MPC in Secure Environment of Hardware Wallet

Ujjwal Kumar
ujjwal@cypherock.com

Amneesh Singh
amneesh@cypherock.com

Vipul Saini
vipul@cypherock.com

Rohan Agarwal
rohan@cypherock.com

August 2023

Abstract

The demand for a lightweight MPC-based Threshold Signature scheme (TSS) has become increasingly urgent to cater to low(computation) power-embedded devices. Existing TSS protocols suffer from computational overheads stemming from complex Zero-Knowledge Proofs (ZKPs) and commitment schemes, making them unsuitable for IoT devices or security cards based on microcontrollers. In this work, we'll introduce our lightweight MPC-based TSS protocols designed specifically for low-power embedded devices. Our primary focus is on integrating the library into Cypherock hardware wallets, enhancing the self-custody infrastructure for Web3 teams. Current TSS solutions predominantly reside on smartphones, PCs, or servers, rendering the party's share vulnerable during distributed key generation and signing processes due to the absence of native TSS algorithms within secure enclaves or Trusted Execution Environments (TEE) of current-generation processors. The risk of loss or theft of party shares further compounds the problem in these solutions. To address these challenges, we propose to implement TSS on Cypherock X1 devices, utilizing a novel architecture that shards the party's individual share further in a hierarchical manner. These shards are then distributed among multiple Java cards, each possessed by the individual party, preventing share loss. Crucially, our library is BIP39 and BIP32 compatible, ensuring the transferability of the shards, and supporting the creation of multiple vaults with any m-of-n scheme. Furthermore, we prioritize privacy by avoiding any utilization or linkage of the party's biometric ID within the architecture.

1 Introduction

This document defines a method to implement MPC processes inside a secure execution environment. The implementation makes use of the pre-established Public Key Infrastructure (PKI) within the hardware ecosystem to establish a

private and authentic secure channel with the parties. Further, we have proposed an extension to the BIP32 protocol with which after the one-time initial setup any individual party would be able to generate verifiable group public keys without other parties being required to come online. All the polynomials required in the signature phase are generated randomly and consumed within the signature phase itself.

2 Background

An ideal MPC solution should be able to fit into the current user behavior of a wallet without any significant impact on how those services are consumed by end-users. In other words, a solution that brings the best of MPC without any major impact on usage behavior.

There are many MPC solutions proposed in the literature, all of which have some overhead/trade-off which makes them practically unusable.

3 Security pillars

The process defined here is aimed to work practically by providing sufficient security and forge-proof functionality. This gives the right balance between usage feasibility and security. This is achieved by the following security pillars:

1. A verifiable secure execution environment for all the processes related to MPC.
2. DKG process from existing literature for key generation.
3. Oblivious Transfer (OT) based MtA (for Multiplicative to Additive) process.

4 Secure Execution Environment

4.1 Verifiability

The execution environment can be verified at any point in time by any other secure execution environment or the end-user. Verification of a secure execution environment is a means to ensure that the processes of MPC are running on software build by a trusted entity.

Verification of the environment is enabled by ECDSA key-pair infused into the hardware of the secure execution environment. The infused private key is non-breachable as it is stored and handled by a secure element.

4.2 Responsibilities

The environment will enforce active safety checks to ensure no important information leaks out of the secure environment. Some active checks are:

1. The environment will provide a mechanism to establish a secure communication channel with hardware of two secure execution environments.

2. Ensure no secret information gets out of the environment in plain text at any step of the process. Such as private-key shares.

