Appendix

badjiea51

September 2025

A Proof of Security

In this section, we present the detailed security proof for the DHSS-Interactive protocol. The strategy reduces the security of the protocol to the hardness of the discrete logarithm problem (DLP) for a challenge value ω. The challenge ω is embedded into the public key Y generated through the `DHSS-KeyGen` protocol, involving multiple participants. The proof proceeds by showing that a forger $\mathcal{F}$ who generates two valid signatures σ and σ' for the same commitment R and distinct messages $m \neq m'$ can be used to compute the discrete logarithm of ω.

A.1 Preliminaries

The security reduction relies on several components:

- $\mathcal{F}$: The forger, capable of producing a valid signature σ with probability ϵ within time t. The key Y is generated as part of the `DHSS-KeyGen` protocol.
- $\mathcal{A}$: The simulator, responsible for simulating honest participants, key generation, and random oracle queries. It invokes $\mathcal{F}$ to perform the forgery.
- $\mathsf{GF}_{\mathcal{A}}$: The Generalized Forking Algorithm, which uses a random tape to interact with $\mathcal{A}$ and produce two forgeries (σ, σ').
- $\mathcal{C}$: The coordination algorithm that, given a challenge ω, invokes the above algorithms to produce (σ, σ') and compute the discrete logarithm of ω.

A.2 Generalized Forking Algorithm

We extend the Generalized Forking Algorithm introduced by Bellare and Neven [4], adapted for concurrent executions. This algorithm ensures that after two invocations of $\mathcal{A}$ with different random oracle responses, it produces two distinct forgeries for the same commitment R.

The algorithm rewinds $\mathcal{A}$ after it makes a specific random oracle query h_J and reruns it with fresh randomness. If $\mathcal{A}$ produces two valid signatures (σ, σ') for the same R but distinct oracle queries $h_J \neq h'_J$, we obtain a successful output.

Algorithm 1: Generalized Forking Algorithm $\mathsf{GF}_{\mathcal{A}}(Y)$

```
Input: $Y$: public key
Output: $(1, h_J, h'_J, \sigma, \sigma')$ or $\perp$
Generate random tape $\beta$;
Sample random oracle outputs $h_1, \ldots, h_{n_r} \leftarrow \mathcal{H}$;
$(J, \sigma) \leftarrow \mathcal{A}(Y, \{h_1, \ldots, h_{n_r}\}; \beta)$;
if $\sigma = \perp$ then
    return $\perp$;
Sample fresh random oracle outputs $h'_J, \ldots, h'_{n_r} \leftarrow \mathcal{H}$;
$(J', \sigma') \leftarrow \mathcal{A}(Y, \{h_1, \ldots, h_{J-1}, h'_J, \ldots, h'_{n_r}\}; \beta)$;
if $J \overset{?}{=} J'$
textbfand $h_J \neq h'_J$ then
    return $(1, h_J, h'_J, \sigma, \sigma')$;
return $\perp$;
```

A.3 Coordination Algorithm

The coordination algorithm $\mathcal{C}$, given a challenge ω, extracts the discrete logarithm using the forgeries (σ, σ') from the Generalized Forking Algorithm.

Algorithm 2: Coordination Algorithm $\mathcal{C}(\omega)$

```
Input: $\omega$: challenge value
Output: Discrete logarithm of $\omega$ or $\perp$
Simulate KeyGen to embed $\omega$ in $Y$;
$(1, h_J, h'_J, \sigma, \sigma') \leftarrow \mathsf{GF}_{\mathcal{A}}(Y)$;
if $(1, h_J, h'_J, \sigma, \sigma') \neq \perp$ then
    Extract $\log_\omega(Y)$ using $h_J, h'_J, \sigma, \sigma'$;
return $\perp$;
```

This algorithm concludes the security reduction by computing the discrete logarithm of ω if the two forgeries are successfully produced.

A.4 Generalized Forking Lemma

The success probability of the generalized forking algorithm is captured by the following lemma from Bellare and Neven [4]:

Lemma 1. *Let* frk *denote the probability of producing two valid forgeries in* $\mathsf{GF}_{\mathcal{A}}$ *over group* $\mathbb{G}$*. Let* ϵ *denote the advantage of solving the discrete logarithm problem in* $\mathbb{G}$*, and* n_r *the number of random oracle outputs. Then:*

$$\epsilon' \geq \mathsf{frk} \geq \mathsf{acc} \cdot \left(\frac{1}{n_r} - \frac{1}{h} \right),$$

where `acc` *is the probability that $\mathcal{A}$ produces a valid forgery.*

This lemma shows that the probability of success is directly tied to the number of oracle outputs and the forger's advantage.

