DGSP: An Efficient Scalable Fully Dynamic Group Signature Scheme Using SPHINCS$^+$

Mojtaba Fadavi[1(✉)], Seyyed Arash Azimi[2], Sabyasachi Karati[3], and Samuel Jaques[1]

[1] Department of Combinatorics & Optimization, University of Waterloo, Waterloo, Canada
mfadavi,sejaques@uwaterloo.ca

[2] CPS2 Lab, Electrical Engineering Department, Shahid Rajaee Teacher Training University, Tehran, Iran
arashazimi7@gmail.com

[3] Cryptology and Security Research Unit, Indian Statistical Institute, Kolkata, India
skarati@isical.ac.in

Abstract. A *group signature scheme* enables users of a group to anonymously sign messages on behalf of the group, while a designated authority can revoke anonymity when needed to ensure user accountability. In this paper, we build a post-quantum fully dynamic group signature scheme from only symmetric encryption and hash functions.

Our protocol, DGSP, achieves the following: (i) the set-up time is effectively constant in the number of signatures that may be issued, with support for up to 2^{64} signatures; (ii) the tracing algorithm run by the authority has constant runtime in the number of users; (iii) the set of users is fully dynamic and users can be revoked or added as needed without system-wide updates; and (iv) forward anonymity, where if a user's secrets are compromised they cannot be used to de-anonymize previous signatures. DGSP is the first group signature based only on symmetric primitives to achieve all of these properties: the previous state-of-the-art in this area is SPHINX-in-the-Head (SITH), which does not achieve (ii) or (iv), and DGMT, which does not achieve (i) and may not achieve (iv). Like DGMT, but unlike SITH, DGSP is *stateful* and users must refresh a local storage of "certificates" to issue new signatures.

We provide a full Rust implementation, showing that our signatures are roughly 5× larger than DGMT but nearly 100× smaller than SITH, and all basic operations run in under 2 milliseconds. We prove security in the standard model based on typical assumptions for symmetric primitives. DGSP is a compelling solution for applications requiring large-scale user support, efficient operations, and conservative post-quantum security.

Keywords: Post-Quantum Cryptography · Group Signature Scheme · Hash-Based Signature Scheme · SPHINCS$^+$

© IACR 2026. This is the extended version of the article published in the proceedings of EUROCRYPT 2026 by Springer-Verlag, available at https://doi.org/10.1007/978-3-032-25330-9_16. It includes the full security proofs and additional exposition on SPHINCS$^+$.

Table of Contents

1 Introduction . . . 3
1.1 Hash-based group signature schemes . . . 4
1.2 Our Contributions . . . 5
1.3 DGSP and Other Post-Quantum GSSs . . . 7
2 Preliminaries . . . 8
Keyed second preimage resistant hash function. . . . 9
2.1 Signature Schemes . . . 9
3 Fully Dynamic Group Signature . . . 10
4 DGSP: An Efficient, Scalable, Fully Dynamic Group Signature Scheme Using SPHINCS$^+$ . . . 16
5 Implementation and Experimental Results . . . 26
6 Comparing DGSP with DGMT and SITH . . . 28
7 Conclusion . . . 30
A Background . . . 36
Pseudorandom Functions. . . . 36
Digital Signatures. . . . 37
Tweakable hash function. . . . 38
WOTS$^+$. . . . 39
Multi-message Signature Scheme (MSS) based on WOTS$^+$. . . . 40
Pseudorandom functions and the message digest. . . . 41
FORS (Forest Of Random Subsets). . . . 41
SPHINCS$^+$. . . . 41
B Security Proofs . . . 44
B.1 Proof of Lemma 1 . . . 44
B.2 Proof of Lemma 2 . . . 45
B.3 Proof of Lemma 3 . . . 47
B.4 Proof of Anonymity 4 . . . 48
B.5 Keyed Second Preimage Resistance . . . 50

1 Introduction

Digital signature schemes are essential for ensuring data integrity, authenticity, and non-repudiation in secure communication, with widespread applications in email, software distribution, and blockchain technologies. Widely used digital signature schemes like (EC)DSA and RSA are secure against classical computers but vulnerable to quantum computers due to Shor's algorithm [53]. The advent of large-scale quantum computers has spurred extensive research in *Post Quantum (PQ)* cryptography, accompanied by global standardization initiatives led by the National Institute of Standards and Technology (NIST) and the European Telecommunications Standards Institute (ETSI) [1,30]. To date, NIST has standardized three PQ cryptographic schemes: (i) ML-KEM [50], for key encapsulation, based on CRYSTALS-Kyber [13]; (ii) ML-DSA [49] for digital signatures, derived from CRYSTALS-Dilithium [25]; and (iii) SLH-DSA [51], a stateless hash-based digital signature scheme using SPHINCS$^+$ [7]. These standards ensure resilience against quantum attacks and are part of the global transition to quantum-safe cryptography. Generally, known proposed PQ-secure signature schemes have based their security on the hardness of computational problems for which no efficient quantum algorithm is known [9,10,23,25,35], or use only hash functions and symmetric-key primitives [16,20,36,37,45,47,48]. The latter has an advantage over the former, as its security against quantum attacks relies on generic, unstructured algorithms such as Grover's search algorithm [34].

Hash-based signature schemes stem from Lamport's *One-Time Signature (OTS)* scheme, which uses hash functions to sign a single message securely [45]. To extend an OTS scheme to a Multi-message signature (MMS) scheme, Merkle introduced a method employing a binary hash tree, known as a Merkle tree, constructed over a set of OTS public keys. The root of this tree serves as the scheme's public key, effectively reducing the public key size. However, this approach increases the signature size, as each signature must include the corresponding OTS public key and its authentication path to the root of the tree [48].

Winternitz One-Time Signature scheme (WOTS) is an efficient hash-based signature scheme that optimizes signature size by reducing key expansion overhead, using iterative hashing in Winternitz chains [48]. When used in a MMS scheme, the public key can be derived from the signature and message, removing the need for explicit inclusion. This makes WOTS ideal for hierarchical hash-based schemes like XMSS and LMS, both standardized by NIST [16,38,47]. WOTS$^+$ is a variant of WOTS whose security relies on second pre-image resistance and pseudorandomness, rather than on collision resistance. It is employed in SPHINCS$^+$ and built using *tweakable hash functions* to mitigate multi-target attacks, ultimately achieving a smaller signature size compared to WOTS [36,39].

Group Signature Schemes (GSS) and Their Variants: GSS, introduced in [21], enable users of a group to sign messages anonymously on behalf of the group, and their signatures can be verified by the group public key. If necessary, a designated authority, such as a group manager or a tracing manager, can reveal the signer's identity to ensure accountability, a process known as signature opening (or tracing) [5,12]. This mechanism ensures both privacy and account-

ability, making GSS a fundamental tool for privacy-sensitive applications such as anonymous credential systems [19], Direct Anonymous Attestation (DAA) [15], anonymous reputation systems [27], and Enhanced Privacy ID (EPID) [11,52]. Signatures with efficient protocols (SEP) are also closely related to group signatures, as they enable privacy-preserving authentication [2]. In SEP-based anonymous-credential presentations, unlinkability can be achieved, but there is no built-in opening authority; limited traceability can be added for revocation using techniques similar to those in DAA and EPID.

GSS can be classified into three categories based on membership dynamics: (i) *Static* GSS that the group membership is determined at the setup, providing simplicity but limited flexibility, (ii) *Partially dynamic* GSS that either allow new users to join post-setup or revoke existing members, and (iii) *Fully dynamic* GSS that allows both joins and revocations, making them well-suited for practical applications, albeit with more complex membership management.

1.1 Hash-based group signature schemes

Group signatures imply public-key encryption [28]. A sketch of the reduction: in the group signature security game, the signing keys of all group members are revealed. To make public key encryption from this, one encrypts a message $m \in \{1, \ldots, N\}$ by using the message m as the index of a group member, selecting any arbitrary message m', and signing m' with the signing key of member m. Decryption is done with the tracing algorithm, since the message is encoded as the signer. Indistinguishability follows from the anonymity of the scheme.

Since one-way functions are insufficient to construct public-key encryption [40], this implies our goal of a hash-based group signature scheme is impossible. Thus, we need to modify the definition of a group signature from [28].

The simplest option is to not give the user's signing keys to the group signature adversary. In this model, we can use a generic construction: the group manager will give a signature σ_i to each member i. To sign a message, a user will create a zero-knowledge proof that they possess a valid signature from the group manager. Applying the Fiat-Shamir transform with a message incorporated into the randomness, such a zero-knowledge proof becomes a signature itself.

To trace users, the tracer is given the secret keys of all the users. If the zero-knowledge proof can be made deterministic by using some secret seed (and can be proven to be produced in this way), the tracer can deanonymize signatures: given a signature, the tracer will iterate over each user in the scheme, and the tracer will try to recreate the signature using that user's secret key; when they succeed, they know that was the user who created the input signature.

The scheme described above was first proposed as SPHINCS-in-the-Head (SITH) [22], which used SPHINCS$^+$ [7] as the underlying signature scheme, MPC-in-the-head for the zero-knowledge proofs [41], and careful use of randomness to ensure traceability.

The appeal of SITH is its conservative security assumptions and its ability for each user to produce arbitrarily many signatures. However, it has large signature sizes, long verification times, and a long brute-force signature opening algorithm.

We also note, as a point of curiosity, that MPC-in-the-head zero knowledge proofs treat the underlying hash functions as *circuits*, not black-box primitives, so the no-go theorems need not apply to this construction.

Another approach in this security model is to give one-time tokens to the users. The tokens are designed to be easily de-anonymizable by the tracer (say, a randomized symmetric encryption of the user ID under a key known only to the tracer) and also randomized, so that they cannot be correlated between signatures. The obvious drawback is the need for each user to store a token for each signature they will need to produce, and to interact with the group manager to obtain a new batch of tokens when they run out.

The benefit of such schemes is signing efficiency. *G-Merkle* (GM) [3] was the first such proposal. Users supply public keys from a one-time-signature (OTS) scheme, and the group manager shuffles these keys and one-time tokens into leaves of a Merkle tree. The root of the Merkle tree is the group public key, and each user receives an authentication path for their public keys. A group signature will include a signature of the actual message with the OTS, then an authentication path for that OTS public key and the one-time token. A follow-up work, GM^{MT} [54] is partially dynamic, using multiple layers of Merkle trees to increase the total number of signatures without impractical initialization times.

GM is static: users cannot be added or removed. Dynamic GM (DGM) [17] is a fully dynamic GSS, but places significant burdens on the group manager: they must stay online for an *interactive* verification process, and they must store secret data that scales linearly with the number of signatures (2^{64} signatures would require $10^{8.7}$ terabytes of storage [55]). In a follow-up work, DGMT [32] removes interactive verification and sharply reduces storage, but the setup time is proportional to the maximum number of messages that can *ever* be signed.

1.2 Our Contributions

We see two main issues with previous approaches. The first is the clear impracticalities of previous schemes: either they do not support enough users or do not support them dynamically [3,32,54], they require interactivity [17], or the signatures are too big (e.g., 571 kB for 2^{10} users with SITH [22]).

The second issue is more subtle: modifying the security game to remove the adversary's access from user's secret keys makes a less powerful model. Specifically, it provides no security guarantees in case of user compromise: if a user is ever compromised by an adversary, the adversary may be able to retroactively de-anonymize all previous signatures. Indeed, SITH would be completely vulnerable against such an attack. We would instead prefer something akin to "forward anonymity": once a signature is produced, the user deletes enough secret material that the signature is untraceable to anyone who compromises the user after that point in time.

Given these requirements, our construction is simple to describe. The group manager will produce a signature keypair $(\mathsf{sk}_{GM}, \mathsf{pk}_{GM})$, and pk_{GM} will be the group signature's public key. A user i will generate n one-time signature keypairs $\{(\mathsf{sk}_{ij}^{OTS}, \mathsf{pk}_{ij}^{OTS})\}_{j=1}^{n}$, and send the public keys to the group manager. For each

public key, the group manager will generate a one-time traceability token t_{ij}, sign both the public key and token, and return the signature σ_{ij} and the token t_{ij} to the user.

To sign a message at a later point, the user will pick one of its OTS signing keys sk_{ij}^{OTS} to sign the message, producing a signature σ_{ij}^{OTS}. They will append the token t_{ij}, the public key pk_{ij}^{OTS}, and the group manager's signature σ_{ij}, and $(\sigma_{ij}^{OTS}, \mathsf{pk}_{ij}^{OTS}, t_{ij}, \sigma_{ij})$ will be the group signature of the message. At this point the user deletes sk_{ij}^{OTS}, pk_{ij}^{OTS}, t_{ij}, and σ_{ij} from its local memory.

To verify, a verifier will check that the σ_{ij}^{OTS} is a valid signature of the message with respect to the public key pk_{ij}^{OTS}, and then check that σ_{ij} is a valid signature for $(\mathsf{pk}_{ij}^{OTS}, t_{ij})$ with respect to the group public key pk_{GM}.

For a traceability token, the group manager will maintain a symmetric tracing key k_t. To make a token, they will select a random value r and encrypt the user's ID i and r under k_t. Tracing a signature is trivial: the group manager will decrypt the token t_{ij}, and this will contain the user's ID. Note here that we treat the group manager and tracer as the same entity. We do this because the group manager is always able to trace users by simply storing a record of all one-time public keys that they signed; however, in practice, the group manager could give the tracing key to another entity, who would be able to trace signatures but not issue them.

This nearly describes our scheme, DGSP (**D**ynamic **G**roup **S**ignature from S**P**HINCS$^+$). There are two subtleties to consider: first, we would also like a "Judge" algorithm, so that the group manager can convince a third party that a signature truly belonged to a particular user; second, precisely how does the user must maintain and delete their randomness to ensure forward anonymity? In the full description of our protocol, we describe carefully how we use randomness and one-way functions to provide both of these properties. Our main contributions are as follows:

i. **Scalability:** DGSP offers exceptional scalability, supporting up to 2^{60} users, matching SITH's claimed scalability and far exceeding DGMT. This advantage arises from its foundation on SPHINCS$^+$.
ii. **Setup Efficiency and Lightweight Design:** Unlike DGMT, which incurs significant setup costs and needs leaf node permutation for anonymity, DGSP leverages the stateless nature of SPHINCS$^+$ to enable efficient setup and random leaf selection. This lightweight design avoids heavy computations and leads to better overall performance while preserving anonymity.
iii. **Efficient Signing, Verification, and Opening:** DGSP achieves exceptional efficiency in signing, verification, and opening, significantly outperforming SITH in speed and resource usage. As shown in Table 8, DGSP's signing and verification are thousands of times faster than SITH, while its signing time is comparable to DGMT but with 668 times faster verification. Unlike SITH's brute-force opening, which is totally infeasible for large-scale systems, DGSP completes opening in under a millisecond.

iv. **Compact and Constant-size Signatures:** DGSP achieves a balance between efficiency and scalability with compact signature sizes, roughly 5× larger than DGMT but nearly 100× smaller than SITH. Unlike SITH, DGSP maintains a constant signature size regardless of group size, ensuring stable performance as the number of users grows.

v. **Trust Level in the Manager**: A key difference among SITH, DGSP, and DGMT is their managerial structure: SITH splits roles among an issuer, a tracer, and a revoker, while DGSP and DGMT rely on a single trusted manager. This distribution in SITH allows adversaries in security games to corrupt some entities, while in DGSP and DGMT, the manager cannot be corrupted. Unlike DGMT, DGSP reduces trust dependency by letting users generate their own key pairs and verify certificates independently. Additionally, like SITH, DGSP employs a *judge* algorithm for public verification of signature attribution. These features enhance managerial accountability, making DGSP considerably more robust than DGMT.

One limitation of our scheme is that it is stateful: to issue a signature, a user must have previously obtained a certificate from the manager. While this may be unsuitable for some applications, in other applications (e.g. a digital currency) it is useful to precisely control how many signatures a user can issue. Moreover, this automatically makes the scheme dynamic: to revoke a user, the manager can simply stop issuing new certificates to that user, and publish any "outstanding" public keys in a revocation list.

1.3 DGSP and Other Post-Quantum GSSs

Table 1 compares the scalability and signature sizes of the most efficient known post-quantum GSSs. Among existing group signature schemes with *non-interactive* verification, only schemes in [22,46] claim to support 2^{60} users; however, [46] is static, and the opening algorithm in [22] is highly inefficient at that scale, making it impractical. All schemes except [22] and [32] are static. Although [32] achieves the shortest signature size among dynamic schemes, it cannot support 2^{60} users and its setup time and algorithms are overall less efficient than those of DGSP (see Section 6 for more details).

Organization. Section 2 briefly outlines the preliminaries, and Section 3 defines a security model for stateful fully dynamic symmetric-key based GSS. Section 4 presents the DGSP construction, with its implementation and a comparison to DGMT and SITH in Sections 5 and 6. Appendices A and B provide extended preliminaries and the security proof, respectively.

Notations. Let $\mathbb{N}$ be the set of positive integers and $\lambda, w \in \mathbb{N}$ be the security parameter and Winternitz parameter. For two given integers $a, b \in \mathbb{N}$, we denote the representation of a in base b by $(a)_b$. For a given list L, we denote the n-th element of this list by $L[n]$. For two given strings x and y, we denote their concatenation by $x \parallel y$. If S is a set or a list, we denote its size by $|S|$ and the operation of randomly selecting an element x from S by $x \xleftarrow{\$} S$. We write $x \leftarrow A(a)$ for an algorithm A that runs on input a and outputs x. We call

Table 1. The signature sizes of the GSSs [9,22,29,31] depend on the number of users (group size), denoted by $N \in \{2^4, 2^5, 2^{10}, 2^{12}, 2^{15}, 2^{20}, 2^{21}, 2^{24}, 2^{60}\}$. All signature sizes are given in kilobytes (KB). Several rows are adapted from Table 1 of [32].

GSS	Signature Type	Based on	Bit Security	N								
				2^4	2^5	2^{10}	2^{12}	2^{15}	2^{20}	2^{21}	2^{24}	2^{60}
ELLNW [31]	Static	Code	80	157	–	–	205	–	–	–	200704	–
LNPS [46]	Static	Lattice	–	203								
BDKLP [9]	Static	Lattice	–	–	126	129	–	–	–	134	–	–
ESZ [29]	Static	Lattice	–	–	12	19	–	–	–	–	–	–
BDKLP [9]	Static	Isogeny	128	–	6	9	–	–	–	15.5	–	–
SITH [22]	Fully dynamic	Symmetric	128	–	–	571	–	–	851	–	–	2000
DGMT [32]	Fully dynamic	Symmetric	128	5.75					–	–	–	–
DGSP **[this work]**	Fully dynamic	Symmetric	128	8.27*								
			192	17.10*								
			256	31.27**								

– scheme either cannot accommodate N users or no information has been reported in their papers.
* for 128-bit and 192-bit security, we consider SPHINCS$^+$-SHAKE-128s-simple and SPHINCS$^+$-SHAKE-192s-simple, respectively, which have $(h, d, k, t) = (63, 7, 14, 2^{12})$ and $(h, d, k, t) = (63, 7, 17, 2^{14})$. See Section 4 for details on the signature size of DGSP.
** for 256-bit security, we consider SPHINCS$^+$-SHAKE-256s-simple. See Table 6.

a function $\epsilon(\cdot) : \mathbb{N} \to \mathbb{R}^+$ negligible in the security parameter λ if for every polynomial $p(\cdot)$ and all sufficiently large values of λ, $\epsilon(\lambda) < \frac{1}{p(\lambda)}$ holds. For a given adversary $\mathcal{A}$ and $r \in \mathbb{N}$, $\mathcal{A}^{f_1,\cdots,f_r}$ indicates that $\mathcal{A}$ is given access to the oracles $f_1, \cdots, f_r$. For $x \in \mathbb{N}$, $\log x$ denotes the base-2 logarithm of x.

2 Preliminaries

Let $m(x)$ and $\ell(x)$ be two polynomials. Our protocol will be built from standard cryptographic building blocks:

- **Hash functions**: A function $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ that maps arbitrary length messages to a fixed-length message digest efficiently.
- **Pseudo-random functions** (PRFs): A keyed function $F : \{0,1\}^\lambda \times \{0,1\}^{m(\lambda)} \to \{0,1\}^{\ell(\lambda)}$, intended to be indistinguishable from a truly random function if the key is unknown. In this work, we will assume standard hash functions are PRFs if the key is added as a prefix or suffix to the hash function's input. This is well known to be false for Merkle-Dåmgard hash functions like SHA2 (via length extensions, e.g.) but is true for sponge-based constructions like SHA3 via the "flat sponge claim" [8]. We only use sponge constructions (SHA3 and SHAKE) in our implementation.
- **Strong Pseudo-random permutations** (SPRPs): A keyed function $\mathsf{E} : \{0,1\}^\lambda \times \{0,1\}^{m(\lambda)} \to \{0,1\}^{m(\lambda)}$ such that for a fixed key k, it is a permutation and indistinguishable from a random permutation, given forward and inverse oracle access. In this work, we use standard block ciphers as SPRPs.
- **Digital Signatures**: A triple of algorithms $(\mathsf{Keygen}, \mathsf{Sign}, \mathsf{Verify})$ that generate keys, sign messages, and verify signatures, respectively. In this work, we will exclusively use hash-based signatures.

Since these are standard notions, we defer formal definitions and security properties to the Appendix A.

