Threshold ECDSA Pseudocode for Coinbase

1 Overview

This document contains pseudocode for ECDSA threshold signatures based on the recent paper of Gennaro and Goldfeder [4]. The exact details of the protocol presented emerged from conversations with Coinbase as to what best suits their needs (e.g. not necessary to attribute misbehavior to a single server). Throughout the document, we will reference specific design decisions and briefly mention the other options available, should it be of interest at some point for us to elaborate further.

Our goal is for this to be a living document that we update with requested clarifications. During the second phase of this engagement, we will also add pseudocode for a dealerless protocol as well as a protocol to dynamically re-share the key.

1.1 Sequential vs. Concurrent Security

The protocol described here is proven secure under sequential composition (i.e. with signatures issued one at a time sequentially) against a static adversary. We conjecture that the protocol is secure sequentially as well as all known attacks of this form are non-practical, although this is not provable in the current protocol. We could provide a variant of our protocol with Universally Composable (UC) concurrent provable security, although this would take an efficiency hit, and based discussions with Coinbase, we are providing the simple protocol which has been proven secure in the standalone sequential model.

Should this decision change once implemented, please let us know and we can provide a document for the UC protocol.

1.2 On our use of the Fiat-Shamir Heuristic

In our academic paper, we assume that the zero knowledge proofs in the protocol are interactive, but in practice these can be made non-interactive using the Fiat-Shamir heuristic in the random oracle model.

However, there is one irregularity in our use of Fiat-Shamir in that we use it for proofs of knowledge, which means that in the security proof, the simulator must extract the adversary's private values. For interactive proofs this is done via "rewinding". Our proofs are three move sigma protocols, which involve the prover committing, the verifier issuing a challenge, and the prover responding. The sigma protocols have a "special soundness" property which means that if the prover commits to its first message and then responds to two different challenges, the verifier can extract the witness. In a "rewinding" argument, the simulator rewinds the adversary back to its first commitment in the sigma protocol and provides a different challenge. The adversary, unaware that it was rewound, responds allowing the prover to extract the witness.

In Fiat-Shamir proofs, the challenge is not supplied by the adversary but by the Random Oracle, and thus all three steps of the protocol are done by the prover: It commits, queries the random oracle, and responds, and then sends the transcript to the verifier. In order to extract by a typical rewinding argument, it is crucial to be able to rewind back to the point in between when the prover commits and gets a challenge. But if the protocol is fully non-interactive the simulator cannot do that since it can only rewind the adversary to the point before it committed. And at this point, the adversary can change its commitment, preventing extraction.

The simplest way to deal with this is by adding an additional round for the FS proofs – in which the adversary sends and commits to its first message. However, in an effort not to add rounds, we

conjecture that there exists a non black-box extractor that allows us to extract the secret values from the adversary.

We are happy to discuss this more, and if you are not comfortable with this assumption, we can add commitment rounds to the proofs. We also mention that related to the point above: the UC version of the protocol does not extract from the FS proofs and thus this issue doesn't arise.

1.3 Accessibility

Throughout this document, we have often appealed to color in the pseudocode to convey important data and simplify the presentation. Upon request, we would be happy to provide an accessible version of this document that does not use color to convey information.

2 Preliminaries

In this section, we will discuss the primitives on which we rely, and provide pseudocode when appropriate.

2.1 ECDSA signatures

The Digital Signature Algorithm (DSA) was proposed by Kravitz in 1991, and adopted by NIST in 1994 as the Digital Signature Standard (DSS) [1, 6]. ECDSA, the elliptic curve variant of DSA, has become quite popular in recent years, especially in cryptocurrencies. Our focus in this document will be on ECDSA.

We will for completeness cover the details of ECDSA here, and indeed the standard ECDSAVerify function will be compatible with the signatures produced by the threshold signing protocol. We do not include pseudocode as we strongly recommend that you choose an existing centralized ECDSA implementation and ensure that you are compatible with the verifier, and we do not recommend implementing this from scratch as there are numerous open-source high quality implementations.

The Public Parameters consist of a cyclic group $\mathcal{G}$ of prime order q, a generator g for $\mathcal{G}$, a hash function $H : \{0,1\}^* \to Z_q$, and another hash function $H' : \mathcal{G} \to Z_q$.

Key-Gen On input the security parameter λ, outputs a private key x chosen uniformly at random in Z_q, and a public key $y = g^x$ computed in $\mathcal{G}$.

Sig On input an arbitrary message M,

- Compute $m = H(M) \in Z_q$
- choose $k \in_R Z_q$
- compute $R = g^{k^{-1}}$ in $\mathcal{G}$ and $r = H'(R) \in Z_q$
- compute $s = k(m + xr) \bmod q$
- output $\sigma = (r, s)$

Ver On input M, σ and y,

- check that $r, s \in Z_q$
- compute $R' = g^{ms^{-1} \bmod q} y^{rs^{-1} \bmod q}$ in $\mathcal{G}$
- Accept (output 1) iff $H'(R') = r$.

The traditional DSA algorithm is obtained by choosing large primes p, q such that $q|(p-1)$ and setting $\mathcal{G}$ to be the order q subgroup of Z_p^*. In this case the multiplication operation in $\mathcal{G}$ is multiplication modulo p. The function H' is defined as $H'(R) = R \bmod q$.

The ECDSA scheme is obtained by choosing $\mathcal{G}$ as a group of points on an elliptic curve of cardinality q. In this case the multiplication operation in $\mathcal{G}$ is the group operation over the curve. The function H' is defined as $H'(R) = R_x \bmod q$ where R_x is the x-coordinate of the point R.

The specific choice of the curve and hash function H will depend on the implementation, and the threshold signing protocol is agnostic to these choices.

2.2 Cryptographic Hash functions

Aside from the hash function used to hash the ECDSA message, our protocol employs a hash function to be used in the commitment scheme as well as for the Random Oracle for making our zero-knowledge proofs non-interactive using Fiat-Shamir. In the pseudocode, we refer to SHA256, but other cryptographically secure hash functions could be used as well (e.g. SHA3).

Notation. When the pseudocode contains a call to SHA256(a, b, c) we mean that the values a, b, and c should be concatenated with an appropriate delimiter in between them to ensure that the separation between protocol values can never be ambiguous. The delimiter should be a character that can never appear in the hashed values (e.g. \$).

2.3 Fiat Shamir Hash function

In our zero knowledge proofs, we require a hash function to non-interactively compute the Fiat-Shamir challenge. We could use a hash function with an appropriate delimiter as discussed in the previous paragraph, but in order to make encoding simple without need to worry about message lengths, we propose the following simple hash function in which we first hash the internal messages and then combine the intermediate hashes into a single digest. Since the intermediate hashes are fixed length, no delimiters are needed for the final digest, and we indicate this using the simple concatenation operator, $||$.

$h \leftarrow$ FS-HASH$(m_1, m_2, \ldots, m_n)$

1. `For` $i = [1, \ldots, n]$
2. Compute $h_i =$ SHA256(m_i)
3. Compute $h =$ SHA256$(h_1||h_2||\cdots||h_n)$
4. `Return` h

2.4 Network assumptions

We assume that each pair of players is connected by a point-to-point authenticated channel. In the pseudocode, sending a message over this channel is denoted by the P2PSend function, and this will be used when players need to send unique messages to other players.

We also make use of a Broadcast function which is called when players need to send the same message to all other players. We differentiate between two uses of broadcast in this document. When we just write Broadcast we refer to sending an identical authenticated message to every other player, although we do not need any guarantees of a reliable broadcast. In other words, we do not need to guarantee that all players see the same message (although the protocol will abort without a signature if someone sent the wrong message.

We will also make use of a reliable broadcast channel, in particular in the distributed key generation protocol. Since we are in the dishonest majority model, a simple echo broadcast suffices. During an echo broadcast, each players utilizes the point-to-point channel to send each player a hash of its view of the the messages sent from all players. If any party receives an inconsistent hash from some other party, it aborts. When we require echo broadcast in the pseudocode, we will denote this with the function EchoBroadcast.

Importantly, whenever a party aborts, they should send a message to all other players indicating that they have aborted.

2.5 Commitment scheme

Throughout the protocol, we use a commitment scheme so that players can commit to values in a manner that will bind them to their choice but also hide their choice until they choose to reveal

it. We require that the commitment schemes is non-malleable and concurrently secure, roughly meaning that given a set of commitments one cannot compute a commitment to a different but related value. In the Random Oracle model, we can achieve this using the simple "canonical" commitment function.

For the commitment function, we could use a cryptographic hash function and separate the two inputs by an appropriate delimiter. Here, we use HMAC, and in our notation, the first input is the HMAC key and the second input is the message.

$[C, D] \leftarrow \mathsf{Commit}(m)$

1. Choose $r \xleftarrow{\$} \{0,1\}^{256}$ // r is a 256-bit random nonce
2. Compute $C = \mathsf{HMAC}(r, m)$
3. Set $D = (m, r)$
4. `Return` $[C, D]$

$m \leftarrow \mathsf{Open}(C, D = (m, r))$

1. `If` $(D \neq \texttt{nil})$,
2. Compute $C_{\text{check}} = \mathsf{HMAC}(r, m)$
3. `If` $(C_{\text{check}} = C)$
4. `Return` m
5. `Return` $\perp$

2.6 Paillier Cryptosystem

The threshold ECDSA protocol makes heavy use of an encryption scheme that is additively homomorphic modulo a large integer N, and we instantiate it with Paillier's cryptosystem [7].

Let $E_{pk}(\cdot)$ denote the encryption algorithm for $\mathcal{E}$ using public key pk. Given ciphertexts $c_1 = E_{\mathsf{pk}}(a)$ and $c_2 = E_{\mathsf{pk}}(b)$, there is an efficiently computable function $+_E$ such that

$$c_1 +_E c_2 = E_{\mathsf{pk}}(a + b \bmod N)$$

The existence of a ciphertext addition operation also implies a scalar multiplication operation, which we denote by $\times_E$. Given an integer $a \in N$ and a ciphertext $c = E_{\mathsf{pk}}(m)$, then we have

$$a \times_E c = E_{\mathsf{pk}}(am \bmod N)$$

We include the pseudocode for Paillier's cryptosystem, but if a suitable library is identified, it may be preferable to use it rather than re-implement it.

