CIRCLE RESEARCH

# Robust Threshold ECDSA Signatures

## Identifying Misbehaving Signers in Real-Time

# Contents

01

# Abstract


## Identifying misbehaving participants in a multi-party computation is better done sooner than later.

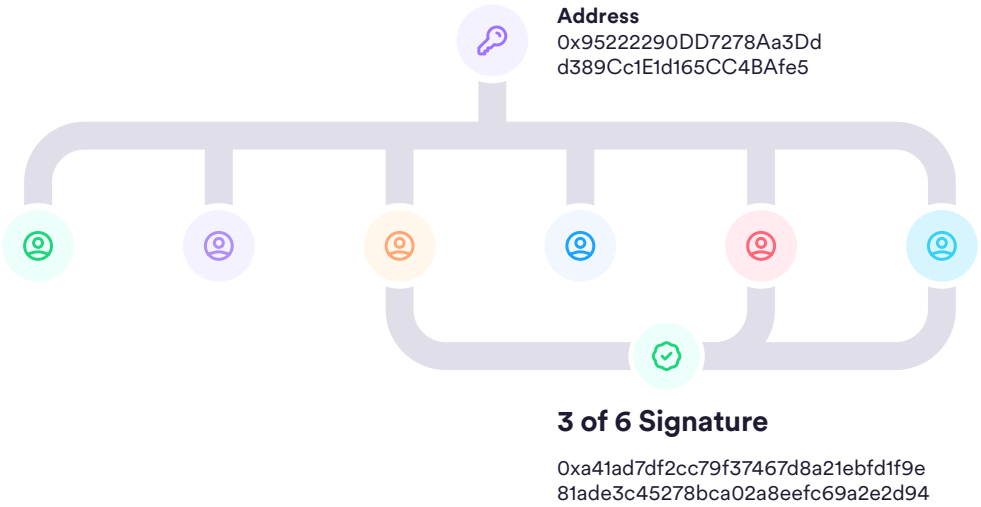
We show how to modify the CGG+21 threshold signature scheme to identify misbehaving signers in-real time. We move the zero-knowledge proofs of correct behavior from the end of the protocol (the traitor-tracing phase) and into the main body of the protocol. The total number of rounds stays the same. Online protocols can proceed in four rounds of communication. Offline protocols require 3 rounds of pre-computation (which can be done in bulk before the message is known) and one round to sign the message. Our protocol requires every message to be broadcast to all participants (echo broadcast is acceptable). In scenarios of 3-of-n signers, the entire message transcript is 141.3 KB. Misbehavior detection may be outsourced to a central authority that verifies the public transcript.

# Introduction

High stakes transactions often require multiple parties to cosign. While many newer blockchains offer native multi-signature support, this is not the case for some older and more established blockchains such as Ethereum.

Threshold signatures allow multiple parties to split the signing key corresponding to a single address. Even in situations where multiple signatures are directly supported, a threshold signature scheme offers more privacy to the address owners because it reveals no information about the key holders. Since threshold signatures are inherently multi-step off-chain protocols, we need a way to hold any cheating signers accountable. This paper modifies the CGG+21 ECDSA threshold signature scheme to identify cheating signers right away instead of after-the-fact.

CGG+21 optimizes the length of messages during signing. If something goes wrong, the participants initiate a traitor-tracing algorithm. They broadcast zero-knowledge proofs that their prior broadcasts were correctly formed. We move these proofs into the main body of the protocol. Participants prove that their message is correctly formed during the round itself. The result is a slower and more expensive protocol, but one that is sufficiently efficient for infrequent transactions and cold storage. The benefit is that misbehaving users are identified immediately.



A $(t,n)$-threshold signature scheme lets any $t + 1 \leq n$ participants sign messages. It supports several protocols:

1. **Key Generation.** The $n$ participants work together to jointly create a public key. This public key looks like an ordinary public key to a signature scheme. Each participant gets as output a share of the secret signing key.
2. **Key Refresh.** A subset of $t + 1$ participants work together to generate new shares of the public key. They may choose to exclude existing members and/or add new members to the group of signers.
3. **Signing.** A subset of $t + 1$ participants work together to generate a signature on a message. The participants do not learn any information about each other's share of the secret.
4. **Verification.** This is the standard signature verification function used in an ordinary signature scheme.

The multi-party key generation, key refresh, and message signing protocols can fail if one of the parties misbehaves. The participants may be forced to abort early or end with invalid output. Identification mechanisms allow participants to determine which participant(s) misbehaved. **Robust** protocols require participants to prove that their output is correct at every step, while **optimistic** protocols reduce message length in each round by performing traitor-tracing only if something goes wrong.