Keyed second preimage resistant hash function. We require a new security notion for hash functions, which we call *keyed second preimage resistance* (kSPR). As we will show in Appendix B, this property is at least as hard as the *interleaved target subset resilience* [7]. This borrows heavily from the techniques of SPHINCS$^+$, since our goal is also to avoid relying on collision resistance. Thus, we consider a hash function, where every time an adversary queries the function, the challenger prepends a random salt to the input. The adversary must then find a second preimage for one of the outputs, using the same salt. Intuitively, the random salt should make each hash look like the output of a new, randomly selected hash function, forcing the adversary to break second preimage resistance for one of these functions. The formal definition is as follows:

Let $\mathsf{H} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ be a keyed function. We will say it is *keyed second preimage resistant* if, for any *probabilistic polynomial-time (PPT)* adversary $\mathcal{A}$ that makes at most Q queries,

$$\mathbf{Adv}^{\mathsf{kSPR}}_{\mathsf{H},\mathcal{A}}(\lambda,\mathsf{Q}) = \left|\Pr\left[\mathbf{Exp}^{\mathsf{kSPR}}_{\mathsf{H},\mathcal{A}}(\lambda,\mathsf{Q}) = 1\right]\right|$$

is negligible in λ, where $\mathbf{Exp}^{\mathsf{kSPR}}_{\mathsf{H},\mathcal{A}}$ is defined in Table 2. Also, we define

$$\mathbf{Adv}^{\mathsf{kSPR}}_{\mathsf{H}}(\lambda,\mathsf{Q}) = \max_{\mathcal{A}\in\mathsf{A}_\mathsf{Q}}\left\{\mathbf{Adv}^{\mathsf{kSPR}}_{\mathsf{H},\mathcal{A}}(\lambda,\mathsf{Q})\right\},$$

where A_Q is the set of all possible PPT adversaries that make almost Q queries.

Table 2. Experiment $\mathbf{Exp}^{\mathsf{kSPR}}_{\mathsf{H},\mathcal{A}}(\lambda,\mathsf{Q})$

$\mathbf{Exp}^{\mathsf{kSPR}}_{\mathsf{H},\mathcal{A}}$	$\mathsf{Eval}(m)$
$\mathcal{Q} \leftarrow \emptyset$;	$k \xleftarrow{\$} \{0,1\}^\lambda$;
$(k', m') \leftarrow \mathcal{A}^{\mathsf{Eval}}()$;	add (k, m) to $\mathcal{Q}$;
return 1 iff $(k', m) \in \mathcal{Q}$ for some $m \neq m'$ and $\lvert\mathcal{Q}\rvert \leq \mathsf{Q}$	return $(k, \mathsf{H}(k, m))$
and $\mathsf{H}(k', m) = \mathsf{H}(k', m')$, but $(k', m') \notin \mathcal{Q}$;	

2.1 Signature Schemes

DGSP will rely on two hash-based signature schemes, SPHINCS$^+$ and WOTS$^+$. These have some of the most conservative assumptions of any post-quantum signature scheme, relying mainly on second preimage resistance of well-studied hash functions. We briefly review them; see Appendix A for details.

SPHINCS$^+$*:* For our long-term signature, we use SPHINCS$^+$ [7], a stateless hash-based signature scheme standardized by NIST as FIPS 205 [51]. Internally, SPHINCS$^+$ uses FORS, a few-time hash-based signature scheme at the bottom of the construction. Each FORS instance is a forest of small Merkle trees generated on demand whose roots are hashed into a single FORS root, which is then signed

by the lowest WOTS$^+$ layer. For DGSP, we will not rely on any internals of SPHINCS$^+$.

SPHINCS$^+$ relies on tweakable hash functions which are single-function multi-target-collision resistant for distinct tweaks, tweakable hash functions which are single-function multi-target decisional second preimage resistant for distinct tweaks, psuedorandom functions, and an interleaved target subset resilient function. We refer to [7,51] for precise definitions of these notions, and only remark that these are likely satisfied by standardized hash functions, and our goal is to rely on assumptions at least as conservative as these.

More precisely, SPHINCS$^+$ can be instantiated with either SHA2 or SHAKE, supports three security levels (128, 192, and 256 bits), and offers two optimization targets, *fast* and *small*, under the simple construction. The 256-bit SHAKE variants with fast and small optimizations are denoted "SPHINCS$^+$-SHAKE-256f-simple" and "SPHINCS$^+$-SHAKE-256s-simple".

WOTS$^+$: WOTS$^+$ is a one-time hash-based signature used internally in SPHINCS$^+$, proven to be secure when instantiated with a second preimage resistant, undetectable function family [36].

For consistency we follow the description of WOTS$^+$ from SPHINCS$^+$, which differs from a generic digital signature scheme in the following ways:

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{WOTS}^+.\mathsf{Keygen}(\mathsf{SK.seed}, \mathsf{PK.seed}, \mathsf{ADRS})$ generates the WOTS$^+$ keys deterministically. In SPHINCS$^+$, SK.seed and PK.seed are the seeds of the full SPHINCS$^+$ scheme, so ADRS is necessary to ensure each WOTS$^+$ instance has different keys. For DGSP, SK.seed and PK.seed will be chosen randomly for each instance of WOTS$^+$, so we will not need ADRS and will always pass it a fixed empty input _. We will need to reference PRF directly in our security proofs.
- $\sigma^W \leftarrow \mathsf{WOTS}^+.\mathsf{Sign}(\mathsf{sk}, \mathrm{msg}, \mathsf{PK.seed}, \mathsf{ADRS})$ uses PK.seed and sk directly, to avoid re-computing the PRF during signing.
- $0/1 \leftarrow \mathsf{WOTS}^+.\mathsf{Verify}(\mathrm{msg}, \sigma^{\mathrm{W}}, \mathsf{PK.seed}, \mathsf{ADRS}, \mathsf{pk})$ verifies signatures.
- $\mathsf{pk}' \leftarrow \mathsf{WOTS}^+.\mathsf{PKRegen}(\mathrm{msg}, \sigma^{\mathrm{W}}, \mathsf{PK.seed}, \mathsf{ADRS})$ computes a candidate public key of a signature from the signature and the message. If $(\mathrm{msg}, \sigma^{\mathrm{W}})$ is a valid pair, the computed public key pk' is equal to the real public key pk. Hence schemes using WOTS$^+$ in a Merkle tree (e.g., SPHINCS$^+$, DGSP) omit pk to save space: an invalid $(\mathrm{msg}, \sigma^{\mathrm{W}})$ yields $\mathsf{pk}' \neq \mathsf{pk}$, which propagates up the path, causes a root mismatch, and verification fails.

3 Fully Dynamic Group Signature

In this section, we present a model for fully dynamic stateful group signature schemes, denoted by $\mathcal{SFDGS}$. Following this, we establish a security model tailored to $\mathcal{SFDGS}$. While our proposed model shares some algorithmic consistency with the model introduced in [32], it differs in its joining and opening algorithms and includes an additional algorithm SFDGS.Judge, introduced in [4], allowing public verification of signature attribution. The proposed model has three types

of entities: (i) A *Trusted manager*, denoted by $\mathcal{M}$, who serves as the central authority ensuring perfect functioning of the group. This manager allows new users to join the group and generates certificates for them, can reveal the identity of a signer and provide proof of the authenticity of its opening, and can revoke the signing ability of misbehaved users and their generated signatures. (ii) *Users*, who individually sign messages anonymously on behalf of the group. (iii) *Verifier*, who confirms the validity of a group signature using the public parameters. Let λ be the security parameter and $\mathsf{SFDGS.SetPr}$ be the *setup parameters*. Our proposed $\mathcal{SFDGS}$ consists of the following PPT algorithms.

- $(\mathsf{SFDGS.SK}, \mathsf{SFDGS.PP}) \leftarrow \mathsf{SFDGS.Kgen}(1^\lambda, \mathsf{SFDGS.SetPr})$: The manager runs this algorithm to generate the *manager's secret key* $\mathsf{SFDGS.SK}$ and the *public parameters* $\mathsf{SFDGS.PP}$, including the *group public key* gpk and other essential public information for signature verification.
- $(\mathsf{id}, \mathsf{state}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{state}_{\mathsf{id}}^{\mathcal{U}})/\perp \leftarrow \mathsf{SFDGS.Join}(\mathsf{Username})$: This is an interactive joining protocol between a user with identity $\mathsf{Username}$, and the manager. Following the model in [5], all communication occurs over secure (i.e., private and authenticated) channels, with the user initiating the protocol by sending their identity $\mathsf{Username}$. Upon receiving this request, the manager who is willing to enroll users, checks the *manager's state* $\mathsf{StatesM}$; if the user is not already a member, it proceeds as follows. (i) It selects an unassigned *identifier* id and sends the pair $(\mathsf{id}, \mathsf{state}_{\mathsf{id}}^{\mathcal{U}})$ to the user, where $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$ contains the information the user needs for future interactions with the manager and signing messages. (ii) It assigns a specific $\mathsf{state}_{\mathsf{id}}^{\mathcal{M}}$, representing the essential information about user id, and stores the pair $(\mathsf{state}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{Username})$ in $\mathsf{StatesM}[\mathsf{id}]$, the id-th entry of $\mathsf{StatesM}$.
- $(\sigma, \mathsf{state}_{\mathsf{id}}^{\mathcal{U}}) \leftarrow \mathsf{SFDGS.Sign}(\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}, \mathrm{msg})$: This algorithm, run by a user, takes the state $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$ of user id along with a message msg and outputs a group signature σ and the updated form of $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$.
  In a stateful hash-based group signature scheme, the user's state $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$ may become invalid at certain points, for example, in DGSP, when the user runs out of one-time keys and needs to request new ones. In such cases, user id must interact with the manager over a secure channel to update $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$. This interaction is specified with the following three functions.
  - $(\mathrm{msg}_{\mathsf{id}}^{\mathcal{U}}, \mathsf{state}_{\mathsf{id}}^{\mathcal{U}}) \leftarrow \mathsf{SFDGS.RequestU}(\mathsf{id}, \mathsf{state}_{\mathsf{id}}^{\mathcal{U}})$: Outputs a state update request for a user.
  - $(\mathrm{msg}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{state}_{\mathsf{id}}^{\mathcal{M}}) \leftarrow \mathsf{SFDGS.ResponseM}(\mathrm{msg}_{\mathsf{id}}^{\mathcal{U}}, \mathsf{state}_{\mathsf{id}}^{\mathcal{M}})$: Run by the manager and returns information necessary for a user to update their state, in response to a request by the user. The manager then updates $\mathsf{StatesM}$ by replacing the first component of $\mathsf{StatesM}[\mathsf{id}]$ with $\mathsf{state}_{\mathsf{id}}^{\mathcal{M}}$, i.e. sets $\mathsf{StatesM}[\mathsf{id}] = (\mathsf{state}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{Username})$.
  - $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}} \leftarrow \mathsf{SFDGS.UpdateU}(\mathrm{msg}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{state}_{\mathsf{id}}^{\mathcal{U}})$: The user receives information from the manager and uses it to update their state.
- $0/1 \leftarrow \mathsf{SFDGS.Verify}(\mathrm{msg}, \sigma, \mathsf{SFDGS.PP})$: This deterministic verification algorithm, run by a verifier, takes as input a message msg, a group signature σ, and the public parameters $\mathsf{SFDGS.PP}$. It outputs 1 if σ is a valid group signature on the message msg with respect to $\mathsf{SFDGS.PP}$, and 0 otherwise.

– $(\mathsf{StatesM}, \mathsf{SFDGS.PP}) \leftarrow \mathsf{SFDGS.Revoke}(\mathsf{SFDGS.SK}, \mathsf{StatesM}, \mathsf{SFDGS.PP}, \mathsf{R})$: This algorithm, run by the manager, revokes users with identifiers in a set R and invalidates their signatures. It takes as input the manager's secret key $\mathsf{SFDGS.SK}$ and state $\mathsf{StatesM}$, the public parameters $\mathsf{SFDGS.PP}$, and the set R of users to be revoked. It updates $\mathsf{StatesM}$ and $\mathsf{SFDGS.PP}$ such that the manager will no longer respond to interactions from revoked users, and $\mathsf{SFDGS.PP}$ will reject any signatures produced by them.
– $(\mathsf{id}, \pi)/\perp \leftarrow \mathsf{SFDGS.Open}(\mathrm{msg}, \sigma, \mathsf{SFDGS.SK})$: This deterministic opening algorithm, run by the manager, takes a message msg and a valid group signature σ on msg along with the manager's secret key $\mathsf{SFDGS.SK}$, and outputs either an identifier-proof pair (id, π), identifying the signer with a proof of attributing σ to the identifier id, or $\perp$ if no attribution is possible.
– $0/1 \leftarrow \mathsf{SFDGS.Judge}((\mathsf{id}, \pi), \mathrm{msg}, \sigma)$: This deterministic algorithm, executable by anyone, takes the identifier-proof pair (id, π), a message msg, and its signature σ to determine whether the signature is correctly attributed to the signer (outputting 1) or not.

Proposed Security Model. The security model proposed in our work for the fully dynamic stateful GSS follows the foundational models established in [5,12,17,18,32]. We specifically adopt the notion of anonymity as introduced in [18], where the adversary is restricted from corrupting the manager during the course of the anonymity game. In this model, we prove that DGSP satisfies four essential properties: *Correctness*, *Unforgeability*, *Anonymity*, and *Traceability*. Each of these properties is defined using the game, where each game is defined between a PPT adversary $\mathcal{A}$ and a *Challenger*. Each of these properties is quantified by the success probability of $\mathcal{A}$ in its game. In these games, the adversary has access to various oracles, each specifying their capabilities, which can be invoked a polynomially many times during the game.

In the following, we first describe the oracles in Table 3, and then in Definition 1 formally define the security properties of the signature scheme. The corresponding security experiments are given in Table 4. The oracles use the following mutable global variables: $\mathcal{H}$, $\mathcal{C}$, and $\mathcal{R}$ represent honest, corrupted, and revoked users, respectively; N, CL and SL refer to the number of registered users, the *challenge list* and *signing list*. The oracles also access two state dictionaries, $\mathsf{StatesU}$ (user states) and $\mathsf{StatesM}$ (manager states), where $\mathsf{StatesM}[\mathsf{id}]$ stores the pair $(\mathsf{state}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{Username})$ and $\mathsf{StatesU}[\mathsf{id}]$ stores $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$, with $\mathsf{id} \leq N$.

– $\mathsf{AddHU}(\mathsf{Username})$: This oracle simulates the join protocol $\mathsf{SFDGS.Join}$ between a user with identity $\mathsf{Username}$ and the manager. If $\mathsf{Username}$ already exists in $\mathsf{StatesM}$, the oracle returns $\perp$; otherwise it returns an unassigned id to the adversary $\mathcal{A}$, stores $(\mathsf{state}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{Username})$ in $\mathsf{StatesM}$ and adds id to $\mathcal{H}$. Clearly, $\mathcal{A}$ does not learn $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$.
– $\mathsf{CorruptHU}(\mathsf{id})$: This oracle allows the adversary $\mathcal{A}$ to corrupt an honest user with identifier id. It removes id from $\mathcal{H}$ and adds it to $\mathcal{C}$. It gives the adversary $\mathcal{A}$ access to $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$. Thus, $\mathcal{A}$ is able to sign messages and interact with the manager and update $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$.

– $\mathsf{RespM}(\mathsf{id}, \mathrm{msg})$: This oracle allows the adversary to send a state update request to the group manager, mainly to update the state of corrupted users.
– $\mathsf{UpdateUser}(\mathsf{id})$: This is a helper function that steps through the interactive update protocol for an honest user and the group manager. As we will describe in the conclusion, our scheme is vulnerable to traffic analysis, so we forbid such attacks in our model by completing this state update interaction **without the adversary's knowledge**.
– $\mathsf{Revoke}(\mathsf{R})$: This oracle lets the adversary $\mathcal{A}$ revoke users in R by updating the manager's state $\mathsf{StatesM}$ and public parameters $\mathsf{SFDGS.PP}$. Specifically, for each $\mathsf{id} \in \mathsf{R}$ the manager updates $\mathsf{state}_{\mathsf{id}}^{\mathcal{M}}$ so that it will no longer respond to the requests of that user. In addition, it removes them from $\mathcal{H}$ and invalidates their previously generated signatures by updating $\mathsf{SFDGS.PP}$.
– $\mathsf{SignHU}(\mathsf{id}, \mathrm{msg})$: This oracle takes an identifier id and a message msg. If $\mathsf{id} \notin \mathcal{H}$, it returns $\perp$; otherwise computes $(\sigma, \mathsf{state}_{\mathsf{id}}^{\mathcal{U}}) \leftarrow \mathsf{SFDGS.Sign}(\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}, \mathrm{msg})$ and returns the signature σ. It also records the pair (msg, σ) in the signing list SL to prevent reuse by the adversary $\mathcal{A}$ in the unforgeability experiment. Finally, the oracle checks whether the updated $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$ remains valid. If it is no longer valid, it updates $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$ by calling $\mathsf{UpdateUser}(\mathsf{id})$.
– $\mathsf{Open}(\mathrm{msg}, \sigma)$: This oracle takes a message-signature pair (msg, σ). If $(\mathrm{msg}, \sigma) \in \mathsf{CL}$ or $\mathsf{SFDGS.Verify}(\mathrm{msg}, \sigma, \mathsf{SFDGS.PP}) = 0$, it returns $(\perp, \text{Invalid Input})$. Otherwise, it returns $\mathsf{SFDGS.Open}(\mathrm{msg}, \sigma, \mathsf{SFDGS.SK})$, which is either a pair (id, π), or $\perp$ if σ cannot be attributed to a user.
– $\mathsf{Ch}_b(\mathsf{id}_0, \mathsf{id}_1, \mathrm{msg})$: This oracle takes the identifiers id_0 and id_1 of two honest users with valid states along with a message msg. The oracle outputs the signature σ_b obtained from $(\sigma_b, \mathsf{state}_{\mathsf{id}_b}^{\mathcal{U}}) \leftarrow \mathsf{SFDGS.Sign}(\mathsf{state}_{\mathsf{id}_b}^{\mathcal{U}}, \mathrm{msg})$. It keeps $(\mathrm{msg}, \sigma_{\mathrm{b}})$ in the challenge list CL to prevent $\mathcal{A}$ from calling the *opening* oracle $\mathsf{Open}(\cdot, \cdot)$ on $(\mathrm{msg}, \sigma_{\mathrm{b}})$ in the Anonymity experiment. This oracle also checks whether the updated $\mathsf{state}_{\mathsf{id}_b}^{\mathcal{U}}$ remains valid. If it is no longer valid, the oracle updates it by running $\mathsf{UpdateUser}(\mathsf{id}_b)$. Importantly for our stateful scheme, both users id_0 and id_1 sign the message (to avoid de-anonymization by the number of valid signatures remaining), but the challenger only returns one signature to the adversary.

We deliberately do not give the adversary an oracle to send arbitrary messages to the users on behalf of the manager (e.g., malformed certificates), because we assume the users and manager communicate through an encrypted and authenticated channel.

The Correctness experiment is modeled as an adversarial game, where the adversary $\mathcal{A}$ corrupts a set of users, possibly all except one. This property ensures that even in such a scenario, the honest user can enroll successfully, generate valid signatures, and be correctly identified by the opening algorithm with a verifiable proof. The Unforgeability property ensures that even if an adversary corrupts all but one user, they cannot produce a valid signature with judge-accepted proof attributing the signature to an honest user unless that user actually produced it. The Anonymity property is defined through an indistinguishability experiment between the adversary and the challenger. The adversary knows all the

users' secret keys except for two, each belonging to a different user. This property ensures that the adversary has only a negligible advantage in distinguishing which of the two users generated a given signature. Notice that in the anonymity game, the adversary can corrupt users at any time (both before and after the challenge), and we only require that the challenge users were uncorrupted when they produced the challenge signature; the adversary can corrupt them afterwards. This formalizes our goal of "forward anonymity". The adversary is also

Table 3: Oracles.