5 Multi-Party Computation

5.1 Distributed Key Generation

This phase is used to identify the parties to be involved in generating the group public key or the vault members. This phase shall also be called as Distributed Key Generation (DKG) phase. The setup process leverages the pre-established PKI of the Cypherock X1 device to verify the participating devices. Based on the same PKI, it is possible to establish a pairwise authentic and private channel for exchanging information (refer Appendix A.1). To initiate the process, every party will generate a secret polynomial of their own (refer Appendix A.2). For threshold ECDSA of **t** of **n** parties, each party, here a wallet on Cypherock X1 will generate a polynomial of degree **t**.

$$S_1(x) = a_0x^0 + a_1x^1 + ... + a_tx^t$$

$$S_n(x) = n_0x^0 + n_1x^1 + ... + n_tx^t$$

Here, $S_i(x)$ represents the secret polynomial generated by the individual 'n' parties. In the next step, the parties will generate shares for each party of their individual polynomial (see below) which will be exchanged over the pairwise authentic and private channel.

$$S_i(x = 1, ..., n) = a_0x^0 + a_1x^1 + ... + a_tx^t$$

Each party upon receiving all their n-shares from the n individual parties, the parties perform arithmetic addition of the shares to generate a sharing of the shared secret polynomial.

$$S(j = 1, ..., n) = \sum_{i=1}^{n} S_i(j)$$

Every participant will have to broadcast their individual share of the Elliptic curve point (in other words the sharing of the public key) calculated as shown below:

$$Q_i = S(i) \cdot G$$

Once the broadcast is complete, each participant can run a check by interpolating in the exponent to verify each individual sharing of the public key.

$$Q_j = ExpInt(Q_1, Q_2, ..., Q_{t+1}; j)$$

$$Q = \lambda_1 \cdot Q_1 + \lambda_2 \cdot Q_2 + ... + \lambda_{t+1} \cdot Q_{t+1}$$

here, ('+' denotes EC addition) and

$$\lambda_i = \prod_{1<m<t+1, m \neq i} \frac{m-j}{m-i}$$

If the check passes, then the group public key is generated by interpolating in the exponent as follows:

$$Q = ExpInt(Q_1, Q_2, ..., Q_{t+1}; 0)$$

5.2 Extending BIP32

In Bitcoin protocol, a transaction structure may contain multiple inputs(utilizing UTXOs) and outputs where one or more outputs may belong to the sender wallet itself, i.e. output may contain a change address derived at a specific change node [2]. If the output does not contain a change address then all the balance amount gets consumed within the miner's fee which might not be the intent of the sender. While constructing the unsigned transaction structure for a single signature scheme, the wallet may simply derive change addresses from the BIP39 seed or use the extended public keys [5] at account node [1] level. But in threshold signature schemes where the number of parties signing the transaction is less than the total number of parties that were involved in the key generation phase, deriving new change addresses in the signing phase remains a challenge.

For threshold ECDSA we propose a mechanism wherein any party would have the capability to derive new group addresses(from group public keys) for receiving funds or sending the funds to change addresses as required while constructing the unsigned transaction, before the signing generation phase. Any threshold number of parties would be able to generate signatures to move funds from such addresses. To do so we extend the initial distributed key generation phase as follows.

1. Parties generate t+1 number of sub-master nodes using hardened key derivation [7] from the master node of their individual wallet. Corresponding to each sub-master node parties derive the private key at the (let index = 0)account node [1] level and (let index = 0)change node [2] level.

2. Parties perform the DKG process to generate the first group public key where the coefficients of the secret polynomial are equal to the private key at the account node level(at index = 0) for each t+1 number of the sub-master node of their individual wallet.

$$P = ExpInt(P_1, P_2, ..., P_{t+1}; 0)$$

3. Parties perform the DKG process to generate the second group public key where the coefficients of the secret polynomial are equal to the private key at the change node level(at index = 0) for each t+1 number of sub-master node.

$$Q = ExpInt(Q_1, Q_2, ..., Q_{t+1}; 0)$$

4. Each party derives and shares the public key P_i and chain code C_i at the account node level(at index = 0) from their individual wallet.

5. Parties derive the public keys Q_i at change node level(at index = 0) of all other party's wallet using the shared public key and chain code following the BIP32 public key derivation [8] approach.

$$Q_i = P_i + HMAC(C_i, (P_i, index))_L$$

$HMAC()_L$ implies left most 256-bits of HMAC-SHA512.

