Identifiable Aborts from Public Information in Multi-TSS

1 Introduction

Digital signatures are a crucial component to modern internet-base systems. For example, users could verify software updates' authenticity from signatures signed by technology companies. Certificate Authorities (CAs) issue certificates attesting to the authenticity of a websites' public key to secure the web, and web servers in turn use those authenticated keys to communicate with clients securely. In cryptocurrencies, digital signatures could be used to authenticate transactions, which means the ability to generate a signature is equal to the ability to spend one's money. From all given instances, it easy to image how catastrophic it can be if the signing key is stolen or loss. Therefore, a key point is how to store signing keys in a manner that is both easy to use and resilient to theft and loss.

Threshold cryptography, and *threshold signatures* in particular, has been gaining traction as an approach to solving this problem. In a threshold signature scheme, signing keys are distributed among several servers which need to act jointly in order to issue a signature. More specifically, in a threshold signature scheme, a key is split into n shares and a parameter t is defined such that an adversary that compromises t or fewer shares is unable to generate a signature and learns no information about the key. On the other hand, in a *threshold optimal* scheme, $t + 1$ shares can be used jointly issue a signature without ever reconstructing the key. Splitting the key in this way could eliminate a single point of failure and allows the honest parties to recover even in the face of partial of compromise.

Identifiable and attribute aborts. The current state-of-the-art threshold ECDSA (Elliptic Curve Digital Signature Algorithm) protocols operate in the dishonest majority model. This model is highly desirable as it allows building threshold signature protocols where the threshold t can take on any value so long as it is less than the total number of players n.

In this model, if parties misbehave, the protocol may abort without producing a signature. Clearly, as $t + 1$ players are required to sign, and the adversary can corrupt up to t nodes, there is no guarantee in the dishonest majority setting that a signature will be issued since there may simply not be $t + 1$ honest nodes. But even if aborts are unavoidable, one may want to identify which player(s) misbehaved and caused the abort. What's more, we not only want the participants to find such player(s), also want other's who did not join in signing stage can identify aborts.

2 Background

2.1 Communication and adversarial model

We assume the existence of a broadcast channel as well as point-to-point channels connecting every pair of players. A simple *echo broadcast* suffices in which each party sends to every other party the hash of all of the broadcasted messages. If any party receives an inconsistent hash from some other part, it aborts and notifies every other party. While unforgeability does not rely on full broadcast, the identification protocol does require broadcast. The use of a broadcast channel is standard in all work of MPC-with-abort and indeed it has been shown that MPC-IA indeed implies the existences of a broadcast channel.

We assume a probabilistic polynomial time malicious adversary, who may deviate from the protocol description arbitrarily. The adversary can corrupt up to t players, and it learns the private state of all corrupted players.

We assume a *rushing* adversary, meaning that the adversary gets to speak last in a given round and, in particular, can choose his message after seeing the honest parties messages.

We assume a dishonest majority, meaning t, the number of players the adversary corrupts, can take on any value up to $n-1$. In this setting, there is no guarantee that the protocol will complete, and we therefore do not attempt to achieve robustness, or the ability to complete the protocol even in the presence of some misbehaving participants. Instead, we show how to identify aborts from the public information given by participants, we guarantee the consistency between the public information and previous broadcasted messages.

2.2 Signature Schemes

A digital signature schemes $\mathcal{S}$ consists of three efficient algorithms:

- $(\mathbf{sk}, \mathbf{pk}) \leftarrow \textbf{Key-Gen}(1^\lambda)$, the randomized key generation algorithm which takes as the security parameter and returns the private signing key **sk** and public verification key **pk**.

- $\sigma \leftarrow \mathbf{Sig}(\mathbf{sk}, m)$, the possibly randomized signing algorithm which takes as input the private key sk and the message to be signed m and outputs a signature, σ. As the signature may be randomized, there may be multiple valid signatures. We denote the set of valid signatures as $\{\mathbf{Sig}(\mathbf{sk}, m)\}$ and require $\sigma \in \{\mathbf{Sig}(\mathbf{sk}, m)\}$

- $b \leftarrow \mathbf{Ver}(\mathbf{pk}, m, \sigma)$, the deterministic verification algorithm, which takes as input a public key **pk**, a message m and a signature σ and outputs a bit b which equals 1 if and only if σ is a valid signature on m under **pk**

[Existential unforgeability] Consider a PPT adversary $\mathcal{A}$ who is given public **pk** output by **Key-Gen** and oracle access to the signing algorithm $\mathbf{Sig}(\mathbf{sk}, \cdot\ )$ with which it can receive signatures on adaptively chosen message of its choosing. Let $\mathcal{M}$ be the set of messages queried by $\mathcal{A}$. A digital signature scheme $\mathcal{S} = (Key-Gen, Sig, Ver)$ is said to be *existentially unforgeable* if there is no such PPT adversary $\mathcal{A}$ that can produce a signature on a message $m \in \mathcal{M}$, except with negligible probability in λ.

