## B Correctness of HHGS

According to the correctness experiment $\mathsf{Exp}^{\mathsf{Correct}}_{\mathcal{A},\mathsf{HHGS}}$, the correctness of $\mathsf{HHGS}$ encompasses two key aspects: (i) the adversary can register honest parties under chosen identities, and all signatures generated by honest parties must pass verification; and (ii) signatures produced by honest parties must be accurately opened (traced) to their legitimate owners.

The registration process in $\mathsf{HHGS}$ primarily involves locating a mount point on the mixed OTS mount sub-tree (MoMSt). Since each OTS is uniquely associated with a leaf node, successful registration is guaranteed as long as there are available vacancies in the tree structure. The validity of an honest party's signature during verification is inherently ensured by the correctness of OTS and $\mathsf{MT}$. Moreover, the oracle queries in the correctness game are simulated using secret keys (including sub-secret keys of the building blocks, such as $\mathsf{AE}$, $\mathsf{PPRF}$, and $\mathsf{PRF}$) generated by the challenger. The correctness of these building blocks ensures that all queries and protocol executions involving them are processed correctly in accordance with the protocol's specifications.

## C Security Proofs of HHGS

### C.1 Proof of Theorem 1

We denote by $\mathsf{Win}^{\mathsf{Trace}}_i$ the event where the $i$-th (modified) $\mathsf{Trace}$ experiment (game) outputs 1.
**Game 0.** This game corresponds to the original $\mathsf{Trace}$ game. Specifically,

$$\Pr[\mathsf{Win}^{\mathsf{Trace}}_0] = \mathsf{Adv}^{\mathsf{Trace}}_{\mathcal{A},\mathsf{HHGS}}(\kappa, \mathsf{SP}).$$

**Game 1.** This game proceeds as before, but the challenger aborts if the adversary $\mathcal{A}$ outputs a forgery $\sigma^*$ containing a value $opk^*_{\mathsf{ID}}||c^*_{\mathsf{ID}}$ such that: i) $opk^*_{\mathsf{ID}}||c^*_{\mathsf{ID}}$ is not generated by the challenger during the simulations of join protocol instances (in handling $\mathsf{AddHM}$ and $\mathsf{AddMM}$ queries); and ii) $\mathsf{H}(opk_{\mathsf{ID}}||c_{\mathsf{ID}}) = \mathsf{H}(opk^*_{\mathsf{ID}}||c^*_{\mathsf{ID}})$, where $opk_{\mathsf{ID}}||c_{\mathsf{ID}}$ is generated by the challenger within the game. That is, $\mathcal{A}$ succeeds in finding a collision of $\mathsf{H}$. It is not hard to see that such a forgery can be used to break the collision-resistance of $\mathsf{H}$. Thus, we have that

$$\Pr[\mathsf{Win}^{\mathsf{Trace}}_0] \leq \Pr[\mathsf{Win}^{\mathsf{Trace}}_1] + \mathsf{Adv}^{\mathsf{CRH}}_{\mathcal{A},\mathsf{H}}(\kappa).$$

**Game 2.** This game is modified to evaluate the security of the Merkle tree scheme. Specifically, the challenger $\mathcal{C}$ aborts if the adversary $\mathcal{A}$ produces a forgery $\sigma^*$ containing a Merkle proof for a subtree (originating either from the OTS hypertree or the mixed OTS mount subtrees) that was not generated by $\mathcal{C}$ during the game, yet the Merkle proof successfully yields a Merkle root generated by $\mathcal{C}$. It is worth noting that there exists at least one honest Merkle root (i.e., $\mathsf{gpk}$). If such an event occurs with non-negligible probability, it indicates a breach of the Merkle tree's unforgeability. Notably, all Merkle trees associated with the

OTS hypertree and the mixed OTS mount subtrees are known to the challenger at the time of their creation. To construct an algorithm $\mathcal{B}$ capable of breaking the security of $\mathsf{M}$ by leveraging $\mathcal{A}$, $\mathcal{B}$ must correctly guess which subtree $\mathcal{A}$ intends to forge. The probability of making a correct guess is approximately $\frac{1}{TN}$, where $TN = \frac{1-2^{\tilde{h}}}{1-2^{\frac{h}{d}}}$ represents the total number of subtrees within each $\mathsf{HHGS}$ instance and $\tilde{h} = h+\frac{h}{d}$ is the total height of the hypertree structure. Importantly, this guess is part of $\mathcal{B}$'s internal setup and does not influence the subsequent progression of the games. If $\mathcal{B}$ successfully identifies the target subtree, it can act as the challenger for $\mathcal{A}$, simulating the game by preparing the tree's leaves (either from $\mathcal{A}$'s $\mathsf{AddMM}$ queries or generated by $\mathcal{B}$ itself) while interacting with the $\mathsf{MT}$-challenger. Should $\mathcal{A}$ successfully forge a Merkle proof, $\mathcal{B}$ can replicate this success to break the $\mathsf{MT}$ with a success probability $\frac{1}{TN}$. Thus, we have:

$$\Pr[\mathsf{Win}_1^{\mathsf{Trace}}] \leq \Pr[\mathsf{Win}_2^{\mathsf{Trace}}] + TN \cdot \mathsf{Adv}_{\mathcal{A},\mathsf{M}}^{\mathsf{UF}}(\kappa).$$

If $\mathcal{C}$ does not abort in this game, the $\mathsf{HHGS}$ instance must contain at least one honest subtree with all leaves (i.e., OTS keys) generated by the challenger.