$\mathsf{AddHU}(\mathsf{Username})$

if $(_, \mathsf{Username}) \in \mathsf{StatesM}$, then return $\perp$;
$N = N + 1$; $(\mathsf{id}, \mathsf{state}^{\mathcal{M}}_{\mathsf{id}}, \mathsf{state}^{\mathcal{U}}_{\mathsf{id}}) \leftarrow \mathsf{SFDGS.Join}(\mathsf{Username})$;
$\mathsf{StatesM}[\mathsf{id}] = (\mathsf{state}^{\mathcal{M}}_{\mathsf{id}}, \mathsf{Username})$; $\mathsf{StatesU}[\mathsf{id}] = \mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$; $\mathcal{H} = \mathcal{H} \cup \{\mathsf{id}\}$;
return id;

$\mathsf{CorruptHU}(\mathsf{id})$

if $\mathsf{id} \notin \mathcal{H}$, then return $\perp$;
$\mathcal{H} = \mathcal{H} \setminus \{\mathsf{id}\}$; $\mathcal{C} = \mathcal{C} \cup \{\mathsf{id}\}$; return $\mathsf{StatesU}[\mathsf{id}]$;

$\mathsf{RespM}(\mathsf{id}, \mathrm{msg})$

$(\mathrm{msg}^{\mathcal{M}}_{\mathsf{id}}, \mathsf{state}^{\mathcal{M}}_{\mathsf{id}}) \leftarrow \mathsf{SFDGS.ResponseM}(\mathrm{msg}, \mathsf{state}^{\mathcal{M}}_{\mathsf{id}})$;
$\mathsf{StatesM}[\mathsf{id}] = (\mathsf{state}^{\mathcal{M}}_{\mathsf{id}}, \mathsf{Username})$; return $\mathrm{msg}^{\mathcal{M}}_{\mathsf{id}}$;

$\mathsf{UpdateUser}(\mathsf{id})$

if $\mathsf{id} \notin \mathcal{H}$, then return $\perp$;
$(\mathrm{msg}^{\mathcal{U}}_{\mathsf{id}}, \mathsf{state}^{\mathcal{U}}_{\mathsf{id}}) \leftarrow \mathsf{SFDGS.RequestU}(\mathsf{id}, \mathsf{StatesU}[\mathsf{id}])$;
$(\mathrm{msg}^{\mathcal{M}}_{\mathsf{id}}, \mathsf{state}^{\mathcal{M}}_{\mathsf{id}}) \leftarrow \mathsf{SFDGS.ResponseM}(\mathrm{msg}^{\mathcal{U}}_{\mathsf{id}}, \mathsf{state}^{\mathcal{M}}_{\mathsf{id}})$;
$\mathsf{state}^{\mathcal{U}}_{\mathsf{id}} \leftarrow \mathsf{SFDGS.UpdateU}(\mathrm{msg}^{\mathcal{M}}_{\mathsf{id}}, \mathsf{state}^{\mathcal{U}}_{\mathsf{id}})$;
$\mathsf{StatesU}[\mathsf{id}] = \mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$; $\mathsf{StatesM}[\mathsf{id}] = (\mathsf{state}^{\mathcal{M}}_{\mathsf{id}}, \mathsf{Username})$;

$\mathsf{Revoke}(\mathsf{R})$

if $\mathsf{R} \not\subset \{1, \cdots, N\}$, then return $\perp$;
$\mathsf{R}' = \mathsf{R} \setminus \mathcal{R}$; $\mathcal{R} = \mathsf{R}' \cup \mathcal{R}$;
$(\mathsf{StatesM}, \mathsf{SFDGS.PP}) \leftarrow \mathsf{SFDGS.Revoke}(\mathsf{SFDGS.SK}, \mathsf{StatesM}, \mathsf{SFDGS.PP}, \mathsf{R}')$;
return $\mathsf{SFDGS.PP}$;

$\mathsf{SignHU}(\mathsf{id}, \mathrm{msg})$

if $\mathsf{id} \notin \mathcal{H}$, then return $\perp$;
$(\sigma, \mathsf{state}^{\mathcal{U}}_{\mathsf{id}}) \leftarrow \mathsf{SFDGS.Sign}(\mathsf{StatesU}[\mathsf{id}], \mathrm{msg})$; $\mathsf{SL} = \mathsf{SL} \cup \{(\mathrm{msg}, \sigma)\}$;
$\mathsf{StatesU}[\mathsf{id}] = \mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$; if $\mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$ is invalid, then $\mathsf{UpdateUser}(\mathsf{id})$;
return σ;

$\mathsf{Open}(\mathrm{msg}, \sigma)$

if $(\mathrm{msg}, \sigma) \in \mathsf{CL} \vee \mathsf{SFDGS.Verify}(\mathrm{msg}, \sigma, \mathsf{SFDGS.PP}) = 0$, then return $(\perp, \text{Invalid Input})$;
return $\mathsf{SFDGS.Open}(\mathrm{msg}, \sigma, \mathsf{SFDGS.SK})$;

$\mathsf{Ch}_b(\mathsf{id}_0, \mathsf{id}_1, \mathrm{msg})$

if $\{\mathsf{id}_0, \mathsf{id}_1\} \not\subset \mathcal{H} \ \vee \ \mathsf{state}^{\mathcal{U}}_{\mathsf{id}_0}$ is invalid $\ \vee \ \mathsf{state}^{\mathcal{U}}_{\mathsf{id}_1}$ is invalid, then return $\perp$;
$(\sigma_0, \mathsf{state}^{\mathcal{U}}_{\mathsf{id}_0}) \leftarrow \mathsf{SFDGS.Sign}(\mathsf{state}^{\mathcal{U}}_{\mathsf{id}_0}, \mathrm{msg})$; $(\sigma_1, \mathsf{state}^{\mathcal{U}}_{\mathsf{id}_1}) \leftarrow \mathsf{SFDGS.Sign}(\mathsf{state}^{\mathcal{U}}_{\mathsf{id}_1}, \mathrm{msg})$;
$\mathsf{CL} = \{(\mathrm{msg}, \sigma_b)\}$;
for $i \in \{0, 1\}$: $\mathsf{StatesU}[\mathsf{id}_i] = \mathsf{state}^{\mathcal{U}}_{\mathsf{id}_i}$; if $\mathsf{state}^{\mathcal{U}}_{\mathsf{id}_i}$ is invalid, then $\mathsf{UpdateUser}(\mathsf{id}_i)$;
return σ_b;

forbidden from revoking the two (honest) challenge users after the challenge. Finally, the Traceability property ensures that every valid signature is correctly attributed to a registered user with a valid proof, i.e. the adversary's advantage in generating a signature that is either untraceable or traceable with an invalid proof is negligible.

Definition 1. *For any security parameter $\lambda \in \mathbb{N}$ and for any PPT adversary $\mathcal{A}$, we say that an $\mathcal{SFDGS}$ provides:*

1. ***Correctness*** *if there exists a negligible function ϵ_1 such that*

$$\mathbf{Adv}^{\mathrm{Corr}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda) = \Pr\left[\mathbf{Exp}^{\mathrm{Corr}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda) = 1\right] \leq \epsilon_1(\lambda),$$

2. ***Unforgeability*** *if there exists a negligible function ϵ_2 such that*

$$\mathbf{Adv}^{\mathrm{Unforg}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda) = \Pr\left[\mathbf{Exp}^{\mathrm{Unforg}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda) = 1\right] \leq \epsilon_2(\lambda),$$

3. ***Anonymity*** *if there exists a negligible function ϵ_3 such that*

$$\mathbf{Adv}^{\mathrm{Anon}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda) = \left|\Pr\left[\mathbf{Exp}^{Anon}_{\mathcal{SFDGS},\mathcal{A}}(\lambda) = 1\right] - \frac{1}{2}\right| \leq \epsilon_3(\lambda),$$

4. ***Traceability*** *if there exists a negligible function ϵ_4 such that*

$$\mathbf{Adv}^{\mathrm{Trace}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda) = \Pr\left[\mathbf{Exp}^{\mathrm{Trace}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda) = 1\right] \leq \epsilon_4(\lambda),$$

where $\mathbf{Exp}^{\mathrm{Corr}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$, $\mathbf{Exp}^{\mathrm{Unforg}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$, $\mathbf{Exp}^{\mathrm{Anon}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$ and $\mathbf{Exp}^{\mathrm{Trace}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$ are defined in Table 4.

Table 4. Experiments

$\mathbf{Exp}^{\mathrm{Corr}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$

$(\mathsf{SFDGS.SK}, \mathsf{SFDGS.PP}) \leftarrow \mathsf{SFDGS.Kgen}(1^\lambda, \mathsf{SFDGS.SetPr})$;
$N = 0$; $\mathsf{id} = \perp$; $\mathcal{H} = \emptyset$; $\mathcal{C} = \emptyset$; $\mathcal{R} = \emptyset$; $\mathsf{StatesU} = \emptyset$; $\mathsf{StatesM} = \emptyset$
$(\mathrm{msg}, \mathsf{id}) \leftarrow \mathcal{A}^{\mathsf{AddHU},\mathsf{CorruptHU},\mathsf{RespM},\mathsf{Revoke}}(\mathsf{SFDGS.PP})$;
if $\mathsf{id} \notin \mathcal{H}$, then **return** 0;
$(\sigma, \mathsf{state}^{\mathcal{U}}_{\mathsf{id}}) \leftarrow \mathsf{SFDGS.Sign}(\mathsf{StatesU}[\mathsf{id}], \mathrm{msg})$;
if $\mathsf{SFDGS.Verify}(\mathrm{msg}, \sigma, \mathsf{SFDGS.PP}) = 0$, then **return** 1;
if $\mathsf{SFDGS.Open}(\mathrm{msg}, \sigma, \mathsf{SFDGS.SK}) = \perp$, then **return** 1;
$(\mathsf{id}^*, \pi^*) \leftarrow \mathsf{SFDGS.Open}(\mathrm{msg}, \sigma, \mathsf{SFDGS.SK})$;
if $\mathsf{id}^* \neq \mathsf{id} \vee \mathsf{DGSP.Judge}((\mathsf{id}^*, \pi^*), \mathrm{msg}, \sigma) = 0$, then **return** 1;
return 0;

$\mathbf{Exp}^{\text{Unforg}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$

$(\mathsf{SFDGS.SK}, \mathsf{SFDGS.PP}) \leftarrow \mathsf{SFDGS.Kgen}(1^\lambda, \mathsf{SFDGS.SetPr})$;
$N = 0$; $\mathcal{H} = \emptyset$; $\mathcal{C} = \emptyset$; $\mathcal{R} = \emptyset$; $\mathsf{SL} = \emptyset$; $\mathsf{StatesU} = \emptyset$; $\mathsf{StatesM} = \emptyset$
$(\text{msg}, \sigma) \leftarrow \mathcal{A}^{\mathsf{AddHU},\mathsf{CorruptHU},\mathsf{RespM},\mathsf{SignHU},\mathsf{Revoke}}(\mathsf{SFDGS.PP})$;
if $(\text{msg}, \sigma) \in \mathsf{SL} \vee \mathsf{SFDGS.Verify}(\text{msg}, \sigma, \mathsf{SFDGS.PP}) = 0$, then **return** 0;
$(\mathsf{id}, \pi) \leftarrow \mathsf{SFDGS.Open}(\text{msg}, \sigma, \mathsf{SFDGS.SK})$;
if $\mathsf{id} \in \mathcal{H} \wedge \mathsf{DGSP.Judge}((\mathsf{id}, \pi), \text{msg}, \sigma) = 1$, then **return** 1;
return 0;

$\mathbf{Exp}^{\text{Anon}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$

$(\mathsf{SFDGS.SK}, \mathsf{SFDGS.PP}) \leftarrow \mathsf{SFDGS.Kgen}(1^\lambda, \mathsf{SFDGS.SetPr})$;
$N = 0$; $\mathcal{H} = \emptyset$; $\mathcal{C} = \emptyset$; $\mathcal{R} = \emptyset$; $\mathsf{CL} = \emptyset$; $\mathsf{StatesU} = \emptyset$; $\mathsf{StatesM} = \emptyset$
$b \xleftarrow{\$} \{0, 1\}$;
$b' \leftarrow \mathcal{A}^{\mathsf{AddHU},\mathsf{CorruptHU},\mathsf{RespM},\mathsf{SignHU},\mathsf{Revoke},\mathsf{Open},\mathsf{Ch}_b}(\mathsf{SFDGS.PP})$;
return $b' = b$;

$\mathbf{Exp}^{\text{Trace}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$

$(\mathsf{SFDGS.SK}, \mathsf{SFDGS.PP}) \leftarrow \mathsf{SFDGS.Kgen}(1^\lambda, \mathsf{SFDGS.SetPr})$;
$N = 0$; $\mathcal{H} = \emptyset$; $\mathcal{C} = \emptyset$; $\mathcal{R} = \emptyset$; $\mathsf{StatesU} = \emptyset$; $\mathsf{StatesM} = \emptyset$
$(\text{msg}, \sigma) \leftarrow \mathcal{A}^{\mathsf{AddHU},\mathsf{CorruptHU},\mathsf{RespM},\mathsf{SignHU},\mathsf{Revoke}}(\mathsf{SFDGS.PP})$;
if $\mathsf{SFDGS.Verify}(\text{msg}, \sigma, \mathsf{SFDGS.PP}) = 0$, then **return** 0;
if $\mathsf{SFDGS.Open}(\sigma, \mathsf{SFDGS.SK}) = \perp$, then **return** 1;
$(\mathsf{id}, \pi) \leftarrow \mathsf{SFDGS.Open}(\sigma, \mathsf{SFDGS.SK})$
if $\mathsf{DGSP.Judge}((\mathsf{id}, \pi), \text{msg}, \sigma) = 0$, then **return** 1;
return 0;

4 DGSP: An Efficient, Scalable, Fully Dynamic Group Signature Scheme Using SPHINCS$^+$

Following the fully dynamic stateful GSS model from Section 3, we present DGSP and its polynomial-time algorithms (DGSP.KG, DGSP.Join, DGSP.Sign, DGSP.Verify, DGSP.Revoke, DGSP.Open, DGSP.Judge). DGSP builds on the stateless hash-based signature scheme SPHINCS$^+$, which uses tweakable hash functions, WOTS$^+$ one-time signatures, and FORS few-time signatures (see Appendix A). It extends SPHINCS$^+$ by adding a WOTS$^+$ *layer* below the FORS trees, allowing the manager to issue certificates while ensuring only legitimate users can sign (see Figure 1). The main symbols of DGSP are listed in Table 5. DGSP's algorithms are as follows:

Table 5. List of Symbols

$\mathsf{SK.seed}$	The secret seed for generating keys of WOTS$^+$ and FORS in SPHINCS$^+$
$\mathsf{SK.prf}$	The secret seed for generating randomness for the message digest
$\mathsf{PK.root}$	The root node of the topmost tree in the tree of SPHINCS$^+$
$\mathsf{PK.seed}$	The random public parameter for using in tweakable hash functions
PK	The public key of SPHINCS$^+$; $(\mathsf{PK.root}, \mathsf{PK.seed})$
SK	The secret key of SPHINCS$^+$; $(\mathsf{SK.seed}, \mathsf{SK.prf}, \mathsf{PK})$
$\mathsf{DGSP.msk}_1$	The manager's secret randomness used in hash function H to generate c_{id}
$\mathsf{DGSP.msk}_2$	The manager's secret key used in SPRP E to generate $\zeta_{\mathsf{id},j}$
$\mathsf{DGSP.msk}$	The manager's master secret key; $(\mathsf{DGSP.msk}_1, \mathsf{DGSP.msk}_2)$
RL	The revocation list
$\mathsf{DGSP.gpk}$	The DGSP group public key which is $\mathsf{PK.root}$
$\mathsf{DGSP.SK}$	The manager's secret key; $(\mathsf{DGSP.msk}, \mathsf{SK})$
$\mathsf{DGSP.PP}$	The DGSP public parameter; $(\mathsf{PK}, \mathsf{RL})$
$\mathsf{DGSP.SetPr}$	The DGSP setup parameter; (h, d, k, t)
id	The identifier of a user
$\mathsf{User.seed}_{\mathsf{id}}$	The secret seed of user id for generating WOTS$^+$ key pairs
$(\mathsf{sk}_{\mathsf{id},j}, \mathsf{pk}_{\mathsf{id},j})$	The j-th secret-public key pair of WOTS$^+$ of the user id
c_{id}	The secret credential of user id; $\mathsf{H}(\mathsf{DGSP.msk}_1 \| \mathsf{id})$
c^*_{id}	The secret identifier of user id for key request authentication; $\mathsf{H}(\mathsf{id} \| c_{\mathsf{id}})$
$\pi_{\mathsf{id},j}$	The proof of attributing a signature to user id; $\mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \parallel c_{\mathsf{id}})$
$\tau_{\mathsf{id},j}$	The manager's proof-binding commitment; $\mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \parallel \pi_{\mathsf{id},j} \parallel \mathsf{id})$
$\zeta_{\mathsf{id},j}$	$\mathsf{E}(\mathsf{DGSP.msk}_2, \mathsf{id} \parallel j)$
$\sigma^S_{\mathsf{id},j}$	$\mathsf{SPHINCS}^+.\mathsf{Sign}(\mathsf{SK}, \mathsf{pk}_{\mathsf{id},j} \parallel \zeta_{\mathsf{id},j} \parallel \tau_{\mathsf{id},j})$
$\mathsf{Cert}_{\mathsf{id},j}$	The manager's certification on $\mathsf{pk}_{\mathsf{id},j}$; $(\zeta_{\mathsf{id},j}, \pi_{\mathsf{id},j}, \sigma^S_{\mathsf{id},j})$
$r_{\mathsf{id},j}$	A λ-bit randomness used for generating $\rho_{\mathsf{id},j}$
$\rho_{\mathsf{id},j}$	A λ-bit randomness for generating extra WOTS$^+$ keys; $\mathsf{H}(r_{\mathsf{id},j})$
$\sigma^W_{\mathsf{id},j}$	$\mathsf{WOTS}^+.\mathsf{Sign}(\mathsf{sk}_{\mathsf{id},j}, \mathrm{msg}, \rho_{\mathsf{id,j}}, \mathsf{ADRS} = _)$
$\mathsf{ctr}^{\mathcal{M}}_{\mathsf{id}}$	The manager's counter for user id tracks the number of certificates received
$\mathsf{ctr}^{\mathcal{U}}_{\mathsf{id}}$	User id's counter for remaining certificates
$\sigma^{\mathrm{DGSP}}_{\mathsf{id},j}$	DGSP's signature; $(\sigma^W_{\mathsf{id},j}, \rho_{\mathsf{id},j}, \zeta_{\mathsf{id},j}, \sigma^S_{\mathsf{id},j}, \tau_{\mathsf{id},j})$

– $(\mathsf{DGSP.SK}, \mathsf{DGSP.PP}) \leftarrow \mathsf{DGSP.KG}(1^\lambda, \mathsf{DGSP.SetPr})$:
During the setup phase, the manager runs the key generation algorithm 1 on the security parameter λ and the setup parameters $\mathsf{DGSP.SetPr} = (h, d, k, t)$ to obtain $(\mathsf{DGSP.SK}, \mathsf{DGSP.PP})$, where $\mathsf{DGSP.SK} = (\mathsf{DGSP.msk}, \mathsf{SK})$ and $\mathsf{DGSP.PP} = (\mathsf{PK}, \mathsf{RL})$ generated as follows:
  - $\mathsf{DGSP.msk} = (\mathsf{DGSP.msk}_1, \mathsf{DGSP.msk}_2)$ is the manager's master secret key; each component is a λ-bit random string. $\mathsf{DGSP.msk}_1$ is used for generating a unique secret credentials for users and $\mathsf{DGSP.msk}_2$ is used for generating the encrypted values, denoted by ζ, that is used to (i) construct certificates, (ii) open signatures, and (iii) revoke signatures/users.
  - $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{SPHINCS}^+.\mathsf{Keygen}(1^\lambda, \mathsf{DGSP.SetPr})$ is the key generation algorithm of SPHINCS$^+$, which generates public and secret (signing) keys. The public key $\mathsf{PK} = (\mathsf{PK.root}, \mathsf{PK.seed})$ consists of two λ-bit strings, where (i) $\mathsf{PK.root}$ is the root node of the topmost tree in the hypertree SPHINCS$^+$, and (ii) $\mathsf{PK.seed} \in \mathcal{P} = \{0,1\}^\lambda$ is a public seed. The secret key is defined as $\mathsf{SK} = (\mathsf{SK.seed}, \mathsf{SK.prf}, \mathsf{PK})$, where $\mathsf{SK.seed}$ and $\mathsf{SK.prf}$ are λ-bit random seeds used to (i) generate WOTS$^+$ and FORS secret keys and (ii) randomize the message digest, respectively. In DGSP, $\mathsf{PK.root}$ is the group public key, denoted by $\mathsf{DGSP.gpk}$. See Appendix A.
  - RL is the revocation list, which is initially empty.

Algorithm 1 DGSP.KG

Input: The security parameter λ and $\mathsf{DGSP.SetPr} = (h, d, k, t)$.
Output: $(\mathsf{DGSP.SK}, \mathsf{DGSP.PP})$

1: $\mathsf{DGSP.msk}_1, \mathsf{DGSP.msk}_2 \xleftarrow{\$} \{0,1\}^\lambda$;
2: $\mathsf{DGSP.msk} = (\mathsf{DGSP.msk}_1, \mathsf{DGSP.msk}_2)$; ▷ The manager's secret key.
3: $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{SPHINCS}^+.\mathsf{Keygen}(1^\lambda, \mathsf{DGSP.SetPr})$;
4: $\mathsf{DGSP.gpk} = \mathsf{PK.root}$;
5: $\mathsf{DGSP.SK} = (\mathsf{DGSP.msk}, \mathsf{SK})$; ▷ Secret key of DGSP
6: $\mathsf{DGSP.PP} = (\mathsf{PK}, \mathsf{RL})$; ▷ Group public parameters of DGSP
7: **return** $(\mathsf{DGSP.SK}, \mathsf{DGSP.PP})$; ▷ Since $\mathsf{DGSP.gpk} = \mathsf{PK.root}$, we do not explicitly include it in $\mathsf{DGSP.PP}$

– $(\mathsf{id}, \mathsf{state}_{\mathsf{id}}^{\mathcal{U}}, \mathsf{state}_{\mathsf{id}}^{\mathcal{M}}) / \perp \leftarrow \mathsf{DGSP.Join}(\mathsf{Username})$:
Membership in DGSP is established through an interactive joining protocol, described in detail in Algorithm 2. The group manager assigns the user a unique identifier id and pseudorandomly derives a *secret credential* $c_{\mathsf{id}} = \mathsf{H}(\mathsf{DGSP.msk}_1 \parallel \mathsf{id})$, which remains secret from the user but is embedded in all of the user's credentials, and a *secret identifier* $c_{\mathsf{id}}^* = \mathsf{H}(\mathsf{id} \parallel c_{\mathsf{id}})$, which is sent to the user. The manager records the user's information as $\mathsf{StatesM}[\mathsf{id}] = (\mathsf{state}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{Username})$, where $\mathsf{state}_{\mathsf{id}}^{\mathcal{M}} = (\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{flag}_{\mathsf{id}})$. Here, $\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}$ is the number of certificates issued to user and $\mathsf{flag}_{\mathsf{id}} \in \{\mathsf{Active}, \mathsf{Revoked}\}$ indicates whether the user is active or revoked. $\mathsf{state}_{\mathsf{id}}^{\mathcal{M}}$ is initially $(0, \mathsf{Active})$. Clearly, possessing the identifier id allows the manager to retrieve the identity

Username, and vice versa. The secret credential c_{id} will be used later for tracing the user.