A.5 Embedding the Challenge Value during KeyGen

The coordination algorithm $\mathcal{C}$, as described in Algorithm 2, performs the setup for the Generalized Forking Algorithm ($\mathrm{GF}_{\mathcal{A}}$) and later extracts the discrete logarithm of the challenge value ω.

Simulating KeyGen. To embed the challenge value ω into the group public key Y, $\mathcal{C}$ simulates the `KeyGen` phase. We assume $n = t$, where n is the total number of participants, and t is the threshold. The forger $\mathcal{F}$ controls $t-1$ participants such that $\mathcal{P}_i \in S \notin \Gamma$, and $\mathcal{A}$ simulates the t-th (honest) participant for $\mathcal{F}$. The general case for $n \neq t$ is similar.

For the first round of the `KeyGen` protocol, $\mathcal{A}$ simulates the honest participant $\mathcal{P}_t$ as follows:

1. Let $\mathbf{C}_i = (\phi_{i1}, \ldots, \phi_{i(t-1)})$ represent the public commitments for participant $\mathcal{P}_i$. To calculate $\mathbf{C}_t$ and distribute shares $f_t(1), \ldots, f_t(t-1)$ to the $t-1$ participants controlled by $\mathcal{F}$, $\mathcal{A}$ proceeds as follows:

- Randomly generate secret shares $\bar{x}_{t1}, \ldots, \bar{x}_{t(t-1)}$ corresponding to $f_t(1), \ldots, f_t(t-1)$.
- Set $\phi_{t0} = \omega$ to embed the challenge value ω.
- Compute ϕ_{tk} for $k = 1, \ldots, t-1$ using Birkhoff interpolation in the exponent:
$$\phi_{tk} = \omega^{\lambda_{k0}} \cdot g^{\sum_{i=1}^{t-1} \lambda_{ki} \cdot \bar{x}_{ti}},$$
where λ_{ki} are Birkhoff coefficients.

$\mathcal{A}$ then broadcasts $\mathbf{C}_t$ for $\mathcal{P}_t$. In the second round, $\mathcal{A}$ sends $(1, \bar{x}_{t1}), \ldots, (t-1, \bar{x}_{t(t-1)})$ to the $t-1$ participants corrupted by $\mathcal{F}$.

Next, $\mathcal{A}$ simulates the proof of knowledge for t_0 by deriving a signature σ:

$$c_t, z \leftarrow \mathbb{Z}_q; \quad R = g^z \cdot \omega^{-c_t}; \quad \sigma = (R, z).$$

$\mathcal{A}$ calculates the public key for $\mathcal{P}_t$ as follows:

$$Y_t = \prod_{j \in \mathrm{QUAL}} \prod_{k=0}^{t-1} \phi_{jk}^{t^k} \mod q.$$

The participants controlled by $\mathcal{F}$ derive their private key shares s_i following the `KeyGen` protocol and compute their public keys as $Y_i = g^{s_i}$. Each party, honest or controlled by $\mathcal{F}$, derives the group public key:

$$Y = \prod_{j \in \mathrm{QUAL}} \phi_{j0}.$$

Additionally, $\mathcal{C}$ obtains $\mathcal{F}$'s secret values $(a_{10}, \ldots, a_{(t-1)0})$ using the extractor for the zero-knowledge proofs generated by $\mathcal{F}$. These values are necessary to convert the discrete logarithm of the group public key Y into the discrete logarithm of the challenge value ω.

A.6 Solving the Discrete Logarithm of the Challenge

Once two forgeries (σ, σ') and the random oracle queries (h_J, h'_J) are obtained from $\mathsf{GF}_{\mathcal{A}}$, $\mathcal{C}$ uses Algorithm 3 to extract the discrete logarithm of the challenge value ω. The advantage ϵ' used later in our proofs denotes $\mathcal{C}(\omega)$'s advantage in solving the discrete logarithm problem for ω.