**Game 3.** In this game, the challenger $\mathcal{C}$ aborts if it fails to correctly guess the index of the OTS key contained in the forgery $\sigma^*$, which resides at the lowest level of the honest subtree. Given that there are at most $2^{\tilde{h}}$ leaves, the probability of making a correct guess is bounded by

$$\Pr[\mathsf{Win}_2^{\mathsf{Trace}}] = 2^{\tilde{h}} \cdot \Pr[\mathsf{Win}_3^{\mathsf{Trace}}].$$

The adversary might attempt to forge the signature of the guessed OTS key or compromise the integrity of the Identity-OPK ciphertext associated. In the subsequent games, several preparations are necessary to reduce the analysis to the security of these primitives, such as replacing their keys with random values.

**Game 4.** In this game, the challenger $\mathcal{C}$ aborts if it fails to correctly guess any forgery cases (when such cases exist): i) **ACase 1**—The adversary $\mathcal{A}$ forges the signature of a OTS key and determines which OTS key was forged. ii) **ACase 2**—The adversary $\mathcal{A}$ forges the Identity-OPK ciphertext and determines which ciphertext was forged. Since the leaf of the forged target (either a OTS or a ciphertext) is already assumed to be known by the challenger (based on the correct guess in the previous game), and each leaf corresponds to at most two OTS keys and two Identity-OPK key ciphertexts (within the obfuscated OTS layer), the probability of making a correct guess is bounded by:

$$\Pr[\mathsf{Win}_3^{\mathsf{Trace}}] = 4 \cdot \Pr[\mathsf{Win}_4^{\mathsf{Trace}}].$$

**Game 5.** The challenger $\mathcal{C}$ changes this game from the previous game according to the guessed attacking cases in the previous games.

- For **ACase 1** and the guessed OTS key is from $\mathsf{GM}$, $\mathcal{C}$ replace the randomness $re$ (output by a $\mathsf{PPRF}$ function) for generating the OTS key with a random value.

- For **ACase 1** and the guessed OTS key is from an honest member, $\mathcal{C}$ replaces the randomness (generated by the $\mathsf{PRG}$ scheme) for generating the OTS key with a random value. Before doing this, $\mathcal{C}$ would play a series hybrid games to replace the seeds of the $\mathsf{PRG}$ which are ancestors of the target one with random values as well. There are at most $2^{\tilde{h}}$ nodes to replace, if the $\mathsf{HHGS}$ instance consists of only one party.
- For **ACase 2**, $\mathcal{C}$ changes the $\mathsf{PRF}$ values for generating the encryption keys and encryption randomness of the target ciphertext with random values in hybrid games.

If there exists an adversary which can distinguish this game from the previous game with non-negligible probability, then it can be used to break one of the primitives in $\{\mathsf{PPRF}, \mathsf{PRF}, \mathsf{PRG}\}$ determined by the guessed attacking cases. Considering the potential number of hybrid games, we have that

$$\Pr[\mathsf{Win}_4^{\mathsf{Trace}}] \leq \Pr[\mathsf{Win}_5^{\mathsf{Trace}}] + 2 \cdot \mathsf{Adv}_{\mathcal{A},\mathsf{F}_1}^{\mathsf{IND\text{-}CMA}}(\kappa, 2) + 2^{\tilde{h}} \cdot \mathsf{Adv}_{\mathcal{A},\mathsf{G}}^{\mathsf{IND}}(\kappa)$$

**Game 6.** Now we can reduce the security of $\mathsf{HHGS}$ to that of $\mathsf{OTS}$. Specifically, the challenger $\mathcal{C}$ aborts if the adversary outputs a valid OTS signature $\bar{\sigma}^*$ in **ACase 1**. If this abort event occurs with non-negligible probability, we build $\mathcal{B}$ making use of $\mathcal{A}$ to break the OTS's security. In the reduction, all other queries of $\mathcal{A}$ can be honestly simulated by $\mathcal{B}$ with her own secrets, whereas the signature of the target OTS key being attacked can be obtained from the OTS challenger when necessary. Thus, we have that

$$\Pr[\mathsf{Win}_5^{\mathsf{Trace}}] \leq \Pr[\mathsf{Win}_7^{\mathsf{Trace}}] + \mathsf{Adv}_{\mathcal{A},\mathsf{S}}^{\mathsf{EUF\text{-}CMA}}(\kappa, 1).$$

**Game 7.** In this final game, the challenger $\mathcal{C}$ aborts if the adversary $\mathcal{A}$ successfully forges a signature containing a valid Identity-OPK key ciphertext that was not generated by $\mathcal{C}$ before the corruption of the corresponding owner. Based on the modifications introduced in the previous game, the encryption key and randomness used for the target ciphertext in **ACase 2** are sampled independently as random values. This setup allows the security to be reduced to the integrity of ciphertexts under $\mathsf{E}$. Additionally, the honest ciphertext utilizing the targeted key can be simulated through the encryption oracle provided by the $\mathsf{AE}$ challenger. Consequently, we establish the following bound:

$$\Pr[\mathsf{Win}_7^{\mathsf{Trace}}] \leq \Pr[\mathsf{Win}_7^{\mathsf{Trace}}] + \mathsf{Adv}_{\mathcal{A},\mathsf{E}}^{\mathsf{CT\text{-}INT}}(\kappa).$$

As the adversary is unable to manipulate any components in the signature of an uncorrupted party (whether a member or $\mathsf{GM}$), the adversary's advantage in the final game is reduced to 0, thereby concluding the proof.