Algorithm 2 DGSP.Join

Input: An identity Username, the manager's state StatesM, and DGSP.SK.
Output: $(\mathsf{id}, \mathsf{state}_{\mathsf{id}}^{\mathcal{U}}, \mathsf{state}_{\mathsf{id}}^{\mathcal{M}})/\perp$

1: The user sends Username as a request to join the group;
2: If $(_, \mathsf{Username}) \in \mathsf{StatesM}$, the Manager outputs $\perp$;
3: The manager takes the next available unassigned $\mathsf{id} \in \mathbb{N}$;
4: The manager generates $c_{\mathsf{id}} = \mathsf{H}(\mathsf{DGSP.msk}_1 \parallel \mathsf{id})$ and $c_{\mathsf{id}}^* = \mathsf{H}(\mathsf{id} \parallel c_{\mathsf{id}})$;
5: The manager sends $(\mathsf{id}, c_{\mathsf{id}}^*)$ to Username;
6: The manager stores $((0, \mathsf{Active}), \mathsf{Username})$ in the id-th entity of the list StatesM;
7: The user generates a secret seed $\mathsf{User.seed}_{\mathsf{id}}$; ▷ $\mathsf{User.seed}_{\mathsf{id}}$ is used by user id as a seed for generating WOTS$^+$ keys
8: The user stores $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}} = (\mathsf{id}, c_{\mathsf{id}}^*, \mathsf{User.seed}_{\mathsf{id}}, \mathsf{ctr}_{\mathsf{id}}^{\mathcal{U}}, \mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}, R, C)$; ▷ Initially $\mathsf{ctr}_{\mathsf{id}}^{\mathcal{U}} = \mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}} = 0$ and $R = C = \emptyset$.

Whenever a user has no remaining signing keys (i.e., $\mathsf{ctr}_{\mathsf{id}}^{\mathcal{U}} = 0$), either because they have just joined the group or have used their last key, the user and the manager follow the procedure shown in Algorithm 3 to update $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$ as explained in Section 3. This procedure enables the user to generate new keys and receive the corresponding certificates, ensuring they always have at least one unused key–certificate pair. The steps are as follows:

1. The user runs the *Certificate Signing Request* subroutine DGSP.RequestU to generate and store a batch of one-time public key pairs. Each batch is a list of pairs: the public key of each key pair and an ephemeral random seed used for key generation. The user will then send a message $\mathsf{msg}_{\mathsf{id}}^{\mathcal{U}}$ containing the user id, the identifier c_{id}^* as a form of authentication, and the list of all of the ephemeral public keys.
2. Upon receiving $\mathsf{msg}_{\mathsf{id}}^{\mathcal{U}} = (\mathsf{id}, \mathrm{c}_{\mathsf{id}}^*, \{\mathsf{pk}_{\mathsf{id,j}}\}_{\mathrm{j}=\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}+1}^{\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}+\mathrm{B}})$, the manager invokes the *Generating Certificate* subroutine, DGSP.ResponseM, to issue certificates, provided that (i) the state of this user is Active, i.e. $\mathsf{flag}_{\mathsf{id}} = \mathsf{Active}$ in $\mathsf{StatesM}[\mathsf{id}]$, and (ii) $c_{\mathsf{id}}^* = \mathsf{H}(\mathsf{id} \parallel c_{\mathsf{id}})$, where the manager first computes $c_{\mathsf{id}} = \mathsf{H}(\mathsf{DGSP.msk}_1 \parallel \mathsf{id})$, then checks the equality. After issuing the certificates, the manager sends $\mathsf{msg}_{\mathsf{id}}^{\mathcal{M}}$ (containing all certificates) to user id and updates $\mathsf{StatesM}[\mathsf{id}]$. See DGSP.ResponseM in Algorithm 3.
   To explain this algorithm, the manager iterates over all public keys sent to them, to generate a certificate for each one. They first add a tag $\zeta_{\mathsf{id},j} = \mathsf{E}(\mathsf{DGSP.msk}_2, \mathsf{id} \| j)$ to the j-th certificate. This will be included in the final group signature, allowing rapid tracing (the manager can simply decrypt and read the user's id). They also compute $\pi_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{id,j} \| c_{id})$, which will be given to the user and can act later as a proof that tracing was done honestly; until that point, it appears completely random to everyone except the group manager, thanks to the secret credential c_{id}.

Since only the group manager knows that c_{id} is associated with user id (and cannot prove this without revealing $\mathsf{DGSP.msk}_1$), the manager also generates $\tau_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{id,j} \| \pi_{\mathsf{id},j} \| \mathsf{id})$, which binds $\pi_{\mathsf{id},j}$ and the user's id to the public key for this certificate, but which looks random without knowledge of $\pi_{\mathsf{id},j}$ (which is random to all parties except the manager).

The manager then signs the public key $\mathsf{pk}_{id,j}$, $\zeta_{\mathsf{id},j}$, and $\tau_{\mathsf{id},j}$ with their long-term signing key. This signature, as well as the information $\zeta_{\mathsf{id},j}$ and $\pi_{\mathsf{id},j}$, form the certificate returned to the user. The signature ensures that the user must include the tracing information $\zeta_{\mathsf{id},j}$ and $\tau_{\mathsf{id},j}$ in any signature they construct.

3. Finally, after receiving $\mathsf{msg}_{\mathsf{id}}^{\mathcal{M}}$, the user id updates $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$ using the *Update State* subroutine $\mathsf{DGSP.UpdateU}$. See $\mathsf{DGSP.UpdateU}$ in Algorithm 3.

4. A final step, not shown, is that the user should verify each signature and verify that $\tau_{\mathsf{id},j}$ was computed correctly. This detects certain malicious group manager behaviour, but falls outside of our security definitions as a fully malicious group manager can still completely undermine the protocol (e.g., issuing certificates to itself on behalf of an unaware user).

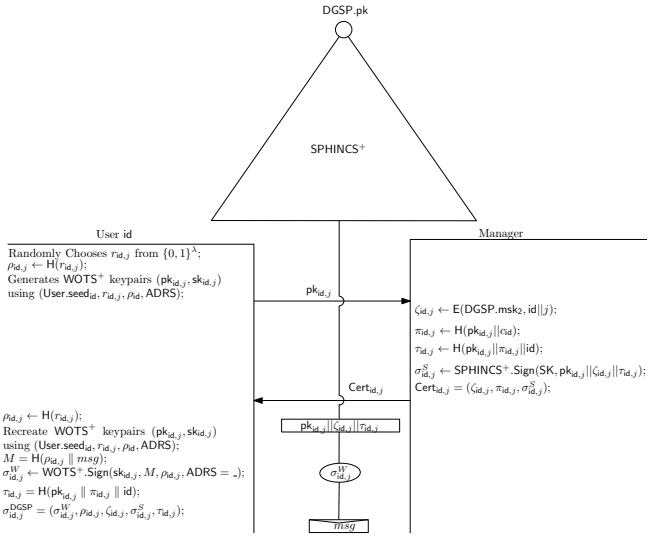

Fig. 1. An overview of DGSP with the batch size $B = 1$.

Algorithm 3 Subroutines of Key Distribution of DGSP

DGSP.RequestU: (Certificate Signing Request)

Input: State $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$ and batch size B.
Output: message $\mathrm{msg}_{\mathsf{id}}^{\mathcal{U}} = (\mathsf{id}, \mathrm{c}_{\mathsf{id}}^*, \{\mathsf{pk}_{\mathsf{id,j}}\}_{\mathrm{j}=\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}+1}^{\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}+\mathrm{B}})$ and updated state $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$.

$P^* = \emptyset$; $R^* = \emptyset$;
for $\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}} + 1 \le j \le \mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}} + B$ **do**
  $r_{\mathsf{id},j} \xleftarrow{\$} \{0,1\}^{\lambda}$;
  $\rho_{\mathsf{id},j} \leftarrow \mathsf{H}(r_{\mathsf{id},j})$;
  $(\mathsf{sk}_{\mathsf{id},j}, \mathsf{pk}_{\mathsf{id},j}) \leftarrow \mathsf{WOTS}^{+}.\mathsf{Keygen}(\mathsf{H}(\mathsf{User.seed}_{\mathsf{id}} \| r_{\mathsf{id},j}), \rho_{\mathsf{id},j}, \mathsf{ADRS} = _)$;
  $P^* = P^* \cup \{\mathsf{pk}_{\mathsf{id},j}\}$;
  $R^* = R^* \cup \{r_{\mathsf{id},j}\}$;
end for
updates R in $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$ as $R \leftarrow R \cup R^*$;
return $(\mathrm{msg}_{\mathsf{id}}^{\mathcal{U}}, \mathsf{state}_{\mathsf{id}}^{\mathcal{U}})$; ▷ $\mathrm{msg}_{\mathsf{id}}^{\mathcal{U}} = (\mathsf{id}, \mathrm{c}_{\mathsf{id}}^*, \mathrm{P}^*)$ with $P^* = \{\mathsf{pk}_{\mathsf{id},j}\}_{j=\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}+1}^{\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}+B}$

DGSP.ResponseM: (Generate Certificate)

Input: The manager's secret key and state $\mathsf{DGSP.SK} = (\mathsf{DGSP.msk}, \mathsf{SK})$ and $\mathsf{StatesM}$, user's message $\mathrm{msg}_{\mathsf{id}}^{\mathcal{U}} = (\mathsf{id}, \mathrm{c}_{\mathsf{id}}^*, \mathrm{P}^*)$, where $P^* = \{\mathsf{pk}_{\mathsf{id},j}\}_{j=\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}+1}^{\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}+B}$.
Output: $\mathrm{msg}_{\mathsf{id}}^{\mathcal{M}} = \{\mathsf{Cert}_{\mathsf{id,j}}\}_{\mathrm{j}=\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}+1}^{\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}+\mathrm{B}}$ and updated state $\mathsf{StatesM}[\mathsf{id}]$.

$\mathrm{msg}_{\mathsf{id}}^{\mathcal{M}} = \emptyset$;
$c_{\mathsf{id}} = \mathsf{H}(\mathsf{DGSP.msk}_1 \| \mathsf{id})$;
if ($\mathsf{StatesM}[\mathsf{id}]$ is not Active) $\vee$ ($c_{\mathsf{id}}^* \neq \mathsf{H}(\mathsf{id} \| c_{\mathsf{id}})$) **then**, **return** $\perp$;
end if
$((\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{Active}), \mathsf{Username}) \leftarrow \mathsf{StatesM}[\mathsf{id}]$;
for $\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}} + 1 \le j \le \mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}} + B$ **do**
  $\zeta_{\mathsf{id},j} = \mathsf{E}(\mathsf{DGSP.msk}_2, \mathsf{id} \| j)$;
  $\pi_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \| c_{\mathsf{id}})$;
  $\tau_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \| \pi_{\mathsf{id},j} \| \mathsf{id})$;
  $\sigma_{\mathsf{id},j}^{S} = \mathsf{SPHINCS}^{+}.\mathsf{Sign}(\mathsf{SK}, \mathsf{pk}_{\mathsf{id},j} \| \zeta_{\mathsf{id},j} \| \tau_{\mathsf{id},j})$;
  $\mathsf{Cert}_{\mathsf{id},j} = (\zeta_{\mathsf{id},j}, \pi_{\mathsf{id},j}, \sigma_{\mathsf{id},j}^{S})$;
  $\mathrm{msg}_{\mathsf{id}}^{\mathcal{M}} = \mathrm{msg}_{\mathsf{id}}^{\mathcal{M}} \cup \{\mathsf{Cert}_{\mathsf{id,j}}\}$;
end for
$\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}} = \mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}} + B$;
$\mathsf{StatesM}[\mathsf{id}] \leftarrow ((\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{Active}), \mathsf{Username})$;
return $(\mathrm{msg}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{StatesM}[\mathsf{id}])$;

DGSP.UpdateU: (Update state)

Input: manager's message $\mathrm{msg}_{\mathsf{id}}^{\mathcal{M}}$ and user's state $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$.
Output: Update the user's state $\mathsf{state}_{\mathsf{id}}^{\mathcal{U}}$

$\mathsf{ctr}_{\mathsf{id}}^{\mathcal{U}} = \mathsf{ctr}_{\mathsf{id}}^{\mathcal{U}} + B$;
$\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}} = \mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}} + B$;
$C = C \cup \mathrm{msg}_{\mathsf{id}}^{\mathcal{M}}$;

Once the user id needs more keys, this user sends $(\mathsf{id}, c^*_{\mathsf{id}})$, along with the public keys of B newly generated WOTS$^+$ keys to the manager, requesting certificates for these public keys. This ensures that the number of signatures the user can generate on behalf of the group is not restricted to B.

– $(\sigma^{\mathrm{DGSP}}_{\mathsf{id},j}, \mathsf{state}^{\mathcal{U}}_{\mathsf{id}}) \leftarrow \mathsf{DGSP.Sign}(\mathsf{state}^{\mathcal{U}}_{\mathsf{id}}, \mathrm{msg})$:
This is detailed in Algorithm 4. When a user id wants to sign a message msg, the user takes an unused random seed $r_{\mathsf{id},j} \in R$ and its corresponding valid certificate $\mathsf{Cert}_{\mathsf{id},j} = (\zeta_{\mathsf{id},j}, \pi_{\mathsf{id},j}, \sigma^S_{\mathsf{id},j}) \in C$ from $\mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$. The user derives $\rho_{\mathsf{id},j}$ (which will be published) from $r_{\mathsf{id},j}$ and obtains $(\mathsf{sk}_{\mathsf{id},j}, \mathsf{pk}_{\mathsf{id},j})$ from $r_{\mathsf{id},j}, \rho_{\mathsf{id},j}$ and $\mathsf{User.seed}_{\mathsf{id}}$, then computes $M = \mathsf{H}(\rho_{\mathsf{id},j} \parallel \mathrm{msg})$. They sign M with the same WOTS$^+$ keypair that was signed in the certificate. This WOTS$^+$ signature, the salt $\rho_{\mathsf{id},j}$, and $\zeta_{\mathsf{id},j}$ and $\sigma^S_{\mathsf{id},j}$ form the certificate $\mathsf{Cert}_{\mathsf{id},j}$ from the group manager form the group signature.
Once this process is done, the user updates their state by securely deleting the random seed $r_{\mathsf{id},j}$ and the certificate itself and decrementing the counter $\mathsf{ctr}^{\mathcal{U}}_{\mathsf{id}}$ by one. This deletes all information that could be used to connect the user to this signature, ensuring forward anonymity: even under a future compromise of the user, the signature cannot be tied to the user.
Effectively, $\rho_{\mathsf{id},j}$ acts as a salt on the message, preventing offline collision attacks as in SPHINCS$^+$, whereas $r_{\mathsf{id},j}$ randomizes the key of each WOTS$^+$ keypair. In our implementation, we use the WOTS$^+$ implementation from SPHINCS$^+$, which incorporates a random salt into key generation as well as signing, and thus the user must generate $\rho_{\mathsf{id},j}$ when they request a certificate. To save space, we derive $\rho_{\mathsf{id},j}$ from $r_{\mathsf{id},j}$. With a different signature scheme we could generate salts at signing time, rather than key generation time; however, re-using the WOTS$^+$ implementation for SPHINCS$^+$ reduces the chances of implementation errors and only requires the user to store an extra λ bits of space for each certificate.

Algorithm 4 DGSP.Sign

Input: A message msg and $\mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$
Output: A group signature $\sigma^{\mathrm{DGSP}}_{\mathsf{id},j}$

1: Select a $r_{\mathsf{id},j} \in R$ and its corresponding $\mathsf{Cert}_{\mathsf{id},j} = (\zeta_{\mathsf{id},j}, \pi_{\mathsf{id},j}, \sigma^S_{\mathsf{id},j}) \in C$;
2: $\rho_{\mathsf{id},j} \leftarrow \mathsf{H}(r_{\mathsf{id},j})$
3: $(\mathsf{sk}_{\mathsf{id},j}, \mathsf{pk}_{\mathsf{id},j}) \leftarrow \mathsf{WOTS}^+.\mathsf{Keygen}(\mathsf{H}(\mathsf{User.seed}_{\mathsf{id}} \| r_{\mathsf{id},j}), \rho_{\mathsf{id},j}, \mathsf{ADRS} = _)$
4: $M = \mathsf{H}(\rho_{\mathsf{id},j} \parallel \mathrm{msg})$;
5: $\sigma^W_{\mathsf{id},j} \leftarrow \mathsf{WOTS}^+.\mathsf{Sign}(\mathsf{sk}_{\mathsf{id},j}, M, \rho_{\mathsf{id},j}, \mathsf{ADRS} = _)$;
6: $\tau_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \parallel \pi_{\mathsf{id},j} \parallel \mathsf{id})$;
7: $\sigma^{\mathrm{DGSP}}_{\mathsf{id},j} = (\sigma^W_{\mathsf{id},j}, \rho_{\mathsf{id},j}, \zeta_{\mathsf{id},j}, \sigma^S_{\mathsf{id},j}, \tau_{\mathsf{id},j})$;
8: Remove $r_{\mathsf{id},j}$ from R and $\mathsf{Cert}_{\mathsf{id},j}$ from C in $\mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$ and decrement $\mathsf{ctr}^{\mathcal{U}}_{\mathsf{id}}$ by one;
9: **return** $\sigma^{\mathrm{DGSP}}_{\mathsf{id},j}$;

Signature Size of DGSP: Given the security parameter λ, Winternitz parameter w, and setup parameters (h, d, k, t), the signature size of SPHINCS$^+$

is $(h + k \cdot (\log t + 1) + d \cdot \mathsf{len} + 1) \cdot \lambda$ bits (see Appendix A). Accordingly, a DGSP signature has the form $\sigma^{\mathrm{DGSP}} = \big(\sigma^W, \rho, \zeta, \sigma^S, \tau\big)$, resulting in a total size of $(h + k \cdot (\log t + 1) + (d + 1) \cdot \mathsf{len} + 3) \cdot \lambda + 128$ bits, as both r and τ are λ bits and ζ is 128 bits.

– $0/1 \leftarrow \mathsf{DGSP.Verify}(\mathrm{msg}, \sigma^{\mathrm{DGSP}}, \mathsf{DGSP.PP})$:
To verify the message-signature pair $(\mathrm{msg}, \sigma^{\mathrm{DGSP}})$, the verifier runs Algorithm 5. It first checks whether the user has been revoked. If $\zeta \notin \mathsf{RL}$, the verifier reconstructs the WOTS$^+$ public key pk' and then verifies the group manager's signature on $\mathsf{pk}' \| \zeta \| \tau$. If all checks pass, it outputs 1; otherwise, it outputs 0.

Algorithm 5 DGSP.Verify

Input: A message-signature pair $(\mathrm{msg}, \sigma^{\mathrm{DGSP}})$, and $\mathsf{DGSP.PP} = (\mathsf{PK}, \mathsf{RL})$
Output: $0/1$

Parse $\sigma^{\mathrm{DGSP}} = \big(\sigma^W, \rho, \zeta, \sigma^S, \tau\big)$;
if $\zeta \in \mathsf{RL}$, **then return** 0;
end if
$M = \mathsf{H}(\rho \parallel \mathrm{msg})$;
$\mathsf{pk}' \leftarrow \mathsf{WOTS}^+.\mathsf{PKRegen}(M, \sigma^W, \rho, \mathsf{ADRS} = _)$;
return $\mathsf{SPHINCS}^+.\mathsf{Verify}(\mathsf{pk}' \parallel \zeta \parallel \tau, \sigma^S, \mathsf{PK})$;

– $(\mathsf{StatesM}, \mathsf{DGSP.PP}) \leftarrow \mathsf{DGSP.Revoke}(\mathsf{DGSP.SK}, \mathsf{StatesM}, \mathsf{DGSP.PP}, \mathsf{R})$:
Revocation uses a list RL supporting both user and signature revocations. For each $\mathsf{id} \in \mathsf{R}$, the manager reads $\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}$ from $\mathsf{StatesM}[\mathsf{id}] = ((\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{flag}_{\mathsf{id}}), \mathsf{Username})$, appends $\{\zeta_{\mathsf{id},j} = \mathsf{E}(\mathsf{DGSP.msk}_2, \mathsf{id} \parallel j)\}_{j=1}^{\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}}$ to RL, and updates the $\mathsf{flag}_{\mathsf{id}}$ from Active to $\mathsf{Revoked}$ in $\mathsf{StatesM}[\mathsf{id}]$ (see Algorithm 6). For signature-level revocation, the system can instead add the corresponding ζ to RL, disabling specific signatures without revoking the user entirely. Although DGSP can support both modes, this paper focuses exclusively on user-level revocation.

