# HHGS: Forward-secure Dynamic Group Signatures from Symmetric Primitives

Xuelian Cao[1], Zheng Yang[2], Daniel Reijsbergen[3], Jianting Ning[4⋆], Junming Ke[5], Zhiqiang Ma[6], and Jianying Zhou[3]

[1] Tsinghua University, Beijing, China
xl-cao@mail.tsinghua.edu.cn
[2] Southwest University, Chongqing, China
[3] Singapore University of Technology and Design, Singapore
{daniel_reijsbergen,jianying_zhou}@sutd.edu.sg
[4] Wuhan University, Wuhan, China, and City University of Macau
jtning88@gmail.com
[5] Hangzhou Research Institute of AI and Holographic Technology, Hangzhou, China
junmingke@zjuqx.com
[6] Fujian Normal University, Fuzhou, China
zhiqiang-ma@hotmail.com

**Abstract.** Group signatures allow a group member to sign messages on behalf of the group while preserving the signer's anonymity, making them invaluable for privacy-sensitive applications. As quantum computing advances, post-quantum security in group signatures becomes essential. Symmetric primitives (SP) offer a promising pathway due to their simplicity, efficiency, and well-understood security foundations. In this paper, we introduce the first *forward-secure dynamic group signature* (FSDGS) framework relying solely on SP. We begin with *hierarchical hypertree group signatures* (HHGS), a basic scheme that securely organizes keys of one-time signatures (OTS) in a hypertree using puncturable pseudorandom functions to enable on-demand key generation and forward security, dynamic enrollment, and which provides resilience against attacks that exploit registration patterns by obfuscating the assignment and usage of keys. We then extend this foundation to HHGS$^+$, which orchestrates multiple HHGS instances in a generic way, significantly extending the total signing capacity to $O(2^{60})$, which outperforms HHGS's closest competitors while keeping signatures below 8 kilobytes. We prove the security of both schemes in the standard model. Our results outline a practical SP-driven pathway toward post-quantum-secure group signatures suitable for resource-constrained client devices.

**Keywords:** Group signatures · Forward security · Post-quantum security · Symmetric primitives · Standard model.

⋆ Jianting Ning is the corresponding author.

## 1 Introduction

Group signatures (GS) serve as an essential component in privacy-preserving applications such as anonymous credentials [10], trusted computing via direct anonymous attestation [15], enhanced privacy identification [26], digital rights management [36], and biometric authentication [16]. GS allow a group member to sign messages on behalf of the group while preserving anonymity, ensuring that a verifier can confirm the signature originates from a legitimate group member without uncovering the signer's identity. This powerful feature makes GS a fundamental cryptographic primitive for modern security architectures.

As quantum computing threatens the foundations of classical public-key cryptographic systems, the development of post-quantum secure group signatures has become imperative. Various post-quantum constructions have been proposed, leveraging lattice [44,9], code [39], isogeny [9], multivariate [40,43], and symmetric primitives (SP) [2,11,18,21,45]. Each approach has its strengths and limitations. For instance, lattice-based schemes offer resistance against quantum attacks and asymptotic efficiency, but the resulting group signatures still have a signature size of several hundred kilobytes [38]. In contrast, symmetric primitives offer simplicity and efficiency and, because the best known quantum attacks (e.g. [28,14]) against them are generic and predictably bounded.

**Related Work.** There are two primary ways for constructing GS schemes from SP. The first approach lies in the generic GS framework by Bellare et al. [4] based on non-interactive zero-knowledge (NIZK) proofs. The core design of the state-of-the-art NIZK-and-SP based GS [11,21] leverages the "MPC-in-the-Head" technique of Ishai et al. [33] being SP-compatible for achieving post-quantum security. However, the introduction of NIZK in [11,21] results in large signature sizes, with group signatures in [11,21] reaching sizes in the order of megabytes. Moreover, the NIZK-based GS schemes [11,21] are only proven secure in the random oracle model because of the use of Fiat-Shamir-like transformations [23]. However, most commonly-deployed hash functions deviate significantly from the ideal properties assumed in the random oracle model. This gap between the theoretical model and practical implementations can lead to invalid security guarantees [19,22]. Accordingly, although these NIZK-based GS schemes can provide full dynamic security properties [13] (such as full anonymity), they cannot achieve forward security because of the static signing keys of participants. Forward security is a fundamental requirement for group signatures, first formalized by Song [42], ensuring that compromise of a (current) signing key does not permit the forgery of valid group signatures for any earlier time period. NIZK-based GS schemes often require substantial computations on the client device (where the group credential is stored) to produce the zero-knowledge proofs. This high cost in both bandwidth and local processing can be particularly prohibitive for resource-constrained devices (e.g., smart cards or IoT devices whose computational and storage resources are limited).

The alternative approach explored in our paper leverages one-time signatures (OTS) [2,18,45] in combination with symmetric primitive-based authentication mechanisms, such as Merkle trees, to generate group authentication credentials

for members' OTS keys. However, compared to NIZK-based schemes, OTS-based group signature schemes may offer weaker security (or fewer functionalities) and have smaller signing capacities. The G-Merkle scheme [2] supports only static group management, as it relies on a single Merkle tree and a pseudorandom permutation (PRP) scheme with fixed inputs. The enhanced $\text{GM}^{\text{MT}}$ [45] employs a hypertree structure of multiple sub-trees to enable dynamic group management (**DGMA**), yet it guarantees anonymity only within a single sub-tree [45, \$4.2]. That is, $\text{GM}^{\text{MT}}$ cannot achieve anonymity across the entire group management lifecycle because an adversary can mount the OTS keys of two challenged parties in different sibling sub-trees rather than the same sub-tree (referred to as a *sibling attack*, see also in Section 4). Consequently, any signatures from sub-trees created before a member is joined is necessarily attributed to the older member. Additionally, G-Merkle and $\text{GM}^{\text{MT}}$ offer limited signing capacities of $O(2^{20})$ and $O(2^{26})$, respectively. Another work following G-Merkle, called dynamic G-Merkle (DGM) [18], allows new users to join the group at any time and distributes members' OTS keys across different sub-trees. After each sub-tree is generated, its root $r_{SMT}$ is transformed into an independent fallback key $F_k$ via the symmetric encryption SE, i.e., $F_k := \text{SE.Dec}(F_n, r_{SMT})$ with an encryption key $F_n$. However, DGM requires the group manager to store all fallback keys and remain online to handle verifier queries concerning the validity of the corresponding fallback keys $F_k$ in signatures. Consequently, it fails to provide members with *self-authenticable* group credentials (**SAGC**), which would allow them to independently validate and generate signatures without relying on the group manager for real-time support. In the latter, DGMT [25] extends DGM to enable SAGC, i.e., allowing non-interactive verification of signatures, by pre-publishing fallback keys. However, this design introduces a centralization threat like G-Merkle: the manager, possessing all signing keys of members, could potentially forge signatures or suffer single-point failures, thus limiting applicability in settings requiring decentralized trust and user-controlled keys (**UCK**). Moreover, to the best of our knowledge, none of the existing OTS-based GS schemes can achieve forward security.

**Motivations.** To address the shortcomings of existing schemes, we focus on the fundamental research problem of how to design a forward-secure dynamic group signature (FSDGS) scheme using symmetric primitives that achieves the following properties: i) forward security, ii) DGMA, iii) SAGC, iv) UCK, v) small signature size, and vi) provable security in the standard model. We will follow the line of OTS-based approach, since it is friendly to resource-constrained devices requiring few OTS signing operations, rather than expensive NIZK operations. Although group credentials (such as Merkle proofs) in OTS-based schemes [2,18,45] might be large, they can be stored on any honest auxiliary device, rather than the device for storing the secret keys in NIZK-based GS schemes. That is, our design will target such a *resource constrained-auxiliary collaboration* (RCAC) scenario, including smart cards with mobile devices, drones with ground stations, and IoT devices with smart gateways. Meanwhile, our

aim is to improve the signing capacity with both SAGC and UCK, and design techniques to avoid sibling attacks (unlike during dynamic group management).

**Our Contributions and Techniques.** This work presents the first SP-based FSDGS framework, addressing the motivated research problem. First, we propose HHGS, an innovative FSDGS scheme that leverages a hierarchical OTS-key management structure to enable efficient DGMA. HHGS is the first secure OTS-based GS to achieve DGMA, SAGC and UCK, alongside the resilience against sibling attacks (a security gap in existing schemes). By leveraging puncturable pseudorandom functions (PPRF), HHGS ensures forward security and significantly expands signing capacity to $O(2^{40})$, surpassing previous OTS-based GS schemes with SAGC and UCK by over $2^{14}$ times under similar join-time efficiency.[7] Second, we extend this design with HHGS$^+$, a scalable enhancement that integrates modular HHGS instances to significantly improve signing capabilities and accommodate larger group sizes. Consequently, the total signing capacity of HHGS$^+$ reaches $O(2^{60})$, accommodating most practical applications, while both schemes can maintain signature sizes under 8 kilobytes. Lastly, we rigorously prove the security of these schemes in the standard model, ensuring that they meet anonymity and traceability.

The core of our design lies in HHGS, which organizes OTS keys of a group manager (GM) and group members into an OTS hypertree (as in [7,31]) to support dynamic group management. The group credential of each member's OTS key is a signature from the hypertree indexed by a unique leaf. The key innovation of HHGS is a tailored obfuscated leaf layer of sub-trees mounted to the hypertree, which consists of two sub-layers (mixed sub-layer and member sub-layer). This design bridges the gap between the determinism of each sub-tree and the dynamic OTS mounting requirements within the sub-tree. All OTS keys in the member sub-layer are from different members joined at any time, each of which is mounted (signed) by its parent, an OTS key, in the mixed sub-layer. The mixed sub-layer encompasses the OTS keys from both members and GM to obfuscate the joining time of these OTS keys. We make use of the fact that the OTS keys from members and GM are generated from the same distribution, so they can be indistinguishable when the owners are uncorrupted. Thus, deterministic sub-trees enable GM to handle dynamic and unpredictable joining requests from members. Our construction addresses sibling attacks by customizing a randomized leaf assignment mechanism. This mechanism employs a pseudorandom function (PRF) to ensure that signing keys are independently distributed across members, preventing adversaries from exploiting shared structural information in signatures to identify members. Since the outputs of existing random functions including PRF are not truly random, potential collisions may occur during the key assignment. To address this, we utilize puncturable pseudorandom functions (PPRFs) in our customized random selection methods to track assigned

[7] Theoretically, all state-of-the-art OTS-based GS schemes supporting SAGC and UCK can achieve signing capacities approaching $O(2^{60})$. However, due to differences in underlying design principles, their practical performance can vary significantly. We aim to compare them under similar (reported) performance conditions.

signing keys. By puncturing previously assigned keys, PPRFs effectively prevent key reuse and resolve collisions. Additionally, PPRFs enable GM to update its secret key after each joining operation, ensuring forward security for its generated signatures. For resource-constrained members, pseudorandom generators serve as a lightweight alternative for generating independent OTS keys and updating secret keys to achieve forward security.

Building on the foundation of HHGS, we introduce HHGS$^+$, an extension designed to efficiently accommodate large group sizes and dynamic workloads. HHGS$^+$ adopts a modular framework by initializing multiple HHGS instances and organizing them within a unified structure. Specifically, a short-range PRF is used to randomly assign an OTS key to a suitable HHGS instance, utilizing collisions among PRF values. A key challenge in HHGS$^+$ lies in randomizing the assignment of OTS keys across HHGS instances while avoiding biases caused by the non-uniform distribution of PRF values, which could lead to earlier full-instance scenarios. To address this, HHGS$^+$ integrates redundancy within each HHGS instance, countering potential biases and mitigating adversarial exploitation of uneven assignments. Through formal analysis, we establish a redundancy model that ensures secure and balanced key distribution while maintaining minimal overhead, enabling HHGS$^+$ to handle dynamic group operations with high scalability and robustness.

**Comparison.** Table 1 summarizes a comparison of our proposed schemes with some existing ones according to underlying construction mechanisms. The first compared category involves group signatures (i.e., G-Merkle [2] and GM$^{\mathrm{MT}}$ [45]) which are composed of OTS signatures and self-authenticable group membership credentials (which exclude DGM [18]). G-Merkle, GM$^{\mathrm{MT}}$ and DGMT differ from the group credential and the features of supporting dynamic group management (DGMT) and user-centric key (UCK) management. G-Merkle uses a single Merkle tree to create group credentials, so it does not provide DGMA. Additionally, both G-Merkle and DGMT does not support UCK, making them less secure and considerably easier to construct compared to other schemes. In contrast, the group credentials in GM$^{\mathrm{MT}}$ are generated using XMSS$^{\mathrm{T}}$ [30] equipped with a hypertree. While it supports DGMA, it fails to provide resilience against sibling attacks (SA) [45, §4.2]. SA applies only to DGMA-enabled GS schemes.

Table 1: Comparison among SP-based Group Signature Schemes.

| Schemes | Sig. Technique | Credential Structure | UCK Support | Group Management | Standard Model | Forward Security | SA Resilience | Group Size | Each Mem. Sig. Capacity | Sig. Size |
|---|---|---|---|---|---|---|---|---|---|---|
| G-Merkle [2] | OTS | Merkle tree | × | Static | √ | × | - | $2^6$ | $2^{14}$ | 1.9 |
| GM$^{\mathrm{MT}}$ [45] | OTS | Hypertree | √ | Dynamic | √ | × | × | $2^{16}$ | $2^{10}$ | 3.3 |
| DGMT [25] | OTS | Hypertree | × | Dynamic | √ | × | √ | $2^7$ | $2^{26.76}$ | 5.344 |
| BEB [11] | NIZK | Merkle tree | √ | Static | × | × | - | $2^{40}$ | unlimit | 6650 |
| SPHINX [21] | NIZK | Hypertree | √ | Dynamic | × | × | √ | $2^{60}$ | unlimit | 2000 |
| HHGS | OTS | Hypertree | √ | Dynamic | √ | √ | √ | $2^{20}$ | $2^{20}$ | 6.41 |
| HHGS$^+$ | OTS | Hypertree | √ | Dynamic | √ | √ | √ | $2^{30}$ | $2^{30}$ | 7.04 |

In the second category, a group signature comprises of a SP-based NIZK proof of the knowledge of a group membership credential, such as BEB [11] and SPHINX [21]. The credential in BEB [11], is a signature of GM based on a single Merkle tree, so it only provides static group management. SPHINX [21] is a fully

dynamic group signature with a hash-based signature, called F-SPHINCS+, and optimized for efficient NIZK proofs. Unlike previous works, it supports large group sizes (up to $2^{60}$).

It is important to note that the comparison between OTS-based group signatures and NIZK-based schemes involves different design goals and security guarantees. In particular, NIZK-based schemes such as BEB and SPHINX can achieve strong anonymity properties, for example full anonymity that holds even against adversaries that obtain all users' secret keys, and they typically provide more compact user-side secret keys. This comes at the cost of larger signature sizes and higher computational overhead for signing and verification. In contrast, our OTS-based constructions are tailored to RCAC scenarios with forward security for resource-constrained devices. They avoid executing computationally intensive NIZK protocols on the client side, at the expense of larger credentials and relatively weaker anonymity guarantees under full key exposure.

Moreover, all compared schemes cannot provide forward security. Unlike NIZK-based GS schemes, our schemes are more suitable for RCAC scenarios involving resource-constrained devices, as they do not require running NIZK operations on devices that store the secret key. Instead, our schemes only need to execute one $\mathsf{PRG}$ operation, one OTS key generation, and two OTS signing operations on the client device (e.g., smart card). Although the group credentials are not small, they can be stored on an auxiliary storage device (e.g., mobile phone), and the leakage of group credentials does not compromise traceability. To further reduce storage costs, the client can register OTS keys on-demand, rather than registering all of them at once. Our schemes make a trade-off between group size and signature capacity. The total OTS keys' amount of our scheme is $O(2^{60})$, and it can *adaptively adjust the ratio of members to their OTS keys*. For simplicity, we consider an equal split (as an example), which can meet the requirements of most practical applications.