## C.2 Proof of Theorem 2

Let $\mathsf{Win}_i^{\mathsf{Anony}}$ denote the event that the $\mathsf{Anony}$ experiment (game) outputs 1 in the $i$-th game.

**Game 0.** The original game, $G_0$, corresponds to the real $\mathsf{Anony}$ experiment (game). We have:

$$\Pr[\mathsf{Win}_0^{\mathsf{Anony}}] = \frac{1}{2} + \mathsf{Adv}_{\mathcal{A},\mathsf{HHGS}}^{\mathsf{Anony}}(\kappa, \mathsf{SP}).$$

**Game 1.** In this game, the challenger $\mathcal{C}$ introduces a new abort condition: it aborts if it fails to guess which two honest group members' OTS keys (i.e., $(osk_{\mathsf{ID}_0^*}$ and $osk_{\mathsf{ID}_1^*})$ in the member sub-layer of the obfuscated OTS layer) are involved in the $\mathcal{O}_{\mathsf{PriTest}}(\mathsf{ID}_0^*, \mathsf{ID}_1^*, m)$ query. Since there are at most $U$ group members, and $E$ OTS keys in total, such that $U \cdot E = 2^{\tilde{h}}$. The probability of a correct guess is bounded by $\frac{1}{2^{\tilde{h}+2}}$. Consequently, we have:

$$\Pr[\mathsf{Win}_0^{\mathsf{Anony}}] = 2^{\tilde{h}+2} \cdot \Pr[\mathsf{Win}_1^{\mathsf{Anony}}].$$

Note that the correct guess of $\mathcal{C}$ implies that it knows the indices of the target OTS keys on the corresponding tree, where the target OTS keys refer to the guessed OTS keys $(osk_{\mathsf{ID}_0^*}, osk_{\mathsf{ID}_1^*})$ and the OTS keys $(osk'_{\mathsf{ID}_0^*}, osk'_{\mathsf{ID}_1^*})$ used to sign them in the obfuscated OTS layers associated with the guessed OTS keys of the two honest members. These target OTS keys would associate with four Identity-OPK ciphertexts, referred to as $(c_{\mathsf{ID}_0^*}, c_{\mathsf{ID}_1^*}, c'_{\mathsf{ID}_0^*}, c'_{\mathsf{ID}_1^*})$, respectively.

**Game 2.** The challenger $\mathcal{C}$ proceeds as in the previous game, but it replaces the random selection key $k_s$, and all the PRF values for generating the target OTS keys and the corresponding IoCs with random values. We may play similar hybrid games to replace them one-by one analogously in the Game 5 of the proof of Theorem 1. Due to the security of $\mathsf{PRG}$ and $\mathsf{PRF}$, we have that

$$\Pr[\mathsf{Win}_1^{\mathsf{Anony}}] \leq \Pr[\mathsf{Win}_2^{\mathsf{Anony}}] + 3 \cdot \mathsf{Adv}_{\mathcal{A},\mathsf{F}_1}^{\mathsf{IND\text{-}CMA}}(\kappa, 3) + 2^{\tilde{h}} \cdot \mathsf{Adv}_{\mathcal{A},\mathsf{G}}^{\mathsf{IND}}(\kappa).$$

As a result, the OTS keys of $\mathsf{GM}$ and uncorrupted members are generated with the same distributions defined by those key generation random values. Hence, the adversary cannot determine between the difference between the $\mathsf{GM}$'s OTS key and the member's OTS key. The OTS mounting scenarios of challenged parties' OTS keys (which are uncorrupted) in the obfuscated layer are hidden from the adversary.

**Game 3.** The game proceeds identically to the previous one, except that the challenge always uses $\hat{\mathsf{ID}}_0^*$ to generate the target ciphertexts in the privacy test query $\mathcal{O}_{\mathsf{PriTest}}(\mathsf{ID}_0^*, \mathsf{ID}_1^*, m)$, regardless of the challenge bit $b$.