Proactive protocols are especially important for offline scenarios such as cold storage. The risks involved are much higher as the amounts of funds at stake can be in the billions of dollars. The signers are often highly trusted individuals who have access to other sensitive resources in their organizations. At the same time, signing takes place in remote locations on offlined computers. Every round of communication requires humans to physically transfer messages. There is very little visibility as to what the offline computer is actually doing. Thus, while the risks are higher, the ability to detect misbehavior is much lower. Also, in terms of efficiency, the time to generate a signature depends more on the number of rounds of communication than the time/space used for individual messages. For all these reasons, a low-round robust signature scheme makes more sense than a low-bandwidth signature scheme with a large number of rounds or a reactive traitor tracing protocol.

GG18 is one of the most widely used ECDSA threshold signature schemes due to its widespread implementation in open source libraries, including by Binance, 0x EigenLabs, and ZenGo. However, developers are moving to the updated CGG+21 protocol (sometimes referred to as CG20, CG21, and CMP) due to a series of vulnerabilities found in GG18. Apache Milagro-MPC library and SafeHeron Multi-Party ECDSA libraries implement CGG+21. See Bibliography for list of open-source implementations.

Both GG18 and CGG+21 abort if something goes wrong. CGG+21 also has the option of trying to identify which participant misbehaved by asking participants to publish zero-knowledge proofs that their prior messages were correctly formed. This step is called the traitor-tracing round. It happens after the protocol completes. The authors state that it is possible to publish these zero-knowledge proofs during the protocol, but decline to do so due to the bandwidth cost. We believe that the cost is not prohibitive for many applications. This document shows developers how to make the necessary modifications.

## Efficiency

The robust protocols are sufficiently efficient for situations where maximizing throughput is not the goal. Using 2048-bit Paillier modulus and a 256-bit ECDSA group, the message size for each of the $t + 1$ participants is:

$$t^2 \cdot (20m + 10k) \; + \; t \cdot (36m + 14k) \; + \; 28m \; + \; 4k$$

Using standard parameters of a 2048-bit Paillier modulus and 256-bit elliptic curves results in a message size of $5.3 \cdot t^2 \; + \; 9.4 \cdot t \; + \; 7.1\,KB$. The table below shows transcript sizes for some common policies.

| Signers | Individual Transcript Size | Total Transcript Size |
|---|---|---|
| 2 signers ($t = 1$) | 21.8 *KB* | 43.6 *KB* |
| 3 signers | 47.1 *KB* | 141.3 *KB* |
| 4 signers | 83.0 *KB* | 332.0 *KB* |
| 5 signers | 129.5 *KB* | 647.5 *KB* |
| 6 signers | 186.6 *KB* | 1.1 *MB* |
| 7 signers | 254.3 *MB* | 1.8 *MB* |

## Communication Rounds

The CGG+21 paper states that its protocol consists of 3 rounds of pre-signing and one round of signing. In case of an online protocol, the entire process completes in 4 rounds. However, the analysis does not quite apply to a cold storage scenario.