2.3 Threshold Signatures

Threshold secret sharing - A (t,n)-threshold secret sharing of a secret x consists of n shares $x_1, ..., x_n$ such that an efficient algorithm exists that takes as input $t+1$ of these shares and outputs the secret but t or fewer shares do not reveal any information about the secret.

Threshold signature schemes - Consider a signature scheme, $\mathcal{S} = (Key-Gen, Sig, Ver)$. A (t,n)-threshold signature scheme $\mathcal{TS}$ for $\mathcal{S}$ enables distributing the signing among a group of n players $P_1, ..., P_n$ such that any group of at least $t+1$ of these players can jointly generate a signature, whereas group of size t or fewer cannot. More formally, $\mathcal{TS}$ consists of two protocols:

- **Thresh-Key-Gen**, the distributed key generation protocol, which takes as input the security parameter 1^λ. Each player P_i receives as output the public key **pk** as well as a private output sk_i, which is P_i's share of the private key. The value $sk_1, ..., sk_n$ constitute a (t,n) threshold secret sharing of the private key **sk**.

- **Thresh-Sig**, the distributed signing protocol which takes as public input a message m to be signed as well as a private input sk_i from each player. It outputs a signature $\sigma \in \{\mathbf{Sig}(\mathbf{sk}, m)\}$.

2.4 Identifiable Aborts from public information

For an identifiable abort, it allows the computation to fail (abort), while guaranteeing that all the honest participants agree on the identity P_i of a corrupted player.

With the help of public information, including broadcasted messages and parameters published by parties who identified aborts, one who doesn't participate in the signing stage can also identify aborts.

2.5 Paillier Cryptosystem

The implemented protocol relies on the Paillier cryptosystem ξ=(**Key-Gen, Enc, Dec**), which is a homomorphic encryption scheme over a large integer *N*.

- **Key-Gen**:

  1. Choose two large prime numbers p and q randomly and independently of each other such that $gcd(pq, (p-1)(q-1)) = 1$. This property is assured if both primes are of equal length.
  2. Compute $n = pq$ and $\lambda = lcm(p-1, q-1)$. Lcm means Least Common Multiple.
  3. Select random integer g where $g \in Z^*_{n^2}$
  4. Ensure n divides the order of g by checking the existence of the following modular multiplicative inverse: $\mu = (L(g^\lambda \; mod \; n^2))^{-1} \; mod \; n$, where function $\boldsymbol{L}$ is defined as $L(x) = \frac{x-1}{n}$

  The public encryption key is (n, g), and the private decryption key is (λ, μ).

- **Encryption**:

  1. Let m be a message to be encrypted where $0 \leq m < n$
  2. Select random r where $0 < r < n$ and $r \in Z^*_{n^2}$(i.e., ensure $gcd(r, n) = 1$)
  3. Compute ciphertext as $c = g^m \cdot r^n \; mod \; n^2$

- **Decryption**

  1. Let c be the ciphertext to decrypt
  2. Compute the plaintext message as $m = L(c^\lambda \; mod \; n^2) \cdot \mu \; mod \; n$

A notable feature of the Paillier cryptosystem is its homomorphic properties along with its non-deterministic encryption. As the encryption function is additively homomorphic, the following identities can be described:

- **Homomorphic addition of plaintexts**: The product of two ciphertexts will decrypt to the sum of their corresponding plaintexts,

  $$D(E(m_1, r_1) \cdot E(m_2, r_2) \; mod \; n^2) \;=\; m_1 + m_2 \; mod \; n \qquad (1)$$

  The product of a ciphertext with a plaintext raising g will decrypt to the sum of the corresponding plaintexts,

  $$D(E(m_1, r_1) \cdot g^{m_2} \; mod \; n^2) \;=\; m_1 + m_2 \; mod \; n \qquad (2)$$

- **Homomorphic multiplication of plaintexts**: An encrypted plaintext raised to the power of another constant k will decrypt to the product of the plaintext and the constant,

  $$D(E(m_1, r_1)^k \; mod \; n^2) = km_1 \; mod \; n \qquad (3)$$

However, given the Paillier encryptions of two messages there is no known way to compute an encryption of the product of these messages without knowing the private key.

2.6 Assumption

DDH. Let $\mathcal{G}$ be a cyclic group of prime order q, generated by g. The DDH Assumption states that the following two distributions over $\mathcal{G}^3$ are computationally indistinguishable: $DH = \left\{(g^a, g^b, g^{ab})\ for\ a, b \in_R Z_q\right\}$ and $R = \left\{(g^a, g^b, g^c)\ for\ a, b, c \in_R Z_q\right\}$. STRONG-RSA. Let N be the product of two safe primes, $N = pq$, with $p = 2p' + 1$ and $q = 2q' + 1$ with p', q' primes. With $\phi(N)$ we denote the Euler function of N, i.e., $\phi(N) = (p-1)(q-1) = p'q'$. With Z_N^* we denote the set of integers between 0 and $N - 1$ and relatively prime to N.

Let e be an integer relatively prime to $\phi(N)$. The RSA Assumption states that it is infeasible to compute e-roots in Z_N^*. That is, given a random element $s \in_R Z_N^*$ it is hard to find x such that $x^e = s\ mod\ N$.