To transition between these modifications, we employ hybrid games that sequentially modify the target ciphertexts one-by-one in each subgame. If an adversary $\mathcal{A}$ exists that can distinguish between two adjacent games, it can be leveraged to construct an algorithm $\mathcal{B}$ capable of breaking the security of $\mathsf{E}$. Specifically, $\mathcal{B}$ queries the $\mathsf{AE}$ challenger with identities $(\mathsf{ID}_0^*, X)$ to simulate the modified target ciphertext, where $X = \mathsf{GM}$ for ciphertexts intended for $\mathsf{GM}$, and $X = \mathsf{ID}_1^*$ otherwise. All other Identity-OPK ciphertexts are generated either by invoking the encryption oracle provided by the $\mathsf{AE}$ challenger or through secret keys produced directly by $\mathcal{B}$ itself. Consequently, we derive:

$$\Pr[\mathsf{Win}_{2}^{\mathsf{Anony}}] \leq \Pr[\mathsf{Win}_{3}^{\mathsf{Anony}}] + 4 \cdot \mathsf{Adv}_{\mathcal{A},\mathsf{E}}^{\mathsf{IND\text{-}CCA}}(\kappa, 4).$$

This game modification ensures that the adversary cannot deduce any information about the identity from the IOCs.

**Game 4.** This game proceeds as before but the challenger randomly chooses an index for each target OTS key from the remaining leaves that can mount without using the output of the $\mathsf{PRF}$ $\mathsf{F}_2$. Due to the security of $\mathsf{F}_2$, we have that

$$\Pr[\mathsf{Win}_{3}^{\mathsf{Anony}}] \leq \Pr[\mathsf{Win}_{4}^{\mathsf{Anony}}] + 2 \cdot \mathsf{Adv}_{\mathcal{A},\mathsf{F}_2}^{\mathsf{IND\text{-}CMA}}(\kappa, 2).$$

In the worst-case scenario, which is most advantageous for the adversary, only the two indices of the target OTS keys remain unknown. All other indices of honest OTS keys can be obtained from $\mathsf{HMSign}$ queries. Consequently, two unrevealed indices remain—one belonging to $\mathsf{ID}_0^*$ and the other to $\mathsf{ID}_1^*$. The attacker's goal is to determine which of these two indices belongs to which party, after obtaining the challenge signature. Despite differing probabilities in earlier selections, the final two unrevealed indices are equally likely to belong to either entity. Since the selection process is independent and random, the attacker has no additional information to distinguish ownership. Therefore, the probability of correctly identifying the owner of the given element is 1/2. This is because the attacker is left with only two remaining choices, one for each entity. Namely, the adversary gains no advantage in this game.

Putting together the advantages in all the above games can only result in negligible advantage, which proves Theorem 2.

## D Tree-based PPRF

In this section, we briefly revisit the key concept of tree-based PPRFs [9,26,1], built from the well-known one-way function based PRF construction, proposed by Goldreich, Goldwasser, and Micali (GGM) [20].

The GGM construction leverages on a pseudorandom generator (PRG) $\mathsf{G} : \{0,1\}^\ell \rightarrow \{0,1\}^{2\ell}$. The output string is then split into two equal parts, denoted as $\mathsf{G}_0(k)$ and $\mathsf{G}_1(k)$, representing the first and second halves, respectively. To define the PRF $\mathsf{F} : \{0,1\}^\ell \times \{0,1\}^n \rightarrow \{0,1\}^\ell$, the GGM construction organizes its domain as a binary tree. The PRF key $k$, generated by $\mathsf{F.Setup}(1^\kappa)$, serves as the root seed. Each leaf corresponds to a unique PRF output, while edges and internal nodes facilitate computation. Edges are labeled 0 or 1, indicating connections to left or right children, respectively, and each node is identified by the binary string of edge labels along the path from the root. For a message $x = x_1 x_2 \cdots x_n \in \{0,1\}^n$, the PRF value is computed by traversing the tree from the root to the leaf specified by $x$. Starting with the root seed $k$, the PRG $\mathsf{G}$ is iteratively applied along the path dictated by the bits of $x$. At each step,

$\mathsf{G}_{x_i}$ (either $\mathsf{G}_0$ or $\mathsf{G}_1$) is applied to the current intermediate value. This process yields the final output:

$$\mathsf{F.Eval}(k, x) = \mathsf{G}_{x_n} \circ \mathsf{G}_{x_{n-1}} \circ \cdots \circ \mathsf{G}_{x_1}(k) \in \{0,1\}^{\ell}.$$

The hierarchical structure of this tree-based construction ensures pseudorandomness and allows for efficient operations. Moreover, its flexibility enables the introduction of puncturing mechanisms. To puncture at $x = x_1 x_2 \ldots x_n$, realizing $\mathsf{PPRF.Punc}(k, x)$ , the initial root key $k$ is replaced with intermediate node evaluations for prefixes $\overline{x_1}, x_1\overline{x_2}, x_1x_2\overline{x_3}, \ldots, x_1x_2\ldots\overline{x_n}$. That is, these values are taken as the new key $k'$ output by $\mathsf{F.Punc}$ algorithm, allowing recomputation of the PRF for all inputs except $x$.

For implement the $\mathsf{RandSelect}$ function in our $\mathsf{FSDGS}$ constructions, we need and additional algorithm of $\mathsf{PPRF}$, i.e., $\mathsf{GetMsg}(k, i)$ to enable the retrieval of the i-th unpunctured message. To implement the $\mathsf{F.GetMsg}(k', i)$ algorithm based on the GGM construction, we devise the following steps:

1. Initialize currentPrefix = $\emptyset$.
2. For each level $j = 1$ to $n$:
   - Compute the number $\nu$ of accessible leaves in the left subtree:
     - Count the number $\mu$ of punctured message according to the prefixes with currentPrefix$||0$;
     - Compute $\nu := 2^{n-j} - \mu$.
   - If $i \leq \nu$, append $0 \to$ currentPrefix. Otherwise, append $1 \to$ currentPrefix and sets $i := i - \nu$.
3. Return currentPrefix as the binary representation of the $i$-th accessible input message.

