{TS} = (\mathsf{Setup}, \mathsf{KeyGen}, (\mathsf{Sign}, \mathsf{Sign}'), \mathsf{Combine}, \mathsf{Verify})$, defined as follows. This definition assumes a centralized key generation mechanism, but can adapted to be fully decentralized via a distributed key generation protocol (DKG).

– $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$: This algorithm generates the public parameters par that are implicitly given as input to all other algorithms.

– $\mathsf{KeyGen}(n, t) \to (\mathcal{PK}, \{\mathcal{PK}_i\}_{i \in [n]}, \{x_i\}_{i \in [n]})$: A randomized protocol that takes as input the total number of signing parties n and the threshold t. The output is the public key $\mathcal{PK}$ representing the set of n parties, n public key shares $\{\mathcal{PK}_i\}_{i \in [n]}$for each party, and the set of secret key shares $\{x_i\}_{i \in [n]}$.

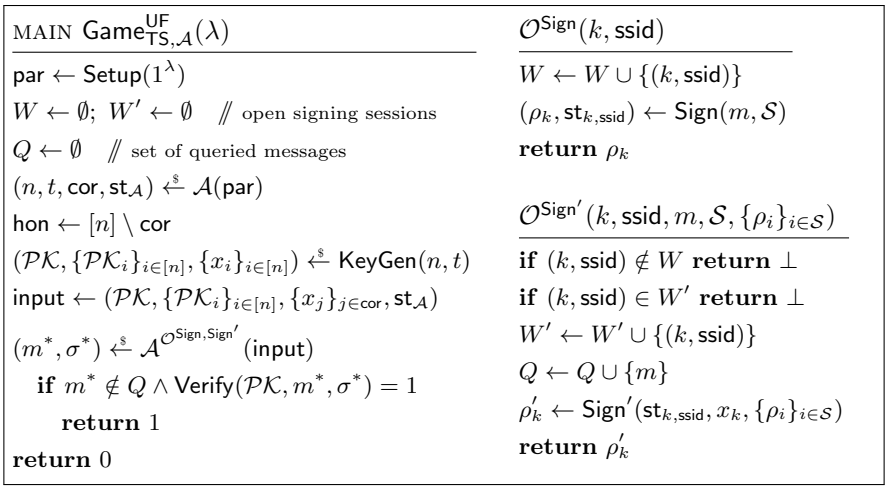

Fig. 2. Static unforgeability games for a threshold signature scheme with two signing rounds. The public parameters par are implicitly given as input to all algorithms, and ρ represents protocol messages defined within the construction.

- $(\mathsf{Sign}, \mathsf{Sign}') \to \rho_i$: A set of randomized algorithms where each Sign algorithm is a single stage in an interactive signing protocol, performed by each signing party in a signing set $\mathcal{S} \subseteq [n], |\mathcal{S}| \geq t$ with respect to a message m. The output from each signing algorithm is a protocol message ρ_i.
- $\mathsf{Combine}(\{(\rho_i, \rho'_i)\}_{i\in\mathcal{S}}) \to (m, \sigma)$: A deterministic algorithm that takes as input the set of protocol messages (ρ_i, ρ'_i) representing the messages sent in Sign and Sign'. It outputs σ as the signature representing the signing set.
- $\mathsf{Verify}(\mathcal{PK}, m, \sigma) \to 0/1$: A deterministic algorithm that takes as input the public key $\mathcal{PK}$, a message m, and signature σ. It outputs 0 or 1 indicating if σ is valid.

2.4 Unforgeability

We employ prior game-based unforgeability definitions for threshold signatures in the literature [4, 7, 8, 6], and show this game in Figure 2.

In the unforgeability game, the adversary begins by choosing the corrupted participants cor. The challenger performs KeyGen to generate the joint public key $\mathcal{PK}$, the individual public key shares $\{\mathcal{PK}_i\}_{i\in[n]}$, and the signing shares $\{x_i\}_{i\in[n]}$. It returns $\mathcal{PK}$, $\{\mathcal{PK}_i\}_{i\in[n]}$, and the set of corrupt signing shares $\{x_j\}_{j\in\mathsf{cor}}$ to the adversary.

After key generation, the adversary may query honest signers at each step in the signing protocol via the oracles $\mathcal{O}^{\mathsf{Sign}}, \mathcal{O}^{\mathsf{Sign}'}$. The adversary is free to choose the set of signers and the message.

The adversary wins if it produces a valid forgery $\sigma^* = (R^*, z^*)$ with respect to the public key $\mathcal{PK}$, on a message m^* that has not been previously queried to the signing oracles.

Definition 2 (Unforgeability). *Let the advantage of an adversary $\mathcal{A}$ playing the (static) unforgeability game $\mathsf{Game}^{\mathsf{UF}}_{\mathcal{A}}(\lambda)$, as defined in Figure 2, be as follows:*

$$\mathit{Adv}^{sec}_{\mathcal{A},\mathsf{TS}}(\lambda) = \left|\Pr[\mathsf{Game}^{\mathsf{UF}}_{\mathcal{A},\mathsf{TS}}(\lambda) = 1]\right|$$

A threshold signature scheme TS is unforgeable *if for all PPT adversaries $\mathcal{A}$, $\mathit{Adv}^{sec}_{\mathcal{A}}(\lambda)$ is negligible.*

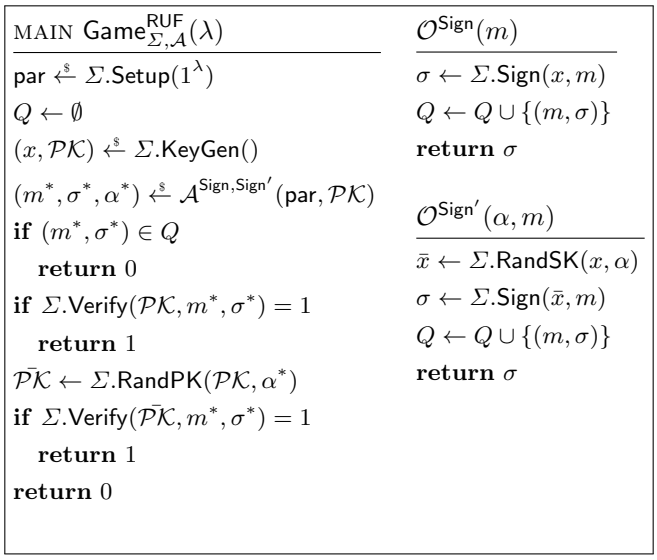

Fig. 3. The Existential Unforgeability Game for a Rerandomized Signature scheme Σ.