The Strong RSA Assumption states that given a random element s in Z_N^* it is hard to find $x, e \neq 1$ such that $x^e = s\ mod\ N$. The assumption differs from the traditional RSA assumption in that we allow the adversary to freely choose the exponent e for which she will be able to compute e-roots.

Assumption 1 *We say that Strong RSA Assumption holds, if for all probabilistic polynomial time adversaries $\mathcal{A}$ the following probability*

$$Prob[N \leftarrow SRSA(n); s \leftarrow Z_N^* : \mathcal{A}(N, s) = (x, e) s.t. x^e = s\ mod\ N] \tag{4}$$

is negligible in n.

2.7 Multiplicative-to-additive share conversion protocol (MtA)

The setting consists of two players, $\mathcal{P}_1$ and $\mathcal{P}_2$, who hold multiplicative shares of a secret x. In particular, $\mathcal{P}_1$ holds a share $a \in Z_q$, and $\mathcal{P}_2$ holds a secret share $b \in Z_q$ such that $x = ab\ mod\ q$. The goal of the **MtA** protocol is to convert these multiplicative shares into additive shares. $\mathcal{P}_1$ receives private output $\alpha \in Z_q$ and $\mathcal{P}_2$ receives private output $\beta \in Z_q$ such that $\alpha + \beta = x = ab\ mod\ q$.

MtAwc. In the basic MtA protocol, the player's inputs are not verified, and indeed the players can cause the protocol to produce an incorrect output by inputting the wrong values a', b'. In the case that $B = g^b$ is public, the protocol can be enhanced to include an extra check that ensures that $\mathcal{P}_2$ inputs the correct value $b = log_g(B)$. This enhanced protocol is denoted as **MtAwc** (for **MtA** with check).

We assume that player $\mathcal{P}_1$ is associated with a public key E_1 for an additively homomorphic scheme ξ defined over an integer N. Let $K > q$ be a bound.

1. $\mathcal{P}_1$ initiates the protocol:
    - Compute $c_A = E_1(a)$
    - Compute a zero knowledge range proof π_A that $\{a : D_1(c_A) = a \bigwedge a < K\}$
    - Send (c_A, π_A) to $\mathcal{P}_2$

2. Upon receiving (c_A, π_A) from $\mathcal{P}_1$, $\mathcal{P}_2$ does the following:
    - Verifies π_A, and aborts if it fails to verify
    - Choose $\beta' \overset{\$}{\leftarrow} Z_N$
    - Set output $\beta = -\beta'$
    - Compute $c_B = b \times_E c_A +_E E_1(\beta') = E_1(ab + \beta')$
    - Compute a zero knowledge range proof π_B^1 that $\{b, \beta' : b < K \bigwedge c_B = b \times_E c_A +_E E_1(\beta')\}$
    - (MtAwc, i.e., if $B = g^b$ is public): Compute a zero knowledge proof of knowledge π_B^2 that he knows $\{b, \beta' : b < K \bigwedge c_B = b \times_E c_A +_E E_1(\beta')\}$
    - Send (c_B, π_B^1, π_B^2) to $\mathcal{P}_1$
3. Upon receiving (c_B, π_B^1, π_B^2) from $\mathcal{P}_2$, $\mathcal{P}_1$ does the following:
    - Verifies π_B^1 and π_B^2 if they are running MtAwc, and aborts if either proof fails to verify
    - Compute $\alpha' = D_1(c_B)$
    - Set output $\alpha = \alpha' \ mod \ q$

In this paper, we don't talk about the correctness and simulating proof of the MtA/MtAwc. From previous researches, the current protocol does not need to have the range proof, and only need to do is to provide zero knowledge proof π_B^2 under discrete logarithm, which parameters of g^b and $g^{\beta'}$ will be public.

2.7.1 Zero Knowledge Proof of b/β'

When $\mathcal{P}_1$ receiving π_B^2, he need to verify the correctness of proofs, and we would first talk about the zero knowledge protocol about b and β'. It's a simple sigma protocol which public share is:

- $N = pq$ is an RSA modulus
- g is an element of Z_N^* of high order
- $v = g^b \ mod \ N$ ($v = g^{\beta'} \ mod \ N$) is public, where $b(\beta')$ is prover's secret
- defined S as the bound for the secret key
- defined $R = 2^{k+k'} \cdot S$, where k and k' are security parameters

Prover

Verifier

$r \in \{0, ..., R-1\}$
$x = g^r \ mod \ N$

$$\xrightarrow{\quad\quad x \quad\quad}$$

$e \in \left\{0, ..., 2^k - 1\right\}$

$$\xleftarrow{\quad\quad e \quad\quad}$$

$y = r - e \cdot b$
$(y = r - e \cdot \beta')$

$$\xrightarrow{\quad y \quad}$$

$$x \overset{?}{=} g^y v^e$$

2.7.2 Zero Knowledge Proof of c_B

Furthermore, $\mathcal{P}_1$ should also verify the construction of $c_B = E_1(ab + \beta')$. It's easy to do that with public shares from proof of b and β', where $\mathcal{P}_1$ could learn g^b and $g^{\beta'}$. Then $\mathcal{P}_1$ decrypt c_B to get a plaintext m' and generate corresponding discrete log $g^{m'}$, and using public shares with its secret to create a discrete log $g^m = (g^b)^a \cdot g^{\beta'}$. Finally, it will check whether $g^m \overset{?}{=} g^{m'}$.