We also provide a rough comparison of the signature sizes (based on results from the relevant literature) across different schemes, considering the corresponding group sizes. While the signatures in our schemes are slightly larger than those of G-Merkle, $\mathrm{GM}^{\mathrm{MT}}$, and DGMT, this increase stems from our simultaneous support for dynamic group management, forward security, user-controlled key management, and sibling attack resilience. However, they are significantly smaller than those of the NIZK-based schemes.

## 2 Preliminaries

**Notation.** We let $\kappa$ be the security parameter and $\emptyset$ be an empty string. The set of integers between 1 and $n$ is represented by $[n] = \{1, \ldots, n\} \subset \mathbb{N}$. We use $\|$ to denote the string concatenation operation, and $\#$ to represent an operation to calculate the size of an element. Moreover, we denote by $y \leftarrow \mathcal{A}(x)$ the execution of algorithm $\mathcal{A}$ on input $x$, resulting in output $y$. The notation $x \xleftarrow{\$} X$ represents the operation of sampling $x$ uniformly at random from a set $X$.

In the following, we manly review the syntax of the main cryptographic building blocks of our upcoming constructions. The security definitions of them are presented in Appendix A. We assume the *parameters* generated by the setup algorithm of a primitive may be implicitly used by its other algorithms.

**Digital Signature.** A digital signature scheme $\mathsf{SIG} = (\mathsf{Setup}, \mathsf{KGen}, \mathsf{Sign}, \mathsf{Verify})$ involves four algorithms. $\mathsf{Setup}(1^\kappa)$ initializes parameters $pms_{\mathsf{SIG}}$, defining the randomness space $\mathcal{RS}_{\mathsf{SIG}}$, secret key $\mathcal{SK}_{\mathsf{SIG}}$, public key space $\mathcal{PK}_{\mathsf{SIG}}$, and signature space $\mathcal{S}_{\mathsf{SIG}}$. $\mathsf{KGen}(rs)$ uses randomness $rs \in \mathcal{RS}_{\mathsf{SIG}}$ to produce a secret key $sk \in \mathcal{SK}_{\mathsf{SIG}}$ and its corresponding public key $pk \in \mathcal{PK}_{\mathsf{SIG}}$. $\mathsf{Sign}(sk, m)$ generates a signature $\sigma \in \mathcal{S}_{\mathsf{SIG}}$ for a message $m \in \mathcal{M}_{\mathsf{SIG}}$ using the secret key $sk$. $\mathsf{Verify}(pk, m, \sigma)$ checks if $\sigma$ is a valid signature on $m$ under $pk$, outputting 1 if valid and 0 otherwise. One-time signature (OTS) schemes, designed for securely signing a single message with a unique key pair, ensure authenticity and resistance to forgery, with security analyzed under adaptive chosen message attacks.

**Merkle Tree.** A Merkle tree scheme consists of four algorithms, $\mathsf{MT} = (\mathsf{Setup}, \mathsf{Build}, \mathsf{GetPrf}, \mathsf{Verify})$. $\mathsf{Setup}(1^\kappa)$ takes as input $1^\kappa$ and outputs the parameter $pms_{\mathsf{MT}}$ defining leaf space $\mathcal{L}_{\mathsf{M}}$ and proof space $\mathcal{PF}_{\mathsf{M}}$. $\mathsf{Build}(\{\mathrm{lf}_i\}_{i\in[N]})$ constructs a Merkle tree instance Tr based on these leaves in $\mathcal{L}_{\mathsf{M}}$, and outputs the initial state $\mathsf{st}_{\mathrm{BDS}}$ for the tree traversal algorithm described in [17] (commonly referred to as the BDS algorithm). Moreover, we let $\mathsf{st}_{\mathrm{BDS}}^{\mathrm{lf}_i} \subseteq \mathsf{st}_{\mathrm{BDS}}$ denote the BDS state for a specific leaf $\mathrm{lf}_i$. $\mathsf{GetPrf}(\mathsf{st}_{\mathrm{BDS}}, \mathrm{lf}_i)$ takes as input the state $\mathsf{st}_{\mathrm{BDS}}$ and a leaf $\mathrm{lf}_i$, and out outputs a Merkle proof $\mathrm{pf}_{\mathrm{lf}_i} \in \mathcal{PF}_{\mathsf{M}}$ that attests to the inclusion of $\mathrm{lf}_i$ in the tree, along with the updated BDS state $\mathsf{st}_{\mathrm{BDS}}$. $\mathsf{Verify}(\mathrm{Tr.Rt}, \mathrm{lf}_i, \mathrm{pf}_{\mathrm{lf}_i})$ takes as input the root Tr.Rt, a leaf node $\mathrm{lf}_i$, and the corresponding proof $\mathrm{pf}_{\mathrm{lf}_i}$, and returns 1 returning 1 if the proof validates $\mathrm{lf}_i$'s inclusion in Tr.Rt, and 0 otherwise. The Merkle proofs of a secure $\mathsf{MT}$ scheme should satisfy unforgeability.

**Authenticated Encryption.** An authenticated encryption (AE) scheme consists of four algorithms $\mathsf{AE} = (\mathsf{Setup}, \mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$. $\mathsf{Setup}(1^\kappa)$ initializes the scheme with security parameter $\kappa$, outputting public parameters $pms_{\mathsf{AE}}$ that define the key space $\mathcal{K}_{\mathsf{AE}}$, plaintext space $\mathcal{M}_{\mathsf{AE}}$, and ciphertext space $\mathcal{C}_{\mathsf{AE}}$. $\mathsf{KGen}(rk)$ generates a symmetric key $k \in \mathcal{K}_{\mathsf{AE}}$ using randomness $rk \in \mathcal{RK}_{\mathsf{AE}}$. $\mathsf{Enc}(k, m)$ encrypts a message $m \in \mathcal{M}_{\mathsf{AE}}$ with key $k$, producing ciphertext $c \in \mathcal{C}_{\mathsf{AE}}$. $\mathsf{Dec}(k, c)$ decrypts $(k, c) \in \mathcal{K}_{\mathsf{AE}} \times \mathcal{C}_{\mathsf{AE}}$, outputting plaintext $m$ or failure symbol $\perp$. For security of $\mathsf{AE}$, we consider the security properties regarding indistinguishability under adaptive chosen-ciphertext attacks, and the integrity of ciphertext.

**Pseudorandom Generators.** A pseudorandom generator consists of three algorithms $\mathsf{PRG} = (\mathsf{Setup}, \mathsf{SGen}, \mathsf{Eval})$. $\mathsf{Setup}(1^\kappa)$ initializes the scheme with security parameter $\kappa$, producing parameters $pms_{\mathsf{PRG}}$ that define the input space $\mathcal{S}_{\mathsf{PRG}}$ and range space $\mathcal{R}_{\mathsf{PRG}}$. $\mathsf{SGen}(pms_{\mathsf{PRG}})$ uses these parameters to output a random seed $s \xleftarrow{\$} \mathcal{S}_{\mathsf{PRG}}$, while the randomness generation algorithm $\mathsf{Eval}(s)$ takes a seed $s \in \mathcal{S}_{\mathsf{PRG}}$ and outputs a pseudorandom value $r \in \mathcal{R}_{\mathsf{PRG}}$. The output of $\mathsf{PRG}$ must be indistinguishable from a truly random value. For a specific PRG scheme $\mathsf{G}$, we may use the shorthand notation $\mathsf{G}(s)$ to represent $\mathsf{G.Eval}(s)$.

**(Puncturable) Pseudo-random Functions.** We define a pseudorandom function (PRF) family with three algorithms $\mathsf{PRF} = (\mathsf{Setup}, \mathsf{KGen}, \mathsf{Eval})$. $\mathsf{Setup}(1^\kappa)$ initializes the scheme with security parameter $\kappa$, producing parameters $pms_{\mathsf{PRF}}$ that define the key space $\mathcal{K}_{\mathsf{PRF}}$, message space $\mathcal{M}_{\mathsf{PRF}}$, and range space $\mathcal{R}_{\mathsf{PRF}}$. $\mathsf{KGen}(pms_{\mathsf{PRF}})$ generates a random secret key $k \xleftarrow{\$} \mathcal{K}_{\mathsf{PRF}}$. $\mathsf{Eval}(k, x)$ computes the evaluation result $r \in \mathcal{R}_{\mathsf{PRF}}$ for a message $x \in \mathcal{M}_{\mathsf{PRF}}$ using the secret key $k$.

A puncturable pseudorandom function (PPRF) family is defined by four algorithms, $\mathsf{PPRF} = (\mathsf{Setup}, \mathsf{KGen}, \mathsf{Eval}, \mathsf{Punc})$, with the same spaces as $\mathsf{PRF}$ for simplicity. The syntax of the first two algorithms mirrors that of $\mathsf{PRF}$. The puncturing algorithm $\mathsf{Punc}(k, x)$ takes as input a key $k \in \mathcal{K}_{\mathsf{PRF}}$ and a message $x \in \mathcal{M}_{\mathsf{PRF}}$, and outputs a punctured key in $\mathcal{K}_{\mathsf{PRF}}$. For a specific (P)PRF function $\mathsf{F}$, we may use the shorthand notation $\mathsf{F}(k, x)$ to represent $\mathsf{F}.\mathsf{Eval}(k, x)$.

**Collision-resistant Hash Functions.** We define a collision-resistant ($\mathsf{CR}$) hash function $\mathsf{CRH} = (\mathsf{Setup}, \mathsf{Eval})$ as a keyed hash function with two algorithms. $\mathsf{Setup}(1^\kappa)$ initializes the scheme with security parameter $\kappa$, producing parameters $pms_{\mathsf{CRH}}$ and a random key $hk \xleftarrow{\$} \mathcal{K}_{\mathsf{CRH}}$, where $pms_{\mathsf{CRH}}$ defines the key space $\mathcal{K}_{\mathsf{CRH}}$, message space $\mathcal{M}_{\mathsf{CRH}}$, and hash value space $\mathcal{Y}_{\mathsf{CRH}}$. $\mathsf{Eval}(hk, m)$ computes a hash value $y \in \mathcal{Y}_{\mathsf{CRH}}$ for a message $m \in \mathcal{M}_{\mathsf{CRH}}$ using key $hk$. For a given $\mathsf{CRH}$ scheme $\mathsf{H}$, we use $\mathsf{H}(m)$ as shorthand for $\mathsf{H}.\mathsf{Eval}(hk, m)$ when the hash key $hk$ is clear from the context.

## 3 Security Model of FSDGS

**Syntax.** A $\mathsf{FSDGS}$ scheme involves a group manager ($\mathsf{GM}$), verifiers, and group members. The $\mathsf{GM}$ is a trusted third party responsible for system initialization and group member management. Each group member has a unique identity $\mathsf{ID}$ and can anonymously create a group signature. Verifier, who might be an outsider or a member of the group, can verify the validity of a group signature. We define the syntax of $\mathsf{FSDGS}$ via the following algorithms $\Sigma = (\mathsf{GMInit}, \mathsf{MInit}, \mathsf{MRegGen}, \mathsf{Join}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Open}, \mathsf{Revoke})$:

- $(\mathsf{Pms}, \mathsf{gpk}, \mathsf{sk}_{\mathsf{GM}}, \mathsf{st}_{\mathsf{GM}}, \mathrm{RL}) \leftarrow \mathsf{GMInit}(1^\kappa, \mathsf{SP})$: The $\mathsf{GM}$ initialization algorithm takes as input the security parameter $1^\kappa$ and a setup parameter $\mathsf{SP}$ required by a specific scheme. This algorithm outputs a system parameter $\mathsf{Pms}$, a group public and private key pair ($\mathsf{gpk}$,$\mathsf{sk}_{\mathsf{GM}}$), the group management state $\mathsf{st}_{\mathsf{GM}}$ of the group manager $\mathsf{GM}$, and a revocation list RL.
- $(sk_{\mathsf{ID}}, \mathsf{st}_{\mathsf{ID}}) \leftarrow \mathsf{MInit}(\mathsf{ID})$: A group member $\mathsf{ID}$ runs the member initialization algorithm to generate its secret key $sk_{\mathsf{ID}}$ and the initial state $\mathsf{st}_{\mathsf{ID}}$.
- $\mathrm{RF}_{\mathsf{ID}} \leftarrow \mathsf{MRegGen}(sk_{\mathsf{ID}})$: A group member $\mathsf{ID}$ uses its secret key $sk_{\mathsf{ID}}$ to generate the registration file $\mathrm{RF}_{\mathsf{ID}} \in \mathcal{RF}$ from the corresponding space $\mathcal{RF}$.
- $(\mathsf{Vt}, \mathrm{RR}_{\mathsf{ID}}) \leftarrow \mathsf{Join}(\mathsf{sk}_{\mathsf{GM}}, \mathsf{st}_{\mathsf{GM}}, \mathsf{ID}, \mathrm{RF}_{\mathsf{ID}})$: The join protocol is an interactive procedure between the group manager and a member $\mathsf{ID}$ seeking to join the group with its registration file $\mathrm{RF}_{\mathsf{ID}}$. The group manager $\mathsf{GM}$ utilizes the group secret key $\mathsf{sk}_{\mathsf{GM}}$ and its local state $\mathsf{st}_{\mathsf{GM}}$. Upon completion of the protocol, $\mathsf{GM}$ updates the group secret key $\mathsf{sk}_{\mathsf{GM}}$ and its management state $\mathsf{st}_{\mathsf{GM}}$, respectively.

$\mathsf{ID}$ will update its state $\mathsf{st}_{\mathsf{ID}}$ using the registration result $\mathrm{RR}_{\mathsf{ID}}$ obtained from $\mathsf{GM}$. Furthermore, the transcript of the protocol's execution is recorded in the variable $\mathsf{Vt}$ which may be $\perp$ indicating any join failure. A group member $\mathsf{ID}$ can register different registration files many times.

• $(sk'_{\mathsf{ID}}, \mathsf{st}'_{\mathsf{ID}}, \sigma) \leftarrow \mathsf{Sign}(sk_{\mathsf{ID}}, \mathsf{st}_{\mathsf{ID}}, m)$: The signing algorithm takes as input the secret key $sk_{\mathsf{ID}}$ of $\mathsf{ID}$, along with its state $\mathsf{st}_{\mathsf{ID}}$, and a message $m \in \mathcal{M}$. The algorithm generates a group signature $\sigma$ on the message $m$ and also outputs an updated secret key $sk'_{\mathsf{ID}}$ and state $\mathsf{st}'_{\mathsf{ID}}$, for $\mathsf{ID}$.

• $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{gpk}, m, \sigma, \mathrm{RL})$: The signature verification algorithm takes as input the group public key $\mathsf{gpk}$, a message $m$, a group signature $\sigma$ on $m$, and a public revocation list RL. It outputs 1 if the signature is valid, and 0 otherwise.

• $\mathsf{ID} \leftarrow \mathsf{Open}(\mathsf{gpk}, \mathsf{sk}_{\mathsf{GM}}, m, \sigma)$: The opening algorithm takes as input the group public key $\mathsf{gpk}$, the group secret key $\mathsf{sk}_{\mathsf{GM}}$, a message $m$, and a group signature $\sigma$ for $m$ to return an identity $\mathsf{ID}$ or the symbol $\perp$ to indicate failure.

• $\mathrm{RL}' \leftarrow \mathsf{Revoke}(\mathsf{gpk}, \mathsf{sk}_{\mathsf{GM}}, \mathrm{RL}, \mathsf{ID}, \mathrm{RSS})$: The revocation algorithm takes as input the group public key $\mathsf{gpk}$, the group secret key $\mathsf{sk}_{\mathsf{GM}}$, the revocation list RL, the revoking identity $\mathsf{ID}$, and a set of group signatures RSS to be revoked. This algorithm updates the revocation list RL to $\mathrm{RL}'$.

**Correctness.** A correct FSDGS scheme should ensure that all honest members' unrevoked signatures on any messages should pass the verification algorithm, i.e., $\Pr[\mathsf{Exp}^{\mathsf{Correct}}_{\mathcal{A},\Sigma}(\kappa, q_s, \mathsf{SP}) \Rightarrow 1] = 1$, where is $\mathsf{Exp}^{\mathsf{Correct}}_{\mathcal{A},\Sigma}(\kappa, q_s, \mathsf{SP})$ the correctness ($\mathsf{Correct}$) experiment defined in Figure 1 for $\mathrm{exp} = \mathsf{Correct}$.