3 Rerandomized Signatures

Let Σ be a standard signature scheme. We now define a signature scheme $\bar{\Sigma}$ with perfectly rerandomized keys. We build on the definition presented by Fleischhacker et al. [5], but additionally include an algorithm to generate the randomizer, to be more explicit.

Definition 3. *A signature scheme $\Sigma = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Verify})$ is a re-randomized signature scheme $\bar{\Sigma}$ if there exists additional PPT algorithms $(\mathsf{GenRand}, \mathsf{RandSK}, \mathsf{RandPK})$ and a randomness distribution $\mathcal{X}$ such that:*

- $\mathsf{GenRand}()$*: Outputs a randomizer α from a distribution $\mathcal{X}$.*
- $\mathsf{RandSK}(x, \alpha)$*: Accepts as input a secret signing key x and a randomizer $\alpha \in \mathcal{X}$. Outputs a randomized secret key $\bar{x}$.*
- $\mathsf{RandPK}(\mathcal{PK}, \alpha)$*: accepts as input a public verification key $\mathcal{PK}$ and a randomizer $\alpha \in \mathcal{X}$. Outputs a randomized public key $\bar{\mathcal{PK}}$.*

Existential Unforgeability. The existential unforgeability of a re-randomized signature scheme requires the scheme to be unforgeable under keys that are not randomized, as well as keys which have been randomized. The existential unforgeability game introduces one additional capability for an adversary over the standard unforgeability game. Here, the adversary is additionally allowed to query for signatures under keys that have been randomized using a randomizer which it is allowed to choose.

We show the game in Figure 3.

Definition 4 (Unforgeability). *Let the advantage of an adversary $\mathcal{A}$ playing the unforgeability game $\mathsf{Game}^{\mathsf{RUF}}_{\mathcal{A},\Sigma}(\lambda)$, as defined in Figure 3, be as follows:*

$$\mathit{Adv}^{rsec}_{\mathcal{A},\Sigma}(\lambda) = \left|\Pr[\mathsf{Game}^{\mathsf{RUF}}_{\mathcal{A},\Sigma}(\lambda) = 1]\right|$$

A re-randoizable signature scheme Σ is unforgeable *if for all PPT adversaries $\mathcal{A}$, $\mathit{Adv}^{rsec}_{\mathcal{A},\Sigma}(\lambda)$ is negligible.*

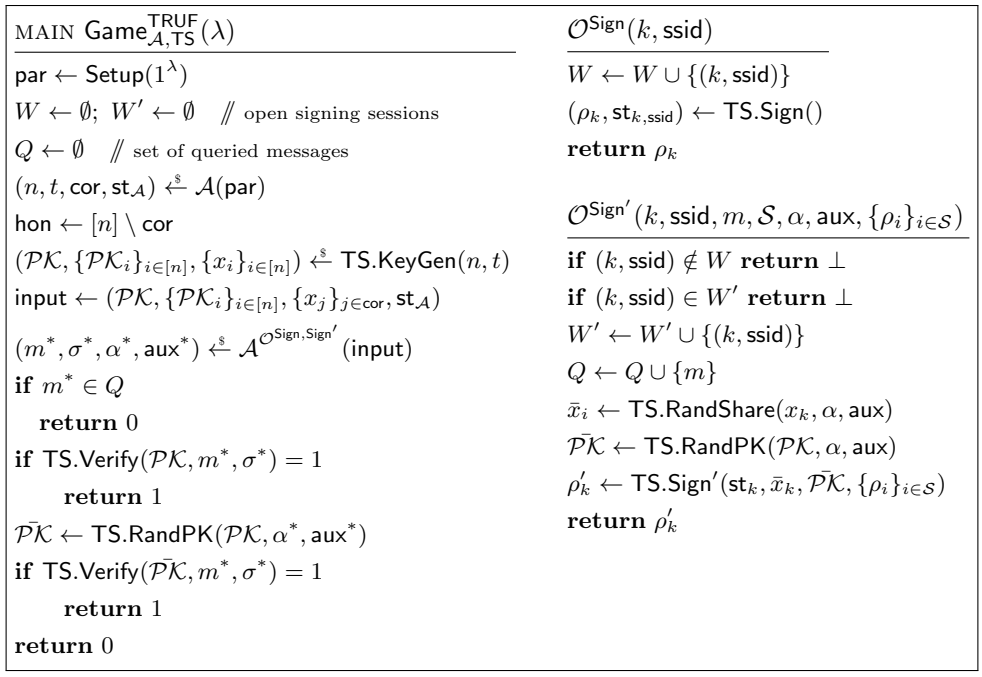

Fig. 4. The (static) unforgeability games for a randomizable threshold signature scheme with two signing rounds.

Re-randomized Schnorr. Fleischhacker et al. [5] proved that re-randomized single-party Schnorr signatures are existentially unforgeable.

4 Randomized Threshold Signature Schemes

We now extend the definition of a threshold signature scheme TS as given in Section 2.3 to one that is perfectly rerandomizable.