3 Protocol

The players run on input $\{\mathcal{G}, g\}$ the cyclic group used by the ECDSA signature scheme. We assume that each player P_i is associated with a public key E_i for an additively homomorphic encryption scheme ξ.

3.1 Key Generation

- **Phase 1.** Each Player P_i selects $u_i \in_R Z_q$; computes $[KGC_i, KGD_i] = Com(g^{u_i})$ and broadcasts KGC_i. Each player P_i broadcasts E_i the public key for Paillier cryptosystem.
  Probable Aborts.

  1. **Commitment Consistency.** If a player P_a found that the commitment was not consistent with the corresponding decommitment, it will cause an abort.
  2. **Consistency in Paillier Cryptosystem.** Each player P_i should provide the zero knowledge proof about their decryption key D_i of Paillier Cryptosystem, which is related to its public encryption key E_i.

- **Phase 2.** Each player P_i broadcasts KGD_i. Let y_i be the value decommitted by P_i. The player P_i performs a (t,n) Feldman-VSS of the value u_i, with y_i as the "free term in the exponent".
  Probable Aborts.

  1. **Feldman-VSS Incorrectness.** A player could complains that the Feldman share it received is inconsistent and therefore does not verify correctly, the protocol will abort.
  2. **Zero Knowledge Proof of x_i** After Feldman share, each player P_i would generate its own secret x_i. Meanwhile, P_i should give the zk-proof of its x_i and therefore if one's proof failed, the protocol will abort.

- **Phase 3.** Let $N_i = p_i q_i$ be the RSA modulus associated with E_i. Each player P_i proves in ZK that he knows x_i using Schnorr's protocol, that N_i is square-free and that h_1 h_2 generate the same group modulo N_i.
  Probable Aborts.

  1. **Composite Discrete Log Proof.** If there is no discrete log relationship between h_1, h_2, then such proof would not pass the verification, the protocol would abort.

3.2 Sign

We now describe the signing protocol, which is run on input m (the hash of the message M being signed) and the output of the key generation protocol described above. We note that the latter protocol is a t-out-of-n protocol (and thus the secret key x is shared using (t,n) Shamir secret-sharing).

Let $S \subseteq [1..n]$ be the set of players participating in the signature protocol. We assume that $|S| = t+1$. For the signing protocol we can share any ephemeral secrets using a $(t, t+1)$ secret sharing scheme, and do not need to use the general (t, n) structure. We note that using the appropriate Lagrangian coefficients $\lambda_{i,S}$ each player is S can locally map its own (t, n) share x_i of x into a $(t, t+1)$ share of x, $w_i = (\lambda_{i,S})(x_i)$, i.e., $x = \sum_{i \in S} w_i$. Since $X_i = g^{x_i}$ and $\lambda_{i,S}$ are public values, all the players can compute $W_i = g^{w_i} = X_i^{\lambda_{i,S}}$.

- **Phase 1.** Each player P_i selects $k_i, \gamma_i \in_R Z_q$; computes $[C_i, D_i] = Com(g^{\gamma_i})$ and broadcast C_i
  Define $k = \sum_{i \in S} k_i, \gamma = \sum_{i \in S} \gamma_i$. Note that

$$\begin{aligned} k\gamma &= \sum_{i,j \in S} k_i \gamma_i \; mod \; q \\ kx &= \sum_{i,j \in S} k_i x_i \; mod \; q \end{aligned} \tag{5}$$

- **Phase 2.** Every pair of players P_i, P_j engages in **two** MtA share conversion subprotocols. Note that the first message for these protocols is the same and is only sent once.

  – P_i, P_j run MtA with shares k_i, γ_j (P_i acts as $\mathcal{P}_1$, while P_j acts as $\mathcal{P}_2$). Let α_{ij} [resp. β_{ij}] be the share received by player P_i [resp. P_j] at the end of the protocol, i.e.

$$k_i \gamma_j = \alpha_{ij} + \beta_{ij} \tag{6}$$

  Player P_i sets $\delta_i = k_i \gamma_i + \sum_{j \neq i} \alpha_{ij} + \sum_{j \neq i} \beta_{ij}$. Note that the δ_i are a $(t, t+1)$ additive sharing of $k\gamma = \sum_{i \in S} \delta_i$.

  – P_i, P_j run MtAwc with shares k_i, w_j respectively. Let μ_{ij} [resp. ν_{ij}] be the shared received by player P_i [resp. P_j] at the end of this protocol, i.e.

$$k_i w_j = \mu_{ij} + \nu_{ij} \tag{7}$$

Player P_i sets $\sigma_i = k_i w_i + \sum_{j \neq i} \mu ij + \sum_{j \neq i} \nu_{ij}$. Note that the σ_i are a $(t, t+1)$ additive sharing of $kx = \sum_{i \in S} \sigma_i$

Probable Aborts.