On the security of Paillier. We note that while the Paillier cryptosystem is widely used in practice, it has not been standardized, relies on non-standard assumptions and is less studied than standardized primitives. We recommend instantiating our protocol with Paillier, but following the approach of [2], we can instantiate our protocol using a sub-protocol that makes use of Oblivious Transfer instead of Paillier, at the cost of bandwidth efficiency.

Our recommendation is to use Paillier as we are comfortable with its security and it yields a significantly better overall protocol with respect to the amount of data communicated between signers, but we can provide the alternative if it is desired.

$(pk, sk) \leftarrow$ PaillierKeyGen(1^κ)
1. Choose a 1024-bit prime P
2. Choose a 1024-bit prime Q
3. Compute $N = P \cdot Q$
4. Compute $\lambda(N) = lcm(P-1, Q-1)$
5. Compute $u = \mathsf{L}((N+1)^{\lambda(N)} \bmod N^2, N)^{-1} \bmod N$
6. Compute $\phi(N) = (P-1) \cdot (Q-1)$
7. Set $pk = N$
8. Set $sk = [N, \lambda(N), \phi(N), u]$
9. `Return` (pk, sk)

$c \leftarrow$ PaillierEncrypt(pk, m)
1. Set $N = pk.N$
2. `If` $m \notin \mathbb{Z}_N$, `Return` $\perp$
3. Choose $r \xleftarrow{\$} \mathbb{Z}_N^*$
4. Compute $c = (N+1)^m r^N \bmod N^2$
5. `Return` c

$(c, r) \leftarrow$ PaillierEncryptAndReturnRandomness(pk, m)
1. Set $N = pk.N$
2. `If` $m \notin \mathbb{Z}_N$, `Return` $\perp$
3. Choose $r \xleftarrow{\$} \mathbb{Z}_N^*$
4. Compute $c = (N+1)^m r^N \bmod N^2$
5. `Return` (c, r)

$m \leftarrow$ PaillierDecrypt(sk, c)
1. Set $N = sk.N$
2. `If` $c \notin \mathbb{Z}_{N^2}$, `Return` $\perp$
3. Compute $m = \mathsf{L}(c^{\lambda(N)} \bmod N^2, N) \cdot u \bmod N$
4. `Return` m

$c_3 \leftarrow$ PaillierAdd(pk, c_1, c_2)
1. Set $N = pk.N$
2. `If` $c_1, c_2 \notin \mathbb{Z}_{N^2}$, `Return` $\perp$
3. `Return` $c_1 \cdot c_2 \bmod N^2$

$c_2 \leftarrow$ PaillierMultiply(pk, a, c_1)
1. Set $N = pk.N$
2. `If` $a \notin \mathbb{Z}_N$, `Return` $\perp$
3. `If` $c_1 \notin \mathbb{Z}_{N^2}$, `Return` $\perp$
4. `Return` $c_1^a \bmod N^2$

$c_3 \leftarrow \mathsf{L}(u, N)$
1. `If` $u \notin \mathbb{Z}_{N^2}$, `Return` $\perp$
2. `If` $u \neq 1 \bmod N$, `Return` $\perp$
3. `Return` $(u-1)/N$

Fig. 1: **The Paillier Cryptosystem**. Paillier's cryptosystem is an additively homomorphic encryption scheme, which supports the addition of two ciphertexts and the multiplication of a ciphertext and a scalar. In a standalone instantiation of Paillier's cryptosystem, we would not need the PaillierEncryptAndReturnRandomness function, but it is often useful (and required in our threshold signing protocol) to return the randomness to facilitate proving statements about the ciphertext. In all cases, it is important that the randomness r is kept private and not passed around as part of the ciphertext.

2.7 Shamir's Secret Sharing (SSS) Scheme

The underlying secret sharing scheme used in the threshold signing protocol is Shamir Secret Sharing (SSS) [8]. For a prime q, Shamir's secret sharing allows one to share a secret $x \in Z_q$ by distributed points on a random degree t polynomial $p(\cdot)$ with x as the constant term:

$$p(x) = x + a_1 x + a_2 x^2 + \cdots + a_t x^t \bmod q$$

.

Each player $\mathcal{P}_i$ is associated with a unique non-zero index p_i and the player's share is $p(p_i)$, the evaluation of the polynomial at p_i. Given $t+1$ shares, the polynomial can be reconstructed using *Lagrange interpolation*, and the secret is obtained by evaluating the polynomial at 0.

We now provide pseudocode for both sharing and revealing a secret using Shamir's secret sharing. We note that we will not use Reveal anywhere in our protocol and have only included it for informational completeness, but it does not need to be implemented.

$[x_1, \ldots, x_n] \leftarrow \mathsf{ShamirShare}(x, t, q, [p_1, \ldots, p_n])$

1. `For` $i = [1, \ldots, t]$ // Generate random coefficients
2. Choose $a_i \xleftarrow{\$} \mathbb{Z}_q$
3. `For` $i = [1, \ldots, n]$
4. `If` $p_i = 0$, `Abort`
5. $x_i = x + a_1 p_i + a_2 p_i^2 + \cdots + a_t p_i^t \bmod q$
6. `Return` $[x_1, \ldots, x_n]$

$x \leftarrow \mathsf{Reveal}(q, [p_1, x_1], \ldots, [p_{t+1}, x_{t+1}])$

1. Set $x = 0$
2. `For` $i = [1, \ldots, t+1]$
3. Set $\ell = x_i$
4. `For` $j = [1, \ldots, t+1]$
5. `If` $i = j$, `Continue`
6. Compute $\ell = \ell \times \frac{p_j}{p_j - p_i} \bmod q$
7. Compute $x = x + \ell \bmod q$
8. `Return` x

Fig. 2: **Shamir's secret sharing scheme.** In the ShamirShare algorithm, the dealer generates a random polynomial of degree t and evaluates it at p_i, the index associated with each player $\mathcal{P}_i$. The dealer then distributed the appropriate share to each player, which they are to keep secret. The Reveal algorithm takes any $t+1$-sized subset of shares, reconstructs the polynomial using Lagrange interpolation, and outputs the secret.

2.8 Verifiable Secret Sharing (VSS)

A verifiable secret sharing scheme builds on the idea of a secret sharing scheme and gives the participants the ability to verify that their shares are consistent with public values that the dealer will publish. Looking forward to the protocols described in this document, when ECDSA key generation is done by a trusted dealer, using Shamir's Secret Sharing suffices. When distributed key generation is used, the players will require a verifiable secret sharing scheme.

Feldman's VSS Here, we describe Feldman's secret sharing scheme which supplements Shamir's scheme. The reveal function is unchanged, so we just describe the new sharing function, FeldmanShare which outputs both the public values as well as the individual shares. We also present the FeldmanVerify, a function that allows any player to verify their shares with respect to the public values.

For each coefficient a_i of the polynomial on which the secret is shared, the dealer will also publish $v_i = g^{a_i} \in \mathcal{G}$, where $\mathcal{G}$ is a group of prime order q and g is a generator for that group. These values will allow each player to construct its share in the exponent and check the result.

$[v_0, \ldots, v_t], [x_1, \ldots, x_n] \leftarrow$ FeldmanShare$(g, x, t, q, [p_1, \ldots, p_n])$

1. Compute $v_0 = g^x$ in $\mathcal{G}$
2. `For` $i = [1, \ldots, t]$ // Generate random coefficients
3. Set $a_i \xleftarrow{\$} \mathbb{Z}_q$
4. Compute $v_i = g^{a_i}$ in $\mathcal{G}$
5. `For` $i = [1, \ldots, n]$
6. `If` $p_i = 0$, `Abort`
7. $x_i = x + a_1 p_i + a_2 p_i^2 + \cdots + a_t p_i^t \bmod q$
8. `Return` $[v_0, \ldots, v_t], [x_1, \ldots, x_n]$

`True/False` $\leftarrow$ FeldmanVerify$(g, q, x_i, p_i[v_0, \ldots, v_t])$

1. Set $v = v_0$
2. `For` $j = [1, \ldots, t]$
3. Compute $c_j = p_i^j \bmod q$
4. Compute $v = v \times v_j^{c_j}$ in $\mathcal{G}$
5. If $v = g^{x_i}$, Return `True`
6. Else Return `False`

Fig. 3: **Feldman's verifiable secret sharing scheme.** In the FeldmanShare function, the dealer generates a Shamir share to be distributed to each player, as well as auxiliary public values $[v_0, \ldots, v_t]$) to enable verification. The FeldmanVerify allows an individual player to check the correctness of their share with respect to the public information. Reveal is unchanged from Shamir's scheme and hence omitted here.

2.9 Security with Abort

Since we are in the dishonest majority model, it is not possible to guarantee that the protocol will always successfully generate a signature as some parties may refuse to participate or send incorrect values. The protocol as described here is secure in the presence of aborts. This means that the protocol may abort without a signature, but security is maintained and the adversary will remain unable to forge messages.

In the protocol described here, it is not always possible to identify which player caused the protocol to fail. In [4], there is a variant of the protocol that supports identification of misbehavior. However, this requires a cryptographic broadcast channel or a bulletin board (/blockchain). Based on our discussions, we did not include the identifiability feature in this document.

When a protocol aborts, the pseudocode contains the `Abort` call. When this is called, the signer should broadcast a message that it has aborted and refuse to participate further in the aborted instantiation of the protocol.

2.10 Notation

In the pseudocode, each function invocation marked in red is a call to a sub-function. Values marked in green should be stored persistently for the duration of the signing protocol as they will be needed in subsequent rounds. We also note such values as return parameters of the function in which they are initially computed.

We use the notation $[x_j]_{j \neq i}$ to denote an array of values $[x_1, x_2, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n]$ with the value x_j corresponding to player $\mathcal{P}_j$ for every other player $\mathcal{P}_{j \neq i}$.

Whenever a function takes as input multiple values from different players, we will require that players not proceed before receiving all of those values. In practice, an optimized implementation may allow players to begin locally computing values before they've received all of the input, but under no circumstances should a player send or broadcast any of the resulting values before it has received all of the inputs sent by other players during the previous round.

3 Key Generation with a trusted dealer

We now describe the key generation protocol in the presence of a trusted dealer. The protocol assumes that each player has a unique, public, non-zero index p_i that is assigned to that player. As a practical matter, we recommend assigning each player a sequential index beginning at 1.

The high-level purpose of the key generation procedure is to output (1) a public ECDSA signing key, (2) private key shares for each signer corresponding to the public signing key, (3) Paillier key pairs for each player and (4) trusted parameters for instantiating the zero-knowledge proofs that will be employed during the signing protocol.