Definition 5. *A threshold signature scheme* $\mathsf{TS} = (\mathsf{Setup}, \mathsf{KeyGen}, (\mathsf{Sign}, \mathsf{Sign}'), \mathsf{Combine}, \mathsf{Verify})$ *is perfectly re-randomizable if there exists additional PPT algorithms* $(\mathsf{GenRand}, \mathsf{RandShare}, \mathsf{RandPKShare}, \mathsf{RandPK})$ *and a randomness distribution* $\mathcal{X}$ *such that:*

- $\mathsf{GenRand}()$*: Outputs a randomizer* α *from a distribution* $\mathcal{X}$*.*
- $\mathsf{RandShare}(x_i, \alpha, \mathsf{aux}) \to \bar{x}_i$*: Accepts as input a secret signing key share* x_i*, a randomizer* $\alpha \in \mathcal{X}$*, and an auxiliary string* $\mathsf{aux} \in \{0,1\}^*$*. Outputs a randomized secret key* $\bar{x}_i$*.*
- $\mathsf{RandPKShare}(\mathcal{PK}_i, \alpha, \mathsf{aux})$*: accepts as input a threshold public verification key share* $\mathcal{PK}_i$ *for a participant* i*, a randomizer* $\alpha \in \mathcal{X}$*, and an auxiliary string* $\mathsf{aux} \in \{0,1\}^*$*. Outputs a randomized public key* $\bar{\mathcal{PK}}_i$*.*
- $\mathsf{RandPK}(\mathcal{PK}, \alpha, \mathsf{aux})$*: accepts as input a threshold public verification key* $\mathcal{PK}$*, a randomizer* $\alpha \in \mathcal{X}$*, and an auxiliary string* $\mathsf{aux} \in \{0,1\}^*$*. Outputs a randomized public key* $\bar{\mathcal{PK}}$*.*

Remark 1 (Including Auxiliary Information in Re-Randomization). Sometimes, it is useful to contribute to the re-randomization of a key using the transcript of some external protocol. In doing so, when the external protocol changes (i.e, by starting a new session), then the re-randomized key will also change. We model this optional contributory factor as the auxiliary string aux.

Unforgeability. The unforgeability of a re-randomizable threshold signature scheme builds upon the notion of unforgeability for the single-party setting. In short, it requires the scheme to be unforgeable under keys

that are not randomized, as well as keys which have been randomized. This notion is reflected in the game, as the adversary can simply submit a randomizer that is equal to zero. The existential unforgeability game introduces the additional capability for an adversary over the standard unforgeability game, where signatures are allowed to be generated under a adversarially-chosen randomizer.

We show the attack game for the unforgeability of a randomized threshold signature scheme in Figure 4.

Definition 6 (Unforgeability). *Let the advantage of an adversary $\mathcal{A}$ playing the re-randomizable unforgeability game $\mathsf{Game}^{\mathsf{TRUF}}_{\mathcal{A},\mathsf{TS}}(\lambda)$ against a re-randomizable threshold signature, as defined in Figure 4, be as follows:*

$$\mathit{Adv}^{rsec}_{\mathcal{A},\mathsf{TS}}(\lambda) = \left|\Pr[\mathsf{Game}^{\mathsf{TRUF}}_{\mathcal{A},\mathsf{TS}}(\lambda) = 1]\right|$$

A re-randomizable threshold signature scheme TS *is* unforgeability *if for all PPT adversaries $\mathcal{A}$, $\mathit{Adv}^{rsec}_{\mathcal{A},\mathsf{TS}}(\lambda)$ is negligible.*

5 Rerandomized FROST

We now introduce an extension to FROST [10, 1], a two-round threshold signature scheme, to allow for rerandomizable keys. We refer to this re-randomized variant as Rerandomized-FROST. We highlight the changes to plain FROST in grey in Figure 4.

The Protocol. We show the exact details of Rerandomized-FROST in Figure 4. In short, the protocol extends plain FROST by deriving additional algorithms to generate a randomizer and randomize the group public key and each signer's private key share.

Rerandomized-FROST is identical to FROST but additionally defines the algorithms GenRand, RandShare, RandPKShare, and RandPK. Similarly to FROST, Rerandomized-FROST assumes an external mechanism to choose the set $\mathcal{S} \subseteq \{1, \ldots, n\}$ of signers, where $t \leq |\mathcal{S}| \leq n$. $\mathcal{S}$ must be required to be ordered to ensure consistency.

The algorithm GenRand simply selects a randomizer α uniformly at random from $\mathbb{Z}_p$. RandShare accepts as input a secret key share x_i, a randomizer α, and an auxiliary string aux. It first hashes aux and α via H_r, and then adds the output to the secret key share, deriving the randomized key share $\bar{x}_i$. It outputs $\bar{x}_i$ as its result.

The algorithm RandPK accepts public key $\mathcal{PK}$, the randomizer α, and auxiliary string aux, It likewise hashes aux and α via H_r, deriving the value $\hat{\alpha}$. The randomized public key that is output is then $\bar{\mathcal{PK}} = \mathcal{PK} \cdot g^{\hat{\alpha}}$.

The algorithm RandPKShare accepts a public key share $\mathcal{PK}_i$ for participant i, a randomizer α, and an auxiliary string aux. It follows the same steps as for RandPK, by hashing aux and α via H_r, and then deriving the value $\hat{\alpha}$. The randomized public key share that is output is then $\bar{\mathcal{PK}}_i = \mathcal{PK}_i \cdot g^{\hat{\alpha}}$.