1. **Zero Knowledge Failed.** The zk-proof in the message from player P_j to P_i would failed if $c_B \neq E_i(k_i \gamma_j + \beta')$ or one of the sub-zk-proofs of γ_j, β' is incorrect, the protocol would abort (Same with k_i, w_j).

- **Phase 3.** Every player P_i broadcasts
  - δ_i and the players reconstruct $\delta = \sum_{i \in S} \delta_i = k\gamma$. The players compute δ^{-1} $mod\ q$
  - $T_i = g^{\sigma_i} h^{l_i}$ with $l_i \in_R Z_q$ and proves in ZK that he knows σ_i, l_i
- **Phase 4.** Each player P_i broadcasts D_i. Let Γ_i be the values decommitted by P_i. The players compute $\Gamma = \prod_{i \in S} \Gamma_i$, and

$$R = \Gamma^{\delta^{-1}} = g^{(\sum_{i \in S} \gamma_i) k^{-1} \gamma^{-1}} = g^{\gamma k^{-1} \gamma^{-1}} = g^{k^{-1}} \tag{8}$$

as well as $r = H'(R)$.
Probable Aborts.

1. **Commitment Inconsistent** Each player should check whether the C_i is consistent with the decommitment D_i.
2. **ZK-proof of g^{γ_i}** Each player should verify that g^{γ_i} is equal with the Γ_i, which is the value decommitted from D_i, if not, then the protocol will abort.

- **Phase 5.** Each player P_i broadcasts $\overline{R}_i = R^{k_i}$ as well as a zero-knowledge proof of consistency between R_i and $E_i(k_i)$, which each player sent as the first message of the MtA protocol in Phase 2. If

$$g \neq \prod_{i \in S} \overline{R}_i \tag{9}$$

the protocol aborts.
Probable Aborts.

1. **Equation Incorrectness** Check the equation (9)
2. **Inconsistency between Parameters** If zk-proof of consistency between R_i and $E_i(k_i)$ failed, the protocol aborts.

- **Phase 6.** Each player P_i broadcasts $S_i = R^{\sigma_i}$ as well as a zero-knowledge proof of consistency between S_i and T_i, which each player sent in Phase 3. If

$$y \neq \prod_{i \in S} S_i \tag{10}$$

the protocol aborts.
Probable Aborts.

1. **Equation Incorrectness** Check the equation (10)
2. **Inconsistency between Parameters** If zk-proof of consistency between S_i and T_i failed, the protocol aborts.

- **Phase 7.** Each player P_i broadcasts $s_i = mk_i + r\sigma_i$ and set $s = \sum s_i$. If the signature (r, s) is correct for m, the player accept, otherwise they abort.
  Probable Aborts.

  1. **Signature Invalid** If the sum of splitted signature is invalid, the protocol aborts.

4 Identifying Aborts

A key problem with all known threshold ECDSA protocols is that in the case of aborts, it is not possible to always identify which party is responsible for causing the signature to fail. In the previous researches, a new protocol has been come up. However, in some real applications, someone who does not participate in the signing protocol could also find which party triggered the aborts from broadcasted messages and public information after aborts. We will show how to identify aborts.

First of all we assume that all messages transferred between are signed, so that it is possible to determine their origin. What's more, we denote outside players who does not participate in signing stage and we should convince them that who cause aborts.

The protocol will abort in case any player deviates from the protocol in a clearly identifiable way by not complying with the protocol instructions - e.g. not sending a message when required. In this case the bad player is clearly identified and removed, and also from outside players' view. Note that we assume a broadcast channel so if a player behaves badly, everybody knows that, and when we want to convince outside players we could use such messages. Note that this requires that *every message* of the protocol has to be reliably broadcast (this includes the pair-wise MtA protocols, which will enable the identification procedure described below).

In this case a message from a player fails to appear, we apply a local timeout bound before marking that player as corrupted, to account for possible delays in message delivery.

We focus our attention here on aborts that are not clearly identifiable as deviations from the protocol - i.e. where the player sent a message of the correct form at the correct time, but the *contents* of the message was crafted in a way that caused the protocol to fail.

4.1 Key Generation

In the key generation protocol, there are 5 possible aborts can occur:

1. **Phase 1.** If a player P_a found that the commitment was not consistent with the corresponding decommitment, it will cause an abort.

2. **Phase 1.** Each player P_i should provide the zero knowledge proof about their decryption key D_i of Paillier Cryptosystem, which is related to its public encryption key E_i.

3. **Phase 2.** A player could complains that the Feldman share it received is inconsistent and therefore does not verify correctly, the protocol will abort.

4. **Phase 2.** After Feldman share, each player P_i would generate its own secret x_i. Meanwhile, P_i should give the zk-proof of its x_i and therefore if one's proof failed, the protocol will abort.

5. **Phase 3.** If there is no discrete log relationship between h_1, h_2, then such proof would not pass the verification, the protocol would abort.