As input, the Dealer receives the index of each player, the signing threshold t such that $t+1$ players[1] must participate to sign, and the curve parameters. The dealer also needs to choose the size of the proof modulus, but we have hard-coded suitable parameters.

We note that in the event that multiple key generations are performed with different signing keys, the proof parameters can be reused as long as the dealer is trusted by all of the signers. For this reason we have separated the dealer into two functions, the first of which is a global setup for generating proof parameters and the second is a per-signing group/key setup.

We note that having a trusted dealer generate the proof parameters (i.e. DealerGenerateProofParams()) will lead to a significant performance increase in the signing protocol. By contrast, having the dealer generate the other parameters (DealerKeyGen()) simplifies the key generation but will not simplify the signing protocol. It thus might make sense to restrict the dealer to running DealerGenerateProofParams() which as noted above can be run once for all key generation instances, and use the distributed key generation procedure for all subsequent instantiations.

$(\tilde{N}, h_1, h_2) \leftarrow$ GenerateProofParams()

1. Choose a 1024-bit safe prime P // $P = 2p + 1$ where both P and p are prime
2. Choose a 1024-bit safe prime Q // $Q = 2q + 1$ where both Q and q are prime
3. Compute $\tilde{N} = P \cdot Q$
4. Choose $f \xleftarrow{\$} \mathbb{Z}_{\tilde{N}^*}$
5. Choose $\alpha \xleftarrow{\$} \mathbb{Z}_{\tilde{N}^*}$
6. Compute $h_1 = f^2 \bmod \tilde{N}$ // square f
7. Compute $h_2 = h_1^{\alpha} \bmod \tilde{N}$
8. `Return` $\tilde{N}, h_1, h_2$

$y, [x_1, x_2, \ldots, x_n], [X_1, X_2, \ldots, X_n] \leftarrow$DealerKeyGen$(g, q, [p_1, p_2, \ldots, p_n], t)$

1. Choose $x \xleftarrow{\$} \mathbb{Z}_q$
2. Compute $y = g^x$
3. $[x_1 \ldots, x_n] \leftarrow$ ShamirShare$(x, t, \{p_1, \ldots, p_n\})$
4. `For` $i = [1, \ldots, n]$
5. Compute $X_i = g^{x_i}$
6. Compute $sk_i, pk_i =$ PaillierKeyGen(1^κ) // generate a Paillier key pair for each player
7. `Return` $y, [x_1, x_2, \ldots, x_n], [X_1, X_2, \ldots, X_n]$

Fig. 4: **Threshold key generation with a trusted dealer.** We split the function into two parts: one for creating the proof parameters which can be run once to establish global parameters for all key generations that trust the dealer running it (DealerGenerateProofParams), and one for generating the ECDSA key shares, ECDSA public key, and the Paillier key for each player (DealerKeyGen).

At the end of the key generation, the dealer distributes the following public parameters to each player:

[1] Note that in this document a threshold value of t implies that $t+1$ signer are needed to generate a signature. This notation is consistent with the academic literature, but inconsistent with e.g. Bitcoin multi-signatures in which (k, n) signatures require the participation of k parties (and not $k+1$).

1. The ECDSA public key: y
2. The index of every player: $p_i, \ldots, p_n$
3. The ECDSA public key share for all players: $[X_i, \ldots, X_n]$
4. The Paillier public keys of all players: $[pk_1, \ldots, pk_n]$
5. The global proof parameters: $\tilde{N}, h_1, h_2$

Additionally, for every player $\mathcal{P}_i$, the dealer distributes the following which is to be kept privately by that player:

1. The ECDSA private key share: x_i
2. The Paillier private key: sk_i

We stress that the ECDSA verification algorithm will only need the public key y. The other parameters are required to securely generate the signature using the distributed signing protocol.

4 Distributed Key Generation (DKG)

In this section, we will describe the protocol for distributed key generation, for which no trusted dealer is required. We note that for fresh addresses this is likely always preferable from a security perspective, but the dealer protocol will still be useful for importing legacy addresses.

We also note that the choice of key generation procedure is not isolated from the signing protocol. When a dealer is employed, it can generate trusted parameters for zero-knowledge proofs. In cases where a single player is trying to prove an identical statement to multiple other players, a single proof using the trusted parameters suffices. By contrast, when no trusted dealer is employed, there is no globally trusted set of parameters and each verifier will require its own proof even for the same statement. Thus from a performance viewpoint, the signing protocol will require more bandwidth and take more compute time when instantiated with a DKG.

As before, the protocol assumes that each player has a unique, public, non-zero index p_i that is assigned to that player and known before the DKG begins. As a practical matter, we recommend assigning each player a sequential index beginning at 1. Also input to the DKG is the signing threshold t such that $t + 1$ players must participate to sign, and the curve parameters.

The output of the protocol will be similar as without a DKG, differing only in the proof parameters. The output consists of: (1) a public ECDSA signing key, (2) private key shares for each signer corresponding to the public signing key, (3) Paillier key pairs for each player and (4) per-player parameters for instantiating the zero-knowledge proofs that will be employed during the signing protocol.

Whereas the protocol would take a security parameter, in this document we hardcode recommended concrete parameters. For the Paillier modulus, we recommend using 2048 bits.

In general, a player should never proceed to send any value in a subsequent round until it has received all values from the previous rounds. We use green to denote values that need to be stored for use in future rounds of the protocol. When an input parameter to a round function needs to be stored for future rounds, we denote that using green on its first appearance, but we do not explicitly list it as an input to subsequent round functions (i.e. we assume that it is stored and can be accessed from any subsequent round function.

4.1 Threshold DKG Protocol

$(D_i, sk_i, pk_i, \tilde{N}_i, h_{1i}, h_{2i}, [v_{i0}, \ldots, v_{it}], [x_{i1}, \ldots, x_{in}]) \leftarrow$DistKeyGenRound1$(g, q, i, [p_1, \ldots, p_n], t)$

1. Choose $u_i \xleftarrow{\$} \mathbb{Z}_q$
2. Compute $[v_{i0}, \ldots, v_{it}], [x_{i1}, \ldots, x_{in}] \leftarrow$ FeldmanShare$(g, u_i, t, q, [p_1, \ldots, p_n])$
3. Compute $[C_i, D_i] =$ Commit$([v_{i0}, \ldots, v_{it}])$
4. Compute $sk_i, pk_i =$ PaillierKeyGen(1^κ) // generate a 2048-bit Paillier key pair
5. Choose a 1024-bit safe prime $P_i = 2p_i + 1$ where both P_i and p_i are also prime
6. Choose a 1024-bit safe prime $Q_i = 2q_i + 1$ where both Q_i and q_i are also prime
7. Compute $\tilde{N}_i = P_i \cdot Q_i$
8. Choose $f \xleftarrow{\$} \mathbb{Z}_{\tilde{N}_i^*}$
9. Choose $\alpha \xleftarrow{\$} \mathbb{Z}_{\tilde{N}_i^*}$
10. Compute $\beta = \alpha^{-1} \bmod p_i q_i$
11. Compute $h_{1i} = f^2 \bmod \tilde{N}_i$
12. Compute $h_{2i} = h_1^\alpha \bmod \tilde{N}_i$
13. Compute $\pi_{1i}^{\text{CDL}} \leftarrow$ProveCompositeDL$(g, q, p_i, q_i, h_{1i}, h_{2i}, \alpha, \tilde{N}_i))$
14. Compute $\pi_{2i}^{\text{CDL}} \leftarrow$ProveCompositeDL$(g, q, p_i, q_i, h_{2i}, h_{1i}, \beta, \tilde{N}_i))$
15. EchoBroadcast $C_i, pk_i, \tilde{N}_i, h_{1i}, h_{2i}, \pi_{1i}^{\text{CDL}}, \pi_{2i}^{\text{CDL}}$ to all other players
16. `Return` $D_i, sk_i, pk_i, \tilde{N}_i, h_{2i}, h_{2i}, [v_{i0}, \ldots, v_{it}], [x_{i1}, \ldots, x_{in}]$ // The returned values are stored locally for use in later rounds of the DKG protocol

$() \leftarrow$DistKeyGenRound2$([C_j, pk_j, \tilde{N}_j, h_{1j}, h_{2j}, \pi_{1j}^{\text{CDL}}, \pi_{2j}^{\text{CDL}}]_{j \neq i})$

1. `For` $j = [1, \ldots, n]$
2. `If` $i = j$, `Continue`
3. `If` VerifyCompositeDL$(\pi_{1j}^{\text{CDL}}, g, q, h_{1j}, h_{2j}, \tilde{N}_j) =$ `False`, `Abort`
4. `If` VerifyCompositeDL$(\pi_{2j}^{\text{CDL}}, g, q, h_{2j}, h_{1j}, \tilde{N}_j) =$ `False`, `Abort`
5. P2PSend x_{ij} to player $\mathcal{P}_j$
6. EchoBroadcast D_i to all other players

$(x_i, y, [X_1, \ldots, X_n]) \leftarrow$DistKeyGenRound3$([D_j, x_{ji}]_{j \neq i})$

1. Set $x_i = x_{ii}$
2. `For` $j = [1, \ldots, n]$
3. `If` $i = j$, `Continue`
4. Compute $[v_{j0}, \ldots, v_{jt}] \leftarrow$Open$(C_j, D_j)$
5. `If` $[v_{j0}, \ldots, v_{jt}] = \perp$, `Abort`
6. `If` FeldmanVerify$(g, q, x_{ji}, p_i, [v_{j0}, \ldots, v_{jt}]) =$ `False`, `Abort`
7. Compute $x_i = x_i + x_{ji} \bmod q$
8. `For` $j = [0, \ldots, t]$
9. Set $v_j = 1$
10. `For` $k = [1, \ldots, n]$
11. Compute $v_j = v_j \cdot v_{kj}$ in $\mathcal{G}$
12. Set $y = v_0$
13. `For` $j = [1, \ldots, n]$
14. Set $X_j = y$
15. `For` $k = [1, \ldots, t]$
16. Compute $c_k = p_j^k \bmod q$
17. Compute $X_j = X_j \times v_k^{c_k}$ in $\mathcal{G}$
18. Compute $\pi_i^{\text{PSF}} =$ProvePSF$(sk_i.N, sk_i.\phi(N), y, g, q, p_i)$
19. EchoBroadcast π_i^{PSF} to all other players
20. `Return` $x_i, y, [X_1, \ldots, X_n]$

$() \leftarrow$DistKeyGenRound4$([\pi_j^{\text{PSF}}]_{j \neq i})$

1. `For` $j = [1, \ldots, n]$
2. `If` $i = j$, `Continue`
3. `If` VerifyPSF$(\pi_j^{\text{PSF}}, pk_j.N, y, g, q, p_j) =$ `False`, `Abort`
4. Broadcast `Success`

Fig. 5: **Threshold distributed key generation protocol.**

5 Going from key generation to signing

After the key generation, each player has a key share that will allow them to participate in the signing protocol. But in a (t, n) configuration, the key generation is done with all n players whereas only a subset of $t+1$ players participate in the signing protocol. Once the group of $t+1$ signers has been identified, the players must convert their Shamir private key shares that were output during the signing protocol to additive shares in which the secret is shared additively among the $t+1$ active participants. Doing this is straightforward and simply requires the players to multiply their shares by the Lagrange coefficients of the active signers. We describe the conversion function here. The function is non-interactive and run locally by each player.