Correctness. Rerandomized-FROST is correct, because the signature it produces is equivalent to a regular rerandomized signature for the rerandomized public key $\bar{\mathcal{PK}} = \mathcal{PK} \cdot g^{\hat{\alpha}}$ where $\hat{\alpha} = \mathsf{H}_\mathsf{r}(\alpha, \mathsf{aux})$, as shown below. Note that $\sum_{i \in \mathcal{S}} x_i \lambda_i = x$ (where x is the original secret key) because x is Shamir secret shared among all parties; also note that $\sum_{i \in \mathcal{S}} \lambda_i = 1$ since it is the interpolation of the constant polynomial $f(X) = 1$. Thus,

$$\begin{aligned} z &= \sum_{i \in \mathcal{S}} r_i + \sum_{i \in \mathcal{S}} s_i a_i + c \sum_{i \in \mathcal{S}} \lambda_i \hat{x}_i \\ &= \sum_{i \in \mathcal{S}} r_i + \sum_{i \in \mathcal{S}} s_i a_i + c\big(\sum_{i \in \mathcal{S}} \lambda_i x_i + \sum_{i \in \mathcal{S}} \lambda_i \hat{\alpha}\big) \\ &= \sum_{i \in \mathcal{S}} r_i + \sum_{i \in \mathcal{S}} s_i a_i + c\big(\sum_{i \in \mathcal{S}} \lambda_i x_i + \hat{\alpha} \sum_{i \in \mathcal{S}} \lambda_i\big) \\ &= r + c(x + \hat{\alpha}) \\ &= r + c\hat{x} \end{aligned} \tag{2}$$

Hence, verification will hold when the protocol is performed honestly.

$\mathsf{Setup}(1^\lambda)$

$(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$

$\mathsf{H}_{\mathsf{non}}, \mathsf{H}_{\mathsf{sig}} : \{0,1\}^* \to \mathbb{Z}_p$

$\mathsf{par} \leftarrow ((\mathbb{G}, p, g), \mathsf{H}_{\mathsf{non}}, \mathsf{H}_{\mathsf{sig}})$

return par

$\mathsf{KeyGen}(n, t)$

$x \xleftarrow{\$} \mathbb{Z}_p; \mathcal{PK} \leftarrow g^x$

$\{(i, x_i)\}_{i\in[n]} \xleftarrow{\$} \mathcal{SS}.\mathsf{IssueShares}(x, n, t)$

// Shamir secret sharing of x

for $i \in [n]$ $\mathcal{PK}_i \leftarrow g^{x_i}$

return $(\mathcal{PK}, \{\mathcal{PK}_i\}_{i\in[n]}, \{x_i\}_{i\in[n]})$

$\mathsf{GenRand}()$

$\alpha \xleftarrow{\$} \mathbb{Z}_p$

return α

$\mathsf{RandShare}(x_i, \alpha, \mathsf{aux})$

$\hat{\alpha} \leftarrow \mathsf{H}_{\mathsf{r}}(\alpha, \mathsf{aux})$

$\bar{x}_i \leftarrow x_i + \hat{\alpha}$

return $\bar{x}_i$

$\mathsf{RandPKShare}(\mathcal{PK}_i, \alpha, \mathsf{aux})$

$\hat{\alpha} \leftarrow \mathsf{H}_{\mathsf{r}}(\alpha, \mathsf{aux})$

$\bar{\mathcal{PK}}_i \leftarrow \mathcal{PK}_i \cdot g^{\hat{\alpha}}$

return $\bar{\mathcal{PK}}_i$

$\mathsf{RandPK}(\mathcal{PK}, \alpha, \mathsf{aux})$

$\hat{\alpha} \leftarrow \mathsf{H}_{\mathsf{r}}(\alpha, \mathsf{aux})$

$\bar{\mathcal{PK}} \leftarrow \mathcal{PK} \cdot g^{\hat{\alpha}}$

return $\bar{\mathcal{PK}}$

$\mathsf{Sign}()$ *// Local signer has index k*

$r_k, s_k \xleftarrow{\$} \mathbb{Z}_p^2$

$R_k \leftarrow g^{r_k}$; $S_k \leftarrow g^{s_k}$;

$\mathsf{st}_k \leftarrow (R_k, r_k, S_k, s_k)$

$\rho_k \leftarrow (R_k, S_k)$

return (ρ_k, st_k)

$\mathsf{Sign}'(\mathsf{st}_k, \bar{x}_i, \bar{\mathcal{PK}}, m, \{\rho_i\}_{i\in\mathcal{S}})$

parse $(R_k, r_k, S_k, s_k) \leftarrow \mathsf{st}_k$

parse $\{(i, R_i, S_i)\}_{i\in S} \leftarrow \{\rho_i\}_{i\in\mathcal{S}}$

return $\perp$ **if** $(R_k, S_k) \notin \{R_i, S_i\}_{i\in S}$

for $i \in \mathcal{S}$ **do**

$a_i \leftarrow \mathsf{H}_{\mathsf{non}}(k, m, \bar{\mathcal{PK}}, \{(i, R_i, S_i)\}_{i\in\mathcal{S}})$

$R \leftarrow \prod_{i\in\mathcal{S}} R_i \cdot {S_i}^{a_i}$

$c \leftarrow \mathsf{H}_{\mathsf{sig}}(\bar{\mathcal{PK}}, R, m)$

$z_k \leftarrow r_k + s_k a_k + c\lambda_k \bar{x}_k$

$\rho'_k \leftarrow z_k$

return ρ'_k

$\mathsf{Combine}(\{(\bar{\mathcal{PK}}, \rho_i, \rho'_i)\}_{i\in\mathcal{S}})$

parse $(R_i, S_i) \leftarrow \rho_i, z_i \leftarrow \rho'_i, i \in \mathcal{S}$

$R \leftarrow \prod_{i\in\mathcal{S}} R_i \cdot {S_i}^{a_i}$

$c \leftarrow \mathsf{H}_{\mathsf{sig}}(\bar{\mathcal{PK}}, R, m)$

$z \leftarrow \sum_{i\in S} z_i$

$\sigma \leftarrow (R, z)$

return σ

$\mathsf{Verify}(\bar{\mathcal{PK}}, m, \sigma)$

// Identical to plain single-party Schnorr

parse $(R, z) \leftarrow \sigma$

$c \leftarrow \mathsf{H}_{\mathsf{sig}}(\bar{\mathcal{PK}}, m, R)$

if $R \cdot \bar{\mathcal{PK}}^c = g^z$

return 1

return 0

Fig. 5. Rerandomized-FROST, which extends the two-round threshold signature scheme FROST to allow for rerandomizable keys. Differences with plain FROST are highlighted in a grey box.