6. Parties verify that point addition of the derived public key at the change node level equals the second group public key.

$$Q = Q_1 + Q_2 + ... + Q_n \quad [Check]$$

7. If the above condition is satisfied parties can safely use the shared individual public key and chain code to derive the change or receive addresses from group public keys at any level below the account node.

5.3 Signature Generation

For threshold ECDSA of **t** of **n** parties, each party, here a Cypherock X1 wallet will generate four polynomials of degree **t** to represent K, A, D and E in the group setting of $n = t+1$ where 't' is as defined in the distributed key generation phase. 'A' will be used to blind 'K' when performing the beaver inversion trick. 'K' is used to generate the same random nonce 'k' as used in the ECDSA signature generation process. 'D' and 'E' will be used to blind the message to be signed. Further, 'D' and 'E' polynomials hold the sharing of '0' which means $D(0) = E(0) = 0$.

In the first stage, participants will follow the same process as explained in the key generation phase to generate the first component of the signature, R. We assume the same check was passed and the value of R is consistent among the parties.

$$R_i = K(i).G$$

$$R = ExpInt(R_1, R_2, ..., R_{t+1}; 0)$$

$$r = R_x$$

At the same time, parties will be generating shares of their individual polynomial for each party (see below representation for polynomials of one such party) which will be exchanged over the pairwise authentic and private channels.

$$K_i(x = 1, ..., t+1) = k_0x^0 + k_1x^1 + ... + k_tx^t$$

$$A_i(x = 1, ..., t+1) = a_0x^0 + a_1x^1 + ... + a_tx^t$$

$$D_i(x = 1, ..., t+1) = d_0x^0 + d_1x^1 + ... + d_tx^t$$

$$E_i(x = 1, ..., t+1) = e_0x^0 + e_1x^1 + ... + e_tx^t$$

Each party will now calculate an authenticator $W = (A * k).G$ as follows:

$$W_i = A(i).R$$

$$W = ExpInt(W_1, W_2, ..., W_{t+1}; 0)$$

For calculating $w = A * k$, every group of parties performs the operations as follows:

$$w = A(0) * k$$

$$w = (\sum_{i=1}^{t+1} A_i(0)) * (\sum_{i=1}^{t+1} K_i(0))$$

$$w = \sum_{i=1,j=1}^{t+1} A_i(0) * K_j(0)$$

$$w = \sum_{i=1}^{t+1} A_i(0) * K_i(0) + \sum_{i=1,j=1;i\neq j}^{t+1} A_i(0) * K_j(0)$$

To calculate $\sum_{i=1,j=1;i\neq j}^{t+1} A_i(0)*K_j(0)$, the two parties will perform a Correlated Oblivious Transfer (refer Appendix A.3) [3] [6] to produce their respective additive shares of each multiplicative term. So $A_i(0) * K_j(0)$ will result into $U1_{ij}, U2_{ij}$ (we will generally refer these as U_{ij}).

At this point, every party will broadcast a sum of their additive terms U_i, where

$$U_i = \sum_{j=1}^{t+1} U_{ij}$$

$$w = \sum_{i=1}^{t+1} A_i(0) * K_i(0) + \sum_{i=1}^{t+1} Ui$$

$$w.G = W \ [check]$$

This enables every party to generate $w = A*k$. The threshold parties should agree on the message to be signed and denoted by M. Now parties will calculate the sharing of k^{-1} as ,

$$k_i^{-1} = A_(i) * w^{-1}$$

Now for calculating the sharing $v = k^{-1} * X$, where X is the group private key, every group of parties performs the following operations.

$$v = k^{-1} * X$$

$$v = (\sum_{i=1}^{t+1} K_i(0)) * Int(x_1, x_2, ..., x_{t+1}; 0)$$

$$v = (\sum_{i=1}^{t+1} K_i(0)) * (\lambda_1 x_1 + \lambda_2 x_2 + ... + \lambda_{t+1} x_{t+1})$$

here,

$$\lambda_i = \prod_{1<m<t+1, m\neq i} \frac{m-j}{m-i}; j = 0$$

now,

$$v = \sum_{i=1,j=1}^{t+1} K_i(0) * \lambda_j x_j$$

$$v = \sum_{i=1}^{t+1} K_i(0) * \lambda_i x_i + \sum_{i=1,j=1;i\neq j}^{t+1} k_i^{-1}(0) * \lambda_j x_j$$