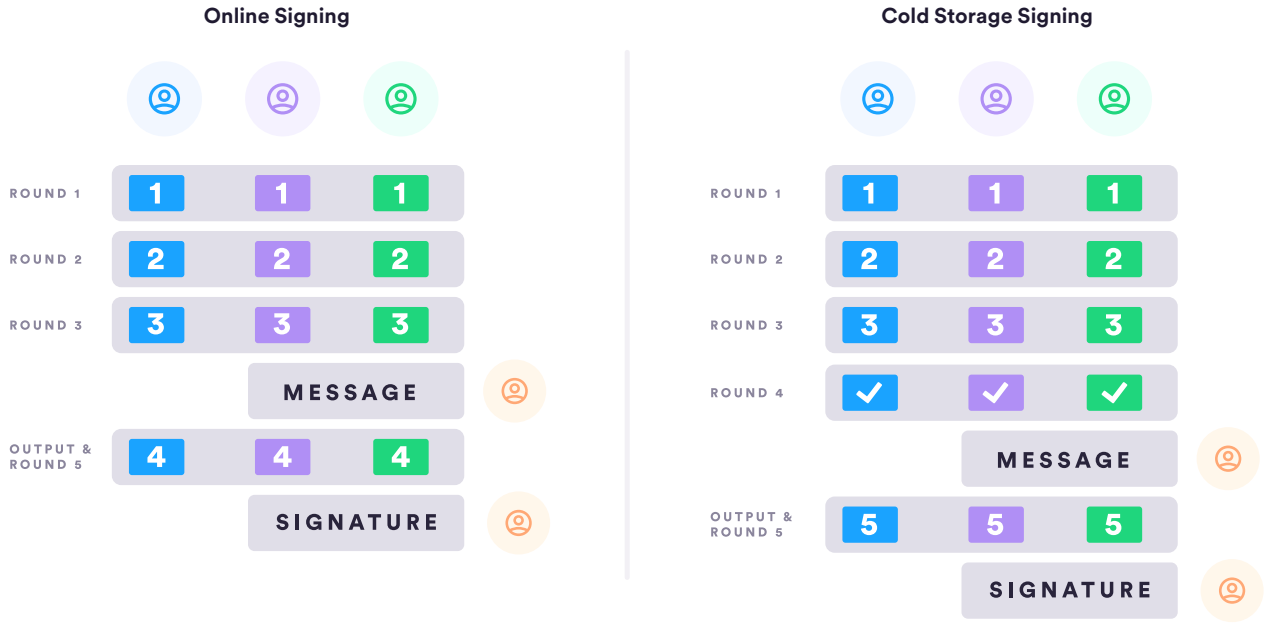
Pre-Signing

- Round 1: all participants broadcast their output.
- Round 2: all participants process output from round 1 and broadcast new output.
- Round 3: all participants process output from round 2 and broadcast new output.
- Output: all participants process output from round 3 and output that everything is ok.

Signing

- Round 5: all participants get the message string and broadcast new output.
- Output: all participants process output from round 1 and compute the signature. Technically, any 3rd party that sees the output of round 1 could compute the signature, but only the actual participants can determine who misbehaved.

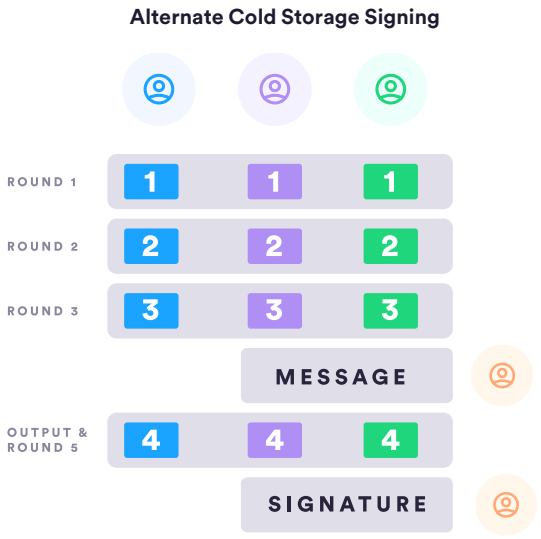
The CGG+21 authors do not count the final output rounds, thus getting three rounds of pre-signing and one round for signing. In some applications such as cold storage, the output rounds would require an extra visit to the cold storage facility.




**Online Signing.** The blue, red, and green participants perform 3 rounds of pre-signing. Then the orange user broadcasts a message to sign. The blue, red, and green participants combine the pre-signing output round with round 5 to broadcast some output.

**Cold Storage Signing.** The blue, red, and green participants perform 4 rounds of pre-signing, because the output round also requires a trip to the cold storage facility to deliver 3 3 3 and compute the acknowledgement ✓ ✓ ✓ which remains in cold storage. Later, the orange participant sends the message to cold storage for round 5. The orange participant is able to assemble the output of round 5 into a signature. A follow-up trip to cold storage is needed only if something goes wrong.

**Alternative Cold Storage Signing.** The pre-signing protocol can pause after round 3. The round 3 output can be delivered to signers with the message. Then the pre-signing output round and round 5 can be combined into one round. A follow-up trip to cold storage is needed only if something goes wrong.



The table below compares this work to GG18, the robust version of GG20, and the most round-efficient optimistic CGG+21 protocol, for a (t,n)-threshold signature. We split the rounds of signing into two categories: the pre-signing stage is a series of precomputation that can proceed without knowing the message, while the signing stage requires the message. For consistency, we use the same analysis of 3 rounds pre-signing and 1 round signing for our work as that is how the authors of GG18, GG20, and CGG+21 counted rounds.

| | GG18 | GG20 | CGG+21 | This work |
|---|---|---|---|---|
| Rounds pre-signing | 4 | 6 | 3 | 3 |
| Rounds signing | 5 | 1 | 1 | 1 |
| Traitor tracing | None | After | After | During |
| Rounds for traitor tracing | n/a | O(1) | $O(t^2)$ | 0 |
| Message length | Low | Low | Low | High |

## Implementation

We built an open-source Golang implementation of robust CGG+21. We added it as a new module for the popular Binance tss-lib, which implements GG18. The two threshold signature algorithms are compatible. Existing deployments can continue to use the same keys that they generated for GG18 to sign with our CGG+21 code. They can also take advantage of the existing tss-lib key refresh and key derivation algorithms.

# Robust CGG+21

This section goes through the CGG+21 threshold signature protocol and explains how to modify it to ensure robustness at every step.

This document only covers message signing. The GG18/CGG+21 key generation protocols and the Binance tss-lib key refresh protocols are compatible with our signing implementation. Their key generation and key refresh protocols already have zero-knowledge proofs of correctness at every step. All of the zero-knowledge proofs are in the CGG+21 paper. The one exception is $\Pi^{aff\text{-}g\text{-}inv}$, which we include here.