$(w_i, [W_1, \ldots, W_{t+1}]) \leftarrow$ ConvertToAdditive$(\mathsf{x}_i, i, [p_1, \ldots, p_{t+1}], [X_1, \ldots, X_{t+1}])$

1. For $j = [1, \ldots, t+1]$ {
2. Set $\ell = 1$
3. For $k = [1, \ldots, t+1]$ {
4. If $j = k$, Continue
5. Compute $\ell = \ell \times \frac{p_k}{p_k - p_j} \bmod q$
6. }
7. Compute $W_j = X_j^{\ell}$ in $\mathcal{G}$
8. If $j = i$ {
9. Compute $w_i = x_i \times \ell \bmod q$
10. }
11. }
12. Return $w_i, [W_1, \ldots, W_{t+1}]$

Fig. 6: **Converting (t, n) key shares to $t+1$ additive shares.** Each player $\mathcal{P}_i$ runs this function locally. Recall that during the key generation protocol, each player obtains a private key share x_i as well as a public key share X_j for every player $\mathcal{P}_j$. The inputs to the function are x_i, the private key share of $\mathcal{P}_i$, the set of indexes for $(t+1)$ players that will actively participate in the signing protocol, as well as their public key shares X_j. The protocol outputs w_i the additive private key share for player $\mathcal{P}_i$ as well as $W_j = g^{w_j}$ for each other player $\mathcal{P}_j$.

6 Threshold signing protocol

Below, we give pseudocode for the signing protocol. The code is symmetric for every player and represents the view point of player $\mathcal{P}_i$.

Although the signing protocol doesn't directly make use of the trusted dealer, the protocol will differ slightly depending on whether a trusted dealer or a DKG was used for the key generation. In particular, having a dealer generated trusted parameters during the key generation protocol simplifies the protocol as it gives a universal set of proof parameters that all players trust. In the absence of a trusted dealer, each player generates their own proof parameters for the zero knowledge proofs (i.e. $\tilde{N}, h_1, h_2$) during the DKG. When proving a statement to other players, player $\mathcal{P}_i$ thus needs to generate a unique proof for each player using the parameters that are provided by and trusted by that player. If there is a trusted dealer, however, all players can use the same proving parameters, reducing the number of proofs.

We note that in the pseudocode we mix together the networking code (i.e. calls to Broadcast and Send) together with the cryptographic code. When organizing the actual code though, it is probably simpler to separate these functionalities by having the cryptographic functions return both the values to store locally and the values to send, and have the calling code take care of the networking/storage.

6.1 Threshold signing with a trusted dealer

We first present the threshold signing protocol that is used for instances in which a trusted dealer generated the key and dealt shares to the players.

$(k_i, \gamma_i, D_i, c_i, r_i) \leftarrow$SignRound1$(w_i, [W_1, \ldots, W_{t+1}], g, q, pk_i, i, \tilde{N}, h_1, h_2)$ // the function parameters are output during KeyGen

1. Choose $k_i \xleftarrow{\$} \mathbb{Z}_q$ // $\mathbb{Z}_q$ are the integers from 0 to $q-1$
2. Choose $\gamma_i \xleftarrow{\$} \mathbb{Z}_q$
3. Compute $\Gamma_i = g^{\gamma_i}$ in $\mathcal{G}$
4. Compute $[C_i, D_i] =$ Commit(Γ_i)
5. Compute $(c_i, r_i) =$ PaillierEncryptAndReturnRandomness(pk_i, k_i)
6. Compute $\pi_i^{\text{Range1}} =$ MtAProveRange1$(g, q, pk_i, \tilde{N}, h_1, h_2, k_i, c_i, r_i)$
7. Broadcast $(C_i, c_i, \pi_i^{\text{Range1}})$ to all other players
8. Return $k_i, \gamma_i, D_i, c_i, r_i$ // The returned values are stored locally for use in later rounds of the signing protocol

$[C_j, \beta_{ji}, \nu_{ji}]_{j\neq i} \leftarrow$SignRound2$([C_j, (c_j, \pi_j^{\text{Range1}})]_{j\neq i})$ // The C_j values are received now, but used in subsequent rounds

1. For $j = [1, \ldots, t+1]$
2. If $i = j$, Continue
3. If MtAVerifyRange1$(\pi_j^{\text{Range1}}, g, q, \tilde{N}, h_1, h_2, c_j) =$ False, Abort
4. Compute $(c_{ji}^{\gamma}, \beta_{ji}, \pi_{ji}^{\text{Range2}}) =$ MtAResponse$(\gamma_i, g, q, pk_j, \tilde{N}, h_1, h_2, c_j)$
5. Compute $(c_{ji}^{w}, \nu_{ji}, \pi_{ji}^{\text{Range3}}) =$ MtAResponse_wc$(w_i, W_i, g, q, pk_j, \tilde{N}, h_1, h_2, c_j)$
6. P2PSend$(c_{ji}^{\gamma}, c_{ji}^{w}, \pi_{ji}^{\text{Range2}}, \pi_{ji}^{\text{Range3}})$ to player $\mathcal{P}_j$
7. Return $[C_j, \beta_{ji}, \nu_{ji}]_{j\neq i}$

$(\delta_i, \sigma_i) \leftarrow$SignRound3$([c_{ij}^{\gamma}, c_{ij}^{w}, \pi_{ij}^{\text{Range2}}, \pi_{ij}^{\text{Range3}}]_{j\neq i})$

1. Compute $\delta_i = k_i\gamma_i \bmod q$
2. Compute $\sigma_i = k_i w_i \bmod q$
3. For $j = [1, \ldots, t+1]$
4. If $i = j$, Continue
5. Compute $\alpha_{ij} =$MtAFinalize$(g, q, sk_i, pk_i, \tilde{N}, h_1, h_2, c_i, c_{ij}^{\gamma}, \pi_{ij}^{\text{Range2}})$
6. If $\alpha_{ij} = \perp$, Abort // $\alpha_{ij} = \perp$ if proof verification failed
7. Compute $\mu_{ij} =$MtAFinalize_wc$(g, q, sk_i, pk_i, \tilde{N}, h_1, h_2, c_i, c_{ij}^{w}, \pi_{ij}^{\text{Range3}}, W_j)$
8. If $\mu_{ij} = \perp$, Abort // $\mu_{ij} = \perp$ if proof verification failed
9. Compute $\delta_i = \delta_i + \alpha_{ij} + \beta_{ji} \bmod q$
10. Compute $\sigma_i = \sigma_i + \mu_{ij} + \nu_{ji} \bmod q$
11. Broadcast δ_i to all other players
12. Return δ_i, σ_i

$\delta \leftarrow$SignRound4$([\delta_j]_{j\neq i})$

1. Set $\delta = \delta_i$
2. For $j = [1, \ldots, t+1]$
3. If $i = j$, Continue
4. Compute $\delta = \delta + \delta_j \bmod q$
5. Broadcast D_i to all other players
6. Return δ

$(r, \bar{R}_i) \leftarrow$SignRound5$([D_j]_{j\neq i})$

1. Compute $R = g^{\gamma_i}$ in $\mathcal{G}$
2. For $j = [1, \ldots, t+1]$
3. If $i = j$, Continue
4. Compute $\Gamma_j =$ Open(C_j, D_j)
5. If $\Gamma_j = \perp$, Abort
6. Else Compute $R = R \cdot \Gamma_j$ in $\mathcal{G}$
7. Compute $R = R^{\delta^{-1}}$ in $\mathcal{G}$
8. Set $r = R_x$ // R_x denotes the x coordinate of the elliptic curve point R
9. Compute $\bar{R}_i = R^{k_i}$
10. Compute $\pi_i^{k^{\text{CONSIST}}} =$ProvePDL$(g, q, R, pk_i, \tilde{N}, h_1, h_2, k_i, \bar{R}_i, c_i, r_i)$
11. Broadcast $(\bar{R}_i, \pi_i^{k^{\text{CONSIST}}})$ to all other players
12. Return $r, \bar{R}_i$

$s_i \leftarrow$SignRound6$(M, [R_j, \pi_j^{k^{\text{CONSIST}}}]_{j\neq i})$

1. Set $V = \bar{R}_i$
2. For $j = [1, \ldots, t+1]$
3. If $i = j$, Continue
4. If VerifyPDL$(\pi_j^{k^{\text{CONSIST}}}, g, q, R, pk_j, \tilde{N}, h_1, h_2, c_j, \bar{R}_j) =$ False, Abort
5. Compute $V = V \cdot \bar{R}_j$ in $\mathcal{G}$
6. If $V \neq g$, Abort
7. Compute $m = H(M) \in \mathbb{Z}_q$ // Hash the message with the hash function used by the centralized ECDSA signer/verifier
8. Compute $s_i = mk_i + r\sigma_i \bmod q$
9. Broadcast s_i to all other players
10. Return s_i

$\sigma \leftarrow$ SignOutput$([s_j]_{j\neq i})$

1. Set $s = s_i$
2. For $j = [1, \ldots, t+1]$
3. If $i = j$, Continue
4. Compute $s = s + s_j \bmod q$
5. Set $\sigma = (r, s)$
6. If ECDSAVerify$(y, \sigma, M) =$ False, Abort
7. Return σ

Fig. 7: **ECDSA threshold signing protocol in the presence of a trusted dealer.**

6.2 Threshold signing in the presence of a DKG

We now present the modified signing protocol when a DKG is used. The protocol is mostly the same as the prior one with the exception that proofs now need to be tailored to each player so that they can verify them with their own parameters.

