Wamu: A Protocol for Computation of Threshold Signatures by Multiple Decentralized Identities

Technical Specification

David Semakula
hello@davidsemakula.com
https://davidsemakula.com

Published: 15th May, 2023
Last Updated: 13th November, 2023
Version: 1.6

Contents

1. Introduction

This document describes the Wamu protocol which augments a state-of-the-art non-interactive threshold signature scheme (e.g. CGGMP20 [1]) by cryptographically associating each signing party with a decentralized identity. This is achieved by:

- Splitting the secret share for each party between the party and the output of a signing operation by its associated decentralized identity, thus making the signing operation a requirement for reconstructing the party's secret share.
- Adding peer-to-peer decentralized identity authentication to the key generation and signing protocols (and optionally to the key refresh protocol) of the threshold signature scheme.
- Defining protocols for identity rotation, access structure modification (i.e. share addition and removal and threshold modification) and share recovery that build on top of the above 2 augmentations.

Wamu is designed to operate in a decentralized, trust-minimized and asynchronous setting with:

- no centralized or trust-based identity infrastructure.
- signing parties being mainstream consumer devices communicating asynchronously.

NOTE: For interoperability with existing wallet solutions, the only requirement for decentralized identity providers is the ability to compute cryptographic signatures for any arbitrary message in such a way that the output signature is 1) deterministic and 2) can be verified in a non-interactive manner.

2. Preliminaries

The rest of this document describes the Wamu protocol in technical detail. For these descriptions, we'll use the following notation:

- P denotes a party.
- I denotes a decentralized identity.
- vk denotes the verifying key (or address) of a decentralized identity.
- sk denotes the secret key of a decentralized identity.
- `Sig` denotes a signing algorithm.
- `Ver` denotes a signature verification algorithm.
- q denotes the prime order of the cyclic group of the elliptic curve.
- $\mathcal{S}$ denotes the set of verified decentralized identities for all parties.
- t denotes the threshold (i.e. the minimum number of signatories required to jointly compute a valid signature using the threshold signature scheme).
- A denotes a predefined prefix chosen to ensure that signatures computed for identity authentication cannot be valid transaction signatures.
- $\|$ denotes concatenation using an unambiguous encoding scheme.

NOTE: While the augmenting protocols in this document are described in relation to the current (circa. 2023) state-of-the-art CGGMP20 [1] non-interactive threshold signature scheme for ECDSA signatures, Wamu is a generic protocol that can be adapted to any non-interactive threshold signature scheme (e.g. GG20 [2], CMP20 [3] and FROST20 [4]) that allows for asynchronous communication between signing parties.

3. Share Splitting and Reconstruction

Given a secret share x for a party P with an associated decentralized identity I, the share splitting and reconstruction protocol describes how to split x between P and the output of a signing operation `Sig` by I so that the output of `Sig` is required to reconstruct the secret share x.

This is achieved by generating a message k (we'll refer to this message as the "signing share") and computing a "sub-share" β (i.e a share of the secret share x) in such a way that k needs to be signed by I using `Sig` to produce another "sub-share" α, such that α and β are shares of x under Shamir's secret-sharing scheme [5].

NOTE: Share splitting and reconstruction is a single-party localized concern that happens after (and is not related to) the distributed key generation (DKG) protocol of the threshold signature scheme.

3.1. Share splitting

Given a secret share x as input and access to the decentralized identity I with secret key sk, the share splitting protocol proceeds as follows:

1. Sample a random message k (i.e. the signing share).
2. Compute a signature $(r, s) \leftarrow \texttt{Sig}(sk, k)$.
3. Compute the first sub-share of x as the point $\alpha = (r, s) \pmod q$.
4. Generate a line L (i.e a polynomial of degree 1) such that α is a point on the line and x is the constant term (i.e. Polynomial Interpolation [6])
5. Compute another point β from L such that $\beta \neq \alpha$, β becomes the second sub-share of x.
6. Erase both α and L from memory.
7. Return the signing share k and the sub-share β.