5.1 Alternative Designs

The design of re-randomized FROST assumes a central party to choose the randomization factor; and hence is trusted with preserving privacy of the scheme. However, this party is *not* trusted with security; even if this party were to act maliciously, then the scheme remains secure.

To remove this trusted party, a distributed key generation (DKG) scheme could instead be used, where all parties would contribute to the randomization factor, but no party would learn the value directly. However, this approach requires additional network rounds of communication, and so requires a tradeoff in increased performance and complexity overhead.

6 Security

We now demonstrate the unforgeability of Rerandomized-FROST.

Theorem 1. Rerandomized-FROST *is unforgeable in the random oracle model, assuming the AOMDL assumption holds in* $\mathbb{G}$*. In other words, for every adversary* $\mathcal{A}$ *that wins the existential unforgeability game against* Rerandomized-FROST*, there exists an adversary* $\mathcal{D}$ *that wins the AOMDL game.*

Concretely, the advantage of $\mathcal{A}$ *is bounded by Equation 3.*

$$\mathsf{Adv}^{sec}_{\mathsf{TS},\mathcal{A}} \leq \sqrt{q \cdot \mathsf{Adv}^{aomdl}_{\mathcal{D}}(\lambda) + 2q^2/p - \mathsf{negl}(\lambda)} \tag{3}$$

where $q = q_h + q_s$*,* q_h *is the number of allowed random oracle queries, and* q_s *is the number of allowed signing queries.*

Proof. Let $\mathcal{B}$ be an adversary playing against the algebraic one-more discrete logarithm game as defined in Figure 1. Let $\mathcal{A}$ be an adversary playing against the unforgeability game against a rerandomized threshold signature scheme as defined in Figure 4, instantiated with Rerandomized-FROST.

The Reduction $\mathcal{B}$*.* To begin, $\mathcal{B}$ receives the set of $2q_S + 1$ challenges $X = \{X_0, \ldots, X_{2q_s}\} \in \mathbb{G}^{2q_s+1}$. It sets the public key $\mathcal{PK} = X_0$ to be the first AOMDL challenge. It runs $\mathcal{A}$ twice, simulating Rerandomized-FROST for each execution. After its first execution, $\mathcal{A}$ outputs a forgery $(m^*, \sigma^* = (R^*, z^*, \alpha^*, \mathsf{aux}^*)$.. Before it runs $\mathcal{A}$ a second time, it re-programs on the challenge query $\mathsf{H}_{\mathsf{sig}}(\mathcal{PK}', R^*, m^*)$ with a freshly sampled random value from $\mathbb{Z}_p$. It then runs $\mathcal{A}$ a second time. By the local forking lemma [2], with non-negligible probability, $\mathcal{A}$ outputs a second forgery $(m^*, \sigma^* = (R^*, z^{**}), \alpha^{**}, \mathsf{aux}^{**})$. Without loss of generality, we assume $\mathcal{A}$ queries $\mathsf{H}_{\mathsf{sig}}$ on its forgeries before outputting them.

We next explain in more detail how $\mathcal{B}$ performs its simulation.

Setup. $\mathcal{B}$ performs setup in the same way as for plain FROST. $\mathcal{B}$ is initialized with the parameters $(\mathbb{G}, g, p)$, and $2q_s + 1$ AOMDL challenges. It initializes tables $\mathsf{Q}_{\mathsf{rand}} \leftarrow \emptyset$, $\mathsf{Q}_{\mathsf{non}} \leftarrow \emptyset$, $\mathsf{Q}_{\mathsf{sig}} \leftarrow \emptyset$ to simulate random oracles $\mathsf{H}_{\mathsf{r}}, \mathsf{H}_{\mathsf{non}}, \mathsf{H}_{\mathsf{sig}}$. It initializes $Q_1 \leftarrow \emptyset, Q_2 \leftarrow \emptyset, Q_3 \leftarrow \emptyset$ to maintain state during its simulation. Finally, it initializes $Q \leftarrow \emptyset$ to maintain the set of messages queried by $\mathcal{A}$ to $\mathcal{O}^{\mathsf{Sign}'}$.

$\mathcal{B}$ then picks random coins ρ, and runs $\mathcal{A}(\mathsf{par}; \rho)$ once on the public parameters par and randomness ρ. $\mathcal{A}$ chooses the total number of potential signers n, the threshold t, and the initial set of corrupted participants $\mathsf{cor} \leftarrow \{j\}, |\mathsf{cor}| \leq t - 1$. $\mathcal{B}$ sets $\mathsf{hon} \leftarrow [n] \setminus \mathsf{cor}$. $\mathcal{B}$ then proceeds to simulating key generation to $\mathcal{A}$.

Simulating KeyGen. Simulating key generation in a trusted dealer manner is the same for Rerandomized-FROST as for the plain FROST setting. In particular, $\mathcal{B}$ simulates a Shamir secret sharing of the discrete logarithm of the AOMDL challenge $\mathcal{PK} = X_0$ by performing the following steps. Assume without loss of generality that $|\mathsf{cor}| = t - 1$.

1. $\mathcal{B}$ samples $(t-1)$ random values $x_j \xleftarrow{\$} \mathbb{Z}_p$ for $j \in \mathsf{cor}$.
2. Let f be the polynomial whose constant term is the challenge $f(0) = \dot{x}$ and for which $f(j) = x_j$ for all $j \in \mathsf{cor}$.

3. For all $1 \leq i \leq n$, $\mathcal{B}$ computes

$$\mathcal{PK}_i = \mathcal{PK}^{\lambda_i^0} \cdot \prod_{i=1}^{t-1} g^{x_k \lambda_i^j}$$

where λ_i^j is the j^{th}Lagrange coefficient interpolating point i, and where $\mathcal{PK}_i$ is implicitly equal to $g^{f(i)}$.