**Security Definition.** The security properties that we consider in this work, are adapted from [4,5,13,21,37], including two critical properties: forward-secure traceability ($\mathsf{Trace}$) and anonymity ($\mathsf{Anony}$) with forward privacy, in a dynamic group management setting. For the security definition, we formalize these properties through the security experiments defined in Figure 1. These experiments are indexed by the variable $\mathrm{exp} \in \{\mathsf{Trace}, \mathsf{Anony}\}$, respectively. Moreover, we let $\mathsf{GetCT}()$ be a generic timestamp function that outputs the current system time, e.g., by counting the steps executed by the Turing Machine.

Informally speaking, forward-Secure traceability requires that even an adversary, capable of corrupting the tracing manager and potentially some or all group members (via $\mathsf{Corrupt}$ queries), cannot produce a valid signature under two conditions (**Trace-Cons**): i) The signature cannot be opened to a non-corrupted group member, ensuring that honest users remain protected; ii) The signature cannot be traced to a corrupted member if it was created in a time period before the adversary accessed that the secret key of a member or the group manager. These properties strengthen the standard traceability requirement, as established in [4,5], to include forward-security, ensuring that prior signatures remain secure even after either a member or group manager is compromised. Anonymity with forward privacy ensures that a PPT adversary cannot distinguish (**Anony-Cons**) which of two honest group members signed a challenged message, provided that $\mathsf{GM}$ remains uncorrupted, the signatures stay unopened, and the signers were uncorrupted when they generated their signatures. This

holds even when the adversary selects the members (adding honest and malicious ones via AddHM and AddMM queries, respectively), the signing message of honest members (using HMSign queries), and the time period. Meanwhile, an adversary may exploit the dynamics of the group management (e.g., registering malicious members, revoking members, opening identities of signatures) trying to gain extra advantage for breaking these two properties.

**Definition 1 (FSDGS Security).** *We say that a correct FSDGS scheme $\Sigma$ is secure if the advantages,* $\mathsf{Adv}^{\mathsf{Trace}}_{\mathcal{A},\Sigma}(\kappa, q_s, \mathsf{SP})$ *and* $\mathsf{Adv}^{\mathsf{Anony}}_{\mathcal{A},\Sigma,\frac{1}{2}}(\kappa, q_s, \mathsf{SP})$ *(as defined in Appendix A), of any PPT adversaries are negligible in the corresponding security experiments.*

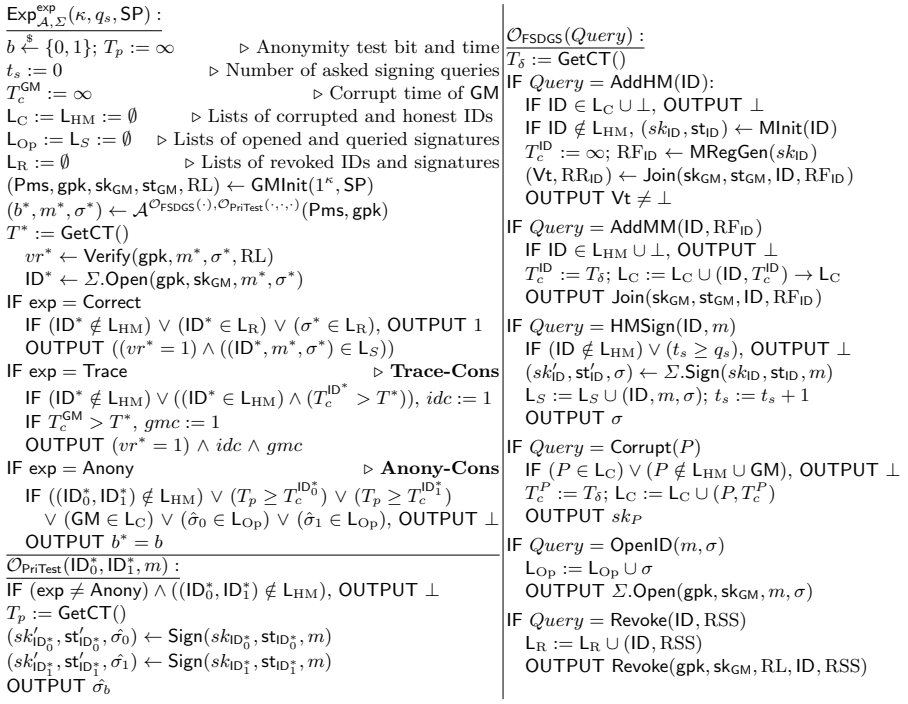



Fig. 1: Security Experiments for Forward Secure Dynamic Group Signature.

## 4 Our FSDGS Constructions

In this section, we introduce two novel FSDGS schemes based on OTS. The first scheme is built upon an OTS hypertree, while the second securely leverages the first as a building block to further extend the signing capability.

### 4.1 An Efficient Solution HHGS with Hypertree

**Construction Overview.** The goals of HHGS include: post-quantum security (G1), efficient dynamic group management (G2), forward security (G3), and provable security in standard model (G4).

*Technical Challenges.* One approach for achieving the above goals is to follow the research line of OTS-based group signatures. A member requests to join a group with its OTS public key and the group manager GM adds the member to the group by signing that member's key using GM's OTS secret key. The member can anonymously sign a message using the OTS secret key corresponding to the public key certified with signatures from GM. A dynamic group with a large size demands a large number of OTS key pairs of GM. Updating the group public key whenever the group changes might pose an inconvenience for existing members. A natural way for achieving G2 is to use a Merkle tree to manage GM's OTS key pairs, where each leaf corresponds to a key pair of GM. Note that the authentication paths of the signatures generated by GM would share nodes in the tree. An adversary can launch a **sibling attack** by exploiting the shared information to infer the identity of the member who generates the group signature and thus break the anonymity. This naturally leads to the question of achieving resilience against sibling attacks in a dynamic group with a large group size. To answer this question, we need to ensure that the OTS key pairs of GM are randomly assigned to members to be joined. A straightforward randomization approach using pseudorandom permutations is impractical since the member can dynamically join the group with unpredictable keys and handling permutations over a large set becomes computationally infeasible. Hence, realizing random selection of a large number of GM's OTS key pairs remains a challenge.

To meet the requirements of G2 and G3, we cannot simply adopt the stateless hypertree used in previous signature schemes, such as those in the SPHINCS+ family [7,31], where secret keys remain static throughout the scheme's lifetime and can derive all keys within the hypertree. However, incorporating dynamic group management (G2) or key-evolving techniques for achieving PFS (G3) inevitably introduces dynamic states for managing the group structure or evolving keys, thereby violating their stateless nature. For instance, the FORS+C layer of SPHINCS+C [31] involves more than $2^{77}$ leaves, which may require extensive state management. Thus, we need to seek an optimized stateful hypertree structure suitable for G2 and G3 from scratch. Meanwhile, we should avoid the repeated selection of the same key pair in the random selection process. In addition, all construction details should satisfy G4, which makes it harder to build, especially when considering adversaries who can adaptively register both honest and malicious members and compromise honest members.

*Construction Ideas.* Our construction starts from organizing OTS key pairs to align with all the aforementioned design goals, following the approach of stateful signature schemes like XMSS-T [32], which offer a more efficient signature structure compared to stateless alternatives such as SPHINCS+C [31]. Our main idea is to adopt a hypertree structure which consists of a GM's OTS hypertree and multiple mixed OTS mount sub-trees (MMT) with obfuscated OTS layers. The

leaves of the trees in the hypertree are used to sign the root of the trees or the sub-trees below. This structure allows GM to generate one tree (sub-tree) on each layer on demand, significantly reducing signature sizes and computational overhead compared to a single large tree. To further optimize GM's computational cost and members' storage requirements, the leaves computed during constructing MMT not only derive from OTS key pairs of GM but also from members' OTS key pairs. This design results in a mixed sub-layer in MMT. However, a security issue arises if the members' OTS secret keys corresponding to their OTS public keys in the mixed sub-layer are used to sign messages directly. In this case, the group signatures generated from these two types of leaves differ in form and can reflect a certain chronological order. An adversary can infer the identity of the member who generates the challenge group signature according the joining order of members. Thus, all keys in the above mixed sub-layer are used to sign the OTS public keys (in the bottom member sub-layer) from members, which will be used to sign messages. To realize the indistinguishability of the group signatures, we bind the OTS public keys of GM and members with fake Identity-OPK ciphertexts (IOCs) and members' IOCs, respectively. The hashes generated from these OTS public keys and ciphertexts serve as the real leaves of MMT.

To realize G3, we use PPRFs to generate OTS key pairs for GM. Within each layer of the hypertree structure, all OTS key pairs of GM are derived using a PPRF specific to that layer. GM maintains the punctured key and state for each PPRF to track which indices and corresponding key pairs have already been used. A used index will be punctured to prevent the reuse of assigned key pairs, thereby ensuring forward security. After each puncturing operation, the BDS state of each tree is updated accordingly, enabling on-demand Merkle proof generation for the remaining available OTS key pairs. Simultaneously, each member uses a pseudorandom generator to generate its OTS key pairs, ensuring the forward security of the signatures.

A critical component of our design is to assign OTS key pairs in the mixed sub-layer to members dynamically. To achieve resilience against sibling attacks, the assignment must be random and independent. Random functions can be used to realize the random selection of OTS key pairs, i.e., determine the index of a leaf in the mixed sub-layer randomly. However, the outputs of existing random functions (like keyed hash functions) are not uniformly random, potentially leading to collisions in indices chosen by GM. There are two main challenges for resolving collisions: how to know which index corresponding to a OTS key pair has been used yet (Q1)? and how to randomly choose an unused key pair from the remaining available key pairs (Q2)? For Q1, PPRFs can be used to trace allocated indices, thereby supporting the subsequent random selection approaches. To answer Q2, we explore two approaches: recomputation and random range. Recomputation iteratively modifies the input of the random function (e.g., appending a counter) and repeats the calculation to generate a new index until an unused index is found. This approach becomes computationally expensive as the number of available indices diminishes. Alternatively, random range allows di-

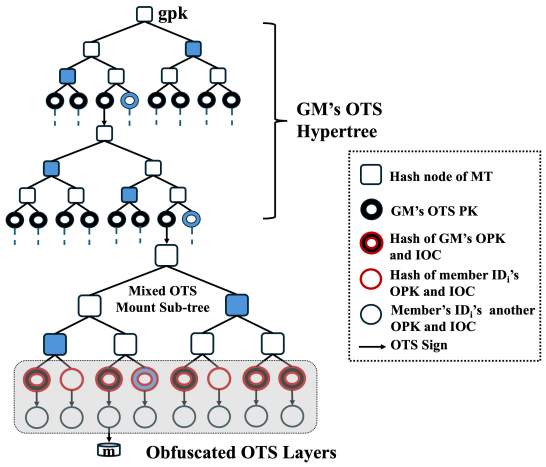



Fig. 2: A visualization of HHGS with a compact structure. The elements highlighted in blue are included in the group credential used for signing $m$.

rect sampling from unused indices tracked by the PPRFs, significantly reducing the overhead associated with collision resolution.

**Construction Preliminaries.** To facilitate the description of our construction, we first review the hypertree structure [7,31] and define functions for obtaining/verifying the authentication credential of a message based on the hypertree.

*Hypertree.* A hypertree (i.e., a tree of trees) of total height $h \in \mathbb{N}$ consists of $d$ layers of trees, each having height $h' = \frac{h}{d}$. On layer $i \in [d]$, the hypertree has $TN_i = 2^{(d-i)h'}$ trees. Let $\mathsf{HI}_i$ denote the index of a OTS key pair at the $i$-th layer. The OTS key pair $(osk^{\mathsf{HI}_1}, opk^{\mathsf{HI}_1})$ belonging to a tree at the bottom layer (i.e., layer 1), which is to be used to sign a message $m$. We denote $\mathsf{HI}_i$ as the first $(d+1-i) \times h'$ bits of $\mathsf{HI}_1$ for $i \in [d]$. On layer $1 < i \leq d$, the OTS key pair $(osk^{\mathsf{HI}_i}, opk^{\mathsf{HI}_i})$ with the index $\mathsf{HI}_i$ is used to sign the root of the tree which has the OTS public key $osk^{\mathsf{HI}_{i-1}}$ as a leaf node. For a message $m$, its authentication credential $\mathsf{Auth}[m] = \{(opk^{\mathsf{HI}_i}, \mathrm{pf}_{\mathrm{lf}}^{\mathsf{HI}_i}, \sigma^{\mathsf{HI}_i})\}_{i\in[d]}$ can be generated (verified) by running the following $\mathsf{GetAuth}$ ($\mathsf{AuthVrfy}$) algorithm. The $i$-th tuple in $\mathsf{Auth}[m]$ contains the OTS public key $opk^{\mathsf{HI}_i}$ for verifying $\sigma^{\mathsf{HI}_i}$, the Merkle proof $\mathrm{pf}_{\mathrm{lf}}^{\mathsf{HI}_i}$ for authenticating $opk^{\mathsf{HI}_i}$, and the signature $\sigma^{\mathsf{HI}_i}$ on the message $m$ ($i = 1$) or on the root of a below tree including $opk^{\mathsf{HI}_{i-1}}$ ($i > 1$).

–$\mathsf{GetAuth}(sk, m, \mathsf{HI}_1)$. Given a secret key $sk$, a message $m$, and an index $\mathsf{HI}_1$, this algorithm outputs the authentication credential $\mathsf{Auth}[m]$ for $m$. For layers 1 to $d$, this algorithm repeatedly uses $sk$ to generate the tree at the $i$-th layer which include the OTS key pair $(osk^{\mathsf{HI}_i}, opk^{\mathsf{HI}_i})$ and computes the Merkle proof $\mathrm{pf}_{\mathrm{lf}}^{\mathsf{HI}_i}$ for the leaf lf associated with $opk^{\mathsf{HI}_i}$. Meanwhile, the root of the tree generated at the end of the previous iteration is signed by the OTS secret key computed in the next iteration. The signature $\sigma^{\mathsf{HI}_1}$ on $m$ is signed by $osk^{\mathsf{HI}_1}$. Finally, this algorithm outputs $\mathsf{Auth}[m] := \{(opk^{\mathsf{HI}_i}, \mathrm{pf}_{\mathrm{lf}}^{\mathsf{HI}_i}, \sigma^{\mathsf{HI}_i})\}_{i\in[d]}$.

–$\mathsf{AuthVrfy}(pk, m, \mathsf{Auth}[m])$. Given a public key $pk$, a message $m$, and an authentication credential $\mathsf{Auth}[m]$, this algorithm checks the validity of $\mathsf{Auth}[m]$. For layers 1 to $d$, this algorithm gets repeated to check whether the leaf lf derived from $opk^{\mathsf{HI}_i}$ belongs to the tree on that layer and whether $\sigma^{\mathsf{HI}_i}$ is a valid signature on the message $m$ ($i = 1$) or the root $\mathrm{Rt}_i$ ($i > 1$). During the verification, the root of each tree is computed with the corresponding leaf and Merkle proof in $\mathsf{Auth}[m]$ by following the root-computation algorithm in [6]. In the final repetition, the generated $\mathrm{Rt}_d$ for the root of the top-most tree on layer $d$ is compared to $pk$. Finally, this algorithm outputs 1 if $\mathrm{Rt}_d = pk$ and 0 otherwise.