Algorithm 3: ExtractDLog$(\omega, h_J, h'_J, \sigma, \sigma', (a_{10}, \ldots, a_{(t-1)0}))$

```
  Input: Challenge value $\omega$, random oracle responses $h_J, h'_J$, forgeries
         $\sigma = (R, z)$, $\sigma' = (R, z')$, secret values $(a_{10}, \ldots, a_{(t-1)0})$
  Output: The discrete logarithm $a_{t0}$ of $\omega$
  // Compute the discrete logarithm of $Y$
1 $\log_g(Y) = \frac{z'-z}{h'_J - h_J}$;
  // Compute $a_{t0}$
2 $a_{t0} = \log_g(Y) - \sum_{i=1}^{t-1} a_{i0}$;
3 return $a_{t0}$, the discrete logarithm of $\omega$;
```

This calculation is possible since

$$R = g^z \cdot Y^{-h_J}, \quad R = g^{z'} \cdot Y^{-h'_J},$$

and thus, when $h_J \neq h'_J$, the discrete logarithm of Y is:

$$\log_g(Y) = \frac{z' - z}{h'_J - h_J}.$$

Finally, by subtracting the known secret values $(a_{10}, \ldots, a_{(t-1)0})$, we obtain $a_{t0} = \log_g(\omega)$.

A.7 Proof of Security for DHSS-Interactive

To demonstrate the security of DHSS-Interactive, we first prove the security of an interactive two-round variant of the DHSS signing operation, which we call `DHSS-Interactive`. This variant simplifies the simulation of zero-knowledge proofs by using a one-time verifiable random function (VRF) for generating binding values ρ_i. In Section 6.2, we discuss how the security proof for DHSS-Interactive extends to the non-interactive DHSS.

A.7.1 DHSS-Interactive Overview

DHSS-Interactive uses the same `KeyGen` protocol as the standard DHSS for generating long-lived keys (see Section **??**). The main difference is how the binding value ρ_i is generated during the signing process, as outlined in the Preprocess phase and in Signing phase.

In DHSS-Interactive, the binding value ρ_i is generated using a one-time VRF. Specifically, each participant computes $\rho_i = a_{ij} + b_{ij} \cdot \mathcal{H}_\rho(m, B)$, where (a_{ij}, b_{ij}) are the one-time VRF keys, and (m, B) are inputs. This modification allows us to prove security under the standard EUF-CMA security notion for signatures.

Preprocess Phase. The preprocess phase in DHSS-Interactive differs from the standard DHSS protocol in two ways: 1. Participants generate one-time VRF keys (a_{ij}, b_{ij}) and their corresponding commitments $(A_{ij} = g^{a_{ij}}, B_{ij} = g^{b_{ij}})$, along with the usual DHSS nonce values (d_{ij}, e_{ij}) and commitments $(D_{ij} = g^{d_{ij}}, E_{ij} = g^{e_{ij}})$. 2. Zero-knowledge proofs of knowledge (ZKPKs) for the VRF keys are generated to ensure secure signing.

In DHSS-Interactive, the preprocess phase must be performed serially, as the simulator needs to extract the discrete logarithm of the adversary's VRF keys by rewinding. However, in the non-interactive setting of DHSS, this phase can be done concurrently.

Signing Phase. The signing phase proceeds as follows: 1. The signing aggregator (β) sends the message m and value B to each participant. 2. Each participant computes $\rho_i = a_{ij} + b_{ij} \cdot \mathcal{H}_\rho(m, B)$, where B is derived from the ordered list of tuples (i, D_{ij}, E_{ij}) for $i \in S$. 3. β collects ρ_i from the participants and broadcasts it. Each participant uses ρ_i to compute R and their response z_i.

A.7.2 Proof of Security for DHSS-Interactive

We now provide a proof of security for DHSS-Interactive under the EUF-CMA security model. The adversary $\mathcal{F}$ that can forge signatures in DHSS-Interactive can be used to compute the discrete logarithm of an arbitrary challenge $\omega \in \mathbb{G}$.