Except type 3 aborts, the others can be identified easily because such aborts are cause by ZK proofs failed or inconsistency between commitment and decommitment. However, for type 3 abort in Phase 2, it means that a player P_j complains about a player P_i meaning that P_j claims the private share he received does not match the public information of P_i's Feldman VSS. In this case it might be useful to identify who the bad player is, in order to remove it from the n players when the key generation protocol is re-run. Here there is ambiguity as if the bad player is P_i or P_j.

A SIMPLE IDENTIFICATION PROTOCOL Notice that if the failure happens during the key generation protocol, it is safe to abandon the protocol and publish the would-be private key since it has not yet been established or used. Thus, if P_j raises a complaint about a share he received from P_i, the simplest identification protocol has him publish the share that he received from P_i in the clear, and indeed anyone can now check whether the share that he received is consistent (recall that we assume that all messages are signed, so the share can be authenticated and P_i cannot be framed by publishing an incorrect share). After the misbehaving player is identified, the key generation protocol will need to be re-run with fresh randomness to establish a secure key.

4.1.1 Verification from Public Information

As mentioned above, only type 3 abort will published new information to the broadcast channel. Therefore, outside player will do as following to verify aborts:

1. Outside player could learn the previous commitment KGC_i and decommitment KGD_i for player P_i, so outside player P_O could re-commit the value, decommitted from KGD_i and check whether it is equal with KGC_i to identify the abort.

2. Each player P_i would provide the proof of their secret key of Paillier cryptosystem, thus it is easy to verify the proof with public encryption key E_i of Paillier cryptosystem.

3. From **SIMPLE IDENTIFICATION PROTOCOL**, P_j would publish its share from P_i in the clear. Apart from the inside players, who participate in key generation stage, outside players could also use the secret share from P_j and P_i's VSS-scheme to check its validity.

4. P_i would give a simple sigma zk proof of its secret x_i and the verification would only need the size of signature, which is *(t,n)*, and the information of y_i, which is also published. Therefore, it is easy to verify the proof for outside player.
   $\mathit{Proof} = \{g^{x_i}, g^r(randomness), challenge = r - e \cdot x_i\}$
   $\mathit{Verification} = \left\{g^r \stackrel{?}{=} g^c \cdot g^{e \cdot x_i}\right\}$

5. For each KGC_i, there is a composite discrete log proof of N_i and also a discrete log statement (contains information about N_i, and corresponding $h_1, h_2 = {h_1}^{xh_i}$). Once the statement is wrong, it would not pass the proof, so outside players could verify the proof by using its statement. $\mathit{Proof} = \{x = {h_1}^r(randomness)\ mod\ N_i, challenge = r + e \cdot xh_i\}$
   $\mathit{Verification} = \left\{x \stackrel{?}{=} {h_1}^c \cdot {h_2}^e\right\}$

4.2 Sign

In our signing protocol, aborts can occur in the following ways:

1. **Phase 2.** The zk-proof in the message from player P_j to P_i would failed if $c_B \neq E_i(k_i\gamma_j + \beta')$ or one of the sub-zk-proofs of γ_j, β' is incorrect, the protocol would abort (Same with k_i, w_j).

2. **Phase 4.** Each player should check whether the C_i is consistent with the decommitment D_i.

3. **Phase 4.** Each player should verify that g^{γ_i} is equal with the Γ_i, which is the value decommitted from D_i, if not, then the protocol will abort.

4. **Phase 5.** Check the equation (9)

5. **Phase 5.** If zk-proof of consistency between R_i and $E_i(k_i)$ failed, the protocol aborts.

6. **Phase 6.** Check the equation (10)

7. **Phase 6.** If zk-proof of consistency between S_i and T_i failed, the protocol aborts.

8. **Phase 7.** If the sum of splitted signature is invalid, the protocol aborts.

For item 1,2,3,5,7, identification of the cheating is simple. In these steps, the aborts result due to the failure of a commitment opening or zero knowledge proof to verify, and the abort is thus attributable to the player who gave the faulty proof or the faulty opening.

For aborts of type 8, i.e. in Phase 7 when the signature *(r,s)* does not verify on message m, we note that if we got to that point then $g = \prod \overline{R}_i$ (where $\overline{R}_i = R^{k_i}$) and $y = \prod S_i$ where $S_i = R^{\sigma_i}$. Note at this phase player P_i should broadcast $s_i = mk_i + r\sigma_i \ mod \ q$. We can check if

$$R^{s_i} = \overline{R}_i^m \cdot S_i^r \tag{11}$$

if all the above equations hold then the signature should verify. Indeed

$$R^s = R^{\sum s_i} = [\prod \overline{R}_i]^m \cdot [\prod S_i]^r = g^m y^r \tag{12}$$

which holds for correct signatures. So we can identify the malicious player by checking for which player Equation (11) does not hold.