Algorithm 6 DGSP.Revoke

Input: A list R of users, $\mathsf{DGSP.SK} = (\mathsf{DGSP.msk}, \mathsf{SK})$, the manager's state $\mathsf{StatesM}$, and $\mathsf{DGSP.PP} = (\mathsf{PK}, \mathsf{RL})$
Output: Update $\mathsf{StatesM}$ and $\mathsf{DGSP.PP}$

for $\mathsf{id} \in \mathsf{R}$ **do**
  if $\mathsf{flag}_{\mathsf{id}}$ in $\mathsf{StatesM}[\mathsf{id}]$ is Active **then**
    $((\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{Active}), \mathsf{Username}) \leftarrow \mathsf{StatesM}[\mathsf{id}]$;
    $\mathsf{RL} = \mathsf{RL} \cup \{\mathsf{E}(\mathsf{DGSP.msk}_2, \mathsf{id} \parallel j)\}_{j=1}^{\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}}$;
    $\mathsf{StatesM}[\mathsf{id}] \leftarrow ((\mathsf{ctr}_{\mathsf{id}}^{\mathcal{M}}, \mathsf{Revoked}), \mathsf{Username})$.
  end if;
end for

– $(\mathsf{id}, \pi_{\mathsf{id},j}) \leftarrow \mathsf{DGSP.Open}(\mathrm{msg}, \sigma^{\mathrm{DGSP}}_{\mathsf{id,j}}, \mathsf{DGSP.SK})$:
To open a valid message-signature pair $(\mathrm{msg}, \sigma^{\mathrm{DGSP}}_{\mathsf{id,j}})$, where $\sigma^{\mathrm{DGSP}}_{\mathsf{id},j} = \left(\sigma^{W}_{\mathsf{id},j}, \rho_{\mathsf{id},j}, \zeta_{\mathsf{id},j}, \sigma^{S}_{\mathsf{id},j}, \tau_{\mathsf{id},j}\right)$, the manager first decrypts $\zeta_{\mathsf{id},j}$ as

$$\mathsf{id} \parallel j = \mathsf{E}^{-1}(\mathsf{DGSP.msk}_2, \zeta_{\mathsf{id},j}).$$

Next, the manager computes the user's secret credential $c_{\mathsf{id}} = \mathsf{H}(\mathsf{DGSP.msk}_1 \parallel \mathsf{id})$ and the proof $\pi_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \parallel c_{\mathsf{id}})$. The manager then outputs id as the identifier of the signer along with the proof $\pi_{\mathsf{id},j}$, which enables public verification of this attribution. See Algorithm 7.

Algorithm 7 DGSP.Open

Input: A valid message-signature pair $(\mathrm{msg}, \sigma^{\mathrm{DGSP}})$, $\mathsf{DGSP.SK} = (\mathsf{DGSP.msk}, \mathsf{SK})$, and the manager's state $\mathsf{StatesM}$.
Output: $(\mathsf{id}, \pi_{\mathsf{id},j})/\ \perp$

Parse $\sigma^{\mathrm{DGSP}} = \left(\sigma^{W}, \rho, \zeta, \sigma^{S}, \tau\right)$;
$\mathsf{id} \parallel j = \mathsf{E}^{-1}(\mathsf{DGSP.msk}_2, \zeta)$;
if $\mathsf{id} > N$ **then return** $\perp$; ▷ N is the number of registered users.
else
  Compute $c_{\mathsf{id}} = \mathsf{H}(\mathsf{DGSP.msk}_1 \parallel \mathsf{id})$;
  Compute $M = \mathsf{H}(\rho \parallel \mathrm{msg})$;
  Compute $\mathsf{pk}_{\mathsf{id},j} \leftarrow \mathsf{WOTS}^{+}.\mathsf{PKRegen}(M, \sigma^{W}, \rho, \mathsf{ADRS} = _)$;
  Compute $\pi_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \parallel c_{\mathsf{id}})$;
end if
return $(\mathsf{id}, \pi_{\mathsf{id},j})$

– $0/1 \leftarrow \mathsf{DGSP.Judge}((\mathsf{id}, \pi_{\mathsf{id},j}), \mathrm{msg}, \sigma^{\mathrm{DGSP}})$:
This algorithm allows anyone to publicly verify that the manager has correctly attributed the message-signature pair $(\mathrm{msg}, \sigma^{\mathrm{DGSP}})$ to its actual signer. To this end, it first computes $\mathsf{pk}_{\mathsf{id},j}$ from the message and WOTS^{+} signature in σ^{DGSP}, then the attribution is accepted (outputting 1) if $\tau_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \parallel \pi_{\mathsf{id},j} \parallel \mathsf{id})$, and rejected (outputting 0) otherwise. See Algorithm 8.

Algorithm 8 DGSP.Judge

Input: A valid pair $(\mathrm{msg}, \sigma^{\mathrm{DGSP}})$ and an identifier-proof pair $(\mathsf{id}, \pi_{\mathsf{id},j})$.
Output: 0/1

Parse $\sigma^{\mathrm{DGSP}}_{\mathsf{id},j} = \left(\sigma^{W}, \rho, \zeta, \sigma^{S}, \tau\right)$;
Compute $M = \mathsf{H}(\rho \parallel \mathrm{msg})$;
Compute $\mathsf{pk}_{\mathsf{id},j} \leftarrow \mathsf{WOTS}^{+}.\mathsf{PKRegen}(M, \sigma^{W}, \rho, \mathsf{ADRS} = _)$;
if $\tau_{\mathsf{id},j} \neq \mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \parallel \pi_{\mathsf{id},j} \parallel \mathsf{id})$ **then return** 0;
end if
return 1;

In Appendix B, we will prove Theorem 1 by establishing Lemmas 1, 2, 3, and 4. In these Lemmas, we use various advantage terms parameterized by a security parameter λ and a certain number of queries Q: $\mathbf{Adv}^{\text{sEU-CMA}}_{\text{SPHINCS}^+}(\lambda, \mathsf{Q})$ is the maximum advantage over all adversaries against SPHINCS$^+$ in the strong existential unforgeability under chosen message attack; $\mathbf{Adv}^{\text{sEU-CMA}}_{\text{WOTS}^+}$ the same but against WOTS$^+$ (implicitly with 1 signing query); $\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, \mathsf{Q})$ is the maximum advantage over all adversaries against the PRF given by $F(k, x) = \mathsf{H}(k\|x)$ or $\mathsf{H}(x\|k)$ for the hash function H; Also $\mathbf{Adv}^{\mathsf{kSPR}}_{\mathsf{H}}(\lambda, \mathsf{Q})$ and $\mathbf{Adv}^{\mathsf{SPRP}}_{\mathsf{E}}(\lambda, \mathsf{Q})$ are the same but against keyed second preimage resistance H and SPRP E.

Theorem 1. *Let* DGSP *be the scheme described in Section 4, constructed using the EU-CMA secure signature scheme* SPHINCS$^+$*, the EU-CMA secure one-time signature scheme* WOTS$^+$*, a strong pseudorandom permutation* E *and a keyed second preimage resistant hash function* H*. Then,* DGSP *satisfies Correctness, Unforgeability, Anonymity, and Traceability, making it a fully dynamic group signature scheme.*

Lemma 1. (**Correctness**) *Let* $\sigma^{DGSP}_{\mathsf{id},j}$ *be a signature produced by an honest user* id *on a message* msg*. Then*

$$1 \leftarrow \mathsf{DGSP.Verify}(\text{msg}, \sigma^{\text{DGSP}}_{\mathsf{id,j}}, \mathsf{DGSP.PP})$$
$$1 \leftarrow \mathsf{DGSP.Judge}(\mathsf{DGSP.Open}(\text{msg}, \sigma^{\text{DGSP}}_{\mathsf{id,j}}, \mathsf{DGSP.SK}), \text{msg}, \sigma^{\text{DGSP}}_{\mathsf{id,j}}).$$

Lemma 2. (**Unforgeability**) *The advantage of any PPT adversary* $\mathcal{A}$ *against the unforgeability game of DGSP is at most:*

$$\begin{aligned}\mathbf{Adv}^{\text{Unforg}}_{\text{DGSP}}(\lambda) \leq\ & \mathbf{Adv}^{\text{sEU}-\text{CMA}}_{\text{SPHINCS}^+}(\lambda, \mathsf{Q}_{\mathsf{SignHU}}) + 2\mathsf{Q}_{\mathsf{AddHU}}\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, \mathsf{Q}_{\mathsf{SignHU}}) + \mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda; \mathsf{Q}_{\mathsf{RespM}}) \\ & + \tfrac{\mathsf{Q}_{\mathsf{RespM}}}{2^\lambda} + \mathsf{Q}_{\mathsf{AddHU}}\mathsf{Q}_{\mathsf{SignHU}}\mathbf{Adv}^{\text{sEU}-\text{CMA}}_{\text{WOTS}^+}(\lambda) + \mathsf{Q}_{\mathsf{AddHU}}\mathbf{Adv}^{\mathsf{kSPR}}_{\mathsf{H}}(\lambda, \mathsf{Q}_{\mathsf{SignHU}}),\end{aligned}$$

where $\mathsf{Q}_{\mathsf{AddHU}}$*,* $\mathsf{Q}_{\mathsf{SignHU}}$*, and* $\mathsf{Q}_{\mathsf{RespM}}$ *are the maximum number of queries to* AddHU*,* SignHU*, and* RespM*, respectively.*

Lemma 3. (**Traceability**) *The advantage of any PPT adversary* $\mathcal{A}$ *against the traceability game of DGSP is at most:*

$$\mathbf{Adv}^{\text{Trace}}_{\text{DGSP}}(\lambda) \leq \mathbf{Adv}^{\text{sEU}-\text{CMA}}_{\text{SPHINCS}^+}(\lambda, \mathsf{Q}_{\mathsf{SignHU}}) + \mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda; \mathsf{Q}_{\mathsf{RespM}}) + \frac{\mathsf{Q}_{\mathsf{RespM}}}{2^\lambda}$$

where $\mathsf{Q}_{\mathsf{SignHU}}$ *and* $\mathsf{Q}_{\mathsf{RespM}}$ *are the maximum number of queries to* SignHU*,* RespM*, respectively.*

Lemma 4. (**Anonymity**) *The advantage of any PPT adversary* $\mathcal{A}$ *against the anonymity of DGSP is at most:*

$$\begin{aligned}\mathbf{Adv}^{\text{Anon}}_{\text{DGSP}}(\lambda) \leq\ & \mathbf{Adv}^{\mathsf{SPRP}}_{\mathsf{E}}(\lambda, \mathsf{Q}_{\mathsf{SignHU}}) + \mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, \mathsf{Q}_{\mathsf{AddHU}} + \mathsf{Q}_{\mathsf{Open}}) + 2\mathsf{Q}^2_{\mathsf{AddHU}}\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, \mathsf{Q}_{\mathsf{SignHU}}) \\ & + 2\mathsf{Q}^2_{\mathsf{AddHU}}\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, 1) + \mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda; \mathsf{Q}_{\mathsf{RespM}}) + \tfrac{\mathsf{Q}_{\mathsf{RespM}}}{2^\lambda},\end{aligned}$$

where $\mathcal{A}$ *makes at most* $\mathsf{Q}_{\mathsf{AddHU}}$*,* $\mathsf{Q}_{\mathsf{SignHU}}$*,*$\mathsf{Q}_{\mathsf{Open}}$*,* $\mathsf{Q}_{\mathsf{RespM}}$ *queries to* AddHU*,* SignHU*,* Open*, and* RespM*, respectively.*

5 Implementation and Experimental Results

In this section, we present our implementation and experimental results for DGSP. Our implementation includes all algorithms from Section 4, namely DGSP.KG, DGSP.Join, DGSP.RequestU, DGSP.ResponseM, DGSP.Sign, DGSP.Verify, DGSP.Open, DGSP.Judge, and DGSP.Revoke. Our implementation is publicly available at

https://github.com/seyyedarashazimi/dgsp .

Experimental Setup: We conducted our experiments on an Ubuntu 24.04 system using the Rust 1.84.0 (stable) toolchain, compiled in release mode. The tests were run on an Intel® Core™ i7-4702MQ CPU operating at 2.20 GHz and equipped with 16 GiB of RAM. To ensure measurement stability, all benchmarks were executed on a single processor core with hyper-threading and turbo-boost disabled. Additionally, each timing result represents the average of one hundred benchmark samples.

Implementation Details: The DGSP scheme is purely implemented in Rust, leveraging the language's strong guarantees in safety, performance, and concurrency. We employ the `pqcrypto-sphincsplus` crate, which provides Rust bindings for the `PQClean` C implementations of SPHINCS$^+$ [42]. All 12 *simple* variants of SPHINCS$^+$, as provided by `PQClean`, are supported. These include three security levels (128, 192, and 256 bits), two hash function families (SHA2 and SHAKE), and two optimization targets (fast and small). We adopt the same naming convention used for different variants of SPHINCS$^+$ in DGSP, replacing SPHINCS$^+$ with DGSP, e.g. if DGSP is instantiated with "SPHINCS$^+$-SHAKE-256f-simple", we denote it as "DGSP-SHAKE-256f-simple".

In our implementation, the most time-consuming and computationally intensive tasks, including DGSP.RequestU and DGSP.ResponseM, are parallelized and processed in batches. However, it is worth emphasizing that we enforced single-thread usage to measure on a single processor in our experiments to maintain consistent benchmarking conditions.

To facilitate flexibility and extensibility, we define interfaces for the two key databases in our scheme, StatesM and RL. These interfaces are designed to accommodate future specialization for diverse use cases. We also offer two concrete implementations of these interfaces, an *in-memory* and an *in-disk* variant. These variants allow choosing between volatile or persistent storage solutions, depending on specific application requirements.

The reference implementation is fully tested and completely documented. We also provide a simple, easy-to-use example demonstrating essential functionality, ensuring that researchers can quickly verify and integrate our scheme.

Benchmark Results: Experimental results of DGSP-SHAKE-256s-simple and DGSP-SHAKE-256f-simple are presented in Table 6 and Table 7, respectively. They use a security parameter $\lambda = 256$, offering 256-bit classic security and 128-bit quantum security, with a Winternitz parameter $w = 16$. They are instantiated in *in-memory* and *in-disk* database storage for group sizes

$N \in \{2^{10}, 2^{25}\}$. The batch sizes are set to $B \in \{1, 8\}$, and the DGSP.SetPr are $(h, d, k, t) = (64, 8, 22, 2^{14})$ and $(h, d, k, t) = (68, 17, 35, 2^9)$, respectively.

DGSP requires no setup or initialization, operating immediately upon deployment. While tested for 2^{10} and 2^{25} users, the reference implementation scales efficiently up to 2^{60} users, given sufficient storage.

Table 6. Benchmarks for the DGSP-SHAKE-256s-simple. Elapsed times are reported in milliseconds (ms) and sizes in bytes (B).

Database variants		in-memory				in-disk			
Number of users		2^{10}		2^{25}		2^{10}		2^{25}	
Certificate batch size		1	8	1	8	1	8	1	8
Time(ms)	DGSP.KG	48.888	48.892	48.921	48.886	48.802	48.923	48.882	48.919
	DGSP.Join	0.0030	0.0030	0.0029	0.0029	0.0250	0.0263	0.0272	0.0280
	DGSP.RequestU	1.3552	10.894	1.3556	10.905	1.4094	11.126	1.4117	11.137
	DGSP.ResponseM	582.70	4661.6	583.53	4661.6	582.93	4664.1	583.52	4667.4
	DGSP.Sign	1.3847	1.3835	1.3863	1.3848	1.4418	1.4396	1.4874	1.4813
	DGSP.Verify	1.8255	1.8248	1.8207	1.8180	1.8515	1.8561	1.8571	1.8529
	DGSP.Open	0.6991	0.6881	0.6901	0.6911	0.7137	0.7060	0.7197	0.7237
	DGSP.Judge	0.6806	0.6813	0.6744	0.6798	0.7016	0.6987	0.7016	0.6986
	DGSP.Revoke	0.0012	0.0044	0.0012	0.0046	0.1773	0.0733	0.0322	0.0797
Size(B)	PK	64 Bytes							
	DGSP.SK	192 Bytes							
	σ_{DGSP}	32016 Bytes							

Table 7. Benchmarks for the DGSP-SHAKE-256f-simple. Elapsed times are reported in milliseconds (ms) and sizes in bytes (B).

Database variants		in-memory				in-disk			
Number of users		2^{10}		2^{25}		2^{10}		2^{25}	
Certificate batch size		1	8	1	8	1	8	1	8
Time(ms)	DGSP.KG	3.0521	3.0534	3.0536	3.0545	3.0585	3.0594	3.0546	3.0558
	DGSP.Join	0.0030	0.0030	0.0029	0.0029	0.0259	0.0255	0.0278	0.0287
	DGSP.RequestU	1.3630	10.985	1.3626	10.988	1.3982	11.045	1.4025	11.051
	DGSP.ResponseM	61.451	491.53	61.552	491.55	61.910	495.04	61.596	492.15
	DGSP.Sign	1.3875	1.3870	1.3886	1.3869	1.4196	1.4202	1.4186	1.4192
	DGSP.Verify	2.7706	2.7678	2.7808	2.7725	2.7626	2.7670	2.7667	2.7778
	DGSP.Open	0.6798	0.6948	0.6987	0.7043	0.6957	0.6957	0.6831	0.6954
	DGSP.Judge	0.6917	0.6912	0.7014	0.6937	0.6914	0.6728	0.6878	0.6877
	DGSP.Revoke	0.0007	0.0027	0.0008	0.0027	0.0305	0.0754	0.0259	0.0909
Size(B)	PK	64 Bytes							
	DGSP.SK	192 Bytes							
	σ_{DGSP}	52080 Bytes							

6 Comparing DGSP with DGMT and SITH

This section compares the performance of DGSP-SHAKE-256s-simple, DGMT, and SITH for the security parameter $\lambda = 256$. Source code for the implementations used in this comparison is available at:

DGSP : https://github.com/seyyedarashazimi/dgsp ,
DGMT : https://github.com/skarati/DGMT ,
SITH : https://github.com/UoS-SCCS/HBGS .

Before comparing the performance of DGSP, DGMT, and SITH, we first examine the experimental platforms used to measure their performance. The Intel i7-4702MQ processor, used to measure the timings of DGSP, belongs to the Haswell architecture family. In contrast, the Intel i7-9700 and i7-8850H processors, used to measure the timings of DGMT and SITH, are based on the more advanced Coffee Lake architecture. Furthermore, the Intel i7-4702MQ processor used in the DGSP experiments has a base frequency of 2.20 GHz, the lowest among the three. Although signature sizes are platform-independent, the performance of operations like Join, Sign, Verify, Open, and Revoke depends heavily on the platform. Notably, despite using the least powerful Intel processor among the three, DGSP still significantly outperforms the other schemes. In addition, it is important to note that the authors of SITH acknowledge that "we did not implement the full protocols" [22]. In contrast, our work provides a complete implementation of the proposed protocol.

For this comparison, we consider group sizes of 2^{25} for DGSP (Table 6, last column), 2^7 for DGMT (Table 7, last column in [32]), and 2^{20} for SITH (Table 3 in [22]). DGMT supports significantly smaller group sizes compared to DGSP and SITH, with 2^7 being the largest group size for which experimental results are available [32]. We also emphasize that DGMT's setup is significantly less efficient than that of DGSP. The performance comparison is summarized in Table 8.

The signature size of DGSP is 5.86 times larger than that of DGMT but 111.38 times smaller than that of SITH. This difference arises because DGSP is based on the stateless signature scheme SPHINCS$^+$, whereas DGMT relies on the stateful signature scheme XMSS^{MT}. In contrast, SITH incorporates both SPHINCS$^+$ and a NIZK-proof system, leading to significantly larger signatures.

- The joining operation in DGSP is 304.6×10^3 times faster than in DGMT and 14.6×10^6 times faster than in SITH.
- During the signing procedure, the user generates the j-th WOTS$^+$ secret key $\mathsf{sk}_{\mathsf{id},j} = (sk_{\mathsf{id},j,1}, \cdots, sk_{\mathsf{id},j,\mathsf{len}})$ using $\mathsf{WOTS}^+.\mathsf{Keygen}$ and signs the message msg by $\mathsf{sk}_{\mathsf{id},j}$ and its corresponding certificate $\mathsf{Cert}_{\mathsf{id},j}$. The use of $\mathsf{WOTS}^+.\mathsf{Keygen}$ in $\mathsf{DGSP.Sign}$ makes DGSP signing 3.71 times slower than DGMT. However, unlike SITH, DGSP avoids constructing NIZK proofs, resulting in a 35.78×10^3 speedup over SITH in signing efficiency.
- The verification operation in DGSP is 660 times faster than in DGMT and 13.82×10^3 times faster than in SITH.

Table 8. Comparison among DGSP, DGMT, and SITH. KB, s, and ms stand for Kilobyte, second, and millisecond. We note that DGMT might be easily adapted to forward anonymous if ephemeral certificates are deleted after use.

		DGSP (this work)	DGMT	SITH
Platform	Processor Model	Intel i7-4702MQ	Intel i7-9700	Intel i7-8850H
	Processor Architecture	Haswell	Coffee Lake	Coffee Lake
	Processor Frequency	2.20 GHz	3.00 GHz	2.60 GHz
Classic Security		256bits	128bits	255bits
Quantum Security		128qbits	128qbits*	128qbits
Underlying Hard Problem		Second Preimage Resistance	Collision Resistance	Second Preimage Resistance
Signature Size		31.27KB	5.34KB	3483KB
Join		0.028ms	8.529s	410.3s
Certificate Generation		583ms	658ms	Not required
Sign		1.481ms	0.399ms	53.0s
Verify		1.852ms	1223ms	25.6s
Open		0.723ms	0.0003ms	–
Revoke		0.079ms	0.134ms	–
Judge		0.698ms	–	–
Language		Rust	C	C++
State		Stateful	Stateful	Stateless
Non-frameability		No	No	Yes
Forward Anonymous		Yes	Yes**	No
Max. # of signatures		2^{64}	2^{27}***	Unlimited

– indicates that the scheme either does not support this feature or that no implementation exists.
* 128 qbits assumes based on [6] that there is no quantum advantage in collision search; otherwise the security drops to 85 qubits, and achieving 128-qubit security would require setting $\lambda = 384$ [14].
** While the published version of DGMT is not forward anonymous, we expect that deleting used credentials would make the scheme forward anonymous.
***Arguably DGMT could support more but the setup time is proportional to the maximum number of signatures. In the experimental data shown this was 2^{27}.