$(k_i, \gamma_i, D_i, c_i, r_i) \leftarrow$SignRound1$(w_i, g, q, pk_i, i, [\tilde{N}_j, h_{1j}, h_{2j}]_{j\in[1,t+1]})$ // the function parameters are output during KeyGen

1. Choose $k_i \xleftarrow{\$} \mathbb{Z}_q$ // $\mathbb{Z}_q$ are the integers from 0 to $q-1$
2. Choose $\gamma_i \xleftarrow{\$} \mathbb{Z}_q$
3. Compute $\Gamma_i = g^{\gamma_i}$ in $\mathcal{G}$
4. Compute $[C_i, D_i] =$ Commit(Γ_i)
5. Compute $(c_i, r_i) =$ PaillierEncryptAndReturnRandomness(pk_i, k_i)
6. `For` $j = [1, \ldots, t+1]$
7. `If` $i = j$, `Continue`
8. Compute $\pi_{ij}^{\text{Range1}} =$ MtAProveRange1$(g, q, pk_i, \tilde{N}_j, h_{1j}, h_{2j}, k_i, c_i, r_i)$
9. P2PSend π_{ij}^{Range1} to player $\mathcal{P}_j$
10. Broadcast (C_i, c_i) to all other players
11. `Return` $k_i, \gamma_i, D_i, c_i, r_i$ // The returned values are stored locally for use in later rounds of the signing protocol

$[C_j, \beta_{ji}, \nu_{ji}]_{j\neq i} \leftarrow$SignRound2$([C_j, (c_j, \pi_{ji}^{\text{Range1}})]_{j\neq i})$ // The C_j values are received now, but used in subsequent rounds

1. `For` $j = [1, \ldots, t+1]$
2. `If` $i = j$, `Continue`
3. `If` MtAVerifyRange1$(\pi_{ji}^{\text{Range1}}, g, q, \tilde{N}_i, h_{1i}, h_{2i}, c_j) =$ `False`, Abort
4. Compute $(c_{ji}^{\gamma}, \beta_{ji}, \pi_{ji}^{\text{Range2}}) =$ MtAResponse$(\gamma_i, g, q, pk_j, \tilde{N}_j, h_{1j}, h_{2j}, c_j)$
5. Compute $(c_{ji}^{w}, \nu_{ji}, \pi_{ji}^{\text{Range3}}) =$ MtAResponse_wc$(w_i, W_i, g, q, pk_j, \tilde{N}_j, h_{1j}, h_{2j}, c_j)$
6. P2PSend$(c_{ji}^{\gamma}, c_{ji}^{w}, \pi_{ji}^{\text{Range2}}, \pi_{ji}^{\text{Range3}})$ to player $\mathcal{P}_j$
7. `Return` $[C_j, \beta_{ji}, \nu_{ji}]_{j\neq i}$

$(\delta_i, \sigma_i) \leftarrow$SignRound3$([c_{ij}^{\gamma}, c_{ij}^{w}, \pi_{ij}^{\text{Range2}}, \pi_{ij}^{\text{Range3}}]_{j\neq i})$

1. Compute $\delta_i = k_i\gamma_i \bmod q$
2. Compute $\sigma_i = k_i w_i \bmod q$
3. `For` $j = [1, \ldots, t+1]$
4. `If` $i = j$, `Continue`
5. Compute $\alpha_{ij} =$MtAFinalize$(g, q, sk_i, pk_i, \tilde{N}_i, h_{1i}, h_{2i}, c_i, c_{ij}^{\gamma}, \pi_{ij}^{\text{Range2}})$
6. `If` $\alpha_{ij} = \perp$, `Abort` // $\alpha_{ij} = \perp$ if proof verification failed
7. Compute $\mu_{ij} =$MtAFinalize_wc$(g, q, sk_i, pk_i, \tilde{N}_i, h_{1i}, h_{2i}, c_i, c_{ij}^{w}, \pi_{ij}^{\text{Range3}}, W_j)$
8. `If` $\mu_{ij} = \perp$, `Abort` // $\mu_{ij} = \perp$ if proof verification failed
9. Compute $\delta_i = \delta_i + \alpha_{ij} + \beta_{ji} \bmod q$
10. Compute $\sigma_i = \sigma_i + \mu_{ij} + \nu_{ji} \bmod q$
11. Broadcast δ_i to all other players
12. `Return` δ_i, σ_i

$\delta \leftarrow$SignRound4$([\delta_j]_{j\neq i})$

1. Set $\delta = \delta_i$
2. `For` $j = [1, \ldots, t+1]$
3. `If` $i = j$, `Continue`
4. Compute $\delta = \delta + \delta_j \bmod q$
5. Broadcast D_i to all other players
6. `Return` δ

$(r, \bar{R}_i) \leftarrow$SignRound5$([D_j]_{j\neq i})$

1. Compute $R = g^{\gamma_i}$ in $\mathcal{G}$
2. `For` $j = [1, \ldots, t+1]$
3. `If` $i = j$, `Continue`
4. Compute $\Gamma_j =$ Open(C_j, D_j)
5. `If` $\Gamma_j = \perp$, `Abort`
6. `Else` Compute $R = R \cdot \Gamma_j$ in $\mathcal{G}$
7. Compute $R = R^{\delta^{-1}}$ in $\mathcal{G}$
8. Set $r = R_x$ // R_x denotes the x coordinate of the elliptic curve point R
9. Compute $\bar{R}_i = R^{k_i}$
10. `For` $j = [1, \ldots, t+1]$
11. `If` $i = j$, `Continue`
12. Compute $\pi_{ij}^{k\text{CONSIST}} =$ProvePDL$(g, q, R, pk_i, \tilde{N}_j, h_{1j}, h_{2j}, k_i, \bar{R}_i, c_i, r_i)$
13. P2PSend $\pi_{ij}^{k\text{CONSIST}}$ to player $\mathcal{P}_j$
14. Broadcast $\bar{R}_i$ to all other players
15. `Return` $r, \bar{R}_i$

$s_i \leftarrow$SignRound6$(M, [R_j, \pi_{ji}^{k\text{CONSIST}}]_{j\neq i})$

1. Set $V = \bar{R}_i$
2. `For` $j = [1, \ldots, t+1]$
3. `If` $i = j$, `Continue`
4. `If` VerifyPDL$(\pi_{ji}^{k\text{CONSIST}}, g, q, R, pk_j, \tilde{N}_i, h_{1i}, h_{2i}, c_j, \bar{R}_j) =$ `False`, `Abort`
5. Compute $V = V \cdot \bar{R}_j$ in $\mathcal{G}$
6. `If` $V \neq g$, `Abort`
7. Compute $m = H(M) \in \mathbb{Z}_q$ // Hash the message with the hash function used by the centralized ECDSA signer/verifier
8. Compute $s_i = mk_i + r\sigma_i \bmod q$
9. Broadcast s_i to all other players
10. `Return` s_i

$\sigma \leftarrow$ SignOutput$([s_j]_{j\neq i})$

1. Set $s = s_i$
2. `For` $j = [1, \ldots, t+1]$
3. `If` $i = j$, `Continue`
4. Compute $s = s + s_j \bmod q$
5. Set $\sigma = (r, s)$
6. `If` ECDSAVerify$(y, \sigma, M) =$ `False`, `Abort`
7. Return σ

Fig. 8: **The ECDSA threshold signing protocol in the presence of a DKG.**

7 One round signing

Notice that in the pseudocode above, the message M is first input to the SignRound6 function. To achieve non-interactive signing, the players simply run the first five rounds of the protocol offline, and indeed they can run many such instantiations in parallel (see introduction regarding concurrent security in our protocol). Then later upon receiving the message to sign, they can complete the protocol by beginning from SignRound6, leading to only a single round in which every player broadcasts a single message.

Indeed, when executing the protocol in the manner, even part of SignRound6 can be performed during the pre-processing phase, and doing so will require the players to store less information for the online phase. We now break up SignRound6 into two functions: one which can be run offline during the pre-processing, and one which is message dependent.

$r, k_i, \sigma_i \leftarrow$SignRound6Offline$([R_j, \pi_{ji}^{k\text{CONSIST}}]_{j \neq i})$

1. Set $V = \bar{R}_i$
2. `For` $j = [1, \ldots, t+1]$
3. `If` $i = j$, `Continue`
4. `If` VerifyPDL$(\pi_j^{k\text{CONSIST}}, g, q, R, pk_j, \tilde{N}_i, h1_i, h2_i, c_j, \bar{R}_j)$ = `False`, `Abort`
5. Compute $V = V \cdot \bar{R}_j$ in $\mathcal{G}$
6. `If` $V \neq g$, `Abort`
7. `Return` r, k_i, σ_i

$s_i \leftarrow$SignRound6Online(M, r, k_i, σ_i)

1. Compute $m = H(M) \in \mathbb{Z}_q$ // Hash the message with the hash function used by the centralized ECDSA signer/verifier
2. Compute $s_i = mk_i + r\sigma_i \bmod q$
3. Broadcast s_i to all other players
4. `Return` s_i

Fig. 9: **Modification for one round signing.** To achieve one round signing, the first five rounds and part of the sixth round can be run during an offline message-independent pre-processing phase. Here we split SignRound6 into two functions showing which parts can be run during pre-processing, and which must be run online once the message is known. Notice that the values returned by SignRound6Offline are the values that must be stored persistently for the online phase, and indeed these are listed as function parameters to SignRound6Online. We use the color orange to denote pseudocode that only applies for the DKG setting, although the difference is quite minor in the functions shown here.

To execute the protocol with one online round, the first five rounds as well as SignRound6Offline are run during a pre-processing phase. Then, during the online phase, only SignRound6Online and SignOutput are run. Notice that the only values that need to be kept for the online round are r, k_i, σ_i. All other values can be discarded.

Keeping state. We stress that just like during the fully online protocol, we require an independent execution of the protocol for each generated signature. The values r, k_i, σ_i must only be ever used in one call of the signing protocol (and even if the protocol aborts, these values must not be re-used).

For safety, we've presented the protocol such that the pre-processing phase is done with a specific set of $t + 1$ signers. This minimizes the chance that a pre-processed value tuple will be accidentally re-used since every signer needs to participate and thus they cannot be partitioned such that different subgroups re-use a tuple. Nevertheless, much care should be taken so that every signers discards the tuple after it is used (no matter whether a signature was successfully generated or the protocol aborted).