The core difficulty is attributing aborts of type 4 or 6: when $\prod_{i \in S} \overline{R}_i \neq g$ or $\prod_{i \in S} S_i \neq y$. At a high level, this means that the distributed values used to compute the signature are wrong, but it gives no indication as to where things went wrong. Indeed, this could be caused by a failure of the MtA protocol itself where a player sent an incorrect ciphertext to another player. But even if the MtA protocols themselves succeeded, the failure could also be caused by players later revealing wrong values that are not consistent with the values they received during the MtA protocols. This could happen if a player reveals the wrong δ_i or Σ_i which would lead to an incorrect R and thus an invalid signature. It could also be caused by a player inputting an incorrect value σ_i in either Phase 3 or Phase 6 (which would lead to an incorrect s).

Recall that when a signature is public it is important that k be kept secret as given k and the signature using k, one can compute the secret key x. Similarly, if a player published its value s_i and k_i, then this would leak its secret share x_i. The problem stems from publishing both k_i as well as a signature share s_i in which k_i was used. However, if s_i has not been published, k_i has no special significance and indeed can be published without leading any information about the key.

Consider now the abort in Phase 5 when $g \neq \prod \overline{R}_i$ (type 4 aborts). At this point, the values s_i in the signature protocol have not been released. Indeed at this point, it is completely acceptable for the players to reveal their values k_i in the clear. And the same is true for the ephemeral value γ_i. Absent the value s_i, the value γ_i need not be kept secret. This means that the MtA protocol with k_i and γ_j can be completely opened. This immediately enables checking all of the values that δ_i is comprised of are made public in Phase 3 and Phase 4.

Therefore the identification protocol for failures of Type 4 in Phase 5 works as follows:

- Each player P_i publishes its values $k_i, \gamma_i, \alpha_{ij}, \beta_{ij}$ for all j as well as the randomness used to encrypt these values during the MtA protocol.

- Every other Player P_j can now verify the correctness of δ_i in the clear. If for any player they do not hold equation (12), the abort is attributed to that player, and the identification protocol terminates.

$$\delta_i = k_i\gamma_i + \sum_{j\neq i}\alpha_{ij} + \sum_{j\neq i}\beta_{ij} \tag{13}$$

Let's now focus on the abort of Type 6 in Phase 6 ($y \neq \prod S_i$). Here players cannot completely open the MtAwc protocol between k_i and w_j since w_j is P_j's long-term secret and, unlike the ephemeral values k_i and γ_j, the value w_i needs to be kept secret even if the signature aborts. We show, however, that it is safe to reveal the value μ_{ij} in the clear, and using these, we can check the correctness of σ_i in the exponent, allowing us to identify the misbehaving player. We now proceed with the details of the identification protocol for failures of Type 6:

- Each player P_i publishes k_i and μ_{ij} as the decryption of the appropriate ciphertext in the MtAwc protocol. Recall that in Paillier's scheme, given a ciphertext and a private key, one can decrypt the plaintext and also recover the randomness used to encrypt, allowing anybody to verify the correctness of the claimed decrypted value by re-encryption.

- Every other player P_l can now verify that the value sent P_i to P_j was k_i and the value sent by P_j to P_i was μ_{ij}.
Moreover, for all j, since g^{w_j}, k_i, μ_{ij} are public, everyone can now compute $g^{\nu_{ji}}$ using the equation:

$$g^{\mu_{ij}} = g^{w_j k_i} g^{\nu_{ji}} \tag{14}$$

And now they can additionally compute

$$g^{\sigma_i} = g^{w_j k_i} \prod_{j\neq i} g^{\mu_{ij}} \prod_{j\neq i} g^{\nu_{ji}} \tag{15}$$

- Each player P_i proves in zero knowledge the consistency between g^{σ_i} that was computed in the previous step and $S_i = R^{\sigma_i}$. If for any player this does not hold, the abort is attributed to that player.

The ZK proof above is classic one. An honest-verifier protocol is described below for completeness. The prover has two values $\Sigma = g^{\sigma}$ and $S = R^{\sigma}$. He sends $\alpha = g^a$ and $\beta = R^a$ for $a \in_R Z_q$. The verifier sends challenge $c \in_R Z_q$. The prover answers with $t = a + c\sigma \ mod\ q$. The verifier checks $g^t = \alpha\Sigma^c$ and $R^t = \beta S^c$.

4.2.1 Verification from public Information

From above discussion, only type 4 and type 6 would published new information to proceed the identification. However, for players who participate in signing stage, the identification of type 1 aborts have used their own decryption key of Paillier cryptosystem, which could not be published to outside players. Therefore, type 1 aborts need players to give more information for verification.

1. We would give proof of MtA and MtAwc separately, since γ_j is totally different with w_j in many ways:
   - **ZK proof of MtA Failed** For MtA in Phase 2, the message from P_j to P_i contains the ciphertext c_B and the corresponding ZK proof of $c_B = \gamma_j \times_E E_i(k_i) +_E E_1(\beta'_{ij}) \overset{?}{=} E_i(k_i\gamma_j + \beta'_{ij})$. The inside logic of previous ZK proof is that
     (a) P_i first decrypt the ciphertext from P_j to get the plaintext $m_B = D_i(c_B) = k_i\gamma_j + \beta'_{ij}$ and generate an exponent g^m
     (b) P_i using its secret k_i and information from discrete log proof of γ_j and β'_{ij} to re-construct an exponent $g^{m'} = (g^{\gamma_j})^{k_i} \cdot g^{\beta'_{ij}}$. Then check the equality between g^m and $g^{m'}$.