**Detailed Algorithms of HHGS.** We define the algorithms of HHGS as follows:

• $\mathsf{GMInit}(1^\kappa, \mathsf{SP})$: Given the security parameter $1^\kappa$ and a setup parameter $\mathsf{SP} = (h, d)$, the group manager GM initializes a set of PPRF schemes as $\{pms_{\mathsf{F}_p^i} := \mathsf{F}_p^i.\mathsf{Setup}(1^\kappa)\}_{0\le i\le d}$, a OTS scheme S as $pms_{\mathsf{S}} := \mathsf{S.Setup}(1^\kappa)$, a MT scheme M as $pms_{\mathsf{M}} := \mathsf{M.Setup}(1^\kappa)$, a AE scheme E as $pms_{\mathsf{E}} := \mathsf{E.Setup}(1^\kappa)$, a PRG scheme G as $pms_{\mathsf{G}} := \mathsf{G.Setup}(1^\kappa)$, a CRH scheme H as $(hk, pms_{\mathsf{H}}) := \mathsf{H.Setup}(1^\kappa)$, the PRF schemes $\mathsf{F}_1$ and $\mathsf{F}_2$ as $pms_{\mathsf{F}_i} := \mathsf{F}_i.\mathsf{Setup}(1^\kappa)$ for $i \in [2]$. We assume that the spaces of each building block are compatible with those of other building blocks when their algorithms interact or invoke each other. The system parameter is set as $\mathsf{Pms} := (\{pms_{\mathsf{F}_p^i}\}_{0\le i\le d}, pms_{\mathsf{S}}, pms_{\mathsf{M}}, pms_{\mathsf{E}}, pms_{\mathsf{G}}, pms_{\mathsf{H}}, pms_{\mathsf{F}_1}, pms_{\mathsf{F}_2}, hk)$.

For $0 \le i \le d$, GM computes a random secret key $k_p^i := \mathsf{F}_p^i.\mathsf{KGen}(pms_{\mathsf{F}_p^i})$ to generate OTS key pairs. The $i$-th layer for $1 \le i \le d$ is a hypertree structure, and the 0-th layer is used to mount the members' OTS keys (which will be defined later). GM also generates a random secret key $k_{\mathsf{GM}} := \mathsf{F}_1.\mathsf{KGen}(pms_{\mathsf{F}_1})$ for $\mathsf{F}_1$. The group secret key $\mathsf{sk}_{\mathsf{GM}}$ consists of the random secret keys of PPRFs and $\mathsf{F}_1$, i.e., $\mathsf{sk}_{\mathsf{GM}} := (\{k_p^i\}_{0\le i\le d}, k_{\mathsf{GM}})$. Then, GM computes the group public key gpk through the following steps:

– Compute the randomnesses $\{rs_{\mathsf{GM}}^{d,1,v}\}_{v\in[2^{h'}]}$ to be used in the generation of leaves $\{\mathrm{lf}_{d,1,v}\}_{v\in[2^{h'}]}$, where each randomness value is defined as $rs_{\mathsf{GM}}^{d,1,v} := \mathsf{F}_p^d(k_p^d, v)$, with $v$ representing a $h'$-bit index of the leaf $\mathrm{lf}_{d,1,v}$ in the top-most tree of the hypertree structure.
– Compute the OTS key pairs $\{(osk_{\mathsf{GM}}^{d,1,v}, opk_{\mathsf{GM}}^{d,1,v}) := \mathsf{S.KGen}(rs_{\mathsf{GM}}^{d,1,v})\}_{v\in[2^{h'}]}$ and the leaves $\{\mathrm{lf}_{d,1,v} := opk_{\mathsf{GM}}^{d,1,v}\}_{v\in[2^{h'}]}$.
– Generate the group public key $\mathsf{gpk} := \mathrm{Tr}_{d,1}.\mathrm{Rt}$, where top-most tree $\mathrm{Tr}_{d,1}$ is built via the tree-building algorithm $(\mathrm{Tr}_{d,1}, \mathsf{st}_{\mathrm{BDS}}^{d,1}) := \mathsf{M.Build}(\{\mathrm{lf}_{d,1,v}\}_{v\in[2^{h'}]})$.

Meanwhile, GM initializes the group management state $\mathsf{st}_{\mathsf{GM}}$ to store a registered identity list $\mathrm{RegIDL} := \emptyset$ recording the identities of registered group members and a revocation list $\mathrm{RL} := \emptyset$.

• $\mathsf{MInit}(\mathsf{ID})$: The member ID computes a secret seed $sd_{\mathsf{ID}} := \mathsf{G.SGen}(pms_{\mathsf{G}})$ and defines its secret key as $sk_{\mathsf{GM}} := sd_{\mathsf{ID}}$. Meanwhile, ID initializes $\mathsf{st}_{\mathsf{ID}} := \emptyset$.

• $\mathsf{MRegGen}(sk_{\mathsf{ID}})$: ID runs this algorithm to generate $\ell_r \in \mathbb{N}$ OTS public keys to be registered. The member first gets the current secret seed $sd_{\mathsf{ID}}$ from its secret key $sk_{\mathsf{GM}}$. For $v \in [\ell_r]$, ID sets the PRG seed as $sd_{\mathsf{ID}}^0 := sd_{\mathsf{ID}}$ and generates the $v$-th OTS key pair $(osk_{\mathsf{ID}}^v, opk_{\mathsf{ID}}^v) := \mathsf{S.KGen}(rs_{\mathsf{ID}}^{v,1})$, where $(rs_{\mathsf{ID}}^{v,1}, rs_{\mathsf{ID}}^{v,2}) := \mathsf{G}(rs_{\mathsf{ID}}^v)$

and $(sd^v_{\mathsf{ID}}, rs^v_{\mathsf{ID}}) := \mathsf{G}(sd^{v-1}_{\mathsf{ID}})$. Finally, the registration file is set as $\mathrm{RF}_{\mathsf{ID}} := \{opk^v_{\mathsf{ID}}\}_{v\in[\ell_r]}$, and the secret key is updated to $sk_{\mathsf{ID}} := sd^{\ell_r}_{\mathsf{ID}}$.

• $\mathsf{Join}(\mathsf{sk}_{\mathsf{GM}}, \mathsf{st}_{\mathsf{GM}}, \mathsf{ID}, \mathrm{RF}_{\mathsf{ID}})$: For each $opk_{\mathsf{ID}} \in \mathrm{RF}_{\mathsf{ID}}$, GM randomly selects an unused *mount index* $\mathrm{MI} := \mathsf{RandSelect}(\mathsf{sk}_{\mathsf{GM}}, \mathsf{ID}, opk_{\mathsf{ID}})$ that is to mount $opk_{\mathsf{ID}}$ (at the layer 0 below the hybertree), where the concrete steps of the random selection function RandSelect is defined later. We further decompose MI into two sub-indices as $\mathrm{MI} = \mathsf{sTI}||\mathrm{sMI}$, where $\mathsf{sTI} := \mathrm{MI}(h)$ comprises the first $h$ bits of MI, indexing a MMT. The remaining $|\mathrm{MI}| - h$ bits specify the leaf index, sMI, within the corresponding MMT. Then, GM adds the tuple $(\mathsf{ID}, opk^{\mathrm{MI}}_{\mathsf{ID}}, \mathrm{MI})$ to a registration cache file $\mathsf{RFCa}^{\mathsf{sTI}}_{\mathsf{MMT}}$ for the sub-tree $\mathsf{MMT}[\mathsf{sTI}]$ with the index sTI. Note that sTI is also the index of the leaf in the hypertree that is used to sign the root of $\mathsf{MMT}[\mathsf{sTI}]$.

GM checks whether $\mathsf{MMT}[\mathsf{sTI}]$ has been created by computing $\mathsf{F}^1_p(k^1_p, \mathsf{sTI})$. It then generates authentication credentials for the member's OTS keys in $\mathsf{RFCa}^{\mathsf{sTI}}_{\mathsf{MMT}}$ based on one of the following two cases for constructing the obfuscated OTS layer in MMT.

– **Case** 1 ($\mathsf{F}^1_p(k^1_p, \mathsf{sTI}) \neq \perp$) for MMT initialization: GM needs to build $\mathsf{MMT}[\mathsf{sTI}]$ using member's keys in $\mathsf{RFCa}^{\mathsf{sTI}}_{\mathsf{MMT}}$ and the supplementary OTS keys generated by itself (for future usage). Let $\mathsf{L}_{\mathrm{mI}}$ be a list storing the indices of members in $\mathsf{RFCa}^{\mathsf{sTI}}_{\mathsf{MMT}}$ and $\overline{\mathsf{L}}_{\mathrm{mI}}$ be the complement of $\mathsf{L}_{\mathrm{mI}}$ such that $\mathsf{L}_{\mathrm{mI}} \cup \overline{\mathsf{L}}_{\mathrm{mI}}$ covers all indices of leaves in $\mathsf{MMT}[\mathsf{sTI}]$. GM prepares the leaves of $\mathsf{MMT}[\mathsf{sTI}]$ based on $\mathsf{L}_{\mathrm{mI}}$ and $\overline{\mathsf{L}}_{\mathrm{mI}}$ respectively as follows.
  - For each index $\mathrm{sMI} \in \mathsf{L}_{\mathrm{mI}}$, GM generates an Identity-OPK ciphertext (IOC) $c^1_{\mathrm{MI}} := \mathsf{E.Enc}(ke^1_{\mathrm{MI}}, \mathsf{ID}; re^1_{\mathrm{MI}})$, where $ke^1_{\mathrm{MI}} := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“K-Mix”}||\mathrm{MI})$ and $re^1_{\mathrm{MI}} := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“R-Mix”}||\mathrm{MI})$. It computes a leaf $\mathrm{lf}_{\mathrm{sMI}} := \mathsf{H}(opk^{\mathrm{MI}}_{\mathsf{ID}}||c^1_{\mathrm{MI}})$ in $\mathsf{MMT}[\mathsf{sTI}]$. It sets $c^2_{\mathrm{MI}} := \emptyset$, $opk^{\mathrm{MI}}_{\mathsf{GM}} := \emptyset$ and $\sigma^{\mathrm{sMI}}_{\mathsf{GM}} := \emptyset$. These values should be completed by the member during signing a message. Additionally, GM also punctures MI as $k^{0\prime}_p := \mathsf{F}^0_p.\mathsf{Punc}(k^0_p, \mathrm{MI})$ at level 0 (in a MMT).
  - For each index $\mathrm{sMI}' \in \overline{\mathsf{L}}_{\mathrm{mI}}$, GM sets $\mathrm{MI}' := \mathsf{sTI}||\mathrm{sMI}'$ and computes the key pair $(osk^{\mathrm{MI}'}_{\mathsf{GM}}, opk^{\mathrm{MI}'}_{\mathsf{GM}}) := \mathsf{S.KGen}(rs^{\mathrm{MI}'}_{\mathsf{GM}})$, where $rs^{\mathrm{MI}'}_{\mathsf{GM}} := \mathsf{F}^0_p(k^0_p, \mathrm{MI}')$. It computes the encryption key $ke^1_{\mathrm{MI}'} := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“K-Mix”}||\mathrm{MI}')$ and randomness $re^1_{\mathrm{MI}'} := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“R-Mix”}||\mathrm{MI}')$ to generate a fake IOC (encrypting GM) for $opk^{\mathrm{MI}'}_{\mathsf{GM}}$, as $c^1_{\mathrm{MI}} := \mathsf{E.Enc}(ke^1_{\mathrm{MI}'}, \mathsf{GM}; re^1_{\mathrm{MI}'})$.

  Then, GM runs $(\mathsf{MMT}[\mathsf{sTI}], \mathsf{st}^{\mathsf{MMT}[\mathsf{sTI}]}_{\mathrm{BDS}}) := \mathsf{M.Build}(\{\mathrm{lf}_v\}_{v\in[2^{|\mathrm{sMI}|}]})$ to build the sub-tree $\mathsf{MMT}[\mathsf{sTI}]$, and obtains $\mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}] := \mathsf{GetAuth}(\{k^i_p\}_{i\in[d]}, \mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}, \mathsf{sTI})$. Next, it derives the Merkle proof of each $\mathrm{lf}_{\mathrm{sMI}}$ associated with ID's OTS key as $\mathrm{pf}_{\mathrm{lf}_{\mathrm{sMI}}} := \mathsf{M.GetPrf}(\mathsf{st}^{\mathsf{MMT}[\mathsf{sTI}]}_{\mathrm{BDS}}, \mathrm{lf}_{\mathrm{sMI}})$. To achieve forward security, GM punctures the used leaves on the hypertree as $k^{i\prime}_p := \mathsf{F}^i_p.\mathsf{Punc}(k^i_p, \mathrm{MI}(w))$ for the PPRF $\mathsf{F}^i_p$ associated with the tree on the $i$-th layer, where $1 \leq i \leq d$ and $\mathrm{MI}(w)$ denotes the first $w = \frac{(d+1-i)h}{d}$ bits of MI. In order to compute the Merkle proofs of the GM's OTS keys indexed by any $\mathrm{sMI}' \in \overline{\mathsf{L}}_{\mathrm{mI}}$ in the future, GM stores the corresponding state $\mathsf{st}^{\mathrm{lf}_{\mathrm{sMI}'}}_{\mathrm{BDS}}$ and $\mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}]$ into $\mathsf{st}_{\mathsf{GM}}$.

– **Case** 2 ($\mathsf{F}_p^1(k_p^1, \mathsf{sTI}) = \perp$) for $\mathsf{MMT}$ usage: In this scenario, each $opk_{\mathsf{ID}}^{\mathrm{MI}}$ in $\mathsf{RFCa}_{\mathsf{MMT}}^{\mathsf{sTI}}$ must reside within the member OTS sub-layer and should therefore be authenticated using $\mathsf{GM}$'s OTS key in the mixed sub-layer. $\mathsf{GM}$ generates the keys $(osk_{\mathsf{GM}}^{\mathrm{MI}}, opk_{\mathsf{GM}}^{\mathrm{MI}}) := \mathsf{S.KGen}(rs_{\mathsf{GM}}^{\mathrm{MI}})$, where $rs_{\mathsf{GM}}^{\mathrm{MI}} := \mathsf{F}_p^0(k_p^0, \mathrm{MI})$ and the IOC $c_{\mathrm{MI}}^1$ for $opk_{\mathsf{GM}}^{\mathrm{MI}}$ as in Case 1. Then, $\mathsf{GM}$ computes the leaf $\mathrm{lf}_{\mathrm{sMI}} := \mathsf{H}(opk_{\mathsf{GM}}^{\mathrm{MI}} || c_{\mathrm{MI}}^1)$ and gets its Merkle proof $\mathrm{pf}_{\mathrm{lf}_{\mathrm{sMI}}} := \mathsf{M.GetPrf}(\mathsf{st}_{\mathrm{BDS}}^{\mathrm{lf}_{\mathrm{sMI}}}, \mathrm{lf}_{\mathrm{sMI}})$, where $\mathsf{st}_{\mathrm{BDS}}^{\mathrm{lf}_{\mathrm{sMI}}} \in \mathsf{st}_{\mathsf{GM}}$. It also retrieves $\mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}]$ from $\mathsf{st}_{\mathsf{GM}}$, and generates an IOC $c_{\mathrm{MI}}^2 := \mathsf{E.Enc}(ke_{\mathrm{MI}}^2, \mathsf{ID}; re_{\mathrm{MI}}^2)$ with $ke_{\mathrm{MI}}^2 := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“K-Mem”}||\mathrm{MI})$ and $re_{\mathrm{MI}}^2 := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“R-Mem”}||\mathrm{MI})$. Lastly, $\mathsf{GM}$ creates the signature $\sigma_{\mathsf{GM}}^{\mathrm{sMI}} := \mathsf{S.Sign}(osk_{\mathsf{GM}}^{\mathrm{sMI}}, opk_{\mathsf{ID}}^{\mathrm{MI}} || c_{\mathrm{MI}}^2)$ over $opk_{\mathsf{ID}}^{\mathrm{MI}}$. $\mathsf{GM}$ removes $\mathsf{st}_{\mathrm{BDS}}^{\mathrm{lf}_{\mathrm{sMI}}}$ and punctures MI as $k_p^{0\prime} := \mathsf{F}_p^0.\mathsf{Punc}(k_p^0, \mathrm{MI})$.