The joint public key is $\mathcal{PK} = g^{f(0)} = X_0$. $\mathcal{B}$ runs $\mathcal{A}^{\mathcal{O}^{\mathsf{Sign},\mathsf{Sign}'}}(\mathcal{PK}, \{\mathcal{PK}_i\}_{i\in[n]}, \{x_j\}_{j\in\mathsf{cor}})$.

Simulating Random Oracles. Simulating the random oracles $\mathsf{H}_{\mathsf{non}}$ and $\mathsf{H}_{\mathsf{sig}}$ is also identical between Rerandomized-FROST and plain FROST. $\mathcal{B}$ simulates random oracle queries by lazy sampling, as follows.

$\underline{\mathsf{H}_{\mathsf{r}}:}$ When $\mathcal{A}$ queries H_{r} on $(\alpha_i, \mathsf{aux}_i)$, $\mathcal{B}$ checks whether $(\alpha_i, \mathsf{aux}_i) \in \mathsf{Q}_{\mathsf{rand}}$ and, if so, returns $\hat{\alpha}$. Else, $\mathcal{B}$ samples $\hat{\alpha} \xleftarrow{\$} \mathbb{Z}_p$, appends $(\alpha_i, \mathsf{aux}_i)$ to $\mathsf{Q}_{\mathsf{rand}}$, and returns $\hat{\alpha}$.

$\underline{\mathsf{H}_{\mathsf{non}}:}$ When $\mathcal{A}$ queries $\mathsf{H}_{\mathsf{non}}$ on $(k, \mathcal{PK}, \{(k, R_i, S_i)\}_{i\in\mathcal{S}})$, $\mathcal{B}$ checks whether $(k, \mathcal{PK}, m, \{(i, R_i, S_i)\}_{i\in\mathcal{S}}, a_k) \in \mathsf{Q}_{\mathsf{non}}$ and, if so, returns a_k. Else, $\mathcal{B}$ samples $a_k \xleftarrow{\$} \mathbb{Z}_p$ and appends $(k, \mathcal{PK}, m, \{(i, R_i, S_i)\}_{i\in\mathcal{S}}, a_k)$ to $\mathsf{Q}_{\mathsf{non}}$.

Then, for all other $j \in \mathcal{S}$, $\mathcal{B}$ samples $a_j \xleftarrow{\$} \mathbb{Z}_p$ and appends $(j, \mathcal{PK}, m, \{(i, R_i, S_i)\}_{i\in\mathcal{S}}, a_k)$ to $\mathsf{Q}_{\mathsf{non}}$. Finally, $\mathcal{B}$ derives $R \leftarrow \prod_{i\in\mathcal{S}} R_i S_i^{a_i}$, and checks if $(\mathcal{PK}, R, m) \in \mathsf{Q}_{\mathsf{sig}}$. If the check holds, then $\mathcal{B}$ returns BadHashEvent. Otherwise, $\mathcal{B}$ samples $c \xleftarrow{\$} \mathbb{Z}_p$, appends $(\mathcal{PK}, m, R, c)$ to $\mathsf{Q}_{\mathsf{sig}}$, and returns c.

$\underline{\mathsf{H}_{\mathsf{sig}}:}$ When $\mathcal{A}$ queries $\mathsf{H}_{\mathsf{sig}}$ on $(\mathcal{PK}, m, R)$, $\mathcal{B}$ checks whether $(\mathcal{PK}, m, R, c) \in \mathsf{Q}_{\mathsf{sig}}$ and, if so, returns c. Else, $\mathcal{B}$ samples $c \xleftarrow{\$} \mathbb{Z}_p$, appends $(\mathcal{PK}, m, R, c)$ to $\mathsf{Q}_{\mathsf{sig}}$, and returns c.

Simulating Randomized Signing. $\mathcal{B}$ handles $\mathcal{A}$'s queries to perform randomized signing as follows.

Round 1 ($\mathcal{O}^{\mathsf{Sign}}(k, \mathsf{ssid})$): When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}}$ on honest participant identifier $k \in \mathsf{hon}$, and session identifier ssid, $\mathcal{B}$ simulates its response in exactly the same way as plain FROST. It picks the next two available AOMDL challenges (X_i, X_{i+1}) and sets $R_i = X_i, S_i = X_{i+1}$. It then outputs (R_i, S_i).

Round 2 ($\mathcal{O}^{\mathsf{Sign}'}(k, \mathsf{ssid}, m, \mathcal{S}, \alpha, \mathsf{aux}, \{R_i, S_i\}_{i\in\mathcal{S}})$): When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}}$ for honest participant identifier $k \in \mathsf{hon}$ and session identifier ssid, $\mathcal{B}$ obtains the blinding factor a_i and the challenge c honestly. $\mathcal{B}$ then performs RandPK honestly; it does not perform RandShare because it does not know its own share. However, it does perform $\mathsf{RandPKShare}(\mathcal{PK}_k, \alpha, \mathsf{aux})$, obtaining the randomized public key $\mathcal{P}\hat{\mathcal{K}}_i$. It then derives $Z_k \leftarrow R_i \cdot S_i^{a_k} \cdot \mathcal{P}\hat{\mathcal{K}}_k^{\,c}$. It queries $\mathcal{O}^{\mathsf{dl}}$ on Z_i and its representation, obtaining z_k in return. It outputs z_k.

Analysis of Simulation. $\mathcal{B}$'s simulation of $\mathsf{H}_{\mathsf{non}}$ and $\mathsf{H}_{\mathsf{sig}}$ are indistinguishable because it performs lazy sampling. The bad event BadHashEvent will occur with negligible probability, because $\mathcal{A}$ would need to guess each a_i for it to occur.