     Obviously, if the proof failed, outside player could not verify the proof because it does not have the decryption key of P_i to decrypt message c_B sent by P_j to P_i. However, recall the solution of type 4 aborts, each player publishes its values $k_i, \gamma_j, \alpha_{ij}, \beta_{ij}$ as well as the randomness used to encrypt these values during the MtA protocol. So, it is easy for outside players to compute $m = k_i\gamma_j + \beta'_{ij}$ and re-encrypt the message m by using the same encryption key E_i and the same randomness and check whether $E_i(m) \overset{?}{=} c_B$.
   - **ZK proof of MtAwc Failed** Similarly, P_i could know g^{w_j} and $g^{\nu_{ij}}$ from the discrete log proof in message c_B, while P_j would give a ZK proof of c_B. If P_i wants to verify the proof, he need to use his decryption key, which outside players do not know.
     Recall the solution of type 6 aborts, let each player P_j publishes the exponent $g^{m'} = g^{w_j k_i + \nu'_{ij}}$, whose distribution is computationally indistinguishable due to discrete log assumption. Then if an outside player want to identify aborts, P_O should verify 2 parts:
     (a) P_O should verify the consistency between m' and the plaintext m, which is the plaintext encrypted in Phase 2 send from P_j to P_i, $E_i(m) = c_B =$. The procedure of proof works as follows:
     Encryption: $E_i(m) = g^m \cdot r^n \ mod \ n^2$, where encryption key is (g, n) and r is randomness.
     Verification: Check the equation (16) holds or not (r_i denote as P_i's published randomness, $E_i = (g, N_i)$:
     $$\frac{c_B}{g^{m'}} = \frac{g^m \cdot r_i^{N_i}}{g^{m'}} \overset{?}{=} r_i^{N_i} \ mod \ N_i^2 \tag{16}$$
     (b) When 1 is correct, using the public information to check whether equation holds:
     $$g^{m'} \overset{?}{=} (g^{w_j})^{k_i} \cdot g^{\nu'_{ij}} \tag{17}$$
     Clearly, only if $m' = m$, then the equation would hold, which means the published $g^{m'}$ is consistent with previous encrypted message m in Phase 2.

2. For type 2 and type 3 aborts in Phase 4, it should check the consistency between the decommitted value Γ_i and g^{γ_i} in discrete log proof in commitment at first, then it should verify the consistency between commitment C_i and decommitment D_i. Since the C_i and D_i are public, outside players can identify aborts easily.

3. For type 4 abort in Phase 5, recall the solution for type 4 abort among the inside players, they just published some parameters to check equation (13) holds or not. From the phase 3 in signing protocol, δ_i has also been broadcasted. Therefore, outside players can simply use these values to find that if any inside player P_i does not hold the equation (13), it would be the one who caused the abort.

4. The verification of consistency between $\overline{R}_i$ and $E_i(k_i)$ need to know the values of $\overline{R}_i, R, E_i$ and corresponding message c_B, P_i received in MtA protocol at Phase 2, and also discrete log proofs. Apparently, such values are broadcasted in the procedure of signing protocol, which means outside players could easily get such values and verify the proof provided from each inside player P_i.

5. For type 6 abort in Phase 6, recall the solution for type 4 abort among the inside players, they just published k_i, μ_{ij}. Thus, it is easy for outside players to compute $g^{\nu_{ij}}$ using same way as equation (14) and then get g^{σ_i} by using equation (15). Finally, for the consistency proof between g^{σ_i} and R^{σ_i}, we do not need another information, and just run the same verification algorithm as inside players ran to identify the abort.

6. For the zk proof of consistency between S_i and T_i, the verification need values of T_i, S_i and R, which are public in phase 3, 4 and phase 6. So, outside players can use such broadcasted values to run the verification of such proofs to identify aborts.

7. If the signature is invalid, recall the inside identification procedure, they need check the equation (11) for each player. While the values of $R, s_i, m, \overline{R}_i$, S_i, r are broadcasted in phase 4,5,6 and 7, outside players can do same equation check for each inside player P_i and find who cause aborts.

5 Reference

1 S. Goldfeder, C. Tech, and O. Labs, "One Round Threshold ECDSA with Identifiable Abort," pp. 1–31.

2 D. Pointcheval, "The composite discrete logarithm and secure authentication," Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 1751, no. january, pp. 113–128, 2000, doi: 10.1007/978-3-540-46588-1_9.

3 T. F. Dahlin, "Paillier zero-knowledge proof," pp. 3–5, 2016.

4 P. Feldman, “A Practical Scheme for Non-interactive Verifiable Secret Sharing Paul Feldman Massachusetts Institute of Technology,” Network, pp. 427–437, 1987.

5 Y. Lindell, “Fast secure two-party ECDSA signing,” Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 10402 LNCS, pp. 613–644, 2017, doi: 10.1007/978-3-319-63715-0_21.