## Basic functions

We review ECDSA and Paillier homomorphic encryption. This is meant to be a brief overview to make the rest of the document easier to understand; developers should refer to the actual ECDSA standard and the Paillier article when implementing these protocols.

**ECDSA signature [FIPS 186-4].** The ECDSA signature has public parameters (*G*,*g*,*q*), where G is an elliptic curve and *g* is a point on the elliptic curve that generates a subgroup of prime order *q*.

1. $(x, y) \leftarrow KeyGen(G, g, q)$ outputs a secret key $x \leftarrow Z_q$ and a public key $y = g^x$.
2. $(r, s) \leftarrow Sign(G, g, q, x, msg)$ outputs a signature $(r, s)$. Briefly, the signer chooses $k \leftarrow Z_q$, and computes $R = g^k$ and $r = H'(R)$ for a specific ECDSA hash function $H'$. Then the signer computes $s = k^{-1}(msg + rx) \bmod q$.
3. $true \mid false \leftarrow Verify(G, g, q, y, msg, r, s)$ outputs true if $(r, s)$ is a valid signature on $msg$. The verifier computes $R = g^{msg \cdot s^{-1} \bmod q} \cdot y^{r \cdot s^{-1} \bmod q}$ and checks if $r = H'(R)$.

**Paillier homomorphic encryption [Paillier99].** This encryption scheme allows us to add and multiply ciphertexts, thus allowing more efficient multi-party protocols and zero-knowledge proofs over arithmetic expressions. We briefly describe the encryption scheme, while omitting some implementation details that have no bearing on our protocols.

1. $(pk,\ sk) \leftarrow KeyGen(1^k)$ chooses two $k$-bit primes $(p, q)$. The private key is a pair of values $(\lambda, \nu)$ that are deterministically precomputed from $(p, q)$. The public key is an RSA modulus $N = pq$.
2. $ctext \leftarrow Encrypt(pk, msg)$ chooses a random $\rho \in Z_N$ and outputs $ctext = (1 + N)^{msg} \cdot \rho^N \bmod N^2$.

3. $(msg, \rho) \leftarrow Decrypt(sk, ctext)$. Computes the message and randomness $\rho$ used to encrypt it. The full decryption algorithm is in Section 5 of Paillier99.
4. $sum \leftarrow A \oplus B$ computes $sum = A \cdot B \bmod N^2$ to get a ciphertext such that $Decrypt(sk, sum) = Decrypt(sk, A) + Decrypt(sk, B)$.
5. $product \leftarrow A \otimes b$ computes $product = A^b \bmod N^2$ to get a ciphertext such that $Decrypt(sk, product) = b \cdot Decrypt(sk, A)$.

For convenience, we use the notation $A \oplus B$ to mean adding two ciphertexts and $A \oplus b$ to mean multiplying ciphertext $A$ by a scalar value $b$. The operations $\oplus$ and $\otimes$ can be done knowing only the public key.

## Parameters

For the protocol to work, the Paillier encryption scheme must use a modulus significantly larger than the elliptic curve parameter $q$. The Binance tss-lib library uses the secp256k curve with $|q|$=256 bits and $|N|$=2048 bits.

For developers who choose their own parameters, it is important to ensure that the elliptic curve is sufficiently smaller than the Paillier modulus. The participants need to perform multiple homomorphic operations on ciphertexts, and the plaintext cannot become greater than the size of the Paillier modulus. The limit below is derived from the operation in Round 5, step 4: $q^2(2t + 1) + q < N$

## Zero knowledge proofs

This table lists the CGG+21 zero-knowledge proofs we use, where to find them, and their sizes.

| Name | Location | Size | Size with $m$ = 2048-bit Pailler modulus, $k$ = 256-bit ECDSA |
|---|---|---|---|
| $\prod^{mul}$ | Appendix C.6 Figure 29 | $9m$ | 2.3 KB |
| $\prod^{mul*}$ | Appendix C.6 Figure 31 | $5m + 3k$ | 1.3 KB |
| $\prod^{log*}$ | Appendix C.2 Figure 25 | $5m + 3k$ | 1.3 KB |
| $\prod^{dec}$ | Appendix C.6 Figure 30 | $6m + 3k$ | 1.6 KB |
| $\prod^{enc}$ | Section 6.1 Figure 14 | $6m + 2k$ | 1.6 KB |
| $\prod^{aff\text{-}p}$ | Appendix C.3 Figure 26 | $13m + 4k$ | 3.4 KB |
| $\prod^{aff\text{-}g}$ | Section 6.2 Figure 15 | $10m + 5k$ | 2.7 KB |
| $\prod^{aff\text{-}g\text{-}inv}$ | Modified Figure 15 | $10m + 5k$ | 2.7 KB |

Developer Notes:

1. The $\prod^{log*}$ protocol proves that some ciphertext $C = Encrypt(x)$ and a point $X = g^x$. The proof should allow the prover to supply an arbitrary points g as part of the statement, and not just default to the base point of the elliptic curve.