– During the opening procedure, DGSP not only traces the identity but also provides proof supporting the correct opening. However, DGMT does not offer such proof of correct opening. As a result, the opening operation in DGSP is 2410 times slower than that of DGMT. In contrast, the revocation operation in DGSP is 1.69 times faster than in DGMT.

As discussed in Section 1, SITH's brute-force tracing algorithm scales linearly with the group size: for N users, its opening requires $N/2$ evaluations of the PRF F and an equal number of comparisons, resulting in $\mathcal{O}(N)$ complexity. This makes large-scale tracing impractical, despite SITH's claim to support $N = 2^{60}$ users. In particular, SITH does not provide timing results for its opening and revocation processes. By contrast, DGSP achieves constant-time opening via a single call to the SPRP E^{-1}.

7 Conclusion

We introduced DGSP, a post-quantum fully dynamic group signature scheme that overcame the limitations of DGMT and Sphinx-in-the-Head (SITH), two of the most efficient schemes in this domain. We formally proved DGSP's security, fully implemented the scheme, and conducted an extensive comparative evaluation, demonstrating its superiority in efficiency and scalability over DGMT and SITH.

By leveraging the stateless property of SPHINCS$^+$, DGSP achieved an optimal balance between efficiency and scalability, supporting up to 2^{60} users while maintaining a short, constant signature size and ensuring efficient setup and algorithms. Although SITH claimed similar scalability, our analysis revealed that its opening algorithm was highly inefficient, making large-scale deployment impractical. Thus, DGSP became the first fully dynamic group signature scheme capable of accommodating 2^{60} users per group public key. Beyond efficiency and scalability, DGSP also improves managerial accountability over DGMT by (i) enabling users to verify their certificates issued by the manager, and (ii) making the manager's signature attribution publicly verifiable. These mechanisms strengthen trust, reduce manager error, and make DGSP a robust, practical fully dynamic group signature scheme.

A limitation of DGSP, as with any stateful group signature, is that DGSP is vulnerable to traffic analysis: users who request many certificates are more likely to be the user who produced any given signature; a user who requests certificates immediately before a signature is produced is likely the same user; if a user has not requested certificates in a long time, they are likely not signing anything. Formalizing and mitigating these kinds of timing attacks is challenging and we do not do this, but we mention this here to warn against using DGSP in deployments where such attacks would be more impactful.

Acknowledgment

The authors acknowledge support of the NSERC Alliance Consortia Quantum Grant "Accelerating the transition to quantum-resistant cryptography" (ALLRP 578463–2022), and the NSERC Discovery Grant program (RGPIN-2024-03996). This work is also supported in part by the Information Security Education and Awareness (ISEA) Project Phase-III initiatives of the Ministry of Electronics and Information Technology (MeitY) of the Government of India under Grant No. F.No. L-14017/1/2022-HRD. The first author initiated this project during the final stage of his postdoctoral appointment at the University of Calgary and gratefully acknowledges Douglas Stebila, David Jao, and Reihaneh Safavi-Naini. The second author thanks Nasour Bagheri for his support throughout this study.

References

1. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Miller, C., Moody, D., Peralta, R., et al.: Status report on the third round of the nist post-quantum cryptography standardization process. US Department of Commerce, NIST (2022)
2. Argo, S., Güneysu, T., Jeudy, C., Land, G., Roux-Langlois, A., Sanders, O.: Practical post-quantum signatures for privacy. In: Luo, B., Liao, X., Xu, J., Kirda, E., Lie, D. (eds.) Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024. pp. 1523–1537. ACM (2024). https://doi.org/10.1145/3658644.3670297, https://doi.org/10.1145/3658644.3670297
3. Bansarkhani, R.E., Misoczki, R.: G-merkle: A hash-based group signature scheme from standard assumptions. In: Lange, T., Steinwandt, R. (eds.) Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10786, pp. 441–463. Springer (2018). https://doi.org/10.1007/978-3-319-79063-3_21
4. Bellare, M., Micciancio, D., Warinschi, B.: Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In: Biham, E. (ed.) Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2656, pp. 614–629. Springer (2003). https://doi.org/10.1007/3-540-39200-9_38
5. Bellare, M., Shi, H., Zhang, C.: Foundations of group signatures: The case of dynamic groups. In: Menezes, A. (ed.) Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3376, pp. 136–153. Springer (2005). https://doi.org/10.1007/978-3-540-30574-3_11
6. Bernstein, D.J.: Cost analysis of hash collisions: Will quantum computers make sharcs obsolete? In: Workshop Record of SHARCS'09: Special-purpose Hardware for Attacking Cryptographic Systems. pp. 105–116 (2009), https://cr.yp.to/hash/collisioncost-20090517.pdf, sHARCS'09, Lausanne, Switzerland, September 9–10, 2009

7. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs$^+$ signature framework. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 2129–2146. ACM (2019). https://doi.org/10.1145/3319535.3363229, https://doi.org/10.1145/3319535.3363229
8. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Cryptographic sponge functions. Tech. Rep. Version 0.1, STMicroelectronics and NXP Semiconductors (January 2011), version 0.1, January 14, 2011
9. Beullens, W., Dobson, S., Katsumata, S., Lai, Y., Pintore, F.: Group signatures and more from isogenies and lattices: generic, simple, and efficient. Des. Codes Cryptogr. **91**(6), 2141–2200 (2023). https://doi.org/10.1007/S10623-023-01192-X
10. Beullens, W., Kleinjung, T., Vercauteren, F.: Csi-fish: Efficient isogeny based signatures through class group computations. In: Galbraith, S.D., Moriai, S. (eds.) Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11921, pp. 227–247. Springer (2019). https://doi.org/10.1007/978-3-030-34578-5_9
11. Boneh, D., Eskandarian, S., Fisch, B.: Post-quantum EPID signatures from symmetric primitives. In: Matsui, M. (ed.) Topics in Cryptology - CT-RSA 2019 - The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4-8, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11405, pp. 251–271. Springer (2019). https://doi.org/10.1007/978-3-030-12612-4_13
12. Bootle, J., Cerulli, A., Chaidos, P., Ghadafi, E., Groth, J.: Foundations of fully dynamic group signatures. In: Manulis, M., Sadeghi, A., Schneider, S.A. (eds.) Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9696, pp. 117–136. Springer (2016). https://doi.org/10.1007/978-3-319-39555-5_7
13. Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - kyber: A cca-secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018. pp. 353–367. IEEE (2018). https://doi.org/10.1109/EUROSP.2018.00032
14. Brassard, G., Høyer, P., Tapp, A.: Quantum cryptanalysis of hash and claw-free functions. In: Lucchesi, C.L., Moura, A.V. (eds.) LATIN '98: Theoretical Informatics, Third Latin American Symposium, Campinas, Brazil, April, 20-24, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1380, pp. 163–169. Springer (1998). https://doi.org/10.1007/BFB0054319
15. Brickell, E.F., Camenisch, J., Chen, L.: Direct anonymous attestation. In: Atluri, V., Pfitzmann, B., McDaniel, P.D. (eds.) Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004. pp. 132–145. ACM (2004). https://doi.org/10.1145/1030083.1030103
16. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS - A practical forward secure signature scheme based on minimal security assumptions. In: Yang, B. (ed.) Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings. Lecture Notes

in Computer Science, vol. 7071, pp. 117–129. Springer (2011). https://doi.org/10.1007/978-3-642-25405-5_8
17. Buser, M., Liu, J.K., Steinfeld, R., Sakzad, A., Sun, S.: DGM: A dynamic and revocable group merkle signature. In: Sako, K., Schneider, S.A., Ryan, P.Y.A. (eds.) Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11735, pp. 194–214. Springer (2019). https://doi.org/10.1007/978-3-030-29959-0_10
18. Camenisch, J., Groth, J.: Group signatures: Better efficiency and new theoretical aspects. In: Blundo, C., Cimato, S. (eds.) Security in Communication Networks, 4th International Conference, SCN 2004, Amalfi, Italy, September 8-10, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3352, pp. 120–133. Springer (2004). https://doi.org/10.1007/978-3-540-30598-9_9
19. Camenisch, J., Herreweghen, E.V.: Design and implementation of the *idemix* anonymous credential system. In: Atluri, V. (ed.) Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002. pp. 21–30. ACM (2002). https://doi.org/10.1145/586110.586114
20. Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., Zaverucha, G.: Post-quantum zero-knowledge and signatures from symmetric-key primitives. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1825–1842. ACM (2017). https://doi.org/10.1145/3133956.3133997
21. Chaum, D., van Heyst, E.: Group signatures. In: Davies, D.W. (ed.) Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings. Lecture Notes in Computer Science, vol. 547, pp. 257–265. Springer (1991). https://doi.org/10.1007/3-540-46416-6_22
22. 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). https://doi.org/10.1145/3638763
23. Courtois, N.T., Finiasz, M., Sendrier, N.: How to achieve a mceliece-based digital signature scheme. In: Boyd, C. (ed.) Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2248, pp. 157–174. Springer (2001). https://doi.org/10.1007/3-540-45682-1_10
24. Dang, Q.H.: Secure hash standard (shs). Federal Information Processing Standards Publication FIPS 180-4, National Institute of Standards and Technology, Gaithersburg, MD (August 2015). https://doi.org/10.6028/NIST.FIPS.180-4
25. Ducas, L., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - dilithium: Digital signatures from module lattices. IACR Cryptol. ePrint Arch. p. 633 (2017), http://eprint.iacr.org/2017/633
26. Dworkin, M.J.: The SHA-3 standard: Permutation-based hash and extendable-output functions. FIPS Publication FIPS 202, National Institute of Standards and Technology, Gaithersburg, MD (Aug 2015). https://doi.org/10.6028/NIST.FIPS.202, https://doi.org/10.6028/NIST.FIPS.202
27. El Kaafarani, A., Katsumata, S., Solomon, R.: Anonymous reputation systems achieving full dynamicity from lattices. In: Meiklejohn, S., Sako, K. (eds.) Financial

Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10957, pp. 388–406. Springer (2018). https://doi.org/10.1007/978-3-662-58387-6_21

28. Emura, K., Hanaoka, G., Sakai, Y., Schuldt, J.C.N.: Group signature implies public-key encryption with non-interactive opening. Int. J. Inf. Sec. **13**(1), 51–62 (2014). https://doi.org/10.1007/S10207-013-0204-Y
29. Esgin, M.F., Steinfeld, R., Zhao, R.K.: Matrict$^+$: More efficient post-quantum private blockchain payments. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022. pp. 1281–1298. IEEE (2022). https://doi.org/10.1109/SP46214.2022.9833655
30. ETSI: Cyber; quantum-safe signatures. ETSI Technical Report TR 103 616 V1.1.1, European Telecommunications Standards Institute (ETSI), Sophia Antipolis, France (September 2021), https://www.etsi.org/deliver/etsi_tr/103600_103699/103616/01.01.01_60/tr_103616v010101p.pdf, version 1.1.1
31. Ezerman, M.F., Lee, H.T., Ling, S., Nguyen, K., Wang, H.: Provably secure group signature schemes from code-based assumptions. IEEE Trans. Inf. Theory **66**(9), 5754–5773 (2020). https://doi.org/10.1109/TIT.2020.2976073
32. Fadavi, M., Karati, S., Erfanian, A., Safavi-Naini, R.: DGMT: A fully dynamic group signature from symmetric-key primitives. Cryptogr. **9**(1), 12 (2025). https://doi.org/10.3390/CRYPTOGRAPHY9010012
33. Goldreich, O.: Foundations of Cryptography: Volume 2, Basic Applications. Cambridge University Press, Cambridge, UK (May 2004). https://doi.org/10.1017/CBO9780511721656
34. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Miller, G.L. (ed.) Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996. pp. 212–219. ACM (1996). https://doi.org/10.1145/237814.237866
35. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: Buhler, J. (ed.) Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1423, pp. 267–288. Springer (1998). https://doi.org/10.1007/BFB0054868
36. Hülsing, A.: WOTS+ - shorter signatures for hash-based signature schemes. IACR Cryptol. ePrint Arch. p. 965 (2017), http://eprint.iacr.org/2017/965
37. Hülsing, A., Butin, D., Gazdag, S., Rijneveld, J., Mohaisen, A.: XMSS: extended merkle signature scheme. RFC **8391**, 1–74 (2018). https://doi.org/10.17487/RFC8391
38. Hülsing, A., Rausch, L., Buchmann, J.: Optimal parameters for XMSS MT. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E.R., Xu, L. (eds.) Security Engineering and Intelligence Informatics - CD-ARES 2013 Workshops: MoCrySEn and SeCIHD, Regensburg, Germany, September 2-6, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8128, pp. 194–208. Springer (2013). https://doi.org/10.1007/978-3-642-40588-4_14, https://doi.org/10.1007/978-3-642-40588-4_14
39. Hülsing, A., Rijneveld, J., Song, F.: Mitigating multi-target attacks in hash-based signatures. In: Cheng, C., Chung, K., Persiano, G., Yang, B. (eds.) Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016,

Proceedings, Part I. Lecture Notes in Computer Science, vol. 9614, pp. 387–416. Springer (2016). https://doi.org/10.1007/978-3-662-49384-7_15
40. Impagliazzo, R., Rudich, S.: Limits on the provable consequences of one-way permutations. In: Johnson, D.S. (ed.) Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA. pp. 44–61. ACM (1989). https://doi.org/10.1145/73007.73012
41. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: Johnson, D.S., Feige, U. (eds.) Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007. pp. 21–30. ACM (2007). https://doi.org/10.1145/1250790.1250794
42. Kannwischer, M.J., Schwabe, P., Stebila, D., Wiggers, T.: Improving software quality in cryptography standardization projects. In: IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022. pp. 19–30. IEEE (2022). https://doi.org/10.1109/EUROSPW55150.2022.00010
43. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Second Edition. CRC Press (2014), https://www.crcpress.com/Introduction-to-Modern-Cryptography-Second-Edition/Katz-Lindell/p/book/9781466570269
44. Kölbl, S., Lauridsen, M.M., Mendel, F., Rechberger, C.: Haraka v2 – efficient short-input hashing for post-quantum applications. IACR Transactions on Symmetric Cryptology **2016**(2), 1–29 (2016). https://doi.org/10.13154/tosc.v2016.i2.1-29
45. Lamport, L.: Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory (Oct 1979)
46. Lyubashevsky, V., Nguyen, N.K., Plançon, M., Seiler, G.: Shorter lattice-based group signatures via "almost free" encryption and other optimizations. In: Tibouchi, M., Wang, H. (eds.) Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 13093, pp. 218–248. Springer (2021). https://doi.org/10.1007/978-3-030-92068-5_8
47. McGrew, D.A., Curcio, M., Fluhrer, S.R.: Leighton-micali hash-based signatures. RFC **8554**, 1–61 (2019). https://doi.org/10.17487/RFC8554
48. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings. Lecture Notes in Computer Science, vol. 435, pp. 218–238. Springer (1989). https://doi.org/10.1007/0-387-34805-0_21
49. National Institute of Standards and Technology: Module-lattice-based digital signature standard. Federal Information Processing Standards Publication FIPS 204, U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD (August 2024). https://doi.org/10.6028/NIST.FIPS.204
50. National Institute of Standards and Technology: Module-lattice-based key-encapsulation mechanism standard. Federal Information Processing Standards Publication FIPS 203, U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD (August 2024). https://doi.org/10.6028/NIST.FIPS.203, https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.203.pdf

51. National Institute of Standards and Technology: Stateless hash-based digital signature standard. Federal Information Processing Standards Publication FIPS 205, U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD (August 2024). https://doi.org/10.6028/NIST.FIPS.205
52. Sanders, O.: EPID with efficient proof of non-revocation. In: Suga, Y., Sakurai, K., Ding, X., Sako, K. (eds.) ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022. pp. 1126–1138. ACM (2022). https://doi.org/10.1145/3488932.3517392, https://doi.org/10.1145/3488932.3517392
53. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994. pp. 124–134. IEEE Computer Society (1994). https://doi.org/10.1109/SFCS.1994.365700
54. Yehia, M., AlTawy, R., Gulliver, T.A.: GM^{mt}: A revocable group merkle multi-tree signature scheme. In: Conti, M., Stevens, M., Krenn, S. (eds.) Cryptology and Network Security - 20th International Conference, CANS 2021, Vienna, Austria, December 13-15, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13099, pp. 136–157. Springer (2021). https://doi.org/10.1007/978-3-030-92548-2_8
55. Yehia, M., AlTawy, R., Gulliver, T.A.: Security analysis of DGM and GM group signature schemes instantiated with XMSS-T. In: Yu, Y., Yung, M. (eds.) Information Security and Cryptology - 17th International Conference, Inscrypt 2021, Virtual Event, August 12-14, 2021, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13007, pp. 61–81. Springer (2021). https://doi.org/10.1007/978-3-030-88323-2_4

A Background

In the following, we provide formal definitions and security properties of some foundational preliminaries required in constructing DGSP [7,33,43]. Let $m(x)$ and $\ell(x)$ be two polynomials.

Pseudorandom Functions. A function $F : \{0,1\}^\lambda \times \{0,1\}^{m(\lambda)} \to \{0,1\}^{\ell(\lambda)}$ is called a *Pseudorandom Function (PRF)* if

1. For all $k \in \{0,1\}^\lambda$ and $x \in \{0,1\}^{m(\lambda)}$, $F(k,x)$ is efficiently computable.
2. For any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ making at most Q queries, the PRF advantage

$$\mathbf{Adv}^{\mathsf{PRF}}_{F,\mathcal{A}}(\lambda,\mathsf{Q}) = \left| \Pr\left[\mathcal{A}^{F(k,\cdot)}(\lambda,\mathsf{Q})=1\right] - \Pr\left[\mathcal{A}^{f(\cdot)}(\lambda,\mathsf{Q})=1\right] \right|$$

is negligible in λ, where $k \xleftarrow{\$} \{0,1\}^\lambda$ and f is chosen randomly from the set of all functions mapping $m(\lambda)$ bits to $\ell(\lambda)$ bits. We define $\mathbf{Adv}^{\mathsf{PRF}}_{F}(\lambda,\mathsf{Q})$ as

$$\mathbf{Adv}^{\mathsf{PRF}}_{F}(\lambda,\mathsf{Q}) = \max_{\mathcal{A}\in\mathsf{A}_q} \left\{ \mathbf{Adv}^{\mathsf{PRF}}_{F,\mathcal{A}}(\lambda,\mathsf{Q}) \right\},$$

where A_q is the set of all possible PPT adversaries making at most Q queries.

Let $\mathsf{Perm}(m(\lambda))$ be the set of all permutations $\pi : \{0,1\}^{m(\lambda)} \to \{0,1\}^{m(\lambda)}$. A function $\mathsf{E} : \{0,1\}^{\lambda} \times \{0,1\}^{m(\lambda)} \to \{0,1\}^{m(\lambda)}$ is called *a Strong Pseudorandom Permutation (SPRP)* if

1. For all $k \in \{0,1\}^{\lambda}$ and $x \in \{0,1\}^{m(\lambda)}$, $\mathsf{E}(k,\cdot)$ is a permutation, and both $\mathsf{E}(k,x)$ and its inverse $\mathsf{E}^{-1}(k,x)$ are efficiently computable, where $\mathsf{E}^{-1} : \{0,1\}^{\lambda} \times \{0,1\}^{m(\lambda)} \to \{0,1\}^{m(\lambda)}$.
2. For any PPT adversary $\mathcal{A}$ making at most Q queries,

$$\mathbf{Adv}^{\mathsf{SPRP}}_{\mathsf{E},\mathcal{A}}(\lambda,\mathsf{Q}) = \left| \Pr\left[\mathbf{Exp}^{\mathsf{SPRP}}_{\mathsf{E},\mathcal{A}}(\lambda,\mathsf{Q}) = 1\right] - \frac{1}{2} \right|$$

is negligible in λ, where $\mathbf{Exp}^{\mathsf{SPRP}}_{\mathsf{E},\mathcal{A}}(\lambda,\mathsf{Q})$ is defined in Table 9.

Table 9. Experiment $\mathbf{Exp}^{\mathsf{SPRP}}_{\mathsf{E},\mathcal{A}}(\lambda,\mathsf{Q})$

$\mathbf{Exp}^{\mathsf{SPRP}}_{\mathsf{E}}(\mathcal{A})$	$\mathsf{Eval}(M)$	$\mathsf{invEval}(C)$
$k \xleftarrow{\$} \{0,1\}^{\lambda}$; $\pi \xleftarrow{\$} \mathsf{Perm}(m(\lambda))$; $b \leftarrow \{0,1\}$; $b' \leftarrow \mathcal{A}^{\mathsf{Eval}(\cdot),\mathsf{invEval}(\cdot)}$; return $(b = b')$;	$C_0 \leftarrow \mathsf{E}(k,M)$; $C_1 \leftarrow \pi(M)$ return C_b;	$M_0 \leftarrow \mathsf{E}^{-1}(k,C)$; $M_1 \leftarrow \pi^{-1}(C)$ return M_b;

Similarly, we define $\mathbf{Adv}^{\mathsf{SPRF}}_{\mathsf{E}}(\mathsf{Q})$ as

$$\mathbf{Adv}^{\mathsf{SPRF}}_{\mathsf{E}}(\lambda,\mathsf{Q}) = \max_{\mathcal{A}\in\mathsf{A}_q} \left\{ \mathbf{Adv}^{\mathsf{SPRF}}_{\mathsf{E},\mathcal{A}}(\lambda,\mathsf{Q}) \right\},$$

where A_q is the set of all possible PPT adversaries making at most Q queries. In real-world applications, SPRPs are typically instantiated using standard symmetric-key block ciphers, such as AES.