The authentication credential for each $opk_{\mathsf{ID}}^{\mathrm{MI}} \in \mathsf{RFCa}_{\mathsf{MMT}}^{\mathsf{sTI}}$ is set as $\mathsf{Auth}[opk_{\mathsf{ID}}^{\mathrm{MI}}] := (opk_{\mathsf{GM}}^{\mathrm{MI}}, c_{\mathrm{MI}}^1, c_{\mathrm{MI}}^2, \mathrm{pf}_{\mathrm{lf}_{\mathrm{sMI}}}, \sigma_{\mathsf{GM}}^{\mathrm{sMI}}, \mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}, \mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}])$. $\mathsf{GM}$ iterates through all OTS keys in $\mathrm{RF}_{\mathsf{ID}}$, processing each as described. It then appends every $\mathsf{Auth}[opk_{\mathsf{ID}}^{\mathrm{MI}}]$ to the registration result $\mathrm{RR}_{\mathsf{ID}}$ and returns it to $\mathsf{ID}$, which updates its state $\mathsf{st}_{\mathsf{ID}}$ accordingly.[8] During the execution of this protocol, the sent registration file and the received registration result of $\mathsf{ID}$ are recorded in the transcript $\mathsf{Vt}$. Meanwhile, $\mathsf{GM}$ adds $\mathsf{ID}$ to the registered identity list RegIDL if $\mathsf{ID}$ has not previously submitted a join request.

We assume that the join sessions for processing different registration requests from members share the state $\mathsf{st}_{\mathsf{GM}}$. These sessions can be collectively handled by $\mathsf{GM}$ through a unified join-hyper routine, reducing the GM's computational and storage costs.

• $\mathsf{Sign}(sk_{\mathsf{ID}}, \mathsf{st}_{\mathsf{ID}}, m)$: A member $\mathsf{ID}$ computes the next secret seed and the current key generation randomness as $(sd_{\mathsf{ID}}^v, rs_{\mathsf{ID}}^v) := \mathsf{G}(sd_{\mathsf{ID}}^{v-1})$ and $(rs_{\mathsf{ID}}^{v,1}, rs_{\mathsf{ID}}^{v,2}) := \mathsf{G}(rs_{\mathsf{ID}}^v)$, respectively. Using the randomness $rs_{\mathsf{ID}}^{v,1}$, $\mathsf{ID}$ generates a OTS key pair $(osk_{\mathsf{ID}}, opk_{\mathsf{ID}}) := \mathsf{S.KGen}(rs_{\mathsf{ID}}^{v,1})$. Next, $\mathsf{ID}$ must determine the sub-layer (either the mixed sub-layer or the member sub-layer) where the OTS public key $opk_{\mathsf{ID}}$ resides within the obfuscated layer. To accomplish this, $\mathsf{ID}$ retrieves the first (i.e., most recent) authentication credential $\mathsf{Auth}[opk_{\mathsf{ID}}]$ from $\mathsf{st}_{\mathsf{ID}}$. If $\sigma_{\mathsf{GM}}^{\mathrm{sMI}} \in \mathsf{Auth}[opk_{\mathsf{ID}}]$ is non-empty, this indicates that $opk_{\mathsf{ID}}$ belongs to the member sub-layer, and $osk_{\mathsf{ID}}$ can be directly used to sign the message $m$, producing the signature $\sigma_{\mathsf{ID}} := \mathsf{S.Sign}(osk_{\mathsf{ID}}, m)$. Eventually, $\mathsf{ID}$ constructs the group signature as $\sigma := (opk_{\mathsf{ID}}, \mathsf{Auth}[opk_{\mathsf{ID}}], \sigma_{\mathsf{ID}})$.

For $\sigma_{\mathsf{GM}}^{\mathrm{sMI}} = \emptyset$, $opk_{\mathsf{ID}}$ must reside in the mixed sub-layer. In this case, $\mathsf{ID}$ generates an additional key pair $(osk_{\mathsf{ID}}', opk_{\mathsf{ID}}') := \mathsf{S.KGen}(rs_{\mathsf{ID}}^{v,2})$ for signing the message, i.e., $\sigma_{\mathsf{ID}} := \mathsf{S.Sign}(osk_{\mathsf{ID}}', m)$. Additionally, $\mathsf{ID}$ randomly generates an IOC $c_{\mathsf{ID}}'$ for $opk_{\mathsf{ID}}'$ as in the $\mathsf{Join}$ protocol but uses the random key $ke' \xleftarrow{\$} \mathcal{K}_{\mathsf{AE}}$ and the randomness $re' \xleftarrow{\$} \mathcal{ER}_{\mathsf{AE}}$. Next, $\mathsf{ID}$ produces the signature $\sigma_{\mathsf{ID}}' := \mathsf{S.Sign}(osk_{\mathsf{ID}}, opk_{\mathsf{ID}}' || c_{\mathsf{ID}}')$ and completes the values regarding $(opk_{\mathsf{GM}}^{\mathrm{MI}}, \sigma_{\mathsf{GM}}^{\mathrm{sMI}}, c_{\mathrm{MI}}^2)$

[8] $\mathsf{GM}$ may optionally returns $ke_{\mathrm{MI}}^1$ and $ke_{\mathrm{MI}}^2$ to enable the member to check the corresponding ciphertext.

in the authentication credential $\mathsf{Auth}[opk_{\mathsf{ID}}]$ by incorporating $opk_{\mathsf{ID}}$, $\sigma'_{\mathsf{ID}}$, and $c'_{\mathsf{ID}}$. The group signature is then defined as $\sigma := (opk'_{\mathsf{ID}}, \mathsf{Auth}[opk_{\mathsf{ID}}], \sigma_{\mathsf{ID}})$.

Finally, ID updates it secret key with the new secret seed, i.e., $sk'_{\mathsf{ID}} := sd^v_{\mathsf{ID}}$, and gets an updated state $\mathsf{st}'_{\mathsf{ID}} := \mathsf{st}_{\mathsf{ID}} \setminus (opk_{\mathsf{ID}}, \mathsf{Auth}[opk_{\mathsf{ID}}])$.

• $\mathsf{Verify}(\mathsf{gpk}, m, \sigma, \mathrm{RL})$: The verifier parses $\sigma = (opk'_{\mathsf{ID}}, \mathsf{Auth}[opk_{\mathsf{ID}}], \sigma_{\mathsf{ID}})$ and $\mathsf{Auth}[opk_{\mathsf{ID}}] = (opk^{\mathrm{MI}}_{\mathsf{GM}}, c^1_{\mathrm{MI}}, c^2_{\mathrm{MI}}, \mathrm{pf}_{\mathrm{lf}_{\mathrm{sMI}}}, \sigma^{\mathrm{sMI}}_{\mathsf{GM}}, \mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}, \mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}])$. This algorithm outputs 0 if one of the following conditions is met: i) $c^1_{\mathrm{MI}} \in \mathrm{RL}$; ii) $\mathsf{AuthVrfy}(\mathsf{gpk}, \mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}, \mathsf{Auth}[\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}]) = 0$; iii) $\mathsf{M.Verify}(\mathsf{MMT}[\mathsf{sTI}].\mathrm{Rt}, \mathrm{lf}_{\mathrm{sMI}}, \mathrm{pf}_{\mathrm{lf}_{\mathrm{sMI}}}) = 0$, where $\mathrm{lf}_{\mathrm{sMI}} = \mathsf{H}(opk^{\mathrm{MI}}_{\mathsf{GM}} || c^1_{\mathrm{MI}})$; iv) $\mathsf{S.Verify}(opk^{\mathrm{MI}}_{\mathsf{GM}}, opk'_{\mathsf{ID}} || c^2_{\mathrm{MI}}, \sigma^{\mathrm{sMI}}_{\mathsf{GM}}) = 0$; v) $\mathsf{S.Verify}(opk'_{\mathsf{ID}}, m, \sigma_{\mathsf{ID}}) = 0$. Otherwise, the verifier outputs 1.

• $\mathsf{Open}(\mathsf{gpk}, \mathsf{sk}_{\mathsf{GM}}, m, \sigma)$: The algorithm outputs 0 if the group signature is not valid, i.e., $\mathsf{Verify}(\mathsf{gpk}, m, \sigma, T, \emptyset) = 0$. GM gets the index MI from $\sigma$ according to its the authentication credential. Next, it computes keys randomness $(ke^1_{\mathrm{MI}}, re^1_{\mathrm{MI}})$ and $(ke^2_{\mathrm{MI}}, re^2_{\mathrm{MI}})$ as in Join protocol based on $k_{\mathsf{GM}}$. GM decrypts $\hat{\mathsf{ID}}_1 := \mathsf{E.Dec}(ke^1_{\mathrm{MI}}, c^1_{\mathrm{MI}}; re^1_{\mathrm{MI}})$ and $\hat{\mathsf{ID}}_2 := \mathsf{E.Dec}(ke^2_{\mathrm{MI}}, c^2_{\mathrm{MI}}; re^2_{\mathrm{MI}})$. If $\hat{\mathsf{ID}}_1 = \mathsf{GM}$ then return $\hat{\mathsf{ID}}_2$; otherwise $\hat{\mathsf{ID}}_1$ is returned.

• $\mathsf{Revoke}(\mathsf{gpk}, \mathsf{sk}_{\mathsf{GM}}, \mathrm{RL}, \mathsf{ID}, \mathrm{RSS})$: For each tuple $(\sigma, m) \in \mathrm{RSS}$, GM returns 0 if $\mathsf{Verify}(\mathsf{gpk}, m, \sigma, \mathrm{RL}) = 0$ or $\mathsf{Open}(\mathsf{gpk}, \mathsf{sk}_{\mathsf{GM}}, m, \sigma) \neq \mathsf{ID}$. Otherwise, GM gets $c^1_{\mathrm{MI}}$ from each $\sigma \in \mathrm{RSS}$ and puts it into RL.

**Instantiations of** $\mathsf{RandSelect}(\mathsf{sk}_{\mathsf{GM}}, \mathsf{ID}, opk_{\mathsf{ID}})$**.** One method for random index selection applies a pseudorandom permutation to shuffle all leaf indices in a HHGS instance [2]. However, this becomes impractical for trees with over $2^{20}$ leaves due to the scaling costs of permutation [2, §4.2].

Alternatively, we propose using a short-range PRF scheme, $\mathsf{F}_2$, with range $\{0,1\}^{\ell_I}$ to generate indices for OTS mounting. The first approach mimics random sampling without replacement, while the second directly selects from remaining available indices. In the first approach, collisions may occur due to the birthday paradox. To address this, the $\mathsf{F}_2$ process can be repeated with different inputs to avoid collisions (referred to as **AP1**). However, as the number of available indices, denoted by $\phi$, decreases, the likelihood of collisions increases. When this happens, we can leverage the PPRF keys, which encode the available keys, to directly select an index from the remaining indices (referred to as **AP2**), provided the number of remaining indices is not excessively large (e.g., $\leq 2^{20}$). We define $\xi$ as the threshold for the number of available indices, marking the point at which we switch from the initial random selection method to the latter approach. To support this second selection method, we introduce an additional algorithm for PPRF, denoted as $\mathsf{GetMsg}(k, i)$. This algorithm enables the retrieval of the $i$-th unpunctured message. Specifically, $\mathsf{GetMsg}(k, i)$ takes as input the current key $k \in \mathcal{K}_{\mathsf{PRF}}$ and returns the $i$-th accessible message $x_i$ from the remaining computable message set. The realization of this algorithm is detailed in Appendix B.

Specifically, GM generates a random selection key, $k_s := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“}RandSel\text{”})$, and computes the index KI as follows:

- **AP1**. If $\phi < \xi$, compute $\mathsf{KI} := \mathsf{F}_2(k_s, \mathsf{ID}||opk_{\mathsf{ID}}||ctr)$ using an incremental counter $ctr$ (initialized to 0) and repeat this process until $\mathsf{F}_p^0.\mathsf{Eval}(k_p^0, \mathsf{KI}) \neq \perp$. Thereafter, $\phi := \phi - 1$.
- **AP2**. Otherwise, compute $tmp := \mathsf{F}_2(k_s, \mathsf{ID}||opk_{\mathsf{ID}})$, then set $tmp := tmp \bmod \phi$, and derive $\mathsf{KI} := \mathsf{F}_p^0.\mathsf{GetMsg}(k_p^0, tmp)$.

The complexity of the above instantiation of RandSelect is determined by the signing $Y = 2^{\tilde{h}}$ of HHGS, where $\tilde{h} = h + \frac{h}{d}$ is the total height of the hypertree structure. The computational cost of **AP1** is determined by the number $n_r$ of PRF recomputations for selecting an unused index. Let $\phi$ denote the remaining indices that can be chosen. The probability of selecting an occupied index is about $1 - \frac{\phi}{Y}$, while considering PRF as a random function. The number of PRF recomputations follows a geometric distribution with success probability $p = \frac{\phi}{Y}$. The expected value of $n_r$ is $\mathbb{E}[n_r] = \frac{Y}{\phi}$, which can be used as the threshold for switching between AP1 and AP2. As $\phi$ decreases, $\mathbb{E}[n_r]$ grows inversely proportional to $\phi$. The total computational cost of **AP1** scales with the reduction in the number of remaining indices, $\phi$. In contrast, the cost of **AP2** is directly proportional to $\phi$, as determined by the cost of GetMsg (analyzed in Appendix B). Consequently, as $\phi$ decreases, the overhead of **AP2** also reduces.

In a nutshell, the signing capacity, $Y$, of HHGS is inherently limited. For instance, when $\mathbb{E}[n_r] = \phi = O(2^{20})$, $Y$ is in $O(2^{40})$.

**Correctness.** According to the correctness experiment $\mathsf{Exp}^{\mathsf{Correct}}_{\mathcal{A},\mathsf{HHGS}}$, the correctness of 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 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 MT. Moreover, the oracle queries in the correctness game are simulated using secret keys (including sub-secret keys of the building blocks, such as AE, PPRF, and 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.

**Security Analysis.** The security of HHGS is demonstrated through the following theorems with detailed proofs in the full version of this paper [20].

**Theorem 1.** *Assume that the* OTS *scheme* S*, the* PPRF *functions* $\{\mathsf{F}_p^i\}_{0 \le i \le d}$*, the* PRF *functions* $\mathsf{F}_1$ *and* $\mathsf{F}_2$*, the* MT *scheme* M*, the* AE *scheme* E*, the* PRG *scheme* G*, and the hash function* H *are secure as defined in Appendix A, then* HHGS *satisfies traceability.*

The proof employs a sequence of game transformations, starting with the Trace experiment, to incrementally restrict the adversary's ability to forge signatures under HHGS. Starting from the original Trace experiment, the challenger

incrementally modifies the game by introducing abort conditions to enable the reduction of security to the hardness of underlying cryptographic primitives.

**Theorem 2.** *Under the same assumptions as Theorem 1,* HHGS *achieves anonymity.*

The core idea involves progressively replacing PRF values, random selection keys, encryption keys, and randomness with random values, leveraging the security of the underlying primitives such as PRF and PRG. Consequently, the OTS keys and ciphertexts associated with the $\mathcal{O}_{\mathsf{PriTest}}(\mathsf{ID}_0^*, \mathsf{ID}_1^*, m)$ query for the challenged members and GM are shown to follow identical distributions, preventing the adversary from distinguishing between them. Finally, the mount indices of the target OTS keys are selected randomly, limiting the adversary's success to a random guess due to the independence of the selection process.

The security of HHGS also ensures its resilience to sibling attacks. The key insight lies in the randomness of the selection process for the leaves at level 0 of the hypertree structure, which prevents the attacker from distinguishing between the positions of the two candidate challenge signatures on the tree. Although the attacker knows the results of the other $2^{\tilde{h}} - 2$ selections, the two remaining unknown selections concerning the challenged identities' OTS keys are drawn randomly and without replacement. Since the selection process is independent and unbiased in the final game, the distribution of these two unknown selections appears identical from the attacker's perspective.

### 4.2 Signing Capability Extension

This section focuses on improving the signing capability of HHGS. A group signature with a large signing capacity is vital for applications with numerous users or frequent signing demands, such as privacy-preserving IoT networks in smart cities or industrial IoT systems. A naive solution to support larger groups and signing capabilities is to increase the parameters of HHGS. However, this is not efficient due to the complexity of the RandSelect algorithm analyzed in the previous section. Alternatively, using multiple independent PPRF keys could address this inefficiency, but it introduces the challenge of privately determining which member's OTS key should be associated with which PPRF key. This requires a method to securely partition members' OTS keys into subgroups.