The complexity of $\mathsf{F.GetMsg}$ primarily depends on the depth of the binary tree, which is determined by the bit-length $n$ of the key, and the number of stored intermediate evaluations, $\phi$, corresponding to the punctured points. The algorithm retrieves the $i$-th accessible input message by iterating through the nodes in the path from the root to the leaf representing the $i$-th unpunctured message. For each level in the tree, the algorithm verifies whether the current node or path is part of the stored intermediate evaluations. This lookup is $O(m)$ for each level since we may need to search among $\phi$ intermediate evaluations. Since there are $n$ levels in the tree, the overall complexity of the algorithm is $O(n \cdot \phi)$. This complexity reflects the worst-case scenario where all $\phi$ punctured points must be checked at each level.

## E Pseudo codes of HHGS

Here, we present the pseudocodes of the main algorithms of HHGS.

**Algorithm 1:** $\mathsf{GMInit}(1^\kappa, \mathsf{SP})$: Initialization algorithm for the group manager.

```
Input: Security parameter $1^\kappa$, setup parameter $\mathsf{SP} = (h, d)$.
Output: System parameters $\mathsf{Pms}$, group secret key $\mathsf{sk}_{\mathsf{GM}}$, and group public key $\mathsf{gpk}$.
1  Initialize PPRF schemes: $\{pms_{\mathsf{F}_p^i} := \mathsf{F}_p^i.\mathsf{Setup}(1^\kappa)\}_{0 \le i \le d}$. Initialize OTS
     scheme: $pms_{\mathsf{S}} := \mathsf{S}.\mathsf{Setup}(1^\kappa)$. Initialize MT scheme: $pms_{\mathsf{M}} := \mathsf{M}.\mathsf{Setup}(1^\kappa)$.
     Initialize AE scheme: $pms_{\mathsf{E}} := \mathsf{E}.\mathsf{Setup}(1^\kappa)$. Initialize PRG scheme:
     $pms_{\mathsf{G}} := \mathsf{G}.\mathsf{Setup}(1^\kappa)$. Initialize CRH scheme: $(hk, pms_{\mathsf{H}}) := \mathsf{H}.\mathsf{Setup}(1^\kappa)$.
     Initialize PRF schemes: $\{pms_{\mathsf{F}_i} := \mathsf{F}_i.\mathsf{Setup}(1^\kappa)\}_{i \in [2]}$. Set system parameters:
     $\mathsf{Pms} := (\{pms_{\mathsf{F}_p^i}\}_{0 \le i \le d}, pms_{\mathsf{S}}, pms_{\mathsf{M}}, pms_{\mathsf{E}}, pms_{\mathsf{G}}, pms_{\mathsf{H}}, pms_{\mathsf{F}_1}, pms_{\mathsf{F}_2}, hk)$.
2  for $i := 0$ to $d$ do
3      $k_p^i := \mathsf{F}_p^i.\mathsf{KGen}(pms_{\mathsf{F}_p^i})$.
4  end for
5  Generate random secret key for $\mathsf{F}_1$: $k_{\mathsf{GM}} := \mathsf{F}_1.\mathsf{KGen}(pms_{\mathsf{F}_1})$.
6  Set group secret key: $\mathsf{sk}_{\mathsf{GM}} := (\{k_p^i\}_{0 \le i \le d}, k_{\mathsf{GM}})$.
7  for $v \in [2^{h'}]$ do
8      $rs_{\mathsf{GM}}^{d,1,v} := \mathsf{F}_p^d(k_p^d, v)$. $(osk_{\mathsf{GM}}^{d,1,v}, opk_{\mathsf{GM}}^{d,1,v}) := \mathsf{S}.\mathsf{KGen}(rs_{\mathsf{GM}}^{d,1,v})$.
       $\mathrm{lf}_{d,1,v} := opk_{\mathsf{GM}}^{d,1,v}$.
9  end for
10 Build the top-most tree: $(\mathrm{Tr}_{d,1}, \mathsf{st}_{\mathrm{BDS}}^{d,1}) := \mathsf{M}.\mathsf{Build}(\{\mathrm{lf}_{d,1,v}\}_{v \in [2^{h'}]})$.
11 Set group public key: $\mathsf{gpk} := \mathrm{Tr}_{d,1}.\mathrm{Rt}$.
12 Set registered identity list: $\mathrm{RegIDL} := \emptyset$. Set revocation list: $\mathrm{RL} := \emptyset$. Set
     group management state: $\mathsf{st}_{\mathsf{GM}} := (\mathrm{RegIDL}, \mathrm{RL})$.
```