To calculate $\sum_{i=1,j=1;i\neq j}^{t+1} K_i(0) * \lambda_j x_j$, the two parties will perform a Correlated Oblivious Transfer (refer Appendix A.3) [3] [6] to produce their respective additive shares of each multiplicative term. So $K_i(0) * \lambda_j x_j$ will result into $U1_{ij}, U2_{ij}$ (we will generally refer these as U_{ij}).

$$U_i = \sum_{j=1}^{t+1} U_{ij}$$

$$v_i = K_i(0) * \lambda_i x_i + Ui$$

With v_i and k_i^{-1} parties can now calculate the sharing of the second component of the signature s as follows:

$$s_i = k_i^{-1} * (Hash(M)) + (Hash(M)) * D(i) + E(i)$$

Each party will broadcast $(v_i * r + \lambda_i * s_i)$. Now to generate s, any party may calculate

$$s = \sum_{i=1}^{t+1} (v_i * r + \lambda_i * s_i)$$

here,

$$\lambda_i = \prod_{1<m<t+1, m\neq i} \frac{m-j}{m-i}; j = 0$$

and hence we get the signature components r and s.

References

[1] *Account Node (Multi-Account Hierarchy for Deterministic Wallets)*. https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki#account. Accessed: September 12, 2023.

[2] *Change Node*. https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki#change. Accessed: September 12, 2023.

[3] Jack Doerner et al. *Threshold ECDSA from ECDSA Assumptions: The Multiparty Case*. Cryptology ePrint Archive, Paper 2019/523. https://eprint.iacr.org/2019/523. 2019. DOI: 10.1109/SP.2019.00024. URL: https://eprint.iacr.org/2019/523.

[4] *ECDH (Elliptic-curve Diffie–Hellman)*. https://en.wikipedia.org/wiki/Elliptic-curve_DiffieHellman. Accessed: September 12, 2023.

[5] *Extended Keys (Hierarchical Deterministic Wallets)*. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki#extended-keys. Accessed: September 12, 2023.

[6] Marcel Keller, Emmanuela Orsini, and Peter Scholl. *MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer*. Cryptology ePrint Archive, Paper 2016/505. https://eprint.iacr.org/2016/505. 2016. DOI: 10.1145/2976749.2978357. URL: https://eprint.iacr.org/2016/505.

[7] *Private CKD (Hierarchical Deterministic Wallets)*. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki#private-parent-key--private-child-key. Accessed: September 12, 2023.

[8] *Public CKD (Hierarchical Deterministic Wallets)*. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki#public-parent-key--public-child-key. Accessed: September 12, 2023.

A Appendix

A.1 Secret channel generation

The group will be able to establish a pairwise private and authentic channel by following the Elliptic Curve Diffie Hellman [4] scheme. The domain parameters (that is, p, a, b, G, n, h) of the ECDH scheme will be pre-established by the verified firmware running on the individual member's device. Suppose Alice wants to establish a shared key with Bob, but the only channel available for them may be eavesdropped by a third party. Also, each party must have a key pair suitable for elliptic curve cryptography, consisting of a private key d (a randomly selected integer in the interval $[1, n-1]$) and a public key represented by a point Q (where $Q = d \cdot G$, that is, the result of adding G to itself d times). Let Alice's key pair be d_A, Q_A and Bob's key pair be d_B, Q_B. Each party must know the other party's public key prior to the execution of the protocol. This will be enabled by the pre-established shared root Xpubs during the provisioning of individual X1 Wallet during production.

Alice computes point $x_k, y_k) = d_A \cdot Q_B$. Bob computes point $x_k, y_k) = d_B \cdot Q_A$. The shared secret is x_k (the x coordinate of the point). Most standardized protocols based on ECDH derive a symmetric key from x_k using some hash-based key derivation function.