Let n_h be the number of queries made to the random oracle, n_p the number of preprocess queries, and n_s the number of signing queries. We state the following theorem:

Theorem 2 (Security of DHSS-Interactive)**.** *If the discrete logarithm problem (DLP) in $\mathbb{G}$ is (τ', ϵ')-hard, then the DHSS-Interactive signature scheme over $\mathbb{G}$ with n signing participants, threshold t , and preprocess batch size π, is $(\tau, n_h, n_p, n_s, \epsilon)$-secure whenever*

$$\epsilon' \le \frac{\epsilon^2}{2n_h + (\pi + 1)n_p + 1}$$

and

$$\tau' = 4\tau + (30\pi n_p + (4t - 2)n_s + (n + t - 1)t + 6) \cdot t_{exp} + O(\pi n_p + n_s + n_h + 1),$$

where t_{exp} is the time complexity of an exponentiation in $\mathbb{G}$, and the number of compromised participants is less than t such that $\mathcal{P}_i \in S \notin T$.

Proof. We proceed by contradiction. Assume that $\mathcal{F}$ can break the unforgeability of DHSS-Interactive in $(\tau, n_h, n_p, n_s, \epsilon)$. We construct an algorithm $\mathcal{C}$ that uses $\mathcal{F}$ to solve the discrete logarithm of a challenge $\omega \in \mathbb{G}$ in (τ', ϵ'). The simulator $\mathcal{A}$ interacts with $\mathcal{F}$, controlling $t-1$ participants and simulating one honest participant $\mathcal{P}_t$.

Preprocess Simulation. In the preprocess phase, described in Algorithm 4, each participant $\mathcal{P}_i$, for $i \in \{1, \ldots, n\}$, generates commitments and nonces as follows:

Signing Simulation. The signing phase follows similar steps, where participants verify commitments and generate their responses. Zero-knowledge proofs for VRF keys are verified using the commitments exchanged in the preprocess phase.

This process is repeated for each signing request from $\mathcal{F}$. By leveraging the interactions, $\mathcal{C}$ can compute the discrete logarithm of ω and thus break the DLP assumption in $\mathbb{G}$.

Conclusion. Therefore, if $\mathcal{F}$ can break the unforgeability of DHSS-Interactive, $\mathcal{C}$ can solve the discrete logarithm problem in $\mathbb{G}$, contradicting the assumption that DLP in $\mathbb{G}$ is hard.

A.8 DHSS-Interactive Signing Protocol

The signing operation for DHSS-Interactive involves two rounds. The signature aggregator (β) coordinates with t participants to generate a signature on a message m.

A.8.1 **Sign**$(m) \to (m, \sigma)$

Round 1: Commitment Exchange 1. β selects a set S of t participants for the signing protocol and retrieves the next available commitments $(D_{ij}, E_{ij}, A_{ij}, B_{ij})$ for each participant $\mathcal{P}_i \in S$. β constructs $B = \langle (i, D_{ij}, E_{ij}) \rangle_{i \in S}$ and sends (m, B) to each participant $\mathcal{P}_i$.

2. Upon receiving (m, B), each participant $\mathcal{P}_i$ verifies that:

- m is a valid message,
- and each tuple $(i, D_{ij}, E_{ij}) \in B$ corresponds to their next available commitments $(D_{ij}, E_{ij}, A_{ij}, B_{ij})$.

If either check fails, $\mathcal{P}_i$ aborts the protocol.