Digital Signatures. A digital signature scheme $\mathcal{DSS} = (\mathsf{Keygen}, \mathsf{Sign}, \mathsf{Verify})$ is a triple of PPT algorithms, where $\mathsf{Keygen}(1^{\lambda})$ outputs a secret-public key pair $(\mathsf{sk}, \mathsf{pk})$; $\mathsf{Sign}(\mathsf{sk}, \mathrm{msg})$ outputs a signature σ under sk for the message msg; $\mathsf{Verify}(\mathrm{msg}, \sigma, \mathsf{pk})$ outputs 1 if and only if σ is a valid signature on msg, such that for all $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Keygen}(1^{\lambda})$ and for any message msg:

$$\mathsf{Verify}(\mathrm{msg}, \mathsf{Sign}(\mathsf{sk}, \mathrm{msg}), \mathsf{pk}) = 1.$$

The standard security notion for digital signature schemes is *existentially unforgeability under chosen message attacks (EU-CMA)*. The success probability of adversary $\mathcal{A}$ that makes at most Q queries is defined as

$$\mathsf{Succ}^{\mathsf{EU\text{-}CMA}}_{\mathcal{DSS},\mathcal{A}}(\lambda,\mathsf{Q}) = \Pr\left[\mathbf{Exp}^{\mathsf{EU\text{-}CMA}}_{\mathcal{DSS},\mathcal{A}}(\lambda,\mathsf{Q}) = 1\right],$$

Table 10. Experiment $\mathbf{Exp}^{\mathsf{EU\text{-}CMA}}_{\mathcal{DSS},\mathcal{A}}(\lambda, \mathsf{Q})$

$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Keygen}(1^\lambda)$
$(\mathrm{msg}^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\mathsf{sk},\cdot)}(\mathsf{pk})$;
Let $\{(\mathrm{msg_i}, \sigma_\mathrm{i})\}_{\mathrm{i}=1}^{\mathsf{Q}}$ be the query-answer pairs of $\mathsf{Sign}(\mathsf{sk}, \cdot)$
Return 1 iff $\mathsf{Verify}(\mathrm{msg}^*, \sigma^*, \mathsf{pk}) = 1$ and $\mathrm{msg}^* \notin \{\mathrm{msg_i}\}_{\mathrm{i}=1}^{\mathsf{Q}}$

where $\mathbf{Exp}^{\mathsf{EU\text{-}CMA}}_{\mathcal{DSS},\mathcal{A}}(\lambda, \mathsf{Q})$ is defined in Table 10.

We call $\mathcal{DSS}$ *EU-CMA-secure*, if the maximum success probability

$$\mathsf{InSec}^{\mathsf{EU\text{-}CMA}}_{\mathcal{DSS}}(\lambda, \mathsf{Q}) = \max_{\mathcal{A} \in \mathsf{A}_q} \left\{ \mathsf{Succ}^{\mathsf{EU\text{-}CMA}}_{\mathcal{DSS},\mathcal{A}}(\lambda, \mathsf{Q}) \right\}$$

is negligible in λ, where A_q is the set of all possible PPT adversaries making at most Q queries to Sign.

A stronger security notion for $\mathcal{DSS}$ is strong existential unforgeability under chosen message attacks (sEU-CMA). This notion extends the experiment $\mathbf{Exp}^{\mathsf{EU\text{-}CMA}}_{\mathcal{DSS},\mathcal{A}}(\lambda, \mathsf{Q})$, with the difference that the adversary $\mathcal{A}$ succeeds not only by producing a valid signature on a new message, but also if $\mathcal{A}$ can generates a new, distinct signature σ^* on any message $\mathrm{msg} \in \{\mathrm{msg_i}\}_{\mathrm{i}=1}^{\mathsf{Q}}$ for which it has already obtained a signature during the signing oracle queries.

Tweakable hash function. SPHINCS$^+$ uses tweakable hash functions, each independently keyed and randomized, to make the scheme resistance against multi-target attacks [7]. These functions are defined as follows:

Definition 2. *Let $\alpha, \lambda \in \mathbb{N}$ and $\mathcal{P}$ be the public parameters space and $\mathcal{T}$ the tweak space. A tweakable hash function is an efficient function*

$$\mathsf{Th} : \mathcal{P} \times \mathcal{T} \times \{0,1\}^\alpha \to \{0,1\}^\lambda, \quad \mathrm{MD} \leftarrow \mathsf{Th}(\mathrm{P}, \mathrm{T}, \mathrm{M}), \tag{1}$$

mapping an α-bit message M *to an λ-bit hash value* MD *using a function key called public parameter* $\mathrm{P} \in \mathcal{P}$ *and a tweak* $\mathrm{T} \in \mathcal{T}$.

Let H be a cryptographic hash functions, e.g., SHA2[24], SHAKE[26], or Haraka[44]. A straightforward instantiation of the tweakable hash function in SPHINCS$^+$ is given by $\mathsf{Th}(P, T, M) = \mathsf{H}(P \parallel T \parallel M)$, which operates within the Quantum Accessible Random Oracle Model (QROM). In the instantiation of SPHINCS$^+$, $\alpha = l\lambda$ for some $l \in \mathbb{N}$, and consequently, Th is denoted by $\mathsf{Th}_l : \mathcal{P} \times \mathcal{T} \times \{0,1\}^{l\lambda} \to \{0,1\}^\lambda$. In particular, the tweakable hash function Th_1 is denoted by F.

To describe the components of SPHINCS$^+$, including WOTS$^+$ and FORS, prior access to SPHINCS$^+$'s key is required. The public key of the signature scheme SPHINCS$^+$ is $\mathsf{PK} = (\mathsf{PK.root}, \mathsf{PK.seed})$, where (i) $\mathsf{PK.root}$ is the root node of the topmost tree in the hypertree SPHINCS$^+$, and (ii) $\mathsf{PK.seed} \in \mathcal{P} = \{0,1\}^\lambda$ is a public seed. Moreover, the tweak element of Th_l, denoted by

$\mathsf{ADRS} \in \mathcal{T} = \{0,1\}^{256}$, is structured to ensure that all functions track the current context by updating addresses after each hash call. Seven distinct types of addresses ADRS are used in SPHINCS$^+$ for: (i) computing the WOTS$^+$ hash chains, (ii) compressing the WOTS$^+$ public key, (iii) computing the hash functions in the Merkle trees, (iv) computing the hash functions within the FORS trees, (v) compressing the tree roots in FORS, (vi) generating WOTS$^+$ secret keys, and (vii) generating FORS secret keys. While the address formats share a common structure, each hash invocation is associated with a distinct ADRS, ensuring the independence of hash inputs and providing resistance against multi-target attacks in the hash-based signature scheme. The secret key is $\mathsf{SK} = (\mathsf{SK.seed}, \mathsf{SK.prf}, \mathsf{PK})$, where $\mathsf{SK.seed}$ and $\mathsf{SK.prf}$ are λ-bit random seeds used to (i) generate WOTS$^+$ and FORS secret keys and (ii) randomize the message digest, respectively.

WOTS$^+$. WOTS$^+$ is a one-time signature scheme where each secret key is used to sign a single message [36]. Reusing a secret key for multiple messages compromises the security of the scheme. In [36], Hülsing proves that WOTS$^+$ is EU-CMA, if the used hash function family is a second preimage resistant family of undetectable one-way functions. WOTS$^+$ is adapted for use in SPHINCS$^+$ and serves two roles: (i) signing lower-layer tree roots to securely link them to upper layers in the hypertree, ensuring verifiable connectivity, and (ii) signing FORS public keys using the leaf nodes of the lowest-layer trees. See Figure 2. WOTS$^+$ consists of the three algorithms $(\mathsf{WOTS^+.Keygen}, \mathsf{WOTS^+.Sign}, \mathsf{WOTS^+.Verify})$ and is configured with two parameters: (i) the security parameter λ, the same as the security parameter of SPHINCS$^+$, and (ii) the Winternitz parameter $w \in \{4, 16, 256\}$. These parameters determine $\mathsf{len} = \mathsf{len}_1 + \mathsf{len}_2$, where

$$\mathsf{len}_1 = \left\lceil \frac{\lambda}{\log(w)} \right\rceil, \qquad \text{and} \qquad \mathsf{len}_2 = \left\lfloor \frac{\log(\mathtt{len}_1 \cdot (w-1))}{\log(w)} \right\rfloor + 1.$$

len is the number of λ-bit string elements in a WOTS$^+$ private key, uncompressed public key, and signature.

The ADRS used in WOTS$^+$ follows the same structure as the ADRS in SPHINCS$^+$. However, it is limited to three address types: (i) computing the WOTS$^+$ hash chains, (ii) compressing the WOTS$^+$ public key, and (iii) generating the WOTS$^+$ secret keys. The value of ADRS is fully determined by the chosen type, the index of the chain, and the step inside the chain. We also set the "layer address" field to a fixed value 73. Since 73 exceeds the highest possible layer address value in any SPHINCS$^+$ version, this choice guarantees it can never clash with an ADRS value used inside SPHINCS$^+$.

WOTS$^+$ algorithms are described below.

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{WOTS^+.Keygen}(\mathsf{SK}, \mathsf{ADRS})$: The key generation algorithm first generates a secret key $\mathsf{sk} = (sk_1, \cdots, sk_{\mathsf{len}})$, where $sk_1, \cdots, sk_{\mathsf{len}} \in \{0,1\}^\lambda$, using the pseudorandom function PRF. The input of PRF are the secret and public seeds $\mathsf{SK.seed}$ and $\mathsf{PK.seed}$ along with a a WOTS$^+$ address ADRS that indicates the address at which this secret key is going to be used. The

WOTS$^+$ compressed public key is derived as

$$\mathsf{pk} = \mathsf{Th}_{\mathsf{len}}(\mathsf{PK.seed}, \mathsf{ADRS}, pk_1 \parallel \cdots \parallel pk_{\mathsf{len}}),$$

where for $i = 1, \cdots, \mathsf{len}$ we have $pk_i = \mathsf{F}^{w-1}(\mathsf{PK.seed}, \mathsf{ADRS}, sk_i)$. Increasing w linearly extends the hash chain length while logarithmically reducing len.

- $\sigma^W \leftarrow \mathsf{WOTS}^+.\mathsf{Sign}(\mathsf{sk}, \mathrm{msg}, \mathsf{PK.seed}, \mathsf{ADRS})$: The signing algorithm generates a signature σ^W on a message $\mathrm{msg} \in \{0,1\}^\lambda$ as follows. It first represents the message msg in the base w as $(m_1, \ldots, m_{\mathsf{len}_1})_w$, i.e., $0 \leq m_1, \ldots, m_{\mathsf{len}_1} \leq w-1$. Then it computes the checksum $C = \sum_{i=1}^{\mathsf{len}_1}(w-1-m_i)$ and represents C in the base w as $(m_{\mathsf{len}_1+1}, \ldots, m_{\mathsf{len}})_w$. Let $\overline{msg} = (m_1, \ldots, m_{\mathsf{len}_1}, m_{\mathsf{len}_1+1}, \ldots, m_{\mathsf{len}})$. The WOTS$^+$ signature on the message msg using the secret key $\mathsf{sk} = (sk_1, \cdots, sk_{\mathsf{len}})$ is $\sigma^W = (\sigma_1^W, \ldots, \sigma_{\mathsf{len}}^W)$ where

  $$(\sigma_1^W, \ldots, \sigma_{\mathsf{len}}^W) = (\mathsf{F}^{m_1}(\mathsf{PK.seed}, \mathsf{ADRS}, sk_1), \ldots, \mathsf{F}^{m_{\mathsf{len}}}(\mathsf{PK.seed}, \mathsf{ADRS}, sk_{\mathsf{len}})).$$

- $0/1 \leftarrow \mathsf{WOTS}^+.\mathsf{Verify}(\mathrm{msg}, \sigma^W, \mathsf{PK.seed}, \mathsf{ADRS}, \mathsf{pk})$: The verification algorithm checks the validity of a message-signature pair (msg, σ^W) as follows. It computes $\overline{msg} = (m_1, \ldots, m_{\mathsf{len}_1}, m_{\mathsf{len}_1+1}, \ldots, m_{\mathsf{len}})$ as discussed above and apply F^{w-1-m_i} to each block σ_i^W, for every $1 \leq i \leq \mathsf{len}$, to obtain $pk'_i = \mathsf{F}^{w-1-m_i}(\sigma_i^W)$. Finally, computes

  $$\mathsf{pk}' = \mathsf{Th}_{\mathsf{len}}(\mathsf{PK.seed}, \mathsf{ADRS}, pk'_1 \parallel \ldots \parallel pk'_{\mathsf{len}}).$$

  The algorithm accepts the signature by outputting 1 if $\mathsf{pk} = \mathsf{pk}'$.

For WOTS$^+$, the EU-CMA and sEU-CMA security notions are identical to that described previously, except that the adversary is restricted to a single oracle query, i.e., $\mathsf{Q} = 1$.

Multi-message Signature Scheme (MSS) based on WOTS$^+$. A Merkle tree of height h supports 2^h signatures under a single public key, at the cost of increasing the signature size. In SPHINCS$^+$, Merkle trees are built from WOTS$^+$ key pairs $\{(\mathsf{sk}_i, \mathsf{pk}_i)\}_{i=0}^{2^h-1}$, where the key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$ is assigned to the i-th leaf node and set as $y_h[i] = \mathsf{pk}_i$, for $0 \leq i \leq 2^h - 1$. Internal nodes are recursively computed using the tweakable hash function Th_2 as

$$y_j[i] = \mathsf{Th}_2(\mathsf{PK.seed}, \mathsf{ADRS}, y_{j+1}[2i] \parallel y_{j+1}[2i+1]),$$

for each level j where $0 \leq j \leq h-1$ and $0 \leq i \leq 2^j - 1$. The root of the tree, $y_0[0]$, serves as the public key of this specific tree. The i-th leaf node of this Merkle tree, $0 \leq i < 2^h$, is used only once to sign a λ-bit message msg. The corresponding signature is denoted as

$$\sigma_i = (i, \sigma_i^W, A.path_i),$$

where $\sigma_i^W = \mathsf{WOTS}^+.\mathsf{Sign}(\mathsf{sk}_i, \mathrm{msg}, \mathsf{PK.seed}, \mathsf{ADRS})$ and $A.path_i$ represents the authentication path. To verify the signature, the verifier reconstructs the Merkle

tree root from the public key pk_i (which can be computed from the message msg and σ_i^W), the index i, and the authentication path $A.path_i$. The signature σ_i is valid if the reconstructed root matches the public key $y_0[0]$.

SPHINCS$^+$ adopts the hypertree construction from XMSS^{MT} to significantly increase the number of signatures per public key [37]. A hypertree is parameterized by integers h and d, representing the total tree height and the number of layers, respectively. In this structure, the leaf nodes of the lowest layer (the $(d-1)$-th layer) are used to sign messages, while the leaf nodes of layers $0 \leq i \leq d-2$ sign the Merkle root of the subsequent $(i+1)$-th layer. The root of the topmost Merkle tree serves as the public key. Importantly, constructing the public key requires only the full computation of the topmost layer. This hierarchical design involves not only validating authentication paths but also verifying the WOTS$^+$ signatures that link different layers. For more details, we refer the reader to [7,37,51].

Pseudorandom functions and the message digest. SPHINCS$^+$ generates *pseudorandom keys* using a pseudorandom function $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^{256} \to \{0,1\}^\lambda$, and derives *randomness* for message compression using another pseudorandom function $\mathsf{PRF}_{\mathrm{msg}} : \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$. The generated randomness is then used to compress the message msg via a keyed hash function

$$\mathsf{H}_{\mathrm{msg}} : \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^m.$$

This output is denoted by MD $\|$ Index, where Index represents the index of the leaf node used to sign the message digest MD.

FORS (Forest Of Random Subsets). FORS is a few-time signature scheme, which is parameterized by integers k and $t = 2^a$, and is designed to sign ka-bit messages. In other words, FORS comprises k Merkle trees, each with a height of a and each is responsible for authenticating the t secret values belonging to one subset out of the k sets. These kt secret leaf nodes within these trees are generated pseudorandomly and temporarily from SK.seed, PK.seed and ADRS via the pseudorandom function PRF, and these k Merkle trees are constructed over them separately. These Merkle trees are generated on demand and not stored. The FORS public key is derived as a single λ-bit hash value by applying Th_k to the roots of all k Merkle trees. Figure 2 illustrates the structure of FORS. For more details, we refer the reader to [7,51].

SPHINCS$^+$. This signature scheme consists of three polynomial-time algorithms (SPHINCS$^+$.Keygen, SPHINCS$^+$.Sign, SPHINCS$^+$.Verify), defined as:

- $(\mathsf{SK}, \mathsf{PK}) \leftarrow$ SPHINCS$^+$.Keygen$(1^\lambda, \mathsf{DGSP.SetPr})$: This key generation algorithm produces a key pair $(\mathsf{SK}, \mathsf{PK})$. The public key PK consists of two λ-bit strings: The root node of the top tree in the hypertree of SPHINCS$^+$, denoted by PK.root, and a random public seed, shown by PK.seed. The

secret key SK consists of two more λ-bit random seeds: $\mathsf{SK.seed}$, used for generating WOTS$^+$ key-pairs and FORS secret keys, and $\mathsf{SK.prf}$, employed to randomize the message digest. Thus, $\mathsf{PK} = (\mathsf{PK.root}, \mathsf{PK.seed})$ and $\mathsf{SK} = (\mathsf{SK.seed}, \mathsf{SK.prf}, \mathsf{PK})$.

– $\sigma^S \leftarrow \mathsf{SPHINCS}^+.\mathsf{Sign}(\mathsf{SK}, \mathrm{msg})$: This algorithm generates a signature

$$\sigma^S = (R, \sigma^F, \sigma^W, \mathit{Auth.path}),$$

where R is a pseudorandom value, σ^F is a FORS signature on a Message Digest MD of the message msg, σ^W is a WOTS$^+$ signature on the FORS public key, and $\mathit{Auth.path}$ is a set of authentication paths and WOTS$^+$ signatures derived from the SPHINCS$^+$ hypertree. This process starts with computing $\mathrm{R} = \mathsf{PRF}_{\mathrm{msg}}(\mathsf{SK.prf}, \mathsf{OptRand}, \mathrm{msg})$, where $\mathsf{OptRand}$ is an optional random string. Then, the message digest MD and the index Index are computed as:

$$\mathrm{MD} \parallel \mathsf{Index} = \mathsf{H}_{\mathrm{msg}}(\mathrm{R}, \mathsf{PK.seed}, \mathsf{PK.root}, \mathrm{msg}).$$

Index determines the index of the leaf node that is used for signing the message and ensures that the hypertree's leaf index is derived deterministically and publicly verifiable. This prevents arbitrary manipulation and providing strong protection against forgery attempts. The message digest MD is a ka-bit string, where each a-bit segment is signed using one of the k FORS trees. Subsequently, the public root of FORS is signed using the Index-th leaf node of the SPHINCS$^+$ hypertree, following the approach used in other hypertree-based signature schemes. See Figure 2 for a visual representation.

– $0/1 \leftarrow \mathsf{SPHINCS}^+.\mathsf{Verify}(\mathrm{msg}, \sigma^{\mathrm{S}}, \mathsf{PK})$: The verification process begins by computing $\mathrm{MD} \parallel \mathsf{Index} = \mathsf{H}_{\mathrm{msg}}(\mathrm{R}, \mathsf{PK.seed}, \mathsf{PK.root}, \mathrm{msg})$ and reconstructing the FORS public key from σ^F and MD. Next, the verifier reconstruct the potential SPHINCS$^+$'s root $\mathsf{PK.root}'$ using the computed FORS public key, Index, σ^W, and $\mathit{Auth.path}$. Finally, the verifier accepts the message-signature pair $(\mathrm{msg}, \sigma^{\mathrm{S}})$ by outputting 1 if $\mathsf{PK.root}' = \mathsf{PK.root}$.

The signature size of SPHINCS$^+$ is $(h + k \cdot (\log t + 1) + d \cdot \mathsf{len} + 1)\lambda$-bit, so for a fixed h, increasing d reduces subtrees' height, speeds up signing, and increases the signature size. For classical and quantum adversaries that make no more than q_h queries to the cryptographic hash function used and q_s signature queries to SPHINCS$^+$ oracle, the insecurity of SPHINCS$^+$ is respectively bounded by

$$\mathcal{O}\left(\frac{q_h}{2^\lambda} + q_h \sum_{0 \le \gamma \le q_s} \left(1 - \left(1 - \frac{1}{t}\right)^\gamma\right)^k \binom{q_s}{\gamma} \left(1 - \frac{1}{2^h}\right)^{q_s - \gamma} \frac{1}{2^{h\gamma}}\right),$$

and

$$\mathcal{O}\left(\frac{q_h^2}{2^\lambda} + q_h^2 \sum_{0 \le \gamma \le q_s} \left(1 - \left(1 - \frac{1}{t}\right)^\gamma\right)^k \binom{q_s}{\gamma} \left(1 - \frac{1}{2^h}\right)^{q_s - \gamma} \frac{1}{2^{h\gamma}}\right).$$

The most effective parameter in security of SPHINCS$^+$ is λ, with three defined security levels: $\lambda = 128$, $\lambda = 192$, and $\lambda = 256$, corresponding to NIST security levels 1, 3, and 5, respectively. Moreover, the hypertree height h directly influences security; taller hypertrees enhance security. In contrast, the parameters w and d do not impact security, but affect performance and signature size, so two optimization variants are considered: size-optimized (suffix "s" for small) and speed-optimized (suffix "f" for fast). In the size-optimized case, subtrees have uniform heights $h/d \in \{7, 8\}$, resulting in shorter signatures but slower signing. In contrast, the speed-optimized case uses subtree heights $h/d \in \{3, 4\}$, prioritizing faster signing at the cost of larger signature sizes. In the initial version of SPHINCS$^+$, the tweakable hash functions are instantiated using SHA2, SHAKE, or Haraka, and can follow either a *simple* or *robust* construction. Robust variants XOR pseudorandom bitmasks with input messages, enhancing security but reducing speed, while simple variants directly process inputs for better performance. Combining security levels, optimization variants, and hash function choices results in 36 possible configurations of SPHINCS$^+$, providing diverse performance and security requirements. For instance, "SPHINCS$^+$ – SHAKE – 256f – simple" denotes the use of SHAKE with a simple, fast structure and a 256-bit output size, corresponding to $\lambda = 256$.