8 Multiplicative-to-Additive Share Conversion Protocol (MtA)

Perhaps the most involved part of the signing protocol is the sub-protocol for converting multiplicative secret shares to additive shares of their product. We present the details of the MtA protocol here as well as complete pseudocode.

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 $x = ab \bmod q$. 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 $\alpha + \beta = x = ab \bmod q$.

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 $\hat{a}, \hat{b}$. 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").

In the protocol, $\mathcal{P}_1$ first encrypts the value a under its own key and sends the ciphertext c_1 to $\mathcal{P}_2$. $\mathcal{P}_2$ then uses the homomorphic property of the encryption scheme to multiply its value b into the ciphertext c_1 and add a mask β' resulting in a new ciphertext c_2 that it sends back to $\mathcal{P}_1$. $\mathcal{P}_1$ decrypts the ciphertext to reveal its share α, and $\mathcal{P}_2$ sets its share to $\beta = -\beta'$.

Note that $\alpha = \mathsf{PaillierDecrypt}(pk_1, c_2) = a \cdot b + \beta'$ and thus $\alpha + (\beta) = a \cdot b$ as desired.

8.1 MtA and MtAwc initiation

In the first phase of the MtA protocol the initiator encrypts its message a and sends the ciphertext c together with a range proof π^{Range1} to the other player.

In this document, we present the range proof here as a sub-function, but we implement the Paillier encryption of the first message directly into the first round of the signing protocol (i.e. SignRound1). We do this primarily for clarity and the ability to write generic functions that will work both for the dealer and dealerless protocols, since in the dealerless version of our protocol we require a single ciphertext for all players, but a unique proof for every other player.

We now present pseudocode for the prover and verifier functions for the initiator's range proof.

As one of the inputs to both MtAProveRange1/MtAVerifyRange1, the prover's Paillier's public key is supplied. Similarly, in MtAResponse/MtAProveRange2/MtAVerifyRange2 the verifier's public key is supplied. In practice this is N, but for clarity, we refer to it as pk in the function header and use the notation $pk.N$ when accessing N to demonstrate that N is stored as part of the respective player's public key.

$\pi \leftarrow \mathsf{MtAProveRange1}(g, q, pk, \tilde{N}, h_1, h_2, a, c, r)$

1. Set $N = pk.N$
2. Choose $\alpha \xleftarrow{\$} \mathbb{Z}_{q^3}$
3. Choose $\beta \xleftarrow{\$} \mathbb{Z}_N^*$
4. Choose $\gamma \xleftarrow{\$} \mathbb{Z}_{q^3\tilde{N}}$
5. Choose $\rho \xleftarrow{\$} \mathbb{Z}_{q\tilde{N}}$
6. Compute $z = h_1^a h_2^\rho \bmod \tilde{N}$
7. Compute $u = (N+1)^\alpha \beta^N \bmod N^2$
8. Compute $w = h_1^\alpha h_2^\gamma \bmod \tilde{N}$
9. Compute $e = \mathsf{FS\text{-}HASH}(g, q, pk, \tilde{N}, h_1, h_2, c, z, u, w)$
10. Compute $s = r^e \beta \bmod N$
11. Compute $s_1 = ea + \alpha$ // computed over the integers
12. Compute $s_2 = e\rho + \gamma$ // computed over the integers
13. Set $\pi = [z, e, s, s_1, s_2]$
14. Return π

`True/False` $\leftarrow \mathsf{MtAVerifyRange1}(\pi = [z, e, s, s_1, s_2]), g, q, pk, \tilde{N}, h_1, h_2, c)$

1. Set $N = pk.N$
2. If $s_1 > q^3$, `Return False` // Check range
3. Compute $\hat{u} = (N+1)^{s_1} s^N c^{-e} \bmod N^2$
4. Compute $\hat{w} = h_1^{s_1} h_2^{s_2} z^{-e} \bmod \tilde{N}$
5. Compute $\hat{e} = \mathsf{FS\text{-}HASH}(g, q, pk, \tilde{N}, h_1, h_2, c, z, \hat{u}, \hat{w})$
6. If $\hat{e} \neq e$, `Return False`
7. `Return True`

Fig. 10: **Initiator proof in the MtA and MtAwc protocols.** This proof is run by the initiator and is identical in both the MtA and MtAwc protocols.

8.2 MtA Response

We now describe the MtA/MtAwc procedures for the respondent that is executed upon receiving the proof and ciphertext from the initiator. The respondent first verifies the received proof (using the procedure from Section 8.1) and then crafts its own ciphertext as well as its own proof using

the MtAResponse and MtAResponse_wc functions. The respondent's functions for MtA and MtAwc are quite similar but subtly different. To avoid duplicating text, we mark in blue the parts of the protocol that are only run during MtAwc.

$(c_2, \beta, \pi^{\mathsf{Range2}}) \leftarrow$ MtAResponse_wc$(b, B, g, q, pk, \tilde{N}, h_1, h_2, c_1)$

1. Compute $c_b =$ PaillierMultiply(pk, b, c_1)
2. Set $N = pk.N$
3. Choose $\beta' \xleftarrow{\$} \mathbb{Z}_N$
4. Compute $(c_\beta, r_\beta) =$ PaillierEncryptAndReturnRandomness(pk, β')
5. Compute $c_2 =$ PaillierAdd(pk, c_b, c_β)
6. Compute $\beta = -\beta' \bmod q$
7. Compute $\pi^{\mathsf{Range2}} =$ MtAProveRange2_wc $(g, q, pk, \tilde{N}, h_1, h_2, b, \beta', r_\beta, c_1, c_2, B)$
8. `Return` $c_2, \beta, \pi^{\mathsf{Range2}}$

Fig. 11: **The MtA and MtAwc response function.** This function is run by the respondent in the MtA protocol after successfully verifying the range proof sent by the initiator. By running this function, the respondent obtains the ciphertext c_2 and the range proof π^{Range2} that he sends back to the initiator as well as the private additive share β which is kept secret and not sent. The protocols for MtA and MtAwc are nearly identical with the only different being that in MtA, the caller invokes the MtAProveRange2 sub-function, whereas in MtAwc, the caller invokes the MtAProveRange2_wc sub-function, which takes an extra parameter: $B = g^b$. To avoid duplicating text, we mark in blue the parts of the protocol that are only run during MtAwc.

8.3 The Respondent's zero knowledge proof

We now present the proving and verification pseudocode for the zero knowledge proofs run by the respondent. As before, we mark in blue the parts of the protocol that are only run during MtAwc. We use the notation from [3] for this proof, but note that x in this proof corresponds to b in the protocol, X corresponds to B, and y corresponds to β.

$\pi \leftarrow$ MtAProveRange2_wc$(g, q, pk, \tilde{N}, h_1, h_2, x, y, r, c_1, c_2, X)$

1. Set $N = pk.N$
2. Choose $\alpha \xleftarrow{\$} \mathbb{Z}_{q^3}$
3. Choose $\rho \xleftarrow{\$} \mathbb{Z}_{q\tilde{N}}$
4. Choose $\rho' \xleftarrow{\$} \mathbb{Z}_{q^3\tilde{N}}$
5. Choose $\sigma \xleftarrow{\$} \mathbb{Z}_{q\tilde{N}}$
6. Choose $\beta \xleftarrow{\$} \mathbb{Z}_N^*$
7. Choose $\gamma \xleftarrow{\$} \mathbb{Z}_N^*$
8. Choose $\tau \xleftarrow{\$} \mathbb{Z}_{q\tilde{N}}$
9. Compute $u = g^{\alpha}$
10. Compute $z = h_1^x h_2^{\rho} \bmod \tilde{N}$
11. Compute $z' = h_1^{\alpha} h_2^{\rho'} \bmod \tilde{N}$
12. Compute $t = h_1^y h_2^{\sigma} \bmod \tilde{N}$
13. Compute $v = c_1^{\alpha}(N+1)^{\gamma}\beta^N \bmod N^2$
14. Compute $w = h_1^{\gamma} h_2^{\tau} \bmod \tilde{N}$
15. Compute $e =$ FS-HASH$(g, q, pk, \tilde{N}, h_1, h_2, X, c_1, c_2, u, z, z', t, v, w)$
16. Computes $s = r^e \beta \bmod N$
17. Compute $s_1 = ex + \alpha$ // computed over the integers
18. Compute $s_2 = e\rho + \rho'$ // computed over the integers
19. Compute $t_1 = ey + \gamma$ // computed over the integers
20. Compute $t_2 = e\sigma + \tau$ // computed over the integers
21. Set $\pi = [z, z', t, e, s, s_1, s_2, t_1, t_2]$
22. `Return` π

`True/False` $\leftarrow$ MtAVerifyRange2_wc$(\pi = [z, z', t, e, s, s_1, s_2, t_1, t_2], g, q, pk, \tilde{N}, h_1, h_2, c_1, c_2, X)$

1. Set $N = pk.N$
2. `If` $s_1 > q^3$, `Return False` // check range
3. Compute $s_1' = s_1 \bmod q$
4. Compute $\hat{u} = g^{s_1'} \cdot X^{-e}$ in $\mathcal{G}$
5. Compute $\hat{z}' = (h_1)^{s_1} \cdot (h_2)^{s_2} z^{-e} \bmod \tilde{N}$
6. Compute $\hat{v} = (c_1)^{s_1} \cdot s^N \cdot (N+1)^{t_1} \cdot c_2^{-e} \bmod N^2$
7. Compute $\hat{w} = (h_1)^{t_1} \cdot (h_2)^{t_2} \cdot t^{-e} \bmod \tilde{N}$
8. Compute $\hat{e} =$ FS-HASH$(g, q, pk, \tilde{N}, h_1, h_2, X, c_1, c_2, \hat{u}, z, \hat{z}', t, \hat{v}, \hat{w})$
9. `If` $\hat{e} \neq e$, `Return False`
10. `Return True`

Fig. 12: **Respondent proof in the** MtA **and** MtAwc **protocols.** This figure shows the proofs for both the MtA and MtAwc protocol. In both, the proof includes a range proof, but in the MtAwc there is an additional check for consistency with $X = g^x$. The proofs are mostly the same, with some extra checks added in the MtAwc checks. For simplicity, we have marked the items in blue that are only run during MtAwc and omitted during MtA.