**Construction Overview.** We address this challenge with $\mathsf{HHGS}^+$, an improved FSDGS scheme that generically builds on HHGS. Specifically, $\mathsf{HHGS}^+$ initializes $W \in \mathbb{N}$ copies of HHGS and employs the MT scheme M to construct a tree, where the leaves are the group public keys $\{\mathsf{gpk}_i\}_{i\in[W]}$ generated by the HHGS instances. To ensure efficiency, we use a single Merkle tree over a hypertree structure. This design minimizes additional tree layers, avoiding the need to store more PPRF keys. Moreover, using multiple independent HHGS instances could enable parallelization of group management.

To securely group members' OTS keys, we randomly associate each OTS key with a specific HHGS instance using a short-range PRF function, which takes as input a OTS key and its owner's identity. Due to the short range of the PRF,

different OTS keys may yield the same PRF value (referred to as OTS collision), enabling the grouping feature. However, as OTS collisions are not uniformly distributed among PRF values, the volume of each HHGS instance poses a challenge. That is, the non-uniform distribution of OTS collisions creates challenges in *balancing the load across* HHGS *instances and ensuring the security.*[9] In an insecure scenario, referred to as the **Full-instance Attack (FIA)**, if a HHGS instance becomes full early in the process (e.g., while registering $\mathsf{ID}_0^*$), it will not be selected in subsequent steps (e.g., during the registration of $\mathsf{ID}_1^*$), potentially creating a bias in the remaining selections. This temporal dependency in the selection process allows an attacker to exploit the evolving probabilities as trees fill, compromising the randomness and enabling prediction of future instance choices. To counter this, each HHGS instance should contain sufficient redundancy (see Lemma 1) to accommodate the non-uniform distribution of PRF values, ensuring they can handle varying numbers of OTS keys securely.

**Detailed Algorithms.** We define the algorithms of $\mathsf{HHGS}^+$ as follows:

• $\mathsf{GMInit}(1^\kappa, \mathsf{SP})$: For $i \in [W]$, GM creates $W \in \mathbb{N}$ HHGS instances as $(\mathsf{Pms}_i, \mathsf{gpk}_i, \mathsf{sk}_{\mathsf{GM}}^i, \mathsf{st}_{\mathsf{GM}}^i, \mathrm{RL}_i) := \mathsf{HHGS.GMInit}(1^\kappa, \mathsf{SP})$. Subsequently, it builds the Merkle tree instance as $(\mathrm{Tr}', \mathsf{st}'_{\mathrm{BDS}}) := \mathsf{M.Build}(\{\mathsf{gpk}_i\}_{i\in[W]})$. GM sets $\mathsf{gpk} := \mathrm{Tr}'.\mathrm{Rt}$, $\mathsf{sk}_{\mathsf{GM}} := \{\mathsf{sk}_{\mathsf{GM}}^i\}_{i\in[W]}$, $\mathsf{Pms} := \{\mathsf{Pms}_i\}_{i\in[W]}$, $\mathsf{st}_{\mathsf{GM}} := \{\mathsf{st}_{\mathsf{GM}}^i\}_{i\in[W]}$, and $\mathrm{RL} := \{\mathrm{RL}_i\}_{i\in[W]}$. We will use the building blocks initiated in HHGS instances such as the PRG scheme G, the PRF schemes $\mathsf{F}_1$ and $\mathsf{F}_2$, and the MT scheme M. Additionally, GM samples a random PRF key for $k_{\mathsf{GM}} := \mathsf{F}_1.\mathsf{KGen}(pms_{\mathsf{PRF}})$. To save space, GM replaces with $k_{\mathsf{GM}}^i := \mathsf{F}_1(k_{\mathsf{GM}}, \text{“}i||SubInstance\text{”})$, so that it can be recovered from $k_{\mathsf{GM}}$.

• The algorithms MInit and MRegGen in $\mathsf{HHGS}^+$ are identical to those in HHGS.

• $\mathsf{Join}(\mathsf{sk}_{\mathsf{GM}}, \mathsf{st}_{\mathsf{GM}}, \mathsf{ID}, \mathrm{RF}_{\mathsf{ID}})$: Although GM can utilize a hyper-join routine to process join requests of members in batches, as implemented in HHGS, we focus here solely on ID for simplicity, to convey the main idea. Let $\mathsf{LR}_{\mathsf{ID}}$ denote the list recording the indices of HHGS instances involved, and $\mathrm{RF}_{\mathsf{ID}}^i$ represent the registration file used in the $i$-th HHGS instance.

For each $opk_{\mathsf{ID}}^j$ in $\mathrm{RF}_{\mathsf{ID}}$ (where $j \in [\#\mathrm{RF}_{\mathsf{ID}}]$), GM randomly assigns it to a HHGS instance and fills $\mathrm{RF}_{\mathsf{ID}}^i$ and $\mathsf{LR}_{\mathsf{ID}}$ accordingly. Specifically, it computes the random selection key $k_j := \mathsf{F}_1(k_{\mathsf{GM}}, \mathsf{ID}||opk_{\mathsf{ID}}^j)$ and determines the HHGS instance index as $\mathsf{il} := \mathsf{F}_2(k_j, \mathsf{ID}||opk_{\mathsf{ID}}^j)$. Next, GM inserts $opk_{\mathsf{ID}}^j$ into $\mathrm{RFCa}_{\mathsf{ID}}^{\mathsf{il}}$ and adds il to $\mathsf{LR}_{\mathsf{ID}}$ if $\mathsf{il} \notin \mathsf{LR}_{\mathsf{ID}}$.

For $i \in \mathsf{LR}_{\mathsf{ID}}$, GM executes $(\mathsf{Vt}_i, \mathrm{RR}_{\mathsf{ID}}^i) \leftarrow \mathsf{Join}(\mathsf{sk}_{\mathsf{GM}}^i, \mathsf{st}_{\mathsf{GM}}^i, \mathsf{ID}, \mathrm{RF}_{\mathsf{ID}}^i, T_\delta)$ with ID, where $\mathrm{RR}_{\mathsf{ID}}^i = (k_{\mathsf{ID}}, \{\mathsf{Auth}[opk_{\mathsf{ID}}^{i,j}]\}_{j\in[\#\mathrm{RF}_{\mathsf{ID}}^i]})$ is the authentication credentials generated for $opk_{\mathsf{ID}}^{i,j}$, and $opk_{\mathsf{ID}}^{i,j}$ is the j-th OTS key in the registration file $\mathrm{RF}_{\mathsf{ID}}^i$ for the ID in the $i$-th HHGS instance. The overall transcript of the protocol is the union of the sub-transcripts from each join session, i.e., $\mathsf{Vt} = \bigcup_{i\in\mathsf{LR}_{\mathsf{ID}}} \mathsf{Vt}_i$.

[9] Note that, the RandSelect function selects nodes randomly from all available nodes (independent of MMTs), while the approach here introduces a structured process, selecting HHGS instances first, creating dependencies in the selection sequence.

Meanwhile, GM appends each authentication credential in $\mathrm{RR}_{\mathsf{ID}}^{i}$ with the corresponding index $i$ to yield a modified registration result set $\{\hat{\mathrm{RR}}_{\mathsf{ID}}^{i}\}_{i\in\mathsf{LR}_{\mathsf{ID}}} = \{(i, \mathsf{Auth}[opk_{\mathsf{ID}}^{i,j}])\}_{i\in\mathsf{LR}_{\mathsf{ID}}, j\in[\#\mathrm{RF}_{\mathsf{ID}}^{i}]}$.

GM builds the states $\hat{\mathsf{st}}_{\mathsf{ID}} = \{\hat{\mathrm{RR}}_{\mathsf{ID}}^{i}\}_{i\in\mathsf{LR}_{\mathsf{ID}}}$ and $\mathsf{st}_{\mathsf{ID}}^{\mathrm{pf}} := \{\mathsf{gpk}_i, \mathrm{pf}_{\mathsf{gpk}_i}\}_{i\in\mathsf{LR}_{\mathsf{ID}}}$ for ID. Finally, it defines $\mathrm{RR}_{\mathsf{ID}} = (k_{\mathsf{ID}}, \mathsf{st}_{\mathsf{ID}}^{\mathrm{pf}}, \hat{\mathsf{st}}_{\mathsf{ID}})$, which is returned to ID. Upon receiving $\mathrm{RR}_{\mathsf{ID}}$, ID updates its states $\mathsf{st} := \mathsf{st} \cup (\mathsf{st}_{\mathsf{ID}}^{\mathrm{pf}}, \hat{\mathsf{st}}_{\mathsf{ID}})$ and secret key $sk_{\mathsf{ID}} := sk_{\mathsf{ID}} \cup k_{\mathsf{ID}}$. If necessary, the member may delete duplicate states and secret keys to save storage.

• $\mathsf{Sign}(sk_{\mathsf{ID}}, \mathsf{st}_{\mathsf{ID}}, m)$: ID first pops up the first tuple $(i, \mathsf{Auth}[opk_{\mathsf{ID}}^{j}])$ from $\hat{\mathsf{st}}_{\mathsf{ID}}$, where $opk_{\mathsf{ID}}^{j}$ is the OTS public key $opk_{\mathsf{ID}}^{j}$ (to be generated within HHGS.Sign later) for verifying the signature about to generate. It then retrieves $(\mathsf{gpk}_i, \mathrm{pf}_{\mathsf{gpk}_i})$ from $\mathsf{st}_{\mathsf{ID}}^{\mathrm{pf}}$ based on $i$. To proceed with the HHGS instance, ID constructs a substate $\mathsf{st}_{\mathsf{ID}}^{i} := \mathsf{Auth}[opk_{\mathsf{ID}}^{j}]$. For singing the message $m$, ID executes $(sk'_{\mathsf{ID}}, \sigma_{\mathsf{ID}}^{i}) := \mathsf{HHGS.Sign}(sk_{\mathsf{ID}}, \mathsf{st}_{\mathsf{ID}}, T_\delta, m)$ and composes the final signature as $\sigma_{\mathsf{ID}} := (\sigma_{\mathsf{ID}}^{i}, \mathsf{gpk}_i, \mathrm{pf}_{\mathsf{gpk}_i})$. Lastly, the tuple $(i, \mathsf{Auth}[opk_{\mathsf{ID}}^{j}])$ is removed from $\mathsf{st}_{\mathsf{ID}}$.

• $\mathsf{Verify}(\mathsf{gpk}, m, \sigma_{\mathsf{ID}}, \mathrm{RL})$: The verifier parses $\sigma_{\mathsf{ID}} = (\sigma_{\mathsf{ID}}^{i}, \mathsf{gpk}_i, \mathrm{pf}_{\mathsf{gpk}_i})$, and it performs two verification steps: i) $mtV := \mathsf{M.Verify}(\mathsf{gpk}, \mathrm{lf}_{\mathsf{gpk}_i}, \mathrm{pf}_{\mathsf{gpk}_i})$, where $(\mathsf{gpk}_i, \mathrm{pf}_{\mathsf{gpk}_i}) \in \sigma_{\mathsf{ID}}$; ii) $dsV := \mathsf{HHGS.Verify}(\mathsf{gpk}_i, m, \sigma_{\mathsf{ID}}^{i}, \mathrm{RL}_i)$. The index $i$ can be inferred analogously from $\sigma_{\mathsf{ID}}$. If $mtV = dsV = 1$, the algorithm outputs 1; otherwise it outputs 0.

• $\mathsf{Open}(\mathsf{gpk}, \mathsf{sk}_{\mathsf{GM}}, m, \sigma_{\mathsf{ID}})$: If $\mathsf{M.Verify}(\mathsf{gpk}, \mathsf{gpk}_i, \mathrm{pf}_{\mathsf{gpk}_i}) = 1$, it outputs the execution result $\mathsf{HHGS.Open}(\mathsf{gpk}_i, \mathsf{sk}_{\mathsf{GM}}^{i}, m,$
$\sigma_{\mathsf{ID}}^{i})$, where $(\mathsf{gpk}_i, \mathrm{pf}_{\mathsf{gpk}_i}, \sigma_{\mathsf{ID}}^{i}) \in \sigma_{\mathsf{ID}}$.

• $\mathsf{Revoke}(\mathsf{gpk}, \mathsf{sk}_{\mathsf{GM}}, \mathrm{RL}, \mathsf{ID}, \mathrm{RSS})$: For each signature $\sigma \in \mathrm{RSS}$, GM gets the index $i$ of the HHGS instance according to the Merkle proofs in $\sigma$, and runs $\mathrm{RL}'_i := \mathsf{HHGS.Revoke}(\mathsf{gpk}_i, \mathsf{sk}_{\mathsf{GM}}^{i}, \mathrm{RL}_i, \mathsf{ID}, \sigma)$. After this, the revocation list is updated to $\mathrm{RL}' := \{\mathrm{RL}'_i\}$.

**Security Analysis.** The correctness of $\mathsf{HHGS}^{+}$ is inherently ensured by the correctness of HHGS, MT, and PRF. Therefore, the analysis is omitted for simplicity. The security of $\mathsf{HHGS}^{+}$ is essentially implied by that of HHGS, along with the security of PRF and M. Here, we mainly present the key security results through the following theorems and outline their proof strategies. We analyze the redundancy in HHGS to resist the FIA, as formally established by Lemma 1, thereby laying the security foundation for $\mathsf{HHGS}^{+}$.

**Lemma 1 (Resilience of FIA).** *Suppose each* HHGS *instance has a maximum capacity of $Y$ leaves, and is expected to register at least $Z$ OTS keys. Given $B = W \cdot Z$ total OTS keys from members, and* $\mathsf{F}_2$ *is a secure* PRF *whose output distribution has a statistical distance of at most $\epsilon_{\mathsf{F}_2}$ from the uniform distribution. Then, the probability that no* HHGS *instance exceeds its capacity is bounded by a negligible probability $\epsilon_{\mathsf{FIA}} = \epsilon_{\mathsf{FIA}}(\kappa)$, provided that $Y$ satisfies the condition $Y \geq Z + \sqrt{2Z \ln\left((1 + O(\epsilon_{\mathsf{F}_2}))/\epsilon_{\mathsf{FIA}}\right)}$.*

*Proof.* The scenario resembles the Coupon Collector's Problem [8], where we aim to collect $Z$ values in each of $W$ instances, and we wish to ensure that no instance exceeds its capacity. The number of $\mathsf{OTS}$ keys involved in each instance can be approximated using a binomial distribution. Let $X_j$ denote the number of values assigned to the $j$-th instance. We aim to find the value of $Y$ such that the probability of any instance exceeding its capacity is negligible. Formally, we want the event $A_j = \{X_j \geq Y\}$ (i.e., the $j$-th $\mathsf{HHGS}$ instance is full) to satisfy $\Pr(A_j) \leq \epsilon_{\mathsf{FIA}}$. The expected number of OTS keys assigned to any $\mathsf{HHGS}$ instance is about $\mathbb{E}[X_j] \approx \frac{B}{W} = Z$. Meanwhile, the statistical distance $\epsilon_{\mathsf{F}_2}$ between the output of $\mathsf{F}_2$ and a uniform distribution introduces a slight deviation from perfect uniformity. This deviation affects the variance, thereby introducing a correction term in the overflow probability bound. Using the Chernoff bound [8], for large $Z$, the probability of the event $A_j$ is given by $\Pr(X_j \geq Y) \leq \exp\left(-\frac{(Y-Z)^2}{2Z}\right)(1 + O(\epsilon_{\mathsf{F}_2}))$, where exp refers to the exponential function. To ensure that this probability is negligible, we require $\exp\left(-\frac{(Y-Z)^2}{2Z}\right)(1 + O(\epsilon_{\mathsf{F}_2})) \leq \epsilon_{\mathsf{FIA}}$. Taking the natural logarithm of both sides $-\frac{(Y-Z)^2}{2Z} \geq \ln((1 + \epsilon_{\mathsf{F}_2})/\epsilon_{\mathsf{FIA}})$. Solving for $Y$, we have $Y \geq Z + \sqrt{2Z \ln((1 + O(\epsilon_{\mathsf{F}_2}))/\epsilon_{\mathsf{FIA}})}$. This complete the proof.

For our target $Z \in O(2^{40})$, $Y$ can be configured around $2^{40} + 2^{20.5}$, ensuring a negligible overflow probability of $\epsilon_{\mathsf{FIA}} \leq 2^{-128}$.