3.2. Share reconstruction

Given a signing share k and a sub-share β as input (i.e. the outputs of the share splitting protocol in section 3.1) and access to the decentralized identity I with secret key sk, the share reconstruction protocol proceeds as follows:

1. Compute a signature $(r, s) \leftarrow \texttt{Sig}(sk, k)$.
2. Compute a sub-share α as the point $\alpha = (r, s) \pmod q$.

3. Generate the line L by performing Polynomial Interpolation [6] using α and β as inputs.
4. Compute x as the constant term of L.
5. Erase both α and L from memory.
6. Return x as the secret share.

NOTE: The signature parameters r and s in $(r, s) \leftarrow \texttt{Sig}(sk, k)$ are already computed modulo q. We use the notation $\alpha \leftarrow (r, s) \pmod{q}$ for the sub-share to make it clear (at a glance) that the sub-shares are computed using finite field arithmetic.

4. Threshold Signature Scheme Augmentations

The general approach for augmenting threshold signature protocols (i.e. key generation and signing - and optionally key refresh) is for each party to sign a non-interactive replay resistant challenge during the first round of communication to prove that it currently controls the associated decentralized identity. The other parties then verify the challenge signature at the beginning of the next round or identify the culprit and halt.

Key generation and key refresh protocols typically include a commitment to secret and random values in their first round while signing includes an arbitrary message, so either a commitment (e.g. for key generation and key refresh) or the message (e.g. for signing) is unambiguously concatenated with a protocol specific prefix and the current timestamp to generate a non-interactive replay resistant challenge.

NOTE: While most threshold signature schemes don't define a key refresh protocol (e.g. GG20 [2] and FROST20 [4]), it is relatively straightforward to derive such a protocol from a standard proactive secret sharing scheme like HJKY95 [7]. However, for applications that require support for access structure modification, it is preferable to derive a key refresh protocol from a share redistribution scheme like DJ97 [8] or WW01 [9], as the latter are more flexible and allow for both proactive security and access structure changes (see section 6 for details and additional considerations).

NOTE: While general (t, n) sharing (and specifically the case where $t < n$) is not formally specified in CGGMP20 [1], it can be derived in a relatively straightforward manner based on GG18 [10] (and GG20 [2]) for the key generation and signing protocols (as described in section 1.2.8 of CGGMP20 [1]) and HJKY95 [7] (or WW01 [9]) for the key refresh protocol. In particular, this entails performing t-out-of-n Feldman's verifiable secret sharing [11] of the secret shares for key generation (as described in section 2.8 and phase 2 of section 3.1 in GG20 [2] and similarly in section 2.6 and phase 2 of section 4.1 in GG18 [10]) or refresh shares for key refresh (with some modifications as described in sections 3.3 and 3.4 of HJKY95 [7] or in section 4 of WW01 [9]), and transforming (t, n) to $(t, t+1)$ shares (using the appropriate Lagrangian coefficients) for pre-signing

and signing (as described in section 3.2 in GG20 [2] and similarly in section 4.2 in GG18 [10]).

4.1. Key Generation

Follow the key generation protocol described in section 3.1 and figure 5 of CGGMP20 [1] to generate ECDSA secret shares with the following modifications:

1. At the end of Round 1, broadcast 2 additional parameters for each P_i associated with the decentralized identity I_i with verifying key vk_i and secret key sk_i as follows:
   - The decentralized identity verifying key vk_i.
   - The current UTC timestamp Δ.
   - The signature $\sigma_i \leftarrow \texttt{Sig}(sk_i, A\|\Delta\|V_i)$.
2. At the beginning of Round 2, for each P_i, verify σ_j from all P_j where $j \neq i$:
   - Verify that $vk_i \in \mathcal{S}$ or report the culprit and halt.
   - Verify σ_j by checking that the output of $\texttt{Ver}(vk_j, A\|\Delta\|V_j, \sigma_j)$ is valid or report the culprit and halt.
3. After the Output phase, follow the share splitting protocol in section 3.1 to split secret share x_i into a signing share k_i and a sub-share β_i for each party P_i.
4. Modify Stored State for each P_i as follows:
   - Don't store x_i.
   - Add vk_i, k_i and β_i.