$\mathcal{B}$'s simulation of Sign and Sign′ are perfect. Because $\mathcal{B}$ outputs two AOMDL challenges $(R_i = X_i, S_i = X_{i+1})$ as its commitments for Sign, these are indistinguishable from values sampled honestly. Because $\mathcal{B}$ queries $\mathcal{O}^{\mathsf{dl}}$ on $Z_i = R_i S_i^{a_i} \mathcal{P}\mathcal{K}_i^{c_i}$, the output signature z_i will be valid and indistinguishable from an honestly generated signature.

Output. At the end of $\mathcal{A}$'s first execution, it will output a forgery $(m^*, \sigma^* = (R^*, z^*), \alpha^*, \mathsf{aux}^*)$. By the local forking lemma [2], with non-negligible probability, $\mathcal{A}$ outputs a second forgery $(m^*, \sigma^* = (R^*, z^{**}), \alpha^{**}, \mathsf{aux}^{**})$.

Extracting the Discrete Logarithm of X_0. $\mathcal{B}$ can extract y_0 which is the discrete logarithm of X_0 as follows. If $\alpha = \perp$, it simply follows the same steps to extract y_0 as in the plain FROST proof (this is the non-rerandomized case). However, if $\alpha \neq \perp$, it solves for y_0 as follows. It first derives $z_0 \leftarrow z^* - c^* \mathsf{H}_{\mathsf{r}}(\alpha^*, \mathsf{aux}^*)$, output during the first execution of $\mathcal{A}$. It then derives $z_1 \leftarrow z^{**} - c^{**} \mathsf{H}_{\mathsf{r}}(\alpha^{**}, \mathsf{aux}^{**})$, output during the second execution of $\mathcal{A}$. It then solves for:

$$y_0 = \frac{z_0 - z_1}{c^* - c^{**}}$$

Extracting $X_1, \ldots, X_{2q_s}$. Extracting the discrete logarithms of the remaining challenges is identical to the proof for FROST [1], with the exception that $\mathcal{D}$ must additionally take into consideration the randomizer when extracting. However, because this value is public, it can extract using similar techniques as in the proof for plain FROST.

For example, in the case where the adversary queries $\mathcal{O}^{\mathsf{Sign}'}$ on the nonce pair (R_i, S_i) but with a different challenges $c_0 \neq c_1$ and different randomizers $a_0 \neq a_1$, $\mathcal{B}$ can derive (r_i, s_i) with the following steps:

1. Derive $z' \leftarrow z_0 - z_1$, cancelling out r_i.
2. Derive $s_i \leftarrow \frac{z' - c_0 x_i \lambda_i - c_0 \alpha_0 \lambda_i - c_1 x_i \lambda_i - c_1 \alpha_1 \lambda_i}{a_0 - a_1}$.
3. Use s_i to obtain $r_i \leftarrow z_0 - s_i a_0 - c_0 x_i \lambda_i - c_0 \alpha_0 \lambda_i$.

Note that the above approach would hold even if $\alpha_0 \neq \alpha_1$, due to the fact that this value is public. Further, in the case where $c_0 = c_1, a_{i0} = a_{i1}$ but $\alpha_0 \neq \alpha_1$, the reduction could still solve for (r_i, s_i) given that these values are known to the reduction.

The remaining cases of when the adversary queries $\mathcal{O}^{\mathsf{Sign}'}$ on the same challenge or the bad event cases are similarly identical to the case of plain FROST, but where additionally (α_0, α_1) are used when solving for (r_i, s_i).

As such, $\mathcal{D}$ can solve for $2q_s + 1$ AOMDL challenges given $2q_s$ queries to the AOMDL solution oracle. This completes the proof. □

7 Implementation

To validate the protocol in practice, we have implemented re-randomized FROST as a Rust crate[1], on top of a regular FROST implementation on the same repository. Since re-randomized FROST uses the same operations as regular FROST, the only thing needed was to implement methods to randomize private and public keys, as well as public key shares. We do provide re-randomized "sign" and "aggregate" functions which do the randomization internally, this makes it easier for callers to use the crates correctly.

These crates provide generic implementations which require the specification of a "ciphersuite", that is, the set of parameters being used which include the elliptic curve and hash functions. Thus we have also implemented two re-randomized FROST ciphersuites[2], one for the JubJub curve and other for the Pallas curve, both used in the Zcash protocol [9]. These allow signing Zcash transactions with FROST, enabling threshold wallets.

The straightfoward approach for implementing re-randomized FROST is to change FROST in the following ways:

- After round 1, when the coordinator receives the commitments from the signers, it generates the randomizer α and derives the "effective" randomizer $\hat{\alpha} \leftarrow \mathsf{H}_{\mathsf{r}}(\alpha, \mathsf{aux})$. Note that in practice this can be simplified, without loss of security, by generating a string of random bytes with the same size as the randomizer and hashing that into H_{r} along with aux, which is easier than generating a random scalar α. As for aux, a sensible approach is to use a byte encoding of the message and the set of commitments, which will bind the randomizer to this specific signing session, ensure that each signing session will use a unique and uniformly random randomizer, even in case of random number generator failure. Other application-specific data might be included along with aux.
- The coordinator then sends the effective randomizer $\hat{\alpha}$ along with the message and the set of commitments to each participant. Note that, in most applications, it will be desirable to encrypt the randomizer so that eavesdroppers can't break unlinkability.
- In round 2, when generating the signature share, each participant re-randomizes their signing key and the group pubic key using the effective randomizer, and the proceed as the regular FROST signing step.

[1] `https://github.com/ZcashFoundation/frost/`
[2] `https://github.com/ZcashFoundation/reddsa/`

Table 1. Benchmark results of regular and re-randomized FROST, in µs.

	2 signers		7 signers		67 signers	
Function	Sign	Aggregate	Sign	Aggregate	Sign	Aggregate
Regular	205	573	396	757	2740	3139
Re-randomized	375	734	573	953	2974	3436
Overhead	170	161	177	196	234	297
Overhead	83%	28%	45%	26%	8%	9%

– After round 2, before aggregating the received signature shares into the final signature, the coordinator also re-randomizes the group public key (and the signers' public key shares) with the effective randomizer and proceeds as the regular FROST aggregation step.