2. The CGG+21 protocol recommends using the zero-knowledge proof $\prod^{aff\text{-}g}$ during round 2 of pre-signing. We found during implementation that we actually needed to make a slight modification to $\prod^{aff\text{-}g}$ to make the protocol work. We provide our variation later in the document for the benefit of other developers. We call this variation $\prod^{aff\text{-}g\text{-}inv}$.

## Setup

A (*t,n*)-threshold signature scheme requires at least $t + 1$ participants to sign a message; any group of $t$ or less participants cannot sign. CGG+21 uses Shamir secret sharing to distribute shares of the secret key $x$ to $n$ participants. When it comes time to sign, a committee of $t + 1$ participants converts their Shamir shares $x_i$ into additive shares $w_i$ such that $x = \Sigma w_i$. The public values $X_i = g^{x_i}$ are similarly converted into a public vector of values $W_i = g^{w_i}$. Each participant can independently perform the conversion for his share and the public values using standard techniques for dealing with Shamir shares. Thus, at the beginning of the signing protocol, every participating party $P_i$ has a secret $w_i$, and everyone knows a public vector of values $W_0,W_1,...,W_t$ such that $W_i = g^{w_i}$. The goal is to compute an ECDSA signature (*r,s*) on common input message *m*.

| Public input | Private input |
|---|---|
| *m* message to sign | |
| *g,y* ECDSA public key | |
| $W_0,W_1,...,W_t$ | $w_i$ *s.t.* $W_i = g^{w_i}$ |
| $pk_0,pk_1,...,pk_t$ Paillier encryption keys | $sk_i$ Pailler secret key |
| $rp_0,rp_1,...,rp_t$ where each $rp_i=(h_1,h_2,\tilde{N})$ is a Ring Pedersen parameter generated by party $P_i$. Used for verifying designated-verifier zero-knowledge proofs. | |

The following protocol is based on Figure 7 of CGG+21. We move some of the zero-knowledge proofs from the traitor-tracing steps to the actual round where the values being verified were initially broadcast. In case a zero-knowledge proof fails to verify, the participants stop and publish the identity of the culprit. For security, it is important to NOT publish a signature unless all proofs verify.

## Signing protocol

We describe each round of the actual signing protocol. All non-proof values in our protocol must be broadcast. Participants **may** send proofs about the values via broadcast or peer-to-peer to the designated verifier. An auditor **may** monitor the entire transcript and verify all proofs using the public verifier parameters.

## Round 1

This round is identical to CGG+21.

1. $P_i$ selects a random pair of values $k_i, \gamma_i \in Z_q$ and computes $K_i = Encrypt(k_i)$ and $G_i = Encrypt(\gamma_i)$ under its own public key $pk_i$.
2. $P_i$ creates for each participant $P_j$ a designated-verifier $\Pi^{enc}$ proof $\psi_{i,j}$ that $P_i$ knows the decryption of $K_i$.

Each participant broadcasts $K_i, G_i, \{\psi_{i,j}\}_{j \neq i}$. Note: the participant will prove $G_i$ is formed correctly in Round 2.

| Name | Size | Size with *m* = 2048-bit Pailler modulus, *k* = 256-bit ECDSA |
|---|---|---|
| $K_i$ | *2m* | |
| $G_i$ | *2m* | |
| $\{\psi_{i,j}\}_{j \neq i}$ | *t • (6m + 2k)* | |
| **Total** | ***4m + t • (6m + 2k)*** | ***1.5 • t + 1 KB*** |

## Round 2

This round modifies CGG+21 to use the $\prod^{aff\text{-}g\text{-}inv}$ proof.

1. $P_i$ verifies each proof $\psi_{j,i}$ that it receives. If there is a problem, terminate and publish all culprits.
2. $P_i$ sets $\Gamma_i = g^{\gamma_i}$ and then prepares a message for every participant $P_j$:
   a. Choose $\beta_{i,j}, \hat{\beta}_{i,j} \leftarrow Z_q$.
   b. $D_{j,i} = (\gamma_i \otimes K_j) \oplus Encrypt(q - \beta_{i,j})$ and $F_{j,i} = Encrypt(\beta_{i,j})$
   c. $\hat{D}_{j,i} = (\gamma_i \otimes K_j) \oplus Encrypt(q - \beta_{i,j})$ and $\hat{F}_{j,i} = Encrypt(\hat{\beta}_{i,j})$
   d. $P_i$ creates for each participant $P_k$ a designated-verifier $\Pi^{aff-g-inv}$ proof $\psi_{i,j,k}$ for the statement $(q, g, G, N_j, N_i, \Gamma_i, F_{j,i}, K_i, D_{j,i})$.
   e. $P_i$ creates for each participant $P_k$ a designated-verifier $\Pi^{aff-g-inv}$ proof $\hat{\psi}_{i,j,k}$ for the statement $(q, g, G, N_j, N_i, \Gamma_i, \hat{F}_{j,i}, K_i, \hat{D}_{j,i})$
   f. $P_i$ creates for each participant $P_j$ a designated-verifier $\Pi^{log*}$ proof $\psi'_{i,j}$ for the statement $(q, g, G, N_i, G_i, \Gamma_i)$, showing that $\Gamma_i = g^{\gamma_i}$ and $G_i = Encrypt(\gamma_i)$.