Algorithm 4: Preprocess Phase for DHSS-Interactive

```
Input: Each participant $\mathcal{P}_i$, where $i \in \{1, \dots, n\}$, performs this phase
       before the signing protocol. $\pi$ denotes the number of
       commitments generated.
Output: List $L_i$ containing commitments and zero-knowledge proofs
        for later use in signing operations.
1  Round 1: Generation of Commitments
2      for $1 \le j \le \pi$ do
3          Generate nonces $d_{ij}, e_{ij}, a_{ij}, b_{ij} \leftarrow \mathbb{Z}_q^*$;
4          Compute commitments:
               $(D_{ij}, E_{ij}, A_{ij}, B_{ij}) = (g^{d_{ij}}, g^{e_{ij}}, g^{a_{ij}}, g^{b_{ij}})$
            Generate auxiliary nonces $k_{aij}, k_{bij} \leftarrow \mathbb{Z}_q$;
5          Compute auxiliary commitments:
               $(R_{aij}, R_{bij}) = (g^{k_{aij}}, g^{k_{bij}})$
            Set $K_{ij} = (D_{ij}, E_{ij}, A_{ij}, B_{ij}, R_{aij}, R_{bij})$;
6          Append $(j, (D_{ij}, E_{ij}, A_{ij}, B_{ij}))$ to the list $L_i$;
7          Store $(d_{ij}, D_{ij}), (e_{ij}, E_{ij}), (a_{ij}, A_{ij}), (b_{ij}, B_{ij})$ for later use in
            signing.
8      Compute $K_i = \mathcal{H}_3(K_{i1}, \dots, K_{i\pi})$;
9      Send $(i, K_i)$ to all other participants;
10 Round 2: Zero-Knowledge Proof of Knowledge
11     foreach $\mathcal{P}_i$ do
12         After receiving $(\ell, K_\ell)$ from all participants, compute
            $\Phi = \mathcal{H}_3(K_1, \dots, K_n)$;
13         Compute the challenge $c_i = \mathcal{H}_4(i, \Phi)$;
14         for $1 \le j \le \pi$ do
15             Derive responses:
                   $\mu_{aij} = k_{aij} + a_{ij} \cdot c_i \quad \text{and} \quad \mu_{bij} = k_{bij} + b_{ij} \cdot c_i$
16         Set $J_i = \langle \mu_{aij}, \mu_{bij} \rangle_{j=1}^{\pi}$;
17         Send $(i, L_i, J_i)$ to all other participants;
18 Verification and Validation
19     foreach $\mathcal{P}_i$ do
20         After receiving $(\ell, L_\ell, J_\ell)$, compute the challenge $c_\ell = \mathcal{H}_4(\ell, \Phi)$;
21         for $1 \le j \le \pi$ do
22             Verify that $D_{\ell j}, E_{\ell j}, A_{\ell j}, B_{\ell j} \in G^*$;
23             Compute:
                   $R'_{a\ell j} = g^{\mu_{a\ell j}} \cdot A_{\ell j}^{-c_\ell}, \quad R'_{b\ell j} = g^{\mu_{b\ell j}} \cdot B_{\ell j}^{-c_\ell}$
                Set $K'_{\ell j} = (D_{\ell j}, E_{\ell j}, A_{\ell j}, B_{\ell j}, R'_{a\ell j}, R'_{b\ell j})$;
24         Compute $K'_\ell = \mathcal{H}_3(K'_{\ell 1}, \dots, K'_{\ell\pi})$;
25         if $K'_\ell \ne K_\ell$ then
26             Abort the protocol;
27         Store $(\ell, L_\ell)$ for signing operations;
```

3. Each participant $\mathcal{P}_i$ computes:

$$\rho_i = a_{ij} + b_{ij} \cdot \mathcal{H}_\rho(m, B),$$

where a_{ij} and b_{ij} are the secret components of the one-time VRF keys. After generating ρ_i, $\mathcal{P}_i$ securely deletes (a_{ij}, A_{ij}) and (b_{ij}, B_{ij}) from local storage and returns ρ_i to β.

Round 2: Response Generation 1. After receiving all $\rho_\ell, \ell \in S$, β distributes the set $\{\rho_\ell\}_{\ell \in S}$ to each participant $\mathcal{P}_i \in S$.

2. Each participant $\mathcal{P}_i$ verifies the validity of ρ_ℓ for each $\ell \in S$ by checking:

$$g^{\rho_\ell} = A_{\ell j} \cdot B_{\ell j}^{H_\rho(m,B)}.$$

If any verification fails, $\mathcal{P}_i$ aborts.

3. Each participant $\mathcal{P}_i$ computes the aggregate commitment R as:

$$R = \prod_{\ell \in S} D_{\ell j} \cdot E_{\ell j}^{\rho_\ell},$$

and derives the challenge c using a hash function:

$$c = \mathcal{H}_2(R, Y, m),$$

where Y is the long-term public key of the group.

4. Each participant $\mathcal{P}_i$ then computes their signature share z_i using their long-term secret share s_i:

$$z_i = d_{ij} + (e_{ij} \cdot \rho_i) + \lambda_i \cdot s_i \cdot c,$$

where λ_i is a Birkhoff coefficient computed based on the set S.

5. Each participant $\mathcal{P}_i$ securely deletes (d_{ij}, D_{ij}) and (e_{ij}, E_{ij}) from their local storage and sends z_i to β.