**Theorem 3.** *Assume that* $\mathsf{HHGS}$ *provides traceability and the* $\mathsf{MT}$ *scheme* $\mathsf{M}$ *is secure, then* $\mathsf{HHGS}^+$ *satisfies traceability.*

The proof of Theorem 3 closely follows that of Theorem 1, with the primary distinction being the need for an additional game to reduce the security of $\mathsf{HHGS}^+$ to that of $\mathsf{MT}$.

Table 2: Performance. Runtime in milliseconds (ms) and size in kilobytes (KB).

| $\kappa$ | $\mathsf{HHGS}$ | | | | | | $\mathsf{HHGS}^+$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Signing Capacity ($d$) | Setup | Join | Sign | Verify | Sig. Size | Signing Capacity ($d$) | Setup | Join | Sign | Verify | Sig. Size |
| 128 | $2^{39}(12)$ | 1.53 | 301.68 | 2.3 | 1.08 | 7.95 | $2^{59}(12)$ | $16 \times 10^5$ | 303.73 | 2.3 | 1.09 | 8.58 |
| | $2^{40}(9)$ | 3.14 | 318.89 | 2.3 | 0.82 | 6.41 | $2^{60}(9)$ | $33 \times 10^5$ | 323.07 | 2.3 | 0.83 | 7.04 |
| | $2^{40}(19)$ | 0.74 | 294.59 | 2.3 | 1.64 | 11.58 | $2^{60}(19)$ | $7.7 \times 10^5$ | 295.29 | 2.3 | 1.65 | 12.21 |
| 256 | $2^{39}(12)$ | 7.29 | 654.62 | 12.87 | 5.86 | 20.25 | $2^{59}(12)$ | $76 \times 10^5$ | 676.22 | 12.87 | 5.87 | 20.88 |
| | $2^{40}(9)$ | 14.68 | 713.46 | 12.87 | 4.40 | 16.24 | $2^{60}(9)$ | $153 \times 10^5$ | 730.85 | 12.87 | 4.41 | 16.87 |
| | $2^{40}(19)$ | 3.62 | 632.91 | 12.87 | 8.77 | 29.72 | $2^{60}(19)$ | $38 \times 10^5$ | 637.34 | 12.87 | 8.78 | 30.35 |

**Theorem 4.** *Assume that* $\mathsf{HHGS}$ *provides anonymity and the* $\mathsf{PRF}$ *families* $(\mathsf{F}_1, \mathsf{F}_2)$ *are secure, then* $\mathsf{HHGS}^+$ *satisfies anonymity.*

To prove this theorem, it suffices to demonstrate that random selection among $\mathsf{HHGS}$ instances does not leak identity-related information, since OTS keys within each $\mathsf{HHGS}$ instance are indistinguishable due to its security. It is

straightforward to observe that if no HHGS instance becomes fully occupied during the lifetime of $\mathsf{HHGS}^+$ (i.e., the resilience against FIA attacks), then OTS keys are randomly assigned to $W$ independent HHGS instances with nearly identical probability (with negligible differences due to the security of F). Using Lemma 1, we show that by properly configuring the size of each HHGS instance, the probability of a successful FIA attack is negligible.

## 5 Evaluation

In this section, we evaluate the performance of our proposed FSDGS schemes.

In the performance analysis, we instantiate the OTS scheme using the state-of-the-art OTS scheme WOTS+C [31,41], and use AES-GCM-SIV [29] for the authenticated encryption E.[10] For the PRG, we use counter-mode AES [3]. The hash functions in our schemes are implemented using the SHA-3 family [24]. The PPRF is implemented with the GGM construction [12]. Meanwhile, we consider 128-bit security against classical attacks and (roughly) double the security parameter to ensure post-quantum security.

For the parameters of WOTS+C, we aim to strike a trade-off between signature size and computational cost by selecting a relatively small number of $2^5$ of signature chains, with a Winternitz parameter of $2^4$ for the length of each chain, and a 9-bit checksum size. The signature size of HHGS is primarily determined by the signing capacity ($Y = 2^{\tilde{h}}$), selected as either $2^{39}$ or $2^{40}$ to accommodate the corresponding group size. The cost of authenticating one OTS in the Join protocol, involve at most $d$+1 OTS signing operations and the cost of RandSelect (about $O(2^{20})$ $\mathsf{F}_2$ operations in the worst case). Let $B$ (e.g., $B \in O(2^{15})$) denote the maximum number of OTS keys a member retains. Consequently, the credentials stored in the member's cache have a size of $O(B \cdot h)$. Members can adjust $B$ based on the storage capacity of their auxiliary storage devices. We consider a practical scenario where, for the first join request, a member must submit at least $B$ OTS keys, and the GM processes join requests from $U$ members in a batch. For simplicity, we assume $B \cdot U = \sqrt{Y}$. Thus, the storage cost of the GM's state, dominated by the PPRF keys, is of magnitude $O(\sqrt{Y})$. The signing process on the member's device is efficient, with the worst-case computational cost dominated by two WOTS+C key generation and signing operations.

We benchmark the performance of the algorithms in our proposed schemes using a Raspberry Pi 3 to simulate the client's signing operation, and a PC with an Intel(R) Core(TM) i5-13500H, 2.60 GHz, and 16 GB RAM, which serves as the verifier and GM. Table 2 provides a summary of the performance of our schemes across key metrics, as in prior works [2,21,45].[11] For simplicity, we do not benchmark the full schemes. The performance of Open and Revoke can be inferred

[10] Although the newer signature scheme [34] provides shorter signatures and more efficient verification, this work adopts WOTS+C for its superior signing efficiency, making it better suited to computation-constrained signers nor powerful verifier.

[11] The source code of our implementation is available at https://github.com/WithoutNi/GS.

from that of Verify. Specifically, the cost of Open is primarily determined by a single Verify operation along with one encryption and one decryption. To revoke a single signature, the overhead of Revoke consists of the combined costs of one Verify and one Open operation. Additionally, the runtime of the join protocol is measured based on handling a single OTS key in the worst-case scenario, dominated by $O(2^{20})$ $\mathsf{F}_2$ operations. The height of the upper layer Merkle tree of $\mathsf{HHGS}^+$ is $W = 2^{20}$ in our benchmark. As a result, the setup time (about a few minutes) for $\mathsf{HHGS}^+$ is $W$ times that of HHGS.

## Acknowledgment

This work was supported by the Natural Science Foundation of China under Grant No. 12441101 and Grant No. 62372386, the National Cryptography Science Foundation of China under Grant No. 2025NCSF02055, and by the National Key Research and Development Program of China under Grant No. 2025YFE0220300.

## References

1. Aviram, N., Gellert, K., Jager, T.: Session resumption protocols and efficient forward security for TLS 1.3 0-rtt. J. Cryptol. **34**(3), 20 (2021)
2. Bansarkhani, R.E., Misoczki, R.: G-merkle: A hash-based group signature scheme from standard assumptions. In: PQCrypto. pp. 441–463. Springer (2018)
3. Barker, E.B., Kelsey, J.M.: Recommendation for random number generation using deterministic random bit generators (revised). US Department of Commerce, Technology Administration, NIST (2007)
4. Bellare, M., Micciancio, D., Warinschi, B.: Foundations of group signatures:formal definitions, simplified requirements, and a construction based on general assumptions. In: EUROCRYPT. pp. 614–629. Springer (2003)
5. Bellare, M., Shi, H., Zhang, C.: Foundations of group signatures: The case of dynamic groups. In: CT-RSA. pp. 136–153. Springer (2005)
6. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: SPHINCS: practical stateless hash-based signatures. In: EUROCRYPT. pp. 368–397. Springer (2015)
7. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs$^+$ signature framework. In: CCS. pp. 2129–2146. ACM (2019)
8. Bertsekas, D., Tsitsiklis, J.N.: Introduction to probability, vol. 1. Athena Scientific (2008)
9. Beullens, W., Dobson, S., Katsumata, S., Lai, Y., Pintore, F.: Group signatures and more from isogenies and lattices: Generic, simple, and efficient. In: EUROCRYPT. pp. 95–126. Springer (2022)
10. Bobolz, J., Diaz, J., Kohlweiss, M.: Foundations of anonymous signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In: Financial Crypto. pp. 121–139. Springer (2024)
11. Boneh, D., Eskandarian, S., Fisch, B.: Post-quantum EPID signatures from symmetric primitives. In: CT-RSA. pp. 251–271. Springer (2019)

12. Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. In: ASIACRYPT. pp. 280–300. Springer (2013)
13. Bootle, J., Cerulli, A., Chaidos, P., Ghadafi, E., Groth, J.: Foundations of fully dynamic group signatures. J. Cryptol. **33**(4), 1822–1870 (2020)
14. Brassard, G., Høyer, P., Tapp, A.: Quantum cryptanalysis of hash and claw-free functions. In: LATIN. pp. 163–169. LNCS, Springer (1998)
15. Brickell, E.F., Camenisch, J., Chen, L.: Direct anonymous attestation. In: CCS. pp. 132–145. ACM (2004)
16. Bringer, J., Chabanne, H., Pointcheval, D., Zimmer, S.: An application of the boneh and shacham group signature scheme to biometric authentication. In: IWSEC. pp. 219–230. Springer (2008)
17. Buchmann, J., Dahmen, E., Schneider, M.: Merkle tree traversal revisited. In: PQCrypto. pp. 63–78. Springer (2008)
18. Buser, M., Liu, J.K., Steinfeld, R., Sakzad, A., Sun, S.: DGM: A dynamic and revocable group merkle signature. In: ESORICS. pp. 194–214. Springer (2019)
19. Canetti, R., Goldreich, O., Halevi, S.: The random oracle methodology, revisited. J. ACM **51**(4), 557–594 (2004)
20. Cao, X., Yang, Z., Reijsbergen, D., Ning, J., Ke, J., Ma, Z., Zhou, J.: HHGS: Forward-secure dynamic group signatures from symmetric primitives. Cryptology ePrint Archive, Paper 2025/2270 (2025), https://eprint.iacr.org/2025/2270
21. Chen, L., Dong, C., Newton, C.J.P., Wang, Y.: Sphinx-in-the-head: Group signatures from symmetric primitives. ACM Trans. Priv. Secur. **27**(1), 11:1–11:35 (2024)
22. Cheval, V., Cremers, C., Dax, A., Hirschi, L., Jacomme, C., Kremer, S.: Hash gone bad: Automated discovery of protocol attacks that exploit hash function weaknesses. In: USENIX Security Symposium. pp. 5899–5916. USENIX Association (2023)
23. Dagdelen, Ö., Fischlin, M., Gagliardoni, T.: The fiat-shamir transformation in a quantum world. In: ASIACRYPT. pp. 62–81. Springer (2013)
24. Dworkin, M.J.: Sha-3 standard: Permutation-based hash and extendable-output functions (2015)
25. Fadavi, M., Karati, S., Erfanian, A., Safavi-Naini, R.: DGMT: A fully dynamic group signature from symmetric-key primitives. Cryptogr. **9**(1), 12 (2025)
26. Faonio, A., Fiore, D., Nizzardo, L., Soriente, C.: Subversion-resilient enhanced privacy ID. In: CT-RSA. pp. 562–588. Springer (2022)
27. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. J. ACM **33**(4), 792–807 (1986)
28. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: STOC. pp. 212–219. ACM (1996)
29. Gueron, S.: Aes-gcm-siv implementations (128 and 256 bit) (2018), https://github.com/Shay-Gueron/AES-GCM-SIV
30. Hülsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A.: XMSS: extended merkle signature scheme. RFC **8391**, 1–74 (2018)
31. Hülsing, A., Kudinov, M.A., Ronen, E., Yogev, E.: SPHINCS+C: compressing SPHINCS+ with (almost) no cost. In: SP. pp. 1435–1453. IEEE (2023)
32. Hülsing, A., Rijneveld, J., Song, F.: Mitigating multi-target attacks in hash-based signatures. In: PKC. pp. 387–416. LNCS, Springer (2016)
33. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge proofs from secure multiparty computation. SIAM J. Comput. **39**(3), 1121–1152 (2009)

34. Khovratovich, D., Kudinov, M., Wagner, B.: At the top of the hypercube - better size-time tradeoffs for hash-based signatures. In: Tauman Kalai, Y., Kamara, S.F. (eds.) CRYPTO 2025. pp. 93–123. Springer Nature Switzerland (2025)
35. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: CCS. pp. 669–684. ACM (2013)
36. Kiayias, A., Yung, M.: Extracting group signatures from traitor tracing schemes. In: EUROCRYPT. pp. 630–648. Springer (2003)
37. Ling, S., Nguyen, K., Wang, H., Xu, Y.: Forward-secure group signatures from lattices. In: PQCrypto. pp. 44–64. Springer (2019)
38. Lyubashevsky, V., Nguyen, N.K., Plançon, M., Seiler, G.: Shorter lattice-based group signatures via "almost free" encryption and other optimizations. In: ASIACRYPT. pp. 218–248. Springer (2021)
39. Nguyen, K., Tang, H., Wang, H., Zeng, N.: New code-based privacy-preserving cryptographic constructions. In: ASIACRYPT. pp. 25–55. Springer (2019)
40. Omar, S., Padhye, S.: Multivariate linkable group signature scheme. In: Proceedings of the International Conference on Computing and Communication Systems. pp. 623–632. Springer, Singapore (2021)
41. Ronen, E.: Software for the sphincs+c scheme (2022), https://github.com/eyalr0/sphincsplusc
42. Song, D.X.: Practical forward secure group signature schemes. In: CCS. pp. 225–234. ACM (2001)
43. Yang, G., Tang, S., Yang, L.: A novel group signature scheme based on MPKC. In: ISPEC. pp. 181–195. Springer (2011)
44. Yang, R., Au, M.H., Zhang, Z., Xu, Q., Yu, Z., Whyte, W.: Efficient lattice-based zero-knowledge arguments with standard soundness: Construction and applications. In: CRYPTO. pp. 147–175. Springer (2019)
45. Yehia, M., AlTawy, R., Gulliver, T.A.: $\mathrm{Gm}^{\mathrm{mt}}$: A revocable group merkle multi-tree signature scheme. In: CANS. pp. 136–157. Springer (2021)

## A Security Notions of Building Blocks

In this section, we define the security notions of the building blocks used in our constructions.

Let $\mathsf{negl}(\kappa) : \mathbb{N} \to \mathbb{R}+$ denote a negligible function, defined such that for every polynomial $P(\kappa)$ there exists a $e_0 \in \mathbb{N}$ s.t. for all $e > e_0$, $\mathsf{negl}(e) \leq 1/P(e)$. We write $\mathcal{A}^{\mathcal{O}}$ to denote the an algorithm $\mathcal{A}$ with access to oracle $\mathcal{O}$ where $\mathcal{O}$ may represent multiple oracles, i.e., $\mathcal{O} = \{\mathcal{O}_1, \ldots, \mathcal{O}_n\}$. For security definitions, we use $\mathsf{Exp}^{\Psi}_{\mathcal{A},\Sigma}(1^\kappa, \rho) \Rightarrow 1$ to denote that the experiment $\mathsf{Exp}$, when instantiated with security parameter $\kappa$, system parameters $\rho$, under an adversary $\mathcal{A}$ attacking the security property $\Psi$ of the primitive $\Sigma$, returns 1 (indicating success). The parameters $\rho$, $\mathcal{A}$, and $\Psi$ may be omitted when clear from context. We define the advantage of $\mathcal{A}$ in this experiment as

$$\mathsf{Adv}^{\Psi}_{\mathcal{A},\Sigma,\lambda} := \left| \Pr[\mathsf{Exp}^{\Psi}_{\mathcal{A},\Sigma}(1^\kappa, \rho) \Rightarrow 1] - \lambda \right|,$$

where $\lambda$ may be omitted if $\lambda = 0$.