**Algorithm 2:** $\mathsf{MRegGen}(sk_{\mathsf{GM}})$: Generate registration file of OTS keys

```
Input: Member secret key $sk_{\mathsf{GM}}$, number of OTS keys $\ell_r$
Output: Registration file $\mathrm{RF}_{\mathsf{ID}}$
1  // Initialize PRG seed from the member's secret key
2  $sd_{\mathsf{ID}}^0 := sd_{\mathsf{ID}}$ extracted from $sk_{\mathsf{GM}}$;
3  // Generate $\ell_r$ OTS key pairs
4  for $v := 1$ to $\ell_r$ do
5      $(sd_{\mathsf{ID}}^v, rs_{\mathsf{ID}}^v) := \mathsf{G}.\mathsf{Eval}(sd_{\mathsf{ID}}^{v-1})$; $(rs_{\mathsf{ID}}^{v,1}, rs_{\mathsf{ID}}^{v,2}) := \mathsf{G}.\mathsf{Eval}(rs_{\mathsf{ID}}^v)$;
       $(osk_{\mathsf{ID}}^v, opk_{\mathsf{ID}}^v) := \mathsf{S}.\mathsf{KGen}(rs_{\mathsf{ID}}^{v,1})$;
6  end for
7  // Assemble the registration file and udpate the SK
8  $\mathrm{RF}_{\mathsf{ID}} := \{opk_{\mathsf{ID}}^v\}_{v \in [\ell_r]}$;
9  $sk_{\mathsf{ID}} := sd_{\mathsf{ID}}^{\ell_r}$;
```