Each participants broadcasts $\Gamma_i$, $\left\{D_{j,i}, \hat{D}_{j,i}, F_{j,i}, \hat{F}_{j,i}\right\}_{j \neq i}$, $\left\{\psi_{i,j,k}, \hat{\psi}_{i,j,k}\right\}_{j,k \neq i}$, $\{\psi'_{i,j}\}_{j,\neq i}$

| Name | Size | Size with $m$ = 2048-bit Pailler modulus, $k$ = 256-bit ECDSA |
|---|---|---|
| $\Gamma_i$ | $2m$ | |
| $\{D_{j,i}, \hat{D}_{j,i}, F_{j,i}, \hat{F}_{j,i}\}_{j\neq i}$ | $t \cdot 8m$ | |
| $\{\psi_{i,j,k}\}_{j,k\neq i}$ | $t^2 \cdot (10m + 5k)$ | |
| $\{\hat{\psi}_{i,j,k}\}_{j,k\neq i}$ | $t^2 \cdot (10m + 5k)$ | |
| **Total** | $\mathbf{t^2 \cdot (20m + 10k) + t \cdot 8m + 2m}$ | $\mathbf{5.1 \cdot t^2 + 2t + 0.5KB}$ |

## Round 3

In this round, the participants must verify the messages intended for other participants as well as those meant for themselves. In addition, the participants will proactively broadcast a proof from the CGG+21 output phase that is normally done in reaction to something going wrong.

1. $P_i$ will verify all proofs designated for its verification. It will terminate and publish any culprits if a proof fails.
   a. $\Pi^{aff-g-inv}$ proof $\left\{\psi_{j,i,k}\right\}_{j,k\neq i}$
   b. $\Pi^{aff-g-inv}$ proof $\left\{\hat{\psi}_{j,i,k}\right\}_{j,k\neq i}$
   c. $\Pi^{log^*}$ proof $\left\{\psi'_{j,i}\right\}_{j\neq i}$
2. $P_i$ will compute $\Gamma = \Pi\Gamma_j$ and $\Delta_i = \Gamma^{k_i}$
   a. For all $\forall j \neq i$: $\alpha_{i,j} = Decrypt(D_{i,j})$ $and$ $\hat{\alpha}_{i,j} = Decrypt(\hat{D}_{i,j})$
   b. $\delta_i = \gamma_i k_i + \sum_{j\neq i}(\alpha_{i,j} + \beta_{i,j}) \; mod \; q$
   c. $\chi_i = w_i k_i + \sum_{j\neq i}(\hat{\alpha}_{i,j} + \hat{\beta}_{i,j}) \; mod \; q$
   d. $P_i$ creates for each participant $P_j$ a designated-verifier $\Pi^{log^*}$ proof $\psi''_{i,j}$ for the statement $(q, \Gamma, G, N_i, K_i, \Delta_i)$, showing that $\Delta_i = \Gamma^k$ and $K_i = Encrypt(k_i)$.
   e. $P_i$ computes $H_i = Encrypt(k_i \cdot \gamma_i)$. Then he creates a single $\Pi^{mul}$ proof $\psi^H{}_i$ for the statement $(q, N_i, G_i, K_i, H_i)$.
   f. $P_i$ computes $\tilde{\Delta}_i = H_i \oplus_{\forall j\neq i} \left(D_{i,j} \oplus F_{j,i}\right)$.
   g. $P_i$ creates for each participant $P_j$ a designated-verifier $\Pi^{dec}$ proof $\psi^{\delta}{}_{i,j}$ for the statement $(q, N_i, \tilde{\Delta}_i, \delta_i)$.

Each participant $P_i$ broadcasts $\delta_i, \Delta_i, H_i, \psi^H{}_i, \left\{\psi''_{i,j}, \psi^{\delta}{}_{i,j}\right\}_{j\neq i}$.

| Name | Size | Size with $m$ = 2048-bit Pailler modulus, $k$ = 256-bit ECDSA |
|---|---|---|
| $\delta_i$, $\Delta_i$ | $2k$ | |
| $H_i$ | $2m$ | |
| $\psi^H_i$ | $9m$ | |
| $\{\psi''_{i,j}\}_{j\neq i}$ | $t \cdot (5m + 3k)$ | |
| $\{\psi^{\delta}_{i,j}\}_{j\neq i}$ | $t \cdot (6m + 5k)$ | |
| **Total** | $t \cdot (11m + 6k) + 11m + 2k$ | $2.8 \cdot t + 2.8KB$ |

## Round 4

This round is the output round of the pre-signing computation. The participants verify all proofs, save state, and stop. No messages are sent. In an on-line signing scenario, the participants can combine Round 4 and Round 5 into a single round.