6. β verifies each z_i, aggregates the signature shares, and publishes the final signature σ in the same manner as plain DHSS.

A.8.2 Signature Verification

The verification of the signature σ follows the same procedure as described in the standard DHSS signature scheme, where the verifier checks the validity of the aggregated signature.

A.9 Simulation and Extraction in DHSS-Interactive

Let $n_r = 2n_h + (\pi + 1)n_p + 1$ denote the maximum number of random oracle outputs required by the adversary $\mathcal{F}$.

Algorithm 5: Algorithm A($Y, \{h_1, \dots, h_{n_r}\}; \beta$)

```
   Input: Public key $Y$, random oracle outputs $\{h_1, \dots, h_{n_r}\}$, random
          tape $\beta$
   Output: Index $J$ and forgery $\sigma$, or $\perp$
   // Initialization
1  Set $ctr \leftarrow 1$, and initialize empty tables $T_\rho$, $T_2$, $T_3$, $T_4$, $J_2$, $C$, and $M$.
   // Run forger F and answer its queries
2  while $\mathcal{F}$ does not output $(m, \sigma = (R, z))$ or $\perp$ do
       // Handle random oracle queries
       // Simulate $H_\rho(m, B)$
3      if $T_\rho[m, B] = \perp$ then
4          $T_\rho[m, B] \leftarrow h_{ctr}$; $ctr \leftarrow ctr + 1$
5      Return $T_\rho[m, B]$
       // Simulate $H_2(R, Y, m)$
6      if $T_2[m, R] = \perp$ then
7          $T_2[R, Y, m] \leftarrow h_{ctr}$; $J_2[R, Y, m] \leftarrow ctr$; $ctr \leftarrow ctr + 1$
8      Return $T_2[R, Y, m]$
       // Simulate $H_3(\tilde{X})$
9      if $T_3[\tilde{X}] = \perp$ then
10         $T_3[\tilde{X}] \leftarrow h_{ctr}$; $ctr \leftarrow ctr + 1$
11     Return $T_3[\tilde{X}]$
       // Simulate $H_4(i, \Phi)$
12     if $T_4[i, \Phi] = \perp$ then
13         $T_4[i, \Phi] \leftarrow h_{ctr}$; $ctr \leftarrow ctr + 1$
14     Return $T_4[i, \Phi]$
   // Simulate Preprocessing Phase
15 for $j \leftarrow 1$
16 $KwTo$ $\pi$ do
17     $\bar{c}_j \leftarrow h_{ctr}$, $C[j] \leftarrow ctr$, $ctr \leftarrow ctr + 1$;
18     $\bar{z}_{tj} \leftarrow \mathbb{Z}_q$ $D_{tj} \leftarrow g^{\bar{z}_{tj}} \cdot Y_t^{-\bar{c}_j}$;
19     Follow the protocol to sample $(e_{tj}, a_{tj}, b_{tj})$ and derive $(E_{tj}, A_{tj}, B_{tj})$;
20     Follow the protocol to sample $(k_{atj}, k_{btj})$ and derive $(R_{atj}, R_{btj})$;
21     Derive $K_t$ honestly, publish $K_t$ to $\mathcal{F}$, and wait for all $K_\ell$ values from
        $\mathcal{F}$;
   // Round 2
22 Derive $L_t$, $\Phi$, and $J_t$ honestly. Send $(t, L_t, J_t)$ to $\mathcal{F}$ and wait for
    $(\ell, L_\ell, J_\ell)$ from $\mathcal{F}$;
23 Reprogram $T_3[K_1, \dots, K_n] = h_{ctr}$, increment $ctr$, and rederive $c_t$ and $J_t$
    honestly;
24 Rewind $\mathcal{F}$ to step 1 of Round 2, just before querying $H_3$ with
    $(K_1, \dots, K_n)$;
25 After $\mathcal{F}$ proceeds, derive the discrete logarithms of each $A_{\ell j}$ and $B_{\ell j}$ for
    dishonest $\mathcal{P}_\ell$ and store for the signing protocol.
```