**Algorithm 3:** $\mathsf{Join}(\mathsf{sk}_{\mathsf{GM}}, \mathsf{st}_{\mathsf{GM}}, \mathsf{ID}, \mathrm{RF}_{\mathsf{ID}})$: Join protocol for the group manager.

```
Input: Group secret key $\mathsf{sk}_{\mathsf{GM}}$, group management state $\mathsf{st}_{\mathsf{GM}}$, identity $\mathsf{ID}$,
       registration file $\mathrm{RF}_{\mathsf{ID}}$.
Output: Registration result $\mathrm{RR}_{\mathsf{ID}}$ and updated states $\mathsf{st}_{\mathsf{GM}}$, $\mathsf{st}_{\mathsf{ID}}$.
1  // Process each OTS public key in the registration file
2  foreach $opk_{\mathsf{ID}} \in RF_{\mathsf{ID}}$ do
3      $\mathrm{MI} := \mathsf{RandSelect}(\mathsf{sk}_{\mathsf{GM}}, \mathsf{ID}, opk_{\mathsf{ID}})$; Decompose MI into $\mathsf{sTI} := \mathrm{MI}(h)$ and
         sMI; Add $(\mathsf{ID}, opk_{\mathsf{ID}}^{\mathrm{MI}}, \mathrm{MI})$ to $\mathsf{RFCa}_{\mathsf{MMT}}^{\mathsf{sTI}}$;
4      // Determine if the sub-tree needs initialization or can be
         reused
5      if $\mathsf{F}_p^1(k_p^1, \mathsf{sTI}) \neq \perp$ then
6          // Case 1: Sub-tree initialization with new OTS leaves
7          Prepare $\mathsf{L}_{\mathrm{mI}}$ and $\overline{\mathsf{L}}_{\mathrm{mI}}$ for $\mathsf{MMT}[\mathsf{sTI}]$;
8          foreach $sMI \in \mathsf{L}_{mI}$ do
9              $ke_{\mathrm{MI}}^1 := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“K-Mix”}||\mathrm{MI})$; $re_{\mathrm{MI}}^1 := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“R-Mix”}||\mathrm{MI})$;
                $c_{\mathrm{MI}}^1 := \mathsf{E.Enc}(ke_{\mathrm{MI}}^1, \mathsf{ID}; re_{\mathrm{MI}}^1)$; $\mathrm{lf}_{\mathrm{sMI}} := \mathsf{H}(opk_{\mathsf{ID}}^{\mathrm{MI}}||c_{\mathrm{MI}}^1)$; Set $c_{\mathrm{MI}}^2 := \emptyset$,
                $opk_{\mathsf{GM}}^{\mathrm{MI}} := \emptyset$, $\sigma_{\mathsf{GM}}^{\mathrm{sMI}} := \emptyset$; $k_p^{0\prime} := \mathsf{F}_p^0.\mathsf{Punc}(k_p^0, \mathrm{MI})$;
10         end foreach
11         foreach $sMI' \in \overline{\mathsf{L}}_{mI}$ do
12             $\mathrm{MI}' := \mathsf{sTI}||\mathrm{sMI}'$; $(osk_{\mathsf{GM}}^{\mathrm{MI}'}, opk_{\mathsf{GM}}^{\mathrm{MI}'}) := \mathsf{S.KGen}(rs_{\mathsf{GM}}^{\mathrm{MI}'})$, where
                $rs_{\mathsf{GM}}^{\mathrm{MI}'} := \mathsf{F}_p^0(k_p^0, \mathrm{MI}')$; $ke_{\mathrm{MI}'}^1 := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“K-Mix”}||\mathrm{MI}')$;
                $re_{\mathrm{MI}'}^1 := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“R-Mix”}||\mathrm{MI}')$; $c_{\mathrm{MI}'}^1 := \mathsf{E.Enc}(ke_{\mathrm{MI}'}^1, \mathsf{GM}; re_{\mathrm{MI}'}^1)$;
13         end foreach
14         // Build and authenticate the new Merkle sub-tree
15         $(\mathsf{MMT}[\mathsf{sTI}], \mathsf{st}_{\mathrm{BDS}}^{\mathsf{MMT}[\mathsf{sTI}]}) := \mathsf{M.Build}(\{\mathrm{lf}_v\}_{v\in[2^{|\mathrm{sMI}|}]})$;
            $\mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}] := \mathsf{GetAuth}(\{k_p^i\}_{i\in[d]}, \mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}, \mathsf{sTI})$; foreach
            $sMI \in \mathsf{L}_{mI}$ do
16             $\mathrm{pf}_{\mathrm{lf}_{\mathrm{sMI}}} := \mathsf{M.GetPrf}(\mathsf{st}_{\mathrm{BDS}}^{\mathsf{MMT}[\mathsf{sTI}]}, \mathrm{lf}_{\mathrm{sMI}})$; for $i := 1$ to $d$ do
17                 $w := \frac{(d+1-i)\cdot h}{d}$; $k_p^{i\prime} := \mathsf{F}_p^i.\mathsf{Punc}(k_p^i, \mathrm{MI}(w))$;
18             end for
19         end foreach
20         Store $\mathsf{st}_{\mathrm{BDS}}^{\mathrm{lf}_{\mathrm{sMI}'}}$ and $\mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}]$ in $\mathsf{st}_{\mathsf{GM}}$;
21     end if
22     else
23         // Case 2: Reuse existing sub-tree and authenticate member
              OTS key
24         $(osk_{\mathsf{GM}}^{\mathrm{MI}}, opk_{\mathsf{GM}}^{\mathrm{MI}}) := \mathsf{S.KGen}(rs_{\mathsf{GM}}^{\mathrm{MI}})$, where $rs_{\mathsf{GM}}^{\mathrm{MI}} := \mathsf{F}_p^0(k_p^0, \mathrm{MI})$;
            $ke_{\mathrm{MI}}^1 := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“K-Mix”}||\mathrm{MI})$; $re_{\mathrm{MI}}^1 := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“R-Mix”}||\mathrm{MI})$;
            $c_{\mathrm{MI}}^1 := \mathsf{E.Enc}(ke_{\mathrm{MI}}^1, \mathsf{GM}; re_{\mathrm{MI}}^1)$; $\mathrm{lf}_{\mathrm{sMI}} := \mathsf{H}(opk_{\mathsf{GM}}^{\mathrm{MI}}||c_{\mathrm{MI}}^1)$;
            $\mathrm{pf}_{\mathrm{lf}_{\mathrm{sMI}}} := \mathsf{M.GetPrf}(\mathsf{st}_{\mathrm{BDS}}^{\mathrm{lf}_{\mathrm{sMI}}}, \mathrm{lf}_{\mathrm{sMI}})$; Retrieve $\mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}]$ from
            $\mathsf{st}_{\mathsf{GM}}$; $ke_{\mathrm{MI}}^2 := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“K-Mem”}||\mathrm{MI})$;
            $re_{\mathrm{MI}}^2 := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“R-Mem”}||\mathrm{MI})$; $c_{\mathrm{MI}}^2 := \mathsf{E.Enc}(ke_{\mathrm{MI}}^2, \mathsf{ID}; re_{\mathrm{MI}}^2)$;
            $\sigma_{\mathsf{GM}}^{\mathrm{sMI}} := \mathsf{S.Sign}(osk_{\mathsf{GM}}^{\mathrm{sMI}}, opk_{\mathsf{ID}}^{\mathrm{MI}}||c_{\mathrm{MI}}^2)$; Remove $\mathsf{st}_{\mathrm{BDS}}^{\mathrm{lf}_{\mathrm{sMI}}}$ and puncture MI;
25     end if
26     $\mathsf{Auth}[opk_{\mathsf{ID}}^{\mathrm{MI}}] :=$
         $(opk_{\mathsf{GM}}^{\mathrm{MI}}, c_{\mathrm{MI}}^1, c_{\mathrm{MI}}^2, \mathrm{pf}_{\mathrm{lf}_{\mathrm{sMI}}}, \sigma_{\mathsf{GM}}^{\mathrm{sMI}}, \mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}, \mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}])$;
27     Append $\mathsf{Auth}[opk_{\mathsf{ID}}^{\mathrm{MI}}]$ to $\mathrm{RR}_{\mathsf{ID}}$;
28 end foreach
29 Update $\mathsf{st}_{\mathsf{GM}}$ and $\mathsf{st}_{\mathsf{ID}}$;
```