The shared secret calculated by both parties is equal, because $d_A \cdot Q_B = d_A \cdot d_B \cdot G = d_B \cdot d_A \cdot G = d_B \cdot Q_A$.

A.2 Polynomial coefficients

A polynomial of degree t will be represented by t+1 number of coefficients where the coefficient of the highest degree term is non-zero. Coefficients of the polynomial are derived as the BIP-32-derived private keys. A derivation path (eg: master/sub-master'/purpose'/coin-type'/account'/change/address-index) shall be used as the derivation, followed by each hardware wallet owned by the party.

A.3 Oblivious Transfer (OT)

A.3.1 Base OT

For a single base OT, Alice has two inputs m_0 and m_1 and Bob has an input bit c and Bob receives m_c from Alice without learning about the other message. We do this using a basic Diffie-Hellman exchange.
Alice and Bob agree upon a curve point G and a prime number p. All calculations are to take place under this prime p.
Alice generates a random scalar a, point multiplies it with G and sends it to Bob as point A

$$A = a.G$$

Bob generates a random scalar b, point multiplies it with G, adds the point A

only if the bit c is 1, and sends it back to Alice as point B
if c = 0

$$B = b.G$$

if c = 1

$$B = (b.G) + A$$

Alice prepares two keys for encryption, k_0 and k_1
k_0 : Alice point multiplies a with point B. The abscissa of this point is k0.

$$k_0 = (a.B)_x$$

k_1 : Alice subtracts point A from point B and then point multiplies a with B. The abscissa of this point is k_1.

$$k_1 = (a.(B - A))_x$$

Alice encrypts m_0 with k_0 and m_1 with k_1 and sends them to Bob as $e_0 and e_1$ respectively.
Bob generates a decryption key k_c by point multiplying b with A and taking the abscissa.

$$k_c = (b.A)_x$$

Bob decrypts e_c, where c is the input bit, with k_c. This is the input messages m_c

A.3.2 Correlated OT (COT)

This part adds a correlation D between m_0 and m_1. In our case m_1 would be $m_0 + D$

$$m_1 = m_0 + D$$

This means that Bob's m_c is m_0 if $c = 0$ or $m_0 + D$ if $c = 1$ ergo

$$m_c = m_0 + c * D$$

A.3.3 Implementation

Now we use the above correlated OT to multiply two terms and get their respective additive shares. Note that we are going to assume that all calculations occur under the previously agreed upon p.
Alice's multiplicative term = x
Bob's multiplicative term = y
Alice's additive share = U
Bob's additive share = V

Mathematically,

$$x * y = U + V$$

Since one OT involves one bit, and say numbers x and y are k bit long, we would require k OT_s

The process for every OT is going to be the same as the standard Diffie-Hellman exchange but with the following changes -

For i_{th} OT
m_0 is a random scalar U_i
The correlation between m_0 and m_1 is going to be x

$$m_0 = U_i$$

$$m_1 = U_i + x$$

From Bob's side the chosen bit c is going to be the i_{th} bit of y

$$c = y_i$$

Subsequently, as noted above

$$m_c = U_i + y_i * x$$

To calculate the additive share U, we calculate the negative weighted sum of U_i with powers of 2:

$$U = -\sum_{i=1}^{k}(2^i * U_i)$$

To calculate the additive share V, we calculate the weighted sum of i_{th} m_c value with powers of 2:

$$V = \sum_{i=1}^{k}(2^i * m_{ci})$$

Upon expanding we can see this is nothing but

$$V = \sum_{i=1}^{k}(2^i * (U_i + y_i * x))$$

$$V = \sum_{i=1}^{k}(2^i * U_i) + x * \sum_{i=1}^{k}(2^i * y_i)$$

$$V = -U + x * \sum_{i=1}^{k}(2^i * y_i)$$

$$V = -U + x * y$$

Thus additive shares for Alice and Bob are obtained.