4.2. Signing

Follow the signing protocol described in sections 4.2 and 4.3 and figure 8 of CGGMP20 [1] to generate an ECDSA signature with the following modifications:

1. Before Round 1, for each party P_i, follow the share reconstruction protocol in section 3.2 to reconstruct secret share x_i.
2. At the end of Round 1, for each P_i associated with the decentralized identity I_i with verifying key vk_i and secret key sk_i, send 2 additional parameters to all P_j where $j \neq i$ as follows:
   - The decentralized identity verifying key vk_i.
   - The current UTC timestamp Δ.
   - The signature $\sigma_i \leftarrow \texttt{Sig}(sk_i, A\|\Delta\|m)$.
3. At the beginning of the Output phase, verify σ_j from all P_j where $j \neq i$ as follows:
   - Verify that $vk_i \in \mathcal{S}$ or report the culprit and halt.
   - Verify that t is within the current epoch for identity authenticated requests or report the culprit and halt.
   - Verify σ_i by checking that the output of $\texttt{Ver}(vk_i, A\|\Delta\|m, \sigma_i)$ is valid or report the culprit and halt.

4.3. Key Refresh

Follow the key refresh protocol described in section 3.2 and figure 6 of CGGMP20 [1] to generate new ECDSA secret shares with the following modifications:

1. At the end of Round 1, broadcast 2 additional parameters for each P_i associated with the decentralized identity I_i with verifying key vk_i and secret key sk_i as follows:
   - The decentralized identity verifying key vk_i.
   - The current UTC timestamp Δ.
   - The signature $\sigma_i \leftarrow \texttt{Sig}(sk_i, A \| \Delta \| V_i)$.
2. At the beginning of Round 2, for each P_i, verify σ_j from all P_j where $j \neq i$ as follows:
   - Verify that $vk_i \in \mathcal{S}$ or report the culprit and halt.
   - Verify σ_i by checking that the output of $\texttt{Ver}(vk_j, A \| \Delta \| V_j, \sigma_j)$ is valid or report the culprit and halt.
3. After the Output phase, follow the share splitting protocol in section 3.1 to split the new secret share x_i^* into a new signing share k_i^* and a new sub-share β_i^* for each party P_i.
4. Modify Stored State for each P_i as follows:
   - Don't store x_i^*.
   - Replace k_i with k_i^* and β_i with β_i^*.

5. Identity Authentication and Quorum Approval

5.1. Identity Authenticated Request

Decentralized identity authenticated requests allow parties to perform or request actions based on their associated decentralized identity.

5.1.1. Identity Authenticated Request Initiation To initiate an identity authenticated request with a command C from a party P_i associated with decentralized identity I_i with verifying key vk_i and secret key sk_i:

1. Read the current UTC timestamp Δ.
2. Compute the signature $\sigma \leftarrow \texttt{Sig}(sk_i, A \| \Delta \| C)$.
3. Broadcast C, vk_i, Δ and σ.

5.1.2. Identity Authenticated Request Verification To verify an identity authenticated request with a command C from a party P_i given its associated decentralized identity verifying key vk_i, a timestamp Δ, a signature σ and a set of verified decentralized identities for all other parties $\mathcal{S}$ as input:

1. Verify that $vk_i \in \mathcal{S}$ or report the culprit and halt.
2. Verify that t is within the current epoch for identity authenticated requests or report the culprit and halt.
3. Verify σ by checking that the output of $\texttt{Ver}(vk_i, A \| \Delta \| C, \sigma)$ is valid or report the culprit and halt.

5.2. Identity Challenge

Identity challenges are used to verify that a party controls a decentralized identity.

5.2.1. Identity Challenge Initiation To issue an identity challenge to a party P_i from all verifying parties P_j where $j \neq i$ for a verified request with command C initiated at timestamp Δ: 1. Sample a random v_j. 2. Broadcast v_j, C and Δ to all parties, such that all parties can compute $v = \|_{j \neq i} v_j$.