8.4 The initiator's finalization function

In the last phase of the MtA/MtAwc protocols, upon receiving the respondent's message, the initiator checks the zero knowledge proof and calculates its own additive secret share. We present the pseudocode for this here.

$\alpha \leftarrow$ MtAFinalize_wc$(g, q, sk, pk, \tilde{N}, h_1, h_2, c_1, c_2, \pi^{\text{Range2}}, B)$

1. `If` MtAVerifyRange2_wc$(\pi^{\text{Range2}}, g, q, pk, \tilde{N}, h_1, h_2, c_1, c_2, B) =$ `False`, `Return` $\perp$
2. Compute $\alpha =$ Decrypt$(sk, c_2) \bmod q$
3. `Return` α

Fig. 13: **The** MtA **finalization function.** Upon receiving $(c_2, \pi^{\text{Range2}})$, $\mathcal{P}_1$ calls MtAFinalize to check the proof's validity compute its additive share α.

9 The zero knowledge proof in SignRound5

In SignRound5 the protocol requires a zero knowledge proof of consistency between a discrete logarithm and a value encrypted in a Paillier ciphertext. In the protocol this is used to prove that the value k_i that a player input to the MtA protocol is the same k_i that they used to compute $\bar{R} = R^{k_i}$.We present the pseudocode for this proof here. In particular, we need to show consistency between a ciphertext c such that $(c, r) = \mathsf{PaillierEncryptAndReturnRandomness}(pk, x)$ and a curve point X such that $X = R^x$.

$\pi \leftarrow \mathsf{ProvePDL}(g, q, R, pk, \tilde{N}, h_1, h_2, x, X, c, r)$

1. Set $N = pk.N$
2. Choose $\alpha \xleftarrow{\$} \mathbb{Z}_{q^3}$
3. Choose $\beta \xleftarrow{\$} \mathbb{Z}_N^*$
4. Choose $\gamma \xleftarrow{\$} \mathbb{Z}_{q^3 \tilde{N}}$
5. Choose $\rho \xleftarrow{\$} \mathbb{Z}_{q\tilde{N}}$
6. Compute $u = R^\alpha$ in G
7. Compute $z = h_1^x h_2^\rho \bmod \tilde{N}$
8. Compute $v = (N+1)^\alpha \beta^N \bmod N^2$
9. Compute $w = h_1^\alpha h_2^\gamma \bmod \tilde{N}$
10. Compute $e = \mathsf{FS\text{-}HASH}(pk, \tilde{N}, h_1, h_2, g, q, R, X, c, u, z, v, w)$
11. Computes $s = r^e \beta \bmod N$
12. Compute $s_1 = ex + \alpha$ // computed over the integers
13. Compute $s_2 = e\rho + \gamma$ // computed over the integers
14. Set $\pi = [z, e, s, s_1, s_2]$
15. `Return` π

`True/False` $\leftarrow \mathsf{VerifyPDL}(\pi = [z, e, s, s_1, s_2], g, q, R, pk, \tilde{N}, h_1, h_2, c, X)$

1. Set $N = pk.N$
2. `If` $s_1 > q^3$, `Return False` // check range
3. Compute $s_1' = s_1 \bmod q$
4. Compute $\hat{u} = R^{s_1'} \cdot X^{-e}$ in $\mathcal{G}$
5. Compute $\hat{v} = s^N \cdot (N+1)^{s_1} \cdot c^{-e} \bmod N^2$
6. Compute $\hat{w} = (h_1)^{s_1} \cdot (h_2)^{s_2} \cdot z^{-e} \bmod \tilde{N}$
7. Compute $\hat{e} = \mathsf{FS\text{-}HASH}(pk, \tilde{N}, h_1, h_2, g, q, R, X, c, \hat{u}, z, \hat{v}, \hat{w})$
8. `If` $\hat{e} \neq e$, `Return False`
9. `Return True`

Fig. 14: **Consistency proof between a Paillier encrypted value and a discrete logarithm.** This figure shows the proof of consistency between a ciphertext c such that $(c, r) = \mathsf{PaillierEncryptAndReturnRandomness}(pk, x)$ and a curve point X such that $X = R^x$.

10 The zero knowledge proofs used in the DKG

In the DKG setting, we cannot trust that the Paillier keys and proof parameters were honestly generated, and we therefore require two additional proofs, the details of which we present here.

10.1 Proof that a Paillier modulus is square free

We now present the proof that a Paillier modulus is square free using the proof from [5]. We note that the proof in its interactive setting begins with a set of challenges $x_i \in \mathbb{Z}_N^*$ supplied by the verifier. However, we want this proof to be non-interactive and thus the prover will supply these challenges. To achieve this, we use a technique that is similar to Fiat-Shamir except since

these challenges are the very first message of the protocol, we must use some external input to the random oracle. In our protocol, we will use y, the ECDSA public key as well as the player's index p_i, and indeed this is the reason that in the DKG we delay providing this proof until the final round when y has been computed.

The exact parameters of the proof can be fine-tuned and have a computation-communication cost trade-off. In particular, the soundness of the proof is related to the smallest prime factors of the modulus, and in particular is $1/d$ where d is the smallest prime factor. Thus, the proof begins by checking that there are "small" factors, and the number of parallel instances needed for soundness directly relates to how high we check. For a security parameter of κ, where we check for prime factors up to t, we need ℓ parallel instances where ℓ is the smallest integer such that $t^\ell > 2^\kappa$. You can choose any suitable parameters here, but in the pseudocode, we use 128-bit security. $t = 1000$ and $\ell = 13$.

$\pi \leftarrow$ProvePSF$(N, \phi(N), y, g, q, p_i)$

1. Set $\ell = 13$
2. Compute $M = N^{-1} \bmod \phi(N)$
3. Compute $[x_1, \ldots, x_\ell] \leftarrow$ GenerateChallenges(g, q, y, p_i, ℓ)
4. `For` $j = [1, \ldots, \ell]$
5. Compute $y_i = x_i^M \bmod \phi(N)$
6. Set $\pi = [y_1, \ldots, y_t]$
7. `Return` π

`True/False` $\leftarrow$ VerifyPSF$(\pi = [y_1, \ldots, y_t], N, y, g, q, p_i)$

1. Set $\ell = 13$
2. Set $t = 1000$
3. `If` $q|N$, `Return False`
4. Compute $[x_1, \ldots, x_\ell] \leftarrow$ GenerateChallenges(g, q, y, p_i, ℓ)
5. `For` $j = [1, \ldots, \ell]$
6. `If` $y_i^N \neq x_i \bmod N$, `Return False`
7. `Return True`

$[x_1, \ldots, x_\ell] \leftarrow$ GenerateChallenges(g, q, y, p_i, ℓ)

1. Set $b = |N|$ // bit length of N
2. Set $h =$ output bit-length of FS-HASH
3. Compute $s = \lceil b/h \rceil$ // number of hash outputs required to obtain b bits
4. Set $j = 0$
5. Set $m = 0$
6. `While` $j \leq \ell$
7. `For` $k = [1, \ldots, s]$
8. Compute $e_{jk} =$ FS-HASH(g, q, y, p_i, j, k, m)
9. Set $x_j = e_{j1} || \cdots || e_{js}$ // where $||$ denotes concatenation
10. Truncate e_j to b bits //Remove the excess $s \cdot h - b$ bits
11. `If` $x_j \in Z_N^*$
12. Set $j = j + 1$
13. Set $m = 0$
14. `Else`
15. Set $m = m + 1$
16. `Return` $[x_1, \ldots, x_\ell]$

Fig. 15: **Zero-knowledge proof that a Paillier modulus is square free.**

10.2 Proof that h_1, h_2 generate the same group

Also in the DKG, we will need to prove that h_1 and h_2 generate the same group modulo $\tilde{N}$. To achieve this, we will run two proofs in parallel showing knowledge of the discrete log of h_1 with respect to h_2 and vice versa. This proof uses binary challenges, and we will do 128 instances in parallel to achieve soundness.

$\pi \leftarrow$ProveCompositeDL$(g, q, p_i, q_i, h_1, h_2, x, N))$

1. Set $\ell = 128$
2. `For` $i = [1, \ldots, \ell]$
3. Choose $\alpha_i \xleftarrow{\$} \mathbb{Z}_{p_i q_i}$
4. Compute $u_i = h^{\alpha_i} \bmod N$
5. Compute $e =$ FS-HASH$(g, q, N, h_1, h_2, [u_1, \ldots, u_\ell])$
6. `For` $i = [1, \ldots, \ell]$
7. Compute $s_i = \alpha_i + e_i x \bmod pq$ // where e_i denotes the ith bit of e
8. Set $\pi = [(u_1, s_1), \ldots, (u_\ell, s_\ell)]$
9. `Return` π

`True/False` $\leftarrow$ VerifyCompositeDL$(\pi = [(u_1, s_1), \ldots, (u_\ell, s_\ell)], g, q, h_1, h_2, N)$

1. Set $\ell = 128$
2. Compute $e =$ FS-HASH$(g, q, N, h_1, h_2, [u_1, \ldots, u_\ell])$
3. `For` $i = [1, \ldots, \ell]$
4. `If` $h_1^{s_i} \neq u_i h_2^{e_i}$, `Return False` // where e_i denotes the ith bit of e
5. `Return True`

Fig. 16: **Proof of knowledge of a discrete log modulo a composite.**

11 Resharing: updating the DKG

In this section, we describe a simple interactive procedure for updating the threshold key generation and "handing-off" the key shares to a new committee. This function is quite flexible and it allows for increasing/decreasing both the threshold (i.e., t) as well as the number of signers (i.e., n). The crucial property of the resharing procedure is that while the underlying sharing of the key is updated, the key itself remains the same.

It must be stressed that any resharing/hand-off procedure necessarily requires that (a sufficiently large subset of) the old signers delete their key shares. In particular, the procedure described here shows how to securely hand off the key to the old signers, but there's no way that we could possibly "invalidate" the old shares other than deleting them.

The procedure described here assumes that there is an existing key that is shared among n players $\mathcal{P}_1, \ldots, \mathcal{P}_n$ with a signing threshold of t, and these players will hand off the key to a new set of $\tilde{n}$ players, $\tilde{\mathcal{P}}_1, \ldots, \tilde{\mathcal{P}}_{\tilde{n}}$ using a signing threshold of $\tilde{t}$. The two sets of players may be entirely disjoint representing a change of the entire set of players, or more likely there will be overlap between the two sets with some new players added or some old players removed. It is also possible that the two sets of players are identical, and this procedure is run just to change the threshold t.