1. Participants $P_i$ computes $\forall j$: $\tilde{\Delta}_j = H_i \oplus_{\forall k \neq j} \left(D_{j,k} \oplus F_{k,j}\right)$.
2. Participant $P_i$ verifies $\forall j$: $\psi^H_j$, $\psi''_{j,i}$, and $\psi^{\delta}_{j,i}$.
3. The participant computes $\delta = \sum \delta_j \, mod \, q$ and $R = \Gamma^{\delta^{-1}}$.

The participant saves $k_i, K_i, \chi_i, \Gamma, R, \left\{\hat{F}_{j,i}, \hat{D}_{j,i}\right\}_{j\neq i}$.

## Round 5 Signing

This round can be combined with round 4. The participant receives the message $m$ that needs to be signed.

1. Participants $P_i$ computes $r = R|_{x-axis}$ and $\sigma_i = k_i \cdot m + r \cdot \chi_i \, mod \, q$.
2. Participants $P_i$ computes $\hat{H}_i = Encrypt(w_i, k_i)$
3. $P_i$ creates for each participant $P_j$ a designated-verifier $\Pi^{mul*}$ proof $\psi^{\hat{H}}_{i,j}$ for the statement $(q, N_i, W_i, K_i, \hat{H}_i)$.
4. Participant $P_i$ computes the encryption of $\sigma_i$ as: $\Sigma_i = \left(m \otimes K_i\right) \oplus \left(r \otimes \left(\hat{H}_i \oplus_{\forall j \neq i} \left(\hat{D}_{i,j} \oplus \hat{F}_{j,i}\right)\right)\right)$.
5. $P_i$ creates for each participant $P_j$ a designated-verifier $\Pi^{dec}$ proof $\psi^{\sigma}_{j,i}$ for the statement $(q, N_i, \Sigma_i, \sigma_i)$.

Each participant $P_i$ broadcasts $\sigma_i, \hat{H}_i, \left\{\psi^{\hat{H}}_{i,j}, \psi^{\alpha}_{i,j}\right\}_{j\neq i}$.

| Name | Size | Size with $m$ = 2048-bit Pailler modulus, $k$ = 256-bit ECDSA |
|---|---|---|
| $\sigma_i$ | $k$ | |
| $\hat{H}_i$ | $2m$ | |
| $\{\psi^{\hat{H}}_{i,j}\}$ | $t \cdot (5m + 3k)$ | |
| $\{\psi^{\alpha}_{i,j}\}_{j \neq i}$ | $t \cdot (6m + 3k)$ | |
| **Total** | $\mathbf{t \cdot (11m + 6k) + 2m + k}$ | $\mathbf{2.8 \cdot t + 0.5KB}$ |

## Round 6

The participants verify the proofs and output the signature. This step can be done by only one participant or trusted a third party observer.

1. Participant $P_i$ computes for each participant $P_j$ the encryption of $\sigma_j$ as
   $\Sigma_j = \left(m \otimes K_j\right) \oplus \left(r \otimes \left(\hat{H}_j \oplus_{\forall k \neq j} \left(\hat{D}_{j,k} \oplus \hat{F}_{k,j}\right)\right)\right)$
2. $P_i$ verifies for each participant $P_j$ proof $\psi^{\hat{H}}_{j,i}$ and $\psi^{\alpha}_{j,i}$.
3. $P_i$ computes $\sigma = \Sigma\sigma_j \; mod \; q$.

The participant outputs the signature is $(r, \sigma)$.

## Robust Key Generation

The key generation protocol in CGG+21 is identical to GG18. It has proofs of correctness at every step because it relies on VSS.

## Robust Key Refresh

CGG+21 does not have a key refresh protocol. Binance published its own as part of tss-lib. It has proofs of correctness at every step because it relies on VSS.

## Proof Aff-G-Inv

CGG+21 Section 6, Figure 14 presents a zero-knowledge proof $\Pi^{aff-g}$ that the prover knows values $(x, y, \rho, \rho_y)$ such that:

- $X = g^x \in G$
- $Y = Encrypt(N_1, y, \rho) = (1 + N_1)^y \cdot \rho^{N_1} \; mod \; N_1^2$
- $D = C^x \cdot Encrypt(N_0, y, \rho_y) = C^x \cdot (1 + N_0)^y \cdot \rho_y^{N_0} \; mod \; N_0^2$

The verifier checks the proof against the statement $(g, G, N_0, N_1, X, Y, C, D)$.