5.2.2. Identity Challenge Response For a party P_i with associated decentralized identity secret key sk_i, to respond to an identity challenge for a command C initiated at timestamp Δ, given v_j from all parties P_j where $j \neq i$:

1. Compute $v = \|_{j \neq i} v_j$.
2. Compute the signature $\sigma \leftarrow \texttt{Sig}(sk_i, A \| \Delta \| C \| v)$.
3. Broadcast C, vk_i, Δ and σ to all verifying parties P_j.

5.2.3. Identity Challenge Response Verification To verify an identity challenge response from a party P_i for a command C initiated at timestamp Δ, given its associated decentralized identity verifying key vk_i, a signature σ and v_j from all verifying parties P_j where $j \neq i$ as input:

1. Compute $v = \|_{j \neq i} v_j$.
2. Verify σ by checking that the output of $\texttt{Ver}(vk_i, A \| \Delta \| C \| v, \sigma)$ is valid or report the culprit and halt.

5.3. Identity Rotation

Identity rotation allows any party to change the decentralized identity associated with its secret share.

Identity rotation for a party P_i from a decentralized identity I_i with verifying key vk_i and secret key sk_i to a decentralized identity I_i^* with verifying key vk_i^* and secret key sk_i^* proceeds as follows:

1. For P_i, initiate an "identity-rotation" request by following the protocol in section 5.1.1.
2. For all P_j where $j \neq i$:
   - Verify the "identity-rotation" request by following the protocol in section 5.1.2.
   - Initiate an identity challenge for P_i by following the protocol in section 5.2.1.
3. For P_i, respond to the identity challenge by following the protocol in section 5.2.2 with the following augmentations:
   - Compute an additional signature $\sigma_i^* \leftarrow \texttt{Sig}(sk_i^*, A \| \Delta \| C \| v)$.
   - Add vk_i^* and σ_i^* to the broadcast parameters.
4. For all P_j where $j \neq i$:

- Verify the identity challenge response from P_i by following the protocol in section 5.2.3.
- Verify that P_i controls the new decentralized identity verifying key vk_i^* as follows:
  - Compute $v = \|_{j \neq i} v_j$:
  - Verify σ^* by checking that the output of $\texttt{Ver}(vk_i^*, A\|\Delta\|C\|v, \sigma^*)$ is valid or report the culprit and halt.
- Modify Stored State as follows:
  - Create $\mathcal{S}^*$ by replacing vk_i with vk_i^* in $\mathcal{S}$.
  - Replace $\mathcal{S}$ with $\mathcal{S}^*$.
- Broadcast confirmation of successful rotation of the verifying key for P_i.

5. For P_i, upon receiving confirmation of successful rotation from a quorum of P_j:
   - Compute the new signing share k_i^* and sub-share β_i^* based on the new decentralized identity I_i^* as follows:
     - Compute the secret share x_i by following the share reconstruction protocol in section 3.2.
     - Follow the share splitting protocol in section 3.1 to split x_i into a new signing share k_i^* and a new sub-share β_i^* based on the new decentralized identity I_i^*.
   - Modify Stored State as follows:
     - Replace vk_i with vk_i^* in $\mathcal{S}$.
     - Replace k_i with k_i^*.
     - Replace β_i with β_i^*.

5.4. Quorum Approved Request

Quorum approved requests allow any verified party to initiate actions that require explicit approval from a quorum of verified parties before execution (e.g. share addition and removal, and threshold modification).