Notice that parts of the distributed key generation that don't directly relate to the shared ECDSA key (e.g. generating proof parameters and Paillier keys) are also run in the resharing procedure mostly unmodified.

Note that during the procedure, the entire set of new players $\tilde{\mathcal{P}}_1, \ldots, \tilde{\mathcal{P}}_{\tilde{n}}$ must be active, but only a threshold number of old players $\mathcal{P}_1, \ldots, \mathcal{P}_{\tilde{t}+1}$ participate. The pseudocode below will have two types of functions – those that are run by the old set of players which will be prepended with Old_ and those run by the new set of players which will be prepended with New_. We assume that every player on the old set and the new set can communicate via point-to-point channels as well as via broadcasting a message to either or both sets as will be specified in the pseudocode via EchoBroadcastToNew and EchoBroadcastToOld.

$(D_i, [v_{i0}, \ldots, v_{i\tilde{t}}], [s_{i1}, \ldots, s_{i\tilde{n}}]) \leftarrow$ Old_ReshareRound1$(g, q, y, i, x_i, [X_1, \ldots, X_{t+1}], [p_1, \ldots, p_{t+1}], [\tilde{p}_1, \ldots, \tilde{p}_{\tilde{n}}], \tilde{t})$

1. Compute $w_i, \_\_ \leftarrow$ ConvertToAdditive$(x_i, i, [p_1, \ldots, p_{t+1}], [X_1, \ldots, X_{t+1}])$
2. Compute $[v_{i0}, \ldots, v_{i\tilde{t}}], [s_{i1}, \ldots, s_{i\tilde{n}}] \leftarrow$ FeldmanShare$(g, w_i, \tilde{t}, q, [\tilde{p}_1, \ldots, \tilde{p}_{\tilde{n}}])$
3. Compute $[C_i, D_i] =$ Commit$([v_{i0}, \ldots, v_{i\tilde{t}}])$
4. EchoBroadcast C_i to all players in the new set $[\tilde{P}_1, \ldots, \tilde{P}_{\tilde{n}}]$
5. `Return` $D_i, [v_{i0}, \ldots, v_{i\tilde{t}}], [s_{i1}, \ldots, s_{i\tilde{n}}]$

$(\tilde{sk}_i, \tilde{pk}_i, \tilde{N}_i, \tilde{h}_{1i}, \tilde{h}_{2i}) \leftarrow$ New_ReshareRound1$(y, [C_j]_{j\in[1,t+1]}, g, q, i, [p_1, \ldots, p_{t+1}], [\tilde{p}_1, \ldots, \tilde{p}_{\tilde{n}}])$

1. Compute $\tilde{sk}_i, \tilde{pk}_i =$ PaillierKeyGen(1^κ) // generate a 2048-bit Paillier key pair
2. Choose a 1024-bit safe prime $\tilde{P}_i = 2\tilde{p}_i + 1$ where both $\tilde{P}_i$ and $\tilde{p}_i$ are also prime
3. Choose a 1024-bit safe prime $\tilde{Q}_i = 2\tilde{q}_i + 1$ where both $\tilde{Q}_i$ and $\tilde{q}_i$ are also prime
4. Compute $\tilde{N}_i = \tilde{P}_i \cdot \tilde{Q}_i$
5. Choose $f \xleftarrow{\$} \mathbb{Z}_{\tilde{N}_i^*}$
6. Choose $\alpha \xleftarrow{\$} \mathbb{Z}_{\tilde{N}_i^*}$
7. Compute $\beta = \alpha^{-1} \bmod \tilde{p}_i\tilde{q}_i$
8. Compute $\tilde{h}_{1i} = f^2 \bmod \tilde{N}_i$
9. Compute $\tilde{h}_{2i} = \tilde{h}_1^{\alpha} \bmod \tilde{N}_i$
10. Compute $\pi_{1i}^{\text{CDL}} \leftarrow$ ProveCompositeDL$(g, q, \tilde{p}_i, \tilde{q}_i, \tilde{h}_{1i}, \tilde{h}_{2i}, \alpha, \tilde{N}_i))$
11. Compute $\pi_{2i}^{\text{CDL}} \leftarrow$ ProveCompositeDL$(g, q, \tilde{p}_i, \tilde{q}_i, \tilde{h}_{2i}, \tilde{h}_{1i}, \beta, \tilde{N}_i))$
12. Compute $\pi_i^{\text{PSF}} =$ ProvePSF$(\tilde{sk}_i.N, \tilde{sk}_i.\phi(N), y, g, q, p_i)$
13. EchoBroadcast $\tilde{pk}_i, \tilde{N}_i, \tilde{h}_{1i}, \tilde{h}_{2i}, \pi_{1i}^{\text{CDL}}, \pi_{2i}^{\text{CDL}}, \pi_i^{\text{PSF}}$ to all other players in the new set $[\tilde{P}_1, \ldots, \tilde{P}_{\tilde{n}}]$
14. EchoBroadcast $\texttt{CommitmentsReceived}_i$ to all other players in the old set $[P_1, \ldots, P_n]$
15. `Return` $\tilde{sk}_i, \tilde{pk}_i, \tilde{N}_i, \tilde{h}_{2i}, \tilde{h}_{2i}$

() $\leftarrow$ Old_ReshareRound2$([\texttt{CommitmentsReceived}_j]_{j\in[1,\tilde{n}],j\neq i})$

1. `For` $j = [1, \ldots, \tilde{n}]$
2. `If` $i = j$, `Continue`
3. P2PSend s_{ij} to player $\tilde{\mathcal{P}}_j$ from the new set
4. EchoBroadcast D_i to all to all players in the new set $[\tilde{P}_1, \ldots, \tilde{P}_{\tilde{n}}]$

$(\tilde{x}_i, [\tilde{X}_1, \ldots, \tilde{X}_{\tilde{n}}]) \leftarrow$ New_ReshareRound2$([\tilde{pk}_j, \tilde{N}_j, \tilde{h}_{1j}, \tilde{h}_{2j}, \pi_{1j}^{\text{CDL}}, \pi_{2j}^{\text{CDL}}, \pi_i^{\text{PSF}}]_{j\in[1,\tilde{n}],j\neq i}, [s_{ji}, D_j]_{j\in[1,t+1]})$

1. `For` $j = [1, \ldots, \tilde{n}]$
2. `If` $i = j$, `Continue`
3. `If` VerifyCompositeDL$(\pi_{1j}^{\text{CDL}}, g, q, h_{1j}, h_{2j}, \tilde{N}_j) =$ `False`, `Abort`
4. `If` VerifyCompositeDL$(\pi_{2j}^{\text{CDL}}, g, q, h_{2j}, h_{1j}, \tilde{N}_j) =$ `False`, `Abort`
5. `If` VerifyPSF$(\pi_j^{\text{PSF}}, pk_j.N, y, g, q, p_j) =$ `False`, `Abort`
6. Set $\tilde{x}_i = 0$
7. `For` $j = [1, \ldots, t+1]$
8. Compute $[v_{j0}, \ldots, v_{j\tilde{t}}] \leftarrow$ Open(C_j, D_j)
9. `If` $[v_{j0}, \ldots, v_{j\tilde{t}}] = \perp$, `Abort`
10. `If` FeldmanVerify$(g, q, s_{ji}, \tilde{p}_i, [v_{j0}, \ldots, v_{j\tilde{t}}]) =$ `False`, `Abort`
11. Compute $\tilde{x}_i = \tilde{x}_i + s_{ji} \bmod q$
12. `For` $j = [0, \ldots, \tilde{t}]$
13. Set $v_j = 1$
14. `For` $k = [1, \ldots, t+1]$
15. Compute $v_j = v_j \cdot v_{kj}$ in $\mathcal{G}$
16. If $y \neq v_0$, `Abort`
17. `For` $j = [1, \ldots, \tilde{n}]$
18. Set $\tilde{X}_j = y$
19. `For` $k = [1, \ldots, \tilde{t}]$
20. Compute $c_k = (\tilde{p}_j)^k \bmod q$
21. Compute $\tilde{X}_j = \tilde{X}_j \times (v_k)^{c_k}$ in $\mathcal{G}$
22. `If` $\texttt{Aborted} =$ `false`, Set $\texttt{status}_i =$ `success`
23. EchoBroadcast $\texttt{status}_i$ to all other players in the new set $[\tilde{P}_1, \ldots, \tilde{P}_{\tilde{n}}]$
24. `Return` $\tilde{x}_i, [\tilde{X}_1, \ldots, \tilde{X}_{\tilde{n}}]$

() $\leftarrow$ New_ReshareRound3$([\texttt{status}_j]_{j\in[1,\tilde{n}]})$

1. `For` $j = [1, \ldots, \tilde{n}]$
2. If $\texttt{status}_j \neq$ `success`, `Abort`
3. Broadcast $\texttt{status}_i$ to all other players in the old set $[P_1, \ldots, P_n]$

() $\leftarrow$ Old_ReshareRound3$([\texttt{status}_j]_{j\in[1,\tilde{n}]})$

1. `For` $j = [1, \ldots, \tilde{n}]$
2. If $\texttt{status}_j \neq$ `success`, `Abort`
3. Else Delete x_i

Fig. 17: The resharing protocol for updating the threshold set and parameters

References

1. Boneh, D.: Digital signature standard. In: Encyclopedia of cryptography and security, pp. 347–347. Springer (2011)
2. Doerner, J., Kondi, Y., Lee, E., et al.: Secure two-party threshold ecdsa from ecdsa assumptions. In: IEEE Symposium on Security and Privacy. p. 0. IEEE (2018)
3. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ecdsa with fast trustless setup. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1179–1194. ACM (2018)
4. Gennaro, R., Goldfeder, S.: One round threshold ecdsa with identifiable abort. In: IACR eprint Report 2020/540 (2020)
5. Gennaro, R., Micciancio, D., Rabin, T.: An efficient non-interactive statistical zero-knowledge proof system for quasi-safe prime products. In: In Proc. of the 5th ACM Conference on Computer and Communications Security (CCS-98. Citeseer (1998)
6. Kravitz, D.W.: Digital signature algorithm (Jul 27 1993), uS Patent 5,231,668
7. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: International Conference on the Theory and Applications of Cryptographic Techniques. pp. 223–238. Springer (1999)
8. Shamir, A.: How to share a secret. Communications of the ACM **22**(11), 612–613 (1979)