During round 2 of the pre-signing protocol in CGG+21 Section 4 Figure 7, the reader is directed to prove a very similar statement that only differs in the $D$ value: the ciphertext is an encryption of $q - y$ instead of $y$ as above. In this proof, the prover knows values $(x, y, \rho, \rho_y)$ such that

- $X = g^x \in G$
- $Y = Encrypt(N_1, y, \rho) = (1 + N_1)^y \cdot \rho^{N_1} \bmod N_1^2$
- $D = C^x \cdot Encrypt(N_0, q - y, \rho_y) = C^x \cdot (1 + N_0)^{(q-y)} \cdot \rho_y^{N_0} \bmod N_1^2$

We now show how to use the $\Pi^{aff-g}$ protocol to build $\Pi^{aff-g-inv}$.

**Aff-G-Inv Proof**. The prover has private input $(x, y, \rho, \rho_y)$ and public input $(q, g, G, N_0, N_1, X, Y, C, D)$.

1. The prover computes $\hat{Y} = \left(Y^{-1}\right) \cdot \left(1 + N_1\right)^{q-N_1} \bmod N_1^2 = \left(1 + N_1\right)^{(q-y)-N_1} \cdot \psi_y^{N_1} \bmod N_1^2$.
2. Then the prover computes $Decrypt(sk_1, \hat{Y})$ to get the message $q - y - N_1$ and randomness $\psi_y$
3. The prover computes $\hat{D} = C^x \cdot (1 + N_0)^{q-y} \cdot \psi^{N_0} \bmod N_0^2$
4. The prover outputs the proof $\Pi^{aff-g}$ using private input $(x, y, \psi, \psi_y)$ and public input $(q, g, G, N1, X, Y, C, \hat{D})$. Note: it is important to use the original $Y$ instead of $\hat{Y}$.

**Aff-G-Inv Verification**. The verifier gets the statement $(q, g, G, N_0, N_1, X, Y, C, \hat{D})$ and an $\Pi^{aff-g}$ proof $\pi$.

1. The verifier computes $\hat{Y} = \left(Y^{-1}\right) \cdot \left(1 + N_1\right)^{q-N_1} \bmod N_1^2$.
2. The verifier creates a new statement $(q, g, G, N_1, X, \hat{Y}, C, \hat{D})$ and verifies $\Pi^{aff-g}$ proof $\pi$.

$\Pi^{aff-g}$ is a sigma protocol. The special honest verifier zero-knowledge property trivially holds on $\Pi^{aff-g-inv}$ because we use the same simulator as for $\Pi^{aff-g}$. Similarly, the same special soundness extractor works on $\Pi^{aff-g-inv}$. Specifically, the proof $\Pi^{aff-g}$ contains values $z_1 = \alpha + ex$, $z_2 = \beta + ey$, $z_1 = \gamma + em$, and $z_4 = \delta + e\mu$ where $e$ is the challenge. It is straightforward to extract secrets $(\alpha, x, \beta, y, \gamma, m, \delta, \nu)$ given a second challenge $e'$ and a set of values $z'_1 = \alpha + e'x$, $z'_2 = \beta + e'y$, $z'_1 = \gamma + e'm$, and $z'_4 = \delta + e'\mu$. Having recovered these values, the simulator has enough information to solve for the remaining secret values.

# 04 Bibliography


1. [Apache]. Apache Incubator Milagro MPC library. https://github.com/apache/incubator-milagro-MPC
2. [Binance19] Binance. Multi-Party Threshold Signature Scheme. 2019. https://github.com/bnb-chain/tss-lib
3. [CGG+21] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts. CCS '20: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. October 2020. Pages 1769–1787. https://eprint.iacr.org/2021/060. Published in 2020 ACM SIGSAC).
4. [FIPS 186-4]. National Institute of Standards and Technology (2013) Digital Signature Standard (DSS). (U.S. Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) 186-4. https://csrc.nist.gov/publications/detail/fips/186/4/final
5. [GG18] Rosario Gennaro and Steven Goldfeder. Fast Multiparty Threshold ECDSA with Fast Trustless Setup. CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. October 2018. Pages 1179–1194. https://eprint.iacr.org/2019/114
6. [GG20] Rosario Gennaro and Steven Goldfeder. One Round Threshold ECDSA with Identifiable Abort. Cryptology ePrint Archive, 2020. https://eprint.iacr.org/2020/540.pdf
7. [Paillier99]. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. Eurocrypt 99. Pages 223-238. https://www.researchgate.net/publication/221348062_Public-Key_Cryptosystems_Based_on_Composite_Degree_Residuosity_Classes
8. [SafeHeron]. Multi-Party ECDSA CPP. https://github.com/Safeheron/multi-party-ecdsa-cpp
9. [0xEigenLabs]. TSS WASM. https://github.com/0xEigenLabs/tss-wasm
10. [ZenGo] Multi-Party ECDSA. https://github.com/ZenGo-X/multi-party-ecdsa

circle.com/circle-research