A quorum approved request with a command C from a party P_i associated with decentralized identity I_i with verifying key vk_i and secret key sk_i proceeds as follows:

1. For P_i, initiate an identity authenticated request by following the protocol in section 5.1.1.
2. For all P_j where $j \neq i$ that approve the requested action:
   - Verify the identity authenticated request by following the protocol in section 5.1.2.
   - Initiate an identity challenge for P_i by following the protocol in section 5.2.1 with the following augmentations:
     - Compute a signature $\sigma_j \leftarrow \texttt{Sig}(sk_j, A\|\Delta\|C\|v_j)$.
     - Add vk_j and σ_j to the broadcast parameters.
3. For P_i, upon receiving an augmented identity challenge from a quorum $\mathcal{S}_c$ such that $\mathcal{S}_c \subseteq \mathcal{S} \wedge |\mathcal{S}_c| \geq t - 1$, respond to the identity challenge by

following the protocol in section 5.2.2 with the following modifications:

- At the beginning of the identity challenge response protocol, verify that approvals have been received from a valid quorum of signatories by checking that $\exists\, \mathcal{S}_c \subseteq \mathcal{S}$ such that $|\mathcal{S}_c| \geq t-1$ and $\forall\, vk_j \in \mathcal{S}_c$ where $j \neq i$, the output of $\texttt{Ver}(vk_j, A\|t\|C\|v_j, \sigma_j)$ is valid or report the culprit and halt.
- Compute v as $v = \|_{j \neq i}\, v_j$ where $v_j \in \mathcal{S}_c$.
- Add $\mathcal{S}_c$ to the broadcast parameters.

4. For all P_j where $j \neq i$:
   - Verify the augmented identity challenge response from P_i by following the protocol in section 5.2.3 with the following modifications:
     - Compute v as $v = \|_{j \neq i}\, v_j$ where $v_j \in \mathcal{S}_c$.
   - Verify that a valid quorum of signatories has approved the request as follows:
     - Verify that $|\mathcal{S}_c| \geq t-1$ or report the culprit and halt.
     - Verify that $\mathcal{S}_c \subseteq \mathcal{S} \wedge vk_i \notin \mathcal{S}_c$ or report the culprit and halt.
     - Verify that $\forall\, vk_j \in \mathcal{S}_c$ where $j \neq i$, the output of $\texttt{Ver}(vk_j, A\|\Delta\|C\|v_j, \sigma_j)$ is valid or report the culprit and halt.

6. Access Structure Modification

Access structure modification allows a quorum of verified parties to perform any of the following actions:

- share addition - issue a secret share to a new party and its associated decentralized identity
- share removal - revoke the secret share of any party.
- threshold modification - change the threshold (i.e. change the size of the quorum).

As noted in section 4, most threshold signature schemes don't define a key refresh protocol, and this is also the case for access structure modification protocols. However, it is similarly relatively straightforward to derive a suitable access structure modification protocol from a standard share redistribution scheme like DJ97 [8] or WW01 [9].

In fact, for applications that require support for access structure modification, it is preferable to replace a key refresh protocol based on (or similar to) a proactive secret sharing scheme like HJKY95 [7] (as is the case for CGGMP20 [1] key refresh) with a protocol based on (or similar to) a share redistribution scheme like DJ97 [8] or WW01 [9] as the latter are more flexible and allow for both proactive security and access structure changes.

NOTE: For threshold signature schemes with identifiable aborts (e.g. CGGMP20 [1], GG20 [2] and FROST20 [4]), key refresh protocols should be derived from verifiable share redistribution schemes like WW01 [9] to preserve the same security model.

Therefore, access structure modification can be achieved by following the augmented key refresh protocol described in section 4.3 of this document, with some modifications based on a verifiable share redistribution scheme like WW01 [9] (or similar) as described above. In particular, this entails each party (from a suitable subset of parties) performing a t'-out-of-n' (where t' and n' denote the new threshold and new number of parties respectively) Feldman's verifiable secret sharing [11] (with some modifications as described in section 4 of WW01 [9]) of its current secret share (i.e. the output from either key generation or the most recent key refresh) with other parties (in the suitable subset).

6.1. Share Addition

Share addition for a new party P_i with associated decentralized identity I_i proceeds as follows:

1. Initiate a quorum approved "share-addition" request by following the protocol in section 5.4.
2. Follow the augmented key refresh protocol described in section 4.3, with verifiable share redistribution modifications as described above and with P_i included as a participant, if the quorum approved request above succeeds.