Algorithm 6: Continue

```
   // Simulate Signature Phase
1  Round 1: Input $(m, B)$, insert $m$ into $M$;
2  Use $(a_{\ell j}, b_{\ell j})$ obtained during preprocessing to derive $\rho_\ell$ for $\ell \in S$
     $(\ell \neq t)$;
3  Derive $\rho_t = a_{tj} + b_{tj} \cdot \mathcal{H}_\rho(m, B)$ and $R$ following the protocol;
4  Program $T_2[m, R] = \bar{c}_j$, $J_2[m, R] = C[j]$; return $\rho_t$;
5  Round 2: Input $(\rho_j, \ldots, \rho_t)$;
6  Compute $z_t = \bar{z}_{tj} + (e_{tj} \cdot \rho_t)$; return $z_t$ to $\mathcal{F}$;
7  if $\mathcal{F}$ outputs $\perp$ then
8      Return $\perp$;
9  else
10     $\mathcal{F}$ outputs $(m, \sigma = (R, z))$
   // Final Check
11 if $T_2[m, Y, R] = \perp$ then
12     $T_2[m, Y, R] \leftarrow h_{ctr}$; $J_2[m, Y, R] \leftarrow ctr$; $ctr \leftarrow ctr + 1$
13 Let $c = T_2[m, Y, R]$;
14 if $R \neq g^z Y^{-c}$ or $m \in M$ then
15     Return $\perp$
16 Return $J = J_2[m, Y, R]$, $\sigma = (R, z)$;
```

A.9.1 Simulating Random Oracle Queries

After the key generation phase (as described in Section 5 of the original manuscript, $\mathcal{A}$ invokes $\mathcal{F}$ to perform a forgery attack. $\mathcal{A}$ simulates the random oracle queries H_ρ, H_2, H_3, and H_4 by maintaining associative tables initialized to empty at the start.

For each random oracle query, $\mathcal{A}$ checks if the output has already been determined. If no value exists, $\mathcal{A}$ assigns the next available output from $\{h_1, \ldots, h_{n_r}\}$, increments the counter *ctr*, and returns the assigned value.

DHSS-Interactive replaces the standard hash function $H_1(i, m, B)$ (used in plain DHSS) with an interactive one-time VRF based on $H_\rho(m, B)$ for better binding.

A.9.2 Simulating Preprocess Phase

In the preprocess stage, $\mathcal{A}$ simulates the honest participant $\mathcal{P}_t$. For each D_{tj}, $\mathcal{A}$ selects $\bar{c}_j$ as the next available h_{ctr}, keeps track of it using $C[j] = ctr$, and then increments *ctr*. $\mathcal{A}$ randomly samples $\bar{z}_{tj} \leftarrow \mathbb{Z}_q$, and computes:

$$D_{tj} = g^{\bar{z}_{tj}} \cdot Y_t^{-\bar{c}_j}.$$

$\mathcal{A}$ computes and publishes its proof of knowledge of (a_{tj}, b_{tj}) values honestly during Round 2. However, $\mathcal{A}$ forks $\mathcal{F}$ in order to extract the discrete logarithms

$(a_{\ell j}, b_{\ell j})$ of the commitments $(A_{\ell j}, B_{\ell j})$ for the players $\mathcal{P}_\ell$ controlled by $\mathcal{F}$. By rewinding $\mathcal{F}$, $\mathcal{A}$ programs the random oracle H_3 to return a different random output Φ'. This allows $\mathcal{A}$ to solve the discrete logarithms.

A.9.3 Simulating the Signing Phase

$\mathcal{F}$ initiates the signing protocol by acting as the signature aggregator (β), sending (m, B) in Round 1. After receiving these values, $\mathcal{A}$ computes both its own ρ_t and the ρ_ℓ values for all other participants. Using the known $(a_{\ell j}, b_{\ell j})$ from the Preprocess phase, $\mathcal{A}$ computes the aggregate commitment R for Round 2, and sets:

$$H_2(R, Y, m) = \bar{c}_j,$$

storing $C[j]$ and the associated counter ctr as $J_2[R, Y, m]$ in a table.

Since $\mathcal{A}$ computes R before $\mathcal{F}$ can, it programs the random oracle H_2 with perfect success, avoiding the need to guess the correct random oracle outputs. By contrast, a signing request in a plain Schnorr simulation succeeds only with probability $1/(n_h + n_s + 1)$.