**Digital Signature.** We say that a SIG scheme $\mathsf{S}$ is correct if for all $m \in \mathcal{M}_{\mathsf{SIG}}$ it holds that

$$\Pr\left[\mathsf{S.Verify}(pk, m, \sigma) = 1 \,\middle|\, \begin{array}{c} pms_{\mathsf{SIG}} \leftarrow \mathsf{S.Setup}(1^\kappa);\ rs \xleftarrow{\$} \mathcal{RS}_{\mathsf{SIG}};\ (sk, pk) \leftarrow \mathsf{S.KGen}(rs) \\ \sigma \leftarrow \mathsf{S.Sign}(sk, m) \end{array}\right] = 1.$$

To capture existential unforgeability under adaptive chosen message attacks (EUF-CMA), we define a security experiment $\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{S}}(\kappa, q)$ as below:

$\underline{\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{S}}(\kappa, q):}$
$pms_{\mathsf{SIG}} \leftarrow \mathsf{S.Setup}(1^\kappa);\ rs \xleftarrow{\$} \mathcal{RS}_{\mathsf{SIG}};\ (sk, pk) \leftarrow \mathsf{S.KGen}(rs)$
$ct := 0;\ \mathsf{L}_S := \emptyset$
$(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{HMSign}(\cdot)}(\kappa, pms_{\mathsf{SIG}}, pk);$
$\mathsf{OUTPUT}\ ((m^*, \sigma^*) \notin \mathsf{L}_S) \wedge (\mathsf{S.Verify}(pk, m^*, \sigma^*) = 1)$

$\underline{\mathsf{HMSign}(m):}$
$\mathsf{IF}\ ct \geq q,\ \mathsf{OUTPUT}\ \perp$
$ct := ct + 1;\ \sigma \leftarrow \mathsf{S.Sign}(sk, m)$
$\mathsf{APPEND}\ (m, \sigma) \rightarrow \mathsf{L}_S$
$\mathsf{OUTPUT}\ \sigma$

**Definition 2 (EUF-CMA for SIG).** *We say that* $\mathsf{S}$ *is an EUF-CMA secure* $\mathsf{SIG}$ *scheme if for any PPT* $\mathcal{A}$*, the advantage* $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{S}}(\kappa, q)$ *of* $\mathcal{A}$ *in* $\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A},\mathsf{S}}(\kappa, q)$ *is negligible. An EUF-CMA secure* $\mathsf{SIG}$ *scheme* $\mathsf{S}$ *with* $q = 1$ *is referred to as a one-time signature (OTS) scheme.*

**Merkle Tree.** We say that a $\mathsf{MT}$ scheme $\mathsf{M}$ is correct if for a polynomial number $N$ defined by $\kappa$, any $\{\mathrm{lf}_i\}_{i\in[N]}$ (s.t. $\mathrm{lf}_i \in \mathcal{L}_{\mathsf{M}}$) and $\mathrm{lf}_j \in \{\mathrm{lf}_i\}_{i\in[N]}$, it holds that

$$\Pr\left[\mathsf{M.Verify}(\mathrm{Tr.Rt}, \mathrm{lf}_j, \mathrm{pf}_{\mathrm{lf}_j}) = 1 \,\middle|\, \begin{array}{c} pms_{\mathsf{MT}} \leftarrow \mathsf{M.Setup}(1^\kappa);\ (\mathrm{Tr}, \mathsf{st}_{\mathrm{BDS}}) \leftarrow \mathsf{M.Build}(\{\mathrm{lf}_i\}_{i\in[N]}) \\ \mathrm{pf}_{\mathrm{lf}_j} \leftarrow \mathsf{M.GetPrf}(\mathsf{st}_{\mathrm{BDS}}, \mathrm{lf}_j) \end{array}\right] = 1.$$

We define a security experiment $\mathsf{Exp}^{\mathsf{UF}}_{\mathcal{A},\mathsf{M}}(\kappa)$ regarding unforgeability ($\mathsf{UF}$) for a $\mathsf{MT}$ scheme $\mathsf{M}$ as below:

$\underline{\mathsf{Exp}^{\mathsf{UF}}_{\mathcal{A},\mathsf{M}}(\kappa):}$
$pms_{\mathsf{MT}} \leftarrow \mathsf{M.Setup}(1^\kappa)\ ;\ (st, \{\mathrm{lf}_i\}_{i\in[N]}) \leftarrow \mathcal{A}_1(\kappa, pms_{\mathsf{MT}})$
$(\mathrm{Tr}, \mathsf{st}_{\mathrm{BDS}}) \leftarrow \mathsf{M.Build}(\{\mathrm{lf}_i\}_{i\in[N]});\ (\mathrm{lf}^*, \mathrm{pf}^*) \leftarrow \mathcal{A}_2(st, \mathrm{Tr})$
$\mathsf{OUTPUT}\ (\mathsf{M.Verify}(\mathrm{Tr.Rt}, \mathrm{lf}^*, \mathrm{pf}^*) = 1) \wedge (\mathrm{lf}^* \notin \{\mathrm{lf}_i\}_{i\in[N]})$

**Definition 3 (Unforgeability of MT).** *We say that* $\mathsf{M}$ *is a secure* $\mathsf{MT}$ *scheme if the advantage* $\mathsf{Adv}^{\mathsf{UF}}_{\mathcal{A},\mathsf{M}}(\kappa)$ *of any PPT adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *in* $\mathsf{Exp}^{\mathsf{UF}}_{\mathcal{A},\mathsf{M}}(\kappa)$ *is negligible.*

**Authenticated Encryption.** An $\mathsf{AE}$ scheme $\mathsf{E}$ is said to be correct if for any $m \in \mathcal{M}_{\mathsf{AE}}$ it holds that

$$\Pr\left[\mathsf{E.Dec}(k, c) = m \,\middle|\, \begin{array}{c} pms_{\mathsf{AE}} \xleftarrow{\$} \mathsf{E.Setup}(1^\kappa);\ rk \xleftarrow{\$} \mathcal{RK}_{\mathsf{AE}};\ k \leftarrow \mathsf{E.KGen}(rk) \\ re \xleftarrow{\$} \mathcal{ER}_{\mathsf{AE}};\ c \leftarrow \mathsf{E.Enc}(k, m; re) \end{array}\right] = 1.$$

In the following, we first define the experiment for the security property regarding indistinguishability under adaptive chosen-ciphertext attacks ($\mathsf{IND\text{-}CCA}$).

**Definition 4 (IND-CCA for AE).** *We say that* $\mathsf{E}$ *is an* $\mathsf{IND\text{-}CCA}$*-secure AE scheme if the advantage* $\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathcal{A},\mathsf{E},\frac{1}{2}}(\kappa)$ *of all PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *in* $\mathsf{Exp}^{\mathsf{IND\text{-}CCA}}_{\mathcal{A},\mathsf{E}}(\kappa)$ *is negligible.*

$\underline{\mathsf{Exp}^{\mathsf{IND\text{-}CCA}}_{\mathcal{A},\mathsf{E}}(\kappa):}$
$pms_{\mathsf{AE}} \xleftarrow{\$} \mathsf{E.Setup}(1^\kappa)$; $b \xleftarrow{\$} \{0,1\}$; $rk \xleftarrow{\$} \mathcal{RK}_{\mathsf{AE}}$; $k \leftarrow \mathsf{E.KGen}(rk)$
$(m_0, m_1, st) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathsf{Dec}}(\cdot)}(\kappa, pms_{\mathsf{AE}})$; $re^* \xleftarrow{\$} \mathcal{ER}_{\mathsf{AE}}$; $c^* \leftarrow \mathsf{E.Enc}(k, m_b; re^*)$
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathsf{Dec}}(\cdot)}(c^*, m_0, m_1, st)$;
OUTPUT $b = b'$

$\underline{\mathcal{O}_{\mathsf{Dec}}(c):}$
If $c = c^*$, OUTPUT $\perp$
$m \leftarrow \mathsf{E.Dec}(k, c)$
OUTPUT $m$

Also, we need the AE to provide integrity CT-INT of ciphertext against adaptive-chosen message attacks, which is formalized by the following experiment.

$\underline{\mathsf{Exp}^{\mathsf{CT\text{-}INT}}_{\mathcal{A},\mathsf{E}}(\kappa):}$
$pms_{\mathsf{AE}} \xleftarrow{\$} \mathsf{E.Setup}(1^\kappa)$; $\mathsf{L_{AE}} := \emptyset$; $rk \xleftarrow{\$} \mathcal{RK}_{\mathsf{AE}}$; $k \xleftarrow{\$} \mathsf{E.KGen}(rk)$
$c^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Enc}}(\cdot)}(\kappa, pms_{\mathsf{AE}})$; $m^* \leftarrow \mathsf{Dec}(k, c^*)$;
OUTPUT $c^* \notin \mathsf{L_{AE}} \wedge m^* \neq \perp$

$\underline{\mathcal{O}_{\mathsf{Enc}}(m):}$
$re \xleftarrow{\$} \mathcal{ER}_{\mathsf{AE}}$
$c \leftarrow \mathsf{E.Enc}(k, m; re)$
APPEND $c \rightarrow \mathsf{L_{AE}}$
Return $c$

**Definition 5 (CT-INT for AE).** *We say that* E *is a* CT-INT*-secure AE if the advantage* $\mathsf{Adv}^{\mathsf{CT\text{-}INT}}_{\mathcal{A},\mathsf{E}}(\kappa)$ *of all PPT adversaries* $\mathcal{A}$ *in* $\mathsf{Exp}^{\mathsf{CT\text{-}INT}}_{\mathcal{A},\mathsf{E}}(\kappa)$ *is negligible.*

**Pseudorandom Generators.** We define the following security experiment $\mathsf{Exp}^{\mathsf{IND}}_{\mathcal{A},\mathsf{G}}(\kappa)$ to formalize the security of a PRG scheme G based on indistinguishability (IND):

$\underline{\mathsf{Exp}^{\mathsf{IND}}_{\mathcal{A},\mathsf{G}}(\kappa):}$
$b \xleftarrow{\$} \{0,1\}$; $pms_{\mathsf{PRG}} \leftarrow \mathsf{G.Setup}(1^\kappa)$; $s \xleftarrow{\$} \mathsf{G.SGen}(pms_{\mathsf{PRG}})$; $r_0 \xleftarrow{\$} \mathcal{R}_{\mathsf{PRG}}$; $r_1 \leftarrow \mathsf{G.Eval}(s)$
$b^* \leftarrow \mathcal{A}(\kappa, pms_{\mathsf{PRG}}, r_b)$;
OUTPUT $b = b^*$

**Definition 6 (IND for PRG).** *We say that* G *is a secure* PRG *scheme if for any probabilistic polynomial time (PPT) adversary* $\mathcal{A}$*, it holds that the advantage* $\mathsf{Adv}^{\mathsf{IND}}_{\mathcal{A},\mathsf{G},\frac{1}{2}}(\kappa)$ *of* $\mathcal{A}$ *in* $\mathsf{Exp}^{\mathsf{IND}}_{\mathcal{A},\mathsf{G}}(\kappa)$ *is negligible.*

**(Puncturable) Pseudo-random Functions.** We say that a PPRF scheme F is correct if for every subset $\mathcal{S} = \{x_1, \ldots, x_q\} \subseteq \mathcal{M}_{\mathsf{PRF}}$ and any $x \in \mathcal{M}_{\mathsf{PRF}} \setminus \mathcal{S}$ it holds that

$$\Pr\left[\mathsf{F.Eval}(k_0, x) = \mathsf{F.Eval}(k_t, x) \,\middle|\, \begin{array}{c} pms_{\mathsf{PRF}} \leftarrow \mathsf{F.Setup}(1^\kappa);\ k \xleftarrow{\$} \mathsf{F.KGen}(pms_{\mathsf{PRF}}) \\ \text{for } i \in [q] : k_i \leftarrow \mathsf{F.Punc}(k_{i-1}, x_i) \end{array}\right] = 1.$$

We define a security experiment $\mathsf{Exp}^{\mathsf{IND\text{-}CMA}}_{\mathcal{A},\mathsf{F}}(\kappa, q_f)$ for (P)PRF below. The boxed lines are exclusively used in the security experiment for PPRF.

**Definition 7 (IND-CMA for (P)PRF).** *We say that* F *is a secure (P)PRF scheme if, in the experiment* $\mathsf{Exp}^{\mathsf{IND\text{-}CMA}}_{\mathcal{A},\mathsf{F}}(\kappa, q_f)$*, the advantage* $\mathsf{Adv}^{\mathsf{IND\text{-}CMA}}_{\mathcal{A},\mathsf{F},\frac{1}{2}}(\kappa, q_f)$ *of any PPT adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *is negligible.*

$\underline{\mathsf{Exp}^{\mathsf{IND\text{-}CMA}}_{\mathcal{A},\mathsf{F}}(\kappa, q_f):}$
$b \xleftarrow{\$} \{0,1\}$; $ct := 0$; $pms_{\mathsf{PRF}} \leftarrow \mathsf{F.Setup}(1^\kappa)$; $k \xleftarrow{\$} \mathsf{F.KGen}(pms_{\mathsf{PRF}})$
$(x^*, st) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathsf{PRF}}(\cdot)}(\kappa, pms_{\mathsf{PRF}})$; $r_0 \xleftarrow{\$} \mathcal{R}_{\mathsf{PRF}}$; $r_1 \leftarrow \mathsf{F.Eval}(k, x^*)$
IF $x^* \in \mathsf{L}_{\mathsf{PRF}}$, OUTPUT $\perp$
$\boxed{k \leftarrow \mathsf{F.Punc}(k, x^*)}$; $b' \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathsf{PRF}}(\cdot)}(st, r_b, \boxed{k})$
OUTPUT $(b' = b)$

$\underline{\mathcal{O}_{\mathsf{PRF}}(x):}$
IF $ct \geq q_f$, OUTPUT $\perp$
APPEND $x \rightarrow \mathsf{L}_{\mathsf{PRF}}$
$r \leftarrow \mathsf{F.Eval}(k, x)$
$\boxed{k \leftarrow \mathsf{F.Punc}(k, x)}$
OUTPUT $r$

In addition, we define the statistical distance (SD) of $\mathsf{F}$ (with a random key $k \xleftarrow{\$} \mathcal{K}_{\mathsf{PRF}}$) to a uniform random function $\mathsf{U} : \mathcal{M}_{\mathsf{PRF}} \rightarrow \mathcal{R}_{\mathsf{PRF}}$, as $\Delta(\mathsf{F}, \mathsf{U}) = \frac{1}{2} \sum_{x \in \mathcal{M}_{\mathsf{PRF}}} \big| \Pr[\mathcal{F}(k, x) = y] - \Pr[\mathsf{U}(x) = y] \big|$.

**Collision-resistant Hash Functions.** We define the security experiment $\mathsf{Exp}^{\mathsf{CR}}_{\mathcal{A},\mathsf{H}}(\kappa)$ for a CRH scheme $\mathsf{H}$ as follows.

$\underline{\mathsf{Exp}^{\mathsf{CR}}_{\mathcal{A},\mathsf{H}}(\kappa):}$
$(hk, pms_{\mathsf{CRH}}) \leftarrow \mathsf{H.Setup}(1^\kappa)$; $(m_0, m_1) \leftarrow \mathcal{A}(\kappa, pms_{\mathsf{CRH}}, hk)$
OUTPUT $(m_0 \neq m_1) \wedge (\mathsf{H.Eval}(hk, m_0) = \mathsf{H.Eval}(hk, m_1))$

**Definition 8.** *We say* $\mathsf{H}$ *is secure if no PPT adversary has non-negligible advantage* $\mathsf{Adv}^{\mathsf{CR}}_{\mathcal{A},\mathsf{H}}(\kappa)$ *in* $\mathsf{Exp}^{\mathsf{CR}}_{\mathcal{A},\mathsf{H}}(\kappa)$.

## B Tree-based PPRF

In this section, we briefly revisit the key concept of tree-based PPRFs [12,35,1], built from the well-known one-way function based PRF construction, proposed by Goldreich, Goldwasser, and Micali (GGM) [27].

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 $x = x_1x_2 \dots x_n$, realizing $\mathsf{PPRF.Punc}(k, x)$ , the initial root key $k$ is replaced with intermediate node evaluations for prefixes $\overline{x_1}, x_1\overline{x_2}, x_1x_2\overline{x_3}, \dots, x_1x_2 \dots \overline{x_n}$. 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 $0 \rightarrow$ currentPrefix. Otherwise, append $1 \rightarrow$ 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.