6.2. Share Removal

Share removal for a party P_i with associated decentralized identity I_i proceeds as follows:

1. Initiate a quorum approved "share-removal" request by following the protocol in section 5.4.
2. Follow the augmented key refresh protocol described in section 4.3, with verifiable share redistribution modifications as described above and without P_i, if the quorum approved request above succeeds.

6.3. Threshold Modification

Threshold modification proceeds as follows:

1. Initiate a quorum approved "threshold-modification" request by following the protocol in section 5.4.
2. Follow the augmented key refresh protocol described in section 4.3, with verifiable share redistribution modifications as described above, if the quorum approved request succeeds.

7. Share Recovery

Share recovery is only possible if the user's decentralized identity either survived or can be recovered after the disastrous event. In either case, there are two options for share recovery depending on:

- A quorum of honest parties surviving the disastrous event.

- A backup (preferably encrypted) of a signing share k and sub-share β pair on user-controlled secondary or device-independent storage.

7.1. Share recovery with a surviving quorum of honest parties

If a quorum of honest parties survives the disastrous event, share recovery can be accomplished based on peer-to-peer decentralized identity authentication.

Share recovery for a party P_i with associated decentralized identity I_i with verifying key vk_i and secret key sk_i proceeds as follows:

1. For P_i, Initiate a "share-recovery" request by following the protocol in section 5.1.1.
2. For all P_j where $j \neq i$:
   - Verify the "share-recovery" request by following the protocol in section 5.1.2.
   - Initiate an identity challenge for P_i by following the protocol in section 5.2.1.
3. For P_i, respond to the identity challenge by following the protocol in section 5.2.2.
4. For all P_j where $j \neq i$, verify the identity challenge response from P_i by following the protocol in section 5.2.3.
5. Follow the key refresh protocol described in section 4.3 if all verifications above pass.

7.2. Share recovery with a backup

7.2.1. Overview of share recovery with a backup From the share splitting and reconstruction protocol in section 3, we note that for any party P, the combination of a signing share k and a sub-share β alone is insufficient to reconstruct the secret share x. This is because a signature of k from the decentralized identity I is required to compute the sub-share α, so that α and β can then be used to reconstruct L and compute the secret share x as the constant term of L.

Therefore, a signing share k and sub-share β pair can be safely backed up to user-controlled secondary (e.g. a secondary device or a flash drive) or device-independent storage (e.g. Apple iCloud [1], Google Drive [2], Microsoft OneDrive [3], Dropbox [4] e.t.c) without exposing the secret share x.

7.2.2. Generating an encrypted backup for share recovery For increased security, a signature of a standardized phrase can be used as entropy for generating an encryption secret which can then be used to encrypt the signing

[1] Apple iCloud. https://www.icloud.com.
[2] Google Drive. https://drive.google.com.
[3] Microsoft OneDrive. https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage.
[4] Dropbox. https://www.dropbox.com.

share k and the sub-share β using a symmetric encryption algorithm before saving them to back up storage.

Given a standardized phrase u, a key derivation function $\mathtt{H}$, a symmetric encryption algorithm $\mathtt{E}$, this proceeds as follows:

1. Compute the signature $\sigma \leftarrow \mathtt{Sig}(sk, u)$.
2. Generate the encryption secret $\varepsilon = \mathtt{H}(\sigma)$.
3. Compute the ciphertext for the signing share k as $k_c = \mathtt{E}_{enc}(k, \varepsilon)$.
4. Compute the ciphertext for the sub-share β as $\beta_c = \mathtt{E}_{enc}(\beta, \varepsilon)$.
5. Erase both σ and ε from memory.
6. Save k_c and β_c to backup storage.

7.2.3. Decrypting an encrypted backup Share recovery would then start by signing this standardized phrase, using the signature to recreate the encryption secret and then decrypting the encrypted backup to retrieve the signing share k and the sub-share β.