A.9.4 Finding the Discrete Logarithm of the Challenge

As detailed in Section A.3, the discrete logarithm of the challenge value ω can be extracted using two forgeries (σ, σ'). The probability of $\mathcal{F}$ succeeding in one run of $\mathcal{A}$ is denoted by ϵ. Using the forking lemma, the probability of extracting the discrete logarithm of ω after two runs is at least $\frac{\epsilon^2}{n_r}$, ignoring negligible terms, where h typically satisfies $h \geq 2^{256}$.

The total time τ' required to extract the discrete logarithm of ω is:

$$\tau' = 4\tau + (30\pi n_p + (4t-2)n_s + (n+t-1)t + 6) \cdot t_{\exp} + O(\pi n_p + n_s + n_h + 1),$$

where $t_{\exp}$ is the time for a single exponentiation.

A.9.5 Computation Overhead

The running time for $\mathcal{C}$ to compute the discrete logarithm via two forgeries from DHSS-Interactive involves:

- $(t-1) \cdot t$ operations to compute C_t, 2 operations to compute R, and $n \cdot t$ operations to compute Y_t during KeyGen.
- In each of two executions of $\mathcal{A}$:
  - 7 operations in each of π iterations of Round 1 during Preprocess simulation,
  - 8π operations to validate two versions of J_ℓ lists in Round 2 of Preprocess simulation,
  - $t-1$ operations to validate the ρ_ℓ values and t operations to compute R in each signing simulation,

  - 2 operations to compute R and verify the output of $\mathcal{F}$.

- Additional operations like table lookups, which sum to $O(\pi n_p + n_s + n_h + 1)$.

The total time overhead for two executions of $\mathcal{A}$ is proportional to $(30\pi n_p + (4t-2)n_s + (n+t-1)t + 6) \cdot t_{\text{exp}} + O(\pi n_p + n_s + n_h + 1)$.

A.10 Extension of DHSS-Interactive to DHSS

In this section, we describe the modifications needed to transform DHSS-Interactive into DHSS by reducing one round of communication in both the Preprocess and Sign phases. We argue in Section 6 why these changes do not compromise the security of the protocol.

A.10.1 Removal of One-Time Verifiable Random Functions for Generating ρ_i

The main difference between DHSS-Interactive and DHSS is the use of one-time verifiable random functions (VRFs) in the former to generate the binding values ρ_i. In DHSS, these values ρ_i are generated via random oracles modeled as hash functions.

Thus, removing the one-time VRFs simplifies the protocol by eliminating the VRF keys (a_{ij}, b_{ij}) and their associated commitments (A_{ij}, B_{ij}). Instead, participants compute ρ_i non-interactively using a random oracle:

$$\rho_i = \mathcal{H}_\rho(m, B).$$

A.10.2 Removal of One Round from the Sign Phase

Without the need for interactive VRFs, each participant can independently compute the ρ_i values of other participants using the random oracle $H_\rho(m, B)$. Consequently, the first round of the Sign phase in DHSS-Interactive, where participants exchange their ρ_i values, becomes unnecessary.

In DHSS, the participants can proceed directly to the second round, where they compute their responses z_i using the precomputed ρ_i values:

$$z_i = d_{ij} + (e_{ij} \cdot \rho_i) + \lambda_i \cdot s_i \cdot c.$$

Here, λ_i represents the Birkhoff coefficient based on the signing set S, and c is the challenge:

$$c = \mathcal{H}_2(R, Y, m).$$

A.10.3 Removal of Proofs of Knowledge of One-Time VRF Keys and One Round of the Preprocess Phase

Since the one-time VRF keys (a_{ij}, b_{ij}) are removed, the associated proofs of knowledge J_i in the Preprocess phase are no longer needed. With the removal of J_i, the K_i values used to aggregate and commit to the VRF keys also become unnecessary.

As a result, the first round of the Preprocess phase, which involves generating and sharing these values, is eliminated entirely in DHSS. Participants now only need to perform the second round of the Preprocess phase, which focuses on generating the necessary commitments for signing.

In summary, the changes made to DHSS-Interactive to form DHSS are as follows:

- Removal of one-time verifiable random functions (VRFs) and their corresponding commitments.
- Elimination of the first round of the Sign phase, as ρ_i values can be computed non-interactively using random oracles.
- Removal of the proofs of knowledge and K_i values in the Preprocess phase, which results in the elimination of the first round of the Preprocess phase.

These modifications reduce the communication overhead and simplify the protocol while maintaining the same level of security.