Using this approach, the coordinator is trusted with the privacy of the signature (i.e. they could leak the randomizer and allow attackers to link different signatures) but they still can't forge signatures. This aligns with applications such as Zcash [9], where the transaction builder is trusted with the privacy of the transaction, including the generation of the zero-knowledge proof, while spend authority is restricted to the holder of the spending key (which allows, for example, using hardware devices to sign transactions, without requiring the expensive zero-knowledge proof creation).

The performance overhead over regular FROST is thus:

For each participant: one fixed-base elliptic curve point multiplication to compute $g^{\hat{\alpha}}$, to randomize the group public key;

For the coordinator: one fixed-base elliptic curve point multiplication to compute $g^{\hat{\alpha}}$, to randomize the group public key and the signer's public key shares (it can be reused in both computations).

The most time-consuming part of regular FROST is the group commitment computation which requires a multi-scalar point multiplication (one scalar for each signer). The precise performance impact will depend on the specific multi-scalar and fixed-base optimizations being employed, but assuming naive non-optimized implementations, the worst-case overhead of re-randomized FROST (which is for two signers), is $(2+1)/2 = 1.5$, i.e. 50%. This overhead is reduced when the number of signers is larger. We expect that this overhead is acceptable in most practical deployments.

To validate this analysis, we have benchmarked our implementation of the Pallas ciphersuite. A 3.7 GHz AMD Ryzen 9 5900X 12-core processor running Ubuntu 22.04 was used. We provide these benchmarks with the purpose of a high-level comparison between regular and re-randomized FROST, and not to provide cycle-accurate state-of-the-art performance numbers; in particular, note that an assembly-optimized elliptic curve implementation was not used. The results are shown in Table 1.

The overhead for signing with two participants (83%) is worse than the naive estimate (50%) because the implementation optimizes the multi-scalar implementation, making the baseline timing faster, and due to additional overheads like point additions for the re-randomized variant. The Aggregate overhead is smaller then the Sign overhead because baseline Aggregate is slower than Sign (it requires a signature verification in addition to the group commitment computation which is shared between the two operations). It can be clearly seen that the relative overhead is reduced when the number of signers increase, since the absolute overhead mostly does not depend on the number of signers (it does require a point addition per signer to randomize the public key shares).

8 Acknowledgements

Thank you to the engineers at the Zcash Foundation and Daira Hopwood for their helpful feedback and discussion.

References

[1] M. Bellare, E. Crites, C. Komlo, M. Maller, S. Tessaro, and C. Zhu. *Better than advertised security for non-interactive threshold signatures*. CRYPTO 2022. To appear. 2022.

[2] M. Bellare, W. Dai, and L. Li. "The Local Forking Lemma and Its Application to Deterministic Encryption". In: *ASIACRYPT 2019, Kobe, Japan, December 8-12, 2019*. Ed. by S. D. Galbraith and S. Moriai. Vol. 11923. LNCS. Springer, 2019, pp. 607–636.

[3] D. Connolly, C. Komlo, I. Goldberg, and C. Wood. *Two-Round Threshold Schnorr Signatures with FROST*. 2022. URL: https://datatracker.ietf.org/doc/draft-irtf-cfrg-frost/.

[4] E. C. Crites, C. Komlo, and M. Maller. "Fully Adaptive Schnorr Threshold Signatures". In: *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part I*. Ed. by H. Handschuh and A. Lysyanskaya. Vol. 14081. Lecture Notes in Computer Science. Springer, 2023, pp. 678–709. DOI: 10.1007/978-3-031-38557-5_22. URL: https://doi.org/10.1007/978-3-031-38557-5_22.

[5] N. Fleischhacker, J. Krupp, G. Malavolta, J. Schneider, D. Schröder, and M. Simkin. "Efficient Unlinkable Sanitizable Signatures from Signatures with Re-randomizable Keys". In: *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part I*. Ed. by C. Cheng, K. Chung, G. Persiano, and B. Yang. Vol. 9614. Lecture Notes in Computer Science. Springer, 2016, pp. 301–330. DOI: 10.1007/978-3-662-49384-7_12. URL: https://doi.org/10.1007/978-3-662-49384-7_12.

[6] R. Gennaro and S. Goldfeder. "Fast Multiparty Threshold ECDSA with Fast Trustless Setup". In: *CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by D. Lie, M. Mannan, M. Backes, and X. Wang. ACM, 2018, pp. 1179–1194.

[7] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. "Robust Threshold DSS Signatures". In: *Inf. Comput.* 164.1 (2001), pp. 54–84.

[8] R. Gennaro, T. Rabin, S. Jarecki, and H. Krawczyk. "Robust and Efficient Sharing of RSA Functions". In: *J. Cryptol.* 20.3 (2007), p. 393.

[9] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. *Zcash Protocol Specification*. 2022. URL: https://zips.z.cash/protocol/protocol.pdf.

[10] C. Komlo and I. Goldberg. "FROST: Flexible Round-Optimized Schnorr Threshold Signatures". In: *SAC 2020, Halifax, NS, Canada (Virtual Event), October 21-23, 2020*. Ed. by O. Dunkelman, M. J. J. Jr., and C. O'Flynn. Vol. 12804. LNCS. Springer, 2020, pp. 34–65.

[11] J. Nick, T. Ruffing, and Y. Seurin. "MuSig2: Simple Two-Round Schnorr Multi-signatures". In: *CRYPTO 2021, Virtual Event, August 16-20, 2021*. Ed. by T. Malkin and C. Peikert. Vol. 12825. LNCS. Springer, 2021, pp. 189–221.

[12] Penumbra Labs. *The Penumbra Protocol*. 2022. URL: https://protocol.penumbra.zone/main/penumbra.html.

[13] A. Shamir. "How to Share a Secret". In: *Commun. ACM* 22.11 (1979), pp. 612–613.