Wamu: A Protocol for Building Threshold Signature Wallets Controlled by Multiple Decentralized Identities

Technical Specification

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

17th May, 2023

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 verification to the key generation and signing protocols (and optionally to the key refresh protocol) of the threshold signature scheme.
- Defining protocols for identity rotation, share addition and removal, 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 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.
- pk denotes the address (or public key) of a decentralized identity.
- sk denotes the secret key of a decentralized identity.
- $KeyGen$ denotes a key generation algorithm.
- Sig denotes a signing algorithm.
- Ver denotes a signature verification algorithm.
- S denotes the set of verified decentralized identities for all parties.
- q denotes the prime order of cyclic group of the elliptic curve.

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] and CMP20 [3]) 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 m (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 m 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 [4].

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 m (i.e. the signing share).
2. Compute a signature $(r, s) = Sig(sk, m)$.
3. Compute the first sub-share of x as the point $\alpha = (r, s \bmod q)$.
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 [5])
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 m and the sub-share β.

3.2. Share reconstruction

Given a signing share m 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 $(r, s) = Sig(sk, m)$.
2. Compute a sub-share α as the point $\alpha = (r, s \bmod q)$.
3. Generate the line L by performing Polynomial Interpolation [5] 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: For ECDSA signatures, the value of the parameter s in $(r, s) = Sig(sk, m)$ is already computed modulo q. We use the notation $\alpha = (r, s \bmod q)$ for the sub-share to make it clear (at a glance) that the sub-shares are computed using finite field arithmetic.

4. 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 address pk_i and secret key sk_i as follows:
   - The decentralized identity address pk_i.
   - The signature $\varphi_i = Sig(sk_i, V_i)$.
2. At the beginning of Round 2, for each P_i, verify φ_j from all P_j where $j \neq i$ by checking that the output of $Ver(pk_j, V_j, \varphi_j)$ 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 m_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 pk_i, m_i, β_i and $S_i = \{pk_j : i \neq j\}$ (i.e the set of verified decentralized identities for all other parties).

5. 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 address pk_i and secret key sk_i as follows:
   - The decentralized identity address pk_i.
   - The signature $\varphi_i = Sig(sk_i, V_i)$.
2. At the beginning of Round 2, for each P_i, verify φ_j from all P_j where $j \neq i$ as follows:
   - Verify that $pk_i \in S_j$ or report the culprit and halt.
   - Verify φ_i by checking that the output of $Ver(pk_j, V_j, \varphi_j)$ 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 m_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 m_i with m_i^* and β_i with β_i^*.

6. 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 address pk_i and secret key sk_i, send 2 additional parameters to all P_j where $j \neq i$ as follows:
   - The decentralized identity address pk_i.
   - The signature $\varphi_i = Sig(sk_i, m)$.
3. At the beginning of the Output phase, verify φ_j from all P_j where $j \neq i$ as follows:
   - Verify that $pk_i \in S_j$ or report the culprit and halt.
   - Verify φ_i by checking that the output of $Ver(pk_i, m, \varphi_i)$ is valid or report the culprit and halt.

7. Identity Authenticated Request Initiation and Verification

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

7.1. 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 address pk_i and secret key sk_i:

1. Read the current UTC timestamp t.
2. Compute the signature $\varphi = Sig(sk_i, t|C)$.
3. Broadcast C, pk_i, t and φ.

7.2. Identity Authenticated Request Verification

To verify an identity authenticated request with a command C from a party P_i given its associated decentralized identity address pk_i, a timestamp t, a signature φ and a set of verified decentralized identities for all other parties S_j as input:

1. Verify that $pk_i \in S_j$ 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 φ by checking that the output of $Ver(pk_i, t|C, \varphi)$ is valid or report the culprit and halt.

8. Identity Challenge

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

8.1. Identity Challenge Initiation To issue an identity challenge to a party P_i from all verifying parties P_j where $j \neq i$: 1. Sample a random v_j. 2. Broadcast v_j to all parties, such that all parties can compute $v = \sum_j v_j$ where $j \neq i$.

8.2. Identity Challenge Response For a party P_i with associated decentralized identity secret key sk_i, to respond to an identity challenge given v_j from all parties P_j where $j \neq i$:

1. Compute $v = \sum_j v_j$ where $j \neq i$.
2. Compute the signature $\psi = Sig(sk_i, v)$.
3. Broadcast ψ to all verifying parties P_j.

8.3. Identity Challenge Verification To verify an identity challenge response from a party P_i given its associated decentralized identity address pk_i, a signature ψ and v_j from all verifying parties P_j where $j \neq i$ as input:

1. Compute $v = \sum_j v_j$ where $j \neq i$.
2. Verify ψ by checking that the output of $Ver(pk_i, v, \psi)$ is valid or report the culprit and halt.

9. 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 address pk_i and secret key sk_i to a decentralized identity I_i^* with address pk_i^* and secret key sk_i^* proceeds as follows:

1. For P_i, initiate an "identity-rotation" request by following the protocol in section 7.1.
2. For all P_j where $j \neq i$:
    - Verify the "identity-rotation" request by following the protocol in section 7.2.
    - Initiate an identity challenge for P_i by following the protocol in section 8.1.
3. For Pi, respond to the identity challenge by following the protocol in section 8.2 with the following augmentations:
    - Generate an additional signature $\psi_i^* = Sig(sk_i^*, v)$.
    - Add pk_i^* and ψ_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 8.3.
- Verify that P_i controls the new decentralized identity address pk_i^* as follows:
  - Compute $v = \sum_j v_j$ where $j \neq i$:
  - Verify ψ^* by checking that the output of $Ver(pk_i^*, v, \psi^*)$ is valid or report the culprit and halt.
- Modify Stored State as follows:
  - Create S_i^* by replacing pk_i with pk_i^* in S_i.
  - Replace S_i with S_i^*.
- Send confirmation of successful rotation of the identity to P_i.