Given a standardized phrase u, a key derivation function $\mathtt{H}$, a symmetric encryption algorithm $\mathtt{E}$, the ciphertext for the signing share k_c and the ciphertext for the sub-share β_c, this proceeds as follows:

1. Compute the signature $\sigma \leftarrow \mathtt{Sig}(sk, u)$.
2. Generate the encryption secret $\varepsilon = \mathtt{H}(\sigma)$.
3. Compute the signing share $k = \mathtt{E}_{dec}(k_c, \varepsilon)$.
4. Compute the sub-share $\beta = \mathtt{E}_{dec}(\beta_c, \varepsilon)$.
5. Erase both σ and ε from memory.
6. Return the signing share k and the sub-share β.

7.2.4. Further security and usability considerations for share recovery with a backup For further improved security and usability, the signing share k can be prefixed with a custom message that alerts the user to the purpose of the signature. This can help reduce the effectiveness of an adversary that gains access to the backup and tries to trick the user into signing m.

Additionally, it's possible to rerun the share splitting protocol to generate a new pair of a signing share k^* and a sub-share β^* such that $k^* \neq k$, $\beta^* \neq \beta$ and $L^* \neq L$ to be specifically used for backup and recovery. This gives us the option to have separate signing shares for backup and recovery with customized prefixes that make it clear to the user that they're signing a backup signing share.

Lastly, the "backup" signing share k^* can be generated based on user input (e.g. a passphrase or security questions) removing the need for it to be backed up together with a sub-share β^* but instead relying on the user to provide this input during recovery as a security-usability tradeoff.

Acknowledgements

This work is funded by a grant from the Ethereum Foundation [5].

References

[1] Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N. and Peled, U. 2020. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security* (New York, NY, USA, 2020), 1769–1787. https://eprint.iacr.org/2021/060.

[2] Gennaro, R. and Goldfeder, S. 2020. One round threshold ECDSA with identifiable abort. Cryptology ePrint Archive, Paper 2020/540. https://eprint.iacr.org/2020/540.

[3] Canetti, R., Makriyannis, N. and Peled, U. 2020. UC non-interactive, proactive, threshold ECDSA. Cryptology ePrint Archive, Paper 2020/492. https://eprint.iacr.org/2020/492.

[4] Komlo, C. and Goldberg, I. 2020. FROST: Flexible round-optimized schnorr threshold signatures. Cryptology ePrint Archive, Paper 2020/852. https://eprint.iacr.org/2020/852.

[5] Shamir, A. 1979. How to share a secret. *Commun. ACM.* 22, 11 (Nov. 1979), 612–613. DOI:https://doi.org/10.1145/359168.359176.

[6] Wikipedia. Polynomial interpolation: *https://en.wikipedia.org/wiki/Polynomial_interpolation*. Accessed: 2023-05-12.

[7] Herzberg, A., Jarecki, S., Krawczyk, H. and Yung, M. 1995. Proactive secret sharing or: How to cope with perpetual leakage. *Advances in cryptology — CRYPT0' 95* (Berlin, Heidelberg, 1995), 339–352. https://doi.org/10.1007/3-540-44750-4_27.

[8] Desmedt, Y. and jodi, S.J. 1997. *Redistributing secret shares to new access structures and its applications.* Technical Report #ISSE-TR-97-01. George Mason University, Fairfax, VA 22030, Department of Computer Science. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=af623000d63f4c0251936b35c057a4d46581b4de.

[9] Wong, T.M. and Wing, J.M. 2001. *Verifiable secret redistribution.* Technical Report #ADA458508. Carnegie Mellon University, Pittsburgh, PA 15213, School of Computer Science. https://apps.dtic.mil/sti/tr/pdf/ADA458508.pdf.

[10] Gennaro, R. and Goldfeder, S. 2018. Fast multiparty threshold ECDSA with fast trustless setup. *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security* (New York, NY, USA, 2018), 1179–1194. https://eprint.iacr.org/2019/114.pdf.

[5] Ethereum Foundation: Ecosystem Support Program. https://esp.ethereum.foundation.

[11] Feldman, P. 1987. A practical scheme for non-interactive verifiable secret sharing. *Proceedings of the 28th annual symposium on foundations of computer science* (USA, 1987), 427–438. https://doi.org/10.1109/SFCS.1987.4.