**Algorithm 4:** $\mathsf{Sign}(sk_{\mathsf{ID}}, \mathsf{st}_{\mathsf{ID}}, m)$: Signature generation by a member

```
Input: Member secret key $sk_{\mathsf{ID}}$, member state $\mathsf{st}_{\mathsf{ID}}$, message $m$
Output: Group signature $\sigma$ and updated states $sk'_{\mathsf{ID}}$, $\mathsf{st}'_{\mathsf{ID}}$
1  // Generate fresh randomness and keys
2  $(sd^{v}_{\mathsf{ID}}, rs^{v}_{\mathsf{ID}}) := \mathsf{G.Eval}(sd^{v-1}_{\mathsf{ID}})$; $(rs^{v,1}_{\mathsf{ID}}, rs^{v,2}_{\mathsf{ID}}) := \mathsf{G.Eval}(rs^{v}_{\mathsf{ID}})$;
    $(osk_{\mathsf{ID}}, opk_{\mathsf{ID}}) := \mathsf{S.KGen}(rs^{v,1}_{\mathsf{ID}})$;
3  // Retrieve latest authentication path
4  $\mathsf{Auth}[opk_{\mathsf{ID}}]$ := first entry from $\mathsf{st}_{\mathsf{ID}}$;
5  // Check which sub-layer the OTS key belongs to
6  if $\sigma^{sMI}_{\mathsf{GM}} \neq \emptyset$ in $\mathsf{Auth}[opk_{\mathsf{ID}}]$ then
7      $\sigma_{\mathsf{ID}} := \mathsf{S.Sign}(osk_{\mathsf{ID}}, m)$;
8      $\sigma := (opk_{\mathsf{ID}}, \mathsf{Auth}[opk_{\mathsf{ID}}], \sigma_{\mathsf{ID}})$;
9  end if
10 else
11     $(osk'_{\mathsf{ID}}, opk'_{\mathsf{ID}}) := \mathsf{S.KGen}(rs^{v,2}_{\mathsf{ID}})$;
12     $\sigma_{\mathsf{ID}} := \mathsf{S.Sign}(osk'_{\mathsf{ID}}, m)$;
13     Generate $ke' \xleftarrow{\$} \mathcal{K}_{\mathsf{AE}}$ and $re' \xleftarrow{\$} \mathcal{ER}_{\mathsf{AE}}$;
14     $c'_{\mathsf{ID}} := \mathsf{E.Enc}(ke', opk'_{\mathsf{ID}}; re')$;
15     $\sigma'_{\mathsf{ID}} := \mathsf{S.Sign}(osk_{\mathsf{ID}}, opk'_{\mathsf{ID}} || c'_{\mathsf{ID}})$;
16     Fill $\mathsf{Auth}[opk_{\mathsf{ID}}]$ with $opk_{\mathsf{ID}}$, $\sigma'_{\mathsf{ID}}$, $c'_{\mathsf{ID}}$;
17     $\sigma := (opk'_{\mathsf{ID}}, \mathsf{Auth}[opk_{\mathsf{ID}}], \sigma_{\mathsf{ID}})$;
18 end if
19 // Update member state and key
20 $sk'_{\mathsf{ID}} := sd^{v}_{\mathsf{ID}}$;
21 $\mathsf{st}'_{\mathsf{ID}} := \mathsf{st}_{\mathsf{ID}} \setminus (opk_{\mathsf{ID}}, \mathsf{Auth}[opk_{\mathsf{ID}}])$;
```

**Algorithm 5:** $\mathsf{Verify}(\mathsf{gpk}, m, \sigma, \mathrm{RL})$: Signature verification algorithm

```
Input: Group public key $\mathsf{gpk}$, message $m$, signature $\sigma$, revocation list RL
Output: 1 if valid, 0 otherwise
// Parse the input signature and authentication path
Parse $\sigma := (opk^{'}_{\mathsf{ID}}, \mathsf{Auth}[opk_{\mathsf{ID}}], \sigma_{\mathsf{ID}})$; Parse $\mathsf{Auth}[opk_{\mathsf{ID}}] :=$
  $(opk^{\mathrm{MI}}_{\mathsf{GM}}, c^1_{\mathrm{MI}}, c^2_{\mathrm{MI}}, \mathrm{pf}_{\mathrm{lf}_{\mathrm{sMI}}}, \sigma^{\mathrm{sMI}}_{\mathsf{GM}}, \mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}, \mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}])$;
// Check revocation status
if $c^1_{MI} \in RL$ then
    return 0;
end if
// Verify the authentication path in the hypertree
if $\mathsf{AuthVrfy}(\mathsf{gpk}, \mathsf{MMT}[\mathsf{sTI}].Rt, \mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].Rt]) = 0$ then
    return 0;
end if
// Verify Merkle proof of GM's leaf in subtree
$\mathrm{lf}_{\mathrm{sMI}} := \mathsf{H}(opk^{\mathrm{MI}}_{\mathsf{GM}} || c^1_{\mathrm{MI}})$; if $\mathsf{M.Verify}(\mathsf{MMT}[\mathsf{sTI}].Rt, lf_{sMI}, pf_{lf_{sMI}}) = 0$ then
    return 0;
end if
// Verify GM's signature on member's OTS key and ciphertext
if $\mathsf{S.Verify}(opk^{MI}_{\mathsf{GM}}, opk^{'}_{\mathsf{ID}} || c^2_{MI}, \sigma^{sMI}_{\mathsf{GM}}) = 0$ then
    return 0;
end if
// Verify member's OTS signature on the message
if $\mathsf{S.Verify}(opk^{'}_{\mathsf{ID}}, m, \sigma_{\mathsf{ID}}) = 0$ then
    return 0;
end if
return 1;
```