5. For P_i, upon receiving confirmation of successful rotation from a quorum of P_j:
   - Compute the new signing share m_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 m_i^* and a new sub-share β_i^* based on the new decentralized identity I_i^*.
   - Modify Stored State as follows:
     - Replace pk_i with pk_i^*.
     - Replace m_i with m_i^*.
     - Replace β_i with β_i^*.

10. Share Addition and Removal

Share addition and removal allows a quorum of verified parties to either issue a secret share to a new party and its associated decentralized identity, or revoke the secret share of any party respectively.

10.1. Share Addition

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

1. For all P_j where $j \neq i$, initiate an identity challenge for P_i by following the protocol in section 7.1.
2. For P_i, respond to the identity challenge by following the protocol in section 7.2.
3. For all P_j where $j \neq i$, verify the identity challenge response from P_i by following the protocol in section 8.3.
4. Follow the key refresh protocol described in section 5 with P_i included as participant if the identity challenge above is passed.

10.2. Share Removal

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

1. Follow the key refresh protocol described in section 5 without P_i.

11. Threshold Modification

Threshold modification allows a quorum of verified parties to change the threshold (i.e. change the size of the quorum).

While threshold modification (or more generally t-out-of-n sharing, and specifically the case where $n > t + 1$) is not formally specified in CGGMP20 [1], it can be derived in a relatively straightforward manner based on GG18 [6] (and GG20 [2]) which CGGMP20 [1] builds upon (see sections 1.2.8, 1.2.1 and 1.2.2 of CGGMP20 [1]). In general, CGGMP20 [1] can be seen as a combination of CMP20 [3] and GG20 [2], and a direct improvement on GG18 [6].

Therefore, threshold modification can be achieved by following the key refresh protocol described in section 3.2 and figure 6 of CGGMP20 [1] and section 5 of this document, with some modifications based on the key generation protocols described in GG18 [6] and GG20 [2], and following the instructions in section 1.2.8 of CGGMP20 [1].

In particular, this entails performing a t-out-of-n Feldman's VSS [7] sharing of the values x_i^k (as defined in section 3.2 of CGGMP20 [1]), with the new threshold t used as the threshold parameter (similarly defined as t) for Feldman's VSS [7] protocol 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 [6]).

NOTE: Similar modifications can be applied to the signing protocol described in section 3.1 and figure 5 of CGGMP20 [1] and section 6 of this document to achieve a t-out-of-n sharing of the secret key for $n \geq t + 1$. In particular, this entails performing a t-out-of-n Feldman's VSS [7] sharing of the value x_i (as defined in section 3.1 of CGGMP20 [1]), based on the same modifications from GG20 [2] and GG18 [6] described above, and following the instructions in section 1.2.8 of CGGMP20 [1].

12. 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 m and sub-share β pair on user-controlled secondary or device-independent storage.

12.1. 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 verification.

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

1. For P_i, Initiate a “share-recovery” request by following the protocol in section 7.1.
2. For all P_j where $j \neq i$:
   - Verify the “share-recovery” request by following the protocol in section 7.2.
   - Initiate an identity challenge for P_i by following the protocol in section 8.1.
3. For P_i, respond to the identity challenge by following the protocol in section 8.2.
4. For all P_j where $j \neq i$, verify the identity challenge response from P_i by following the protocol in section 8.3.
5. Follow the key refresh protocol described in section 5 if all verifications above pass.

12.2. Share recovery with a backup on user-controlled secondary or device-independent storage

12.2.1. 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 m and a sub-share β alone is insufficient to reconstruct the secret share x. This is because a signature of m 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 m 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.

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

[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.

Given a standardized phase k, a key derivation function H, a symmetric encryption algorithm E, this proceeds as follows:

1. Compute the signature $\phi = Sig(sk, k)$.
2. Generate the encryption secret $\varepsilon = H(\phi)$.
3. Compute the ciphertext for the signing share m as $m_c = E_{enc}(m, \varepsilon)$.
4. Compute the ciphertext for the sub-share β as $\beta_c = E_{enc}(\beta, \varepsilon)$.
5. Erase both ϕ and ε from memory.
6. Save m_c and β_c to backup storage.

12.2.3. 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 m and the sub-share β.

Given a standardized phase k, a key derivation function H, a symmetric encryption algorithm E, the ciphertext for the signing share m_c and the ciphertext for the sub-share β_c, this proceeds as follows:

1. Compute the signature $\phi = Sig(sk, k)$.
2. Generate the encryption secret $\varepsilon = H(\phi)$.
3. Compute the signing share $m = E_{dec}(m_c, \varepsilon)$.
4. Compute the sub-share $\beta = E_{dec}(\beta_c, \varepsilon)$.
5. Erase both ϕ and ε from memory.
6. Return the signing share m and the sub-share β.

12.2.4. Further security and usability considerations for share recovery with a backup For further improved security and usability, the signing share m 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 m^* and a sub-share β^* such that $m^* \neq m$, $\beta^* \neq \beta$ 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 m^* 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.

13. Acknowledgements

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

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

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

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

[6] 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://doi.org/10.1145/3243734.3243859.

[7] 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.