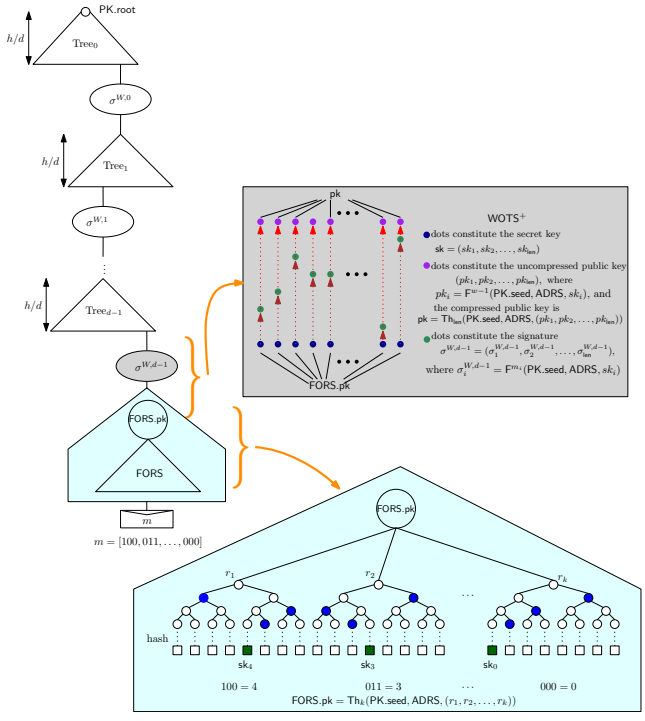

Fig. 2. An overview of SPHINCS$^+$

B Security Proofs

As we explain in Section 3, we follow the adversarial game model of [12,18] to establish the security properties of DGSP. This section presents the proof of Theorem 1, which is derived through Lemmas 1, 2, 3, and 4.

We will use the following straightforward lemma:

Lemma 5. *In any of our security games, an adversary who is restricted to calling* RespM *only with corrupted user IDs, loses only a factor* $\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda; \mathsf{Q}_{\mathsf{RespM}}) + \frac{\mathsf{Q}_{\mathsf{RespM}}}{2^{\lambda}}$ *compared to an adversary with full access to* RespM.

Proof. We see that if an adversary calls RespM with an id that is not a corrupted user, the manager will respond with $\perp$ unless the adversary also sends c^*_{id} such that $c^*_{\mathsf{id}} = \mathsf{H}(\mathsf{id} \| c_{\mathsf{id}})$. Since the adversary does not know c_{id}, we first replace these with fully random values (this game hop loses at most $\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda; \mathsf{Q}_{\mathsf{RespM}})$), and then the adversary only has a $\frac{\mathsf{Q}_{\mathsf{RespM}}}{2^{\lambda}}$ chance of ever guessing a correct value by the union bound.

We will not include this advantage loss in the lemmas below, but include it in the main text.

B.1 Proof of Lemma 1

In the correctness game $\mathbf{Exp}^{\mathrm{Corr}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$, a PPT adversary $\mathcal{A}$ can use the oracles AddHU and Revoke to add honest users and revoke users, CorruptHU to corrupt users, potentially all but one, and RespM to interact with the manager and update the states of the corrupted users. Despite the presence of such an adversary, an honest user can establish a secure channel with the manager and securely receive $(\mathsf{id}, c^*_{\mathsf{id}})$, ensuring that only the user is aware of $(\mathsf{id}, c^*_{\mathsf{id}})$.

For simplicity, we assume that the user asks certificate for only one WOTS$^+$ public key, i.e., $B = 1$. Thus, the honest user generates $(\mathsf{sk}_{\mathsf{id},j}, \mathsf{pk}_{\mathsf{id},j})$ using the secret seed $\mathsf{User.seed}_{\mathsf{id}}$ and randoms $r_{\mathsf{id},j}$ and $\rho_{\mathsf{id},j}$, then sends $(\mathsf{id}, c^*_{\mathsf{id}}, \mathsf{pk}_{\mathsf{id},j})$ to the manager through the secure channel. The manager then executes the subroutine DGSP.ResponseM to authenticate the user. Since the user is honest, the manager will accept their request and generates $(\zeta_{\mathsf{id},j}, \pi_{\mathsf{id},j}, \tau_{\mathsf{id},j}, \sigma^S_{\mathsf{id},j})$ and returns them to the user. Upon receiving $(\zeta_{\mathsf{id},j}, \pi_{\mathsf{id},j}, \sigma^S_{\mathsf{id},j})$, the user computes $\tau_{\mathsf{id},j}$. The user regenerates $(\mathsf{sk}_{\mathsf{id},j}, \mathsf{pk}_{\mathsf{id},j})$ using $\mathsf{User.seed}_{\mathsf{id}}$ and $r_{\mathsf{id},j}$ and produces a group signature

$$\sigma^{\mathrm{DGSP}}_{\mathsf{id},j} = \left(\sigma^W_{\mathsf{id},j}, \rho_{\mathsf{id},j}, \zeta_{\mathsf{id},j}, \sigma^S_{\mathsf{id},j}, \tau_{\mathsf{id},j}\right)$$

on a message msg. Since the user is honest

$$1 \leftarrow \mathsf{DGSP.Verify}(\mathrm{msg}, \sigma^{\mathrm{DGSP}}_{\mathsf{id,j}}, \mathsf{DGSP.PP}).$$

The honest manager outputs $(\mathsf{id}, \pi_{\mathsf{id},j}) \leftarrow \mathsf{DGSP.Open}(\mathrm{msg}, \sigma^{\mathrm{DGSP}}_{\mathsf{id,j}}, \mathsf{DGSP.SK})$. Clearly, $1 \leftarrow \mathsf{DGSP.Judge}((\mathsf{id}, \pi_{\mathsf{id},j}), \mathrm{msg}, \sigma^{\mathrm{DGSP}}_{\mathsf{id,j}})$ as $\sigma^S_{\mathsf{id},j}$ is a valid signature on $\mathsf{pk}_{\mathsf{id},j} \| \zeta_{\mathsf{id},j} \| \tau_{\mathsf{id},j}$ and H is second preimage resistant. □

B.2 Proof of Lemma 2

In the unforgeability security game $\mathbf{Exp}^{\text{Unforg}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$, a challenger first runs the $\mathsf{DGSP.KG}$ to create the secret and public keys of the system. The PPT adversary $\mathcal{A}$ is then given access to the oracles AddHU, $\mathsf{CorruptHU}$, RespM, SignHU, and Revoke. Therefore, $\mathcal{A}$ is allowed to obtain the signatures of every honest user on arbitrary messages. Each message-signature pair is stored in the signing list SL, preventing $\mathcal{A}$ from using these signatures as forgeries for their corresponding messages. $\mathcal{A}$ can revoke users, so their signatures will not be verified. Additionally, $\mathcal{A}$ can corrupt users to obtain their $\mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$s, update $\mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$s, and sign messages on their behalf, however, the generated signatures cannot be used as forgeries as those users are not honest.

Suppose that the adversary $\mathcal{A}$ has forged a signature

$$\sigma^{\text{DGSP}*} = \left(\sigma^{W*}, \rho^*, \zeta^*, \sigma^{S*}, \tau^*\right).$$

on a message msg^* at the end of the query phase, i.e.,

$$1 \leftarrow \mathsf{DGSP.Verify}(\text{msg}^*, \sigma^{\text{DGSP}*}, \mathsf{DGSP.PP}), \qquad (\text{msg}^*, \sigma^{\text{DGSP}*}) \notin \mathsf{SL},$$
$$(\mathsf{id}^*, \pi^*) \leftarrow \mathsf{DGSP.Open}(\text{msg}^*, \sigma^{\text{DGSP}*}, \mathsf{DGSP.SK}), \qquad \text{with } \mathsf{id}^* \in \mathcal{H}, \text{ and}$$
$$1 \leftarrow \mathsf{DGSP.Judge}((\mathsf{id}^*, \pi^*), \text{msg}^*, \sigma^{\text{DGSP}*}).$$

To upper-bound the adversary's success probability, we consider three cases:

1. $\mathcal{A}$'s forgery contains a SPHINCS$^+$ forgery: a signature for SPHINCS$^+$ for a message that $\mathcal{A}$ did not previously query;
2. $\mathcal{A}$'s forgery does not contain a SPHINCS$^+$ forgery, but does contain a WOTS$^+$ forgery: some msg^* and σ^{W*} that can be used to derive a public key pk^* such that pk^* was the output of a previous signature request, but σ^{W*} is not.
3. All other cases.

Clearly, the advantage of $\mathcal{A}$ is bounded above by the sum of the advantage of $\mathcal{A}$ in each of the three cases.

Case 1: We build an adversary $\mathcal{A}_{\text{SPHINCS}^+}$ against the unforgeability of SPHINCS$^+$. $\mathcal{A}_{\text{SPHINCS}^+}$ will play the role of a challenger in the DGSP unforgeability game. During calls to SignHU, $\mathcal{A}_{\text{SPHINCS}^+}$ will forward any challenges to the SPHINCS$^+$ challenger and use the signatures that are provided as a response. When the DGSP adversary produces its forgery, $\mathcal{A}_{\text{SPHINCS}^+}$ will simply present the message-signature pair $(\mathsf{pk}^* \| \zeta^* \| \tau^*, \sigma^{S*})$ as its signature forgery.

Clearly this is a perfect simulation for the DGSP adversary $\mathcal{A}$, and if $\mathcal{A}$ produces a forged signature against SPHINCS$^+$, $\mathcal{A}_{\text{SPHINCS}^+}$ wins its game. Thus, the probability of $\mathcal{A}$ winning in case 1 is at most $\mathbf{Adv}^{\text{EU}-\text{CMA}}_{\text{SPHINCS}^+}(\mathsf{Q}_{\mathsf{SignHU}})$.

Case 2: We will proceed through a series of games. Here we let Game_0 be the original unforgeability game, restricted to case 2 (i.e., we declare it a loss for the adversary if its forgery contains a WOTS$^+$ signature given by a SignHU query, or if the forgery contains a SPHINCS$^+$ signature that was *not* given by a SignHU query).

Since the SPHINCS$^+$ signature is not a forgery, we have $\sigma^{S*} = \sigma^S_{\mathsf{id},j}$ for some id, j, where $\sigma^S_{\mathsf{id},j}$ is a signature of some honestly-generated triple $\mathsf{pk}_{\mathsf{id},j}\|\zeta_{\mathsf{id},j}\|\tau_{\mathsf{id},j} = \mathsf{pk}^*\|\zeta^*\|\tau^*$. Because $\zeta_{id,j} = \mathsf{F}(\mathsf{DGSP.msk}_2, \mathsf{id}\|j)$, and this value id is extracted during DGSP.Open, the user id must belong to an uncorrupted honest user, as our unforgeability game requires that DGSP.Open opens the forged signature to an honest user.

We then hop to Game$_1$, which differs as follows: the challenger will select a random index i from 1 to $\mathsf{Q}_{\mathsf{AddHU}}$ (i.e., an upper-bound on the adversary's queries). The ith user generated from AddHU will be the "special" user. The special user will be simulated exactly like other users, except the adversary wins only if their forgery is based on the id which matches the special users.

This is perfectly indistinguishable to the adversary, except the adversary's advantage is multiplied by a factor of $\frac{1}{\mathsf{Q}_{\mathsf{AddHU}}}$, the probability that the correct special user is chosen.

Now we hop to Game$_2$, where we replace the input to WOTS$^+$ key generation for the special user with a uniformly random string, instead of derived from $(\mathsf{User.seed}_{\mathsf{id}}, r_{\mathsf{id},j})$ using H as a "key prefix" PRF. Distinguishing these two games is easily seen to be equivalent to winning the PRF game against H; a hybrid adversary will replace the WOTS$^+$ signing keys for this user with outputs of $\mathcal{O}^{\mathsf{PRF}}(k, r_{\mathsf{id},j})$ and this will match Game$_1$ or Game$_2$ depending on whether this is a PRF or a truly random function. This would be distinguishable if the adversary corrupted the special user; however, the adversary cannot win in this case since their forgery (which must belong to the special user thanks to the hop to Game$_1$) must belong to an uncorrupted user. Hence, $|\mathbf{Adv}^{G_1}_{\mathcal{A}} - \mathbf{Adv}^{G_2}_{\mathcal{A}}| \leq \mathbf{Adv}^{\mathsf{PRF}}_F(\mathsf{Q}_{\mathsf{SignHU}})$.

Then we can directly use Game$_2$ to build an adversary $\mathcal{A}_{\text{WOTS}^+}$ against WOTS$^+$. It will play the role of the challenger against $\mathcal{A}$. It will randomly select an index j from 1 to $\mathsf{Q}_{\mathsf{SignHU}}$, and on the jth query that $\mathcal{A}$ makes to SignHU for the special user, the challenger will replace the public key of this signature with the public key of the WOTS$^+$ challenge and use its one chosen message query for M, derived from the message msg that $\mathcal{A}$ provides to SignHU and the randomly selected $\rho_{\mathsf{id},j}$.

When (and if) $\mathcal{A}$ presents its forgery, $\mathcal{A}_{\text{WOTS}^+}$ will derive $M^* = \mathsf{H}(\rho^*\|\text{msg}^*)$ and supply (M^*, σ^{W*}) as the forgery for WOTS$^+$, where $\text{msg}^* \neq \text{msg}$. With probability at least $\frac{1}{\mathsf{Q}_{\mathsf{SignHU}}}$, this injects the challenge public key into precisely the WOTS$^+$ signature for which $\mathcal{A}$ will produce a forgery and without requiring more than one chosen message query to the WOTS$^+$ challenger.

This is indistinguishable to the adversary, as the signing keys for this user were selected uniformly at random. Thus, since this is case 2, the DGSP forgery contains a WOTS$^+$ forgery: the value of M^* derived above does not match the value M that was queried in the WOTS$^+$ sEU-CMA security game to create $\sigma^W_{\mathsf{id},j}$.

Altogether, this tells us that

$$\mathbf{Adv}^{\text{Case2}}_{\mathcal{A}} \leq \mathsf{Q}_{\mathsf{AddHU}}\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\mathsf{Q}_{\mathsf{SignHU}}) + \mathsf{Q}_{\mathsf{AddHU}}\mathsf{Q}_{\mathsf{SignHU}}\mathbf{Adv}^{\text{sEU}-\text{CMA}}_{\text{WOTS}^+}.$$

Case 3: As argued for Case 2, since there is no SPHINCS$^+$ forgery, the components of the forgery must match $\sigma^* = \sigma^S_{\mathsf{id},j} = \text{SPHINCS}^+.\mathsf{Sign}(\mathsf{sk}, \mathsf{pk}_{\mathsf{id},\mathsf{j}} \| \zeta_{\mathsf{id},\mathsf{j}} \| \tau_{\mathsf{id},\mathsf{j}})$, where $\zeta_{\mathsf{id},j} = \zeta^*$ and $\tau_{\mathsf{id},j} = \tau^*$ (otherwise we would have a SPHINCS$^+$ forgery), and id must correspond to an honest user.

Firstly, during normal certificate generation, $\zeta_{id,j} = \mathsf{E}(\mathsf{DGSP.msk}_2, \mathsf{id} \parallel j)$ and $\tau_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \parallel \pi_{\mathsf{id},j} \parallel \mathsf{id})$, so DGSP.Open will output $(\mathsf{id}, \pi_{\mathsf{id},j})$ such that $\mathsf{DGSP.Judge}$ outputs 1 on input id.

From msg*, ρ^*, and σ^{W*}, the challenger can uniquely determine a public key pk^*. This must equal $\mathsf{pk}_{\mathsf{id},j}$ or else the signature will not verify (or the SPHINCS$^+$ signature was forged). Since we have assumed $\mathcal{A}$ does not forge a WOTS$^+$ signature, this means $M^* = M_{\mathsf{id},j}$, where $M_{\mathsf{id},j} = \mathsf{H}(\rho_{\mathsf{id},j} \| \text{msg})$ and $M^* = \mathsf{H}(\rho^* \| \text{msg}^*)$ for some msg that was input to a SignHU query. We know that $\rho^* = \rho_{\mathsf{id},j}$ thanks to the signature $\sigma^* = \sigma^S_{\mathsf{id},j}$. Thus, the adversary's only chance to win in this case is for msg $\neq$ msg*, but $\mathsf{H}(\rho_{\mathsf{id},j} \| \text{msg}) = \mathsf{H}(\rho_{\mathsf{id},\mathsf{j}} \| \text{msg}^*)$.

This is effectively keyed second preimage resistance, except that $\rho_{\mathsf{id},j}$ is not uniformly random: it is the output of $\mathsf{H}(r_{id,j})$. Thus, we need to make the same game hop as in Case 2 to guess the "special" user for whom the adversary forges a signature (losing a factor of $\mathsf{Q}_{\mathsf{AddHU}}$ in doing so), then make a game hop where we replace $\rho_{\mathsf{id},j}$ for all j for the special user id, with uniformly random strings. Distinguishing this hop clearly reduces to the pseuorandomness of the PRF (using an empty string as the "message"), since $r_{\mathsf{id},j}$ is unknown to the adversary for the special user.

Finally, we see that if the adversary wins in this new game in this case, the adversary has broken the keyed second preimage resistance of the hash function $\mathsf{H}(\rho \| \text{msg})$. We can easily construct a reduction by using the Eval oracle in the kSPR game to sample r for the signature query. This means the adversary's probability of winning in the case 3 is at most $\mathbf{Adv}^{\mathsf{kSPR}}_{\mathsf{H}}(\lambda, \mathsf{Q}_{\mathsf{SignHU}})$.

Combining all of the terms through the game hops gives

$$\begin{aligned}\mathbf{Adv}^{\text{Unforg}}_{\text{DGSP}}(\lambda) \leq\ & \mathbf{Adv}^{\text{sEU}-\text{CMA}}_{\text{SPHINCS}^+}(\lambda, \mathsf{Q}_{\mathsf{SignHU}}) + 2\mathsf{Q}_{\mathsf{AddHU}}\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, \mathsf{Q}_{\mathsf{SignHU}}) \\ & + \mathsf{Q}_{\mathsf{AddHU}}\mathsf{Q}_{\mathsf{SignHU}}\mathbf{Adv}^{\text{sEU}-\text{CMA}}_{\text{WOTS}^+}(\lambda) + \mathsf{Q}_{\mathsf{AddHU}}\mathbf{Adv}^{\mathsf{kSPR}}_{\mathsf{H}}(\lambda, \mathsf{Q}_{\mathsf{SignHU}})\end{aligned}$$

□

B.3 Proof of Lemma 3

In the traceability game $\mathbf{Exp}^{\text{Trace}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$, the challenger runs the $\mathsf{DGSP.KG}$ to generate the secret and public keys of the system. The PPT adversary $\mathcal{A}$ has access to oracles AddHU, Revoke, $\mathsf{CorruptHU}$, RespM, and SignHU oracles and can add and revoke users, corrupt them to obtain their $\mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$s, update their $\mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$s, and obtain all signatures of honest users. At the end of the query phase, the adversary $\mathcal{A}$ outputs a valid message-signature pair $(\text{msg}, \sigma^{\text{DGSP}})$, where

$$\sigma^{\text{DGSP}} = \left(\sigma^W, \rho, \zeta, \sigma^S, \tau\right)$$

and $\mathsf{DGSP.Open}(\sigma^{\text{DGSP}}, \mathsf{DGSP.SK}) = \perp$.

We first consider the case where the adversary never received the above tuple from a query. More precisely, we use σ^W and msg to derive some WOTS$^+$ public key $\mathsf{PK}^{\mathrm{WOTS}^+}$; we suppose the adversary never received $(\mathsf{PK}^{\mathrm{WOTS}^+}, \zeta, \sigma^S, \tau)$ during the query phase.

In this case, if the message-signature pair $(\mathrm{msg}, \sigma^{\mathrm{DGSP}})$ is valid, it follows that we can build an adversary $\mathcal{A}_{\mathrm{SPHINCS}^+}$ against the unforgeability of SPHINCS$^+$. $\mathcal{A}_{\mathrm{SPHINCS}^+}$ will play the role of a challenger in the DGSP traceability game. During calls to SignHU, $\mathcal{A}_{\mathrm{SPHINCS}^+}$ will forward any challenges to the SPHINCS$^+$ challenger and use the signatures that are provided as a response. When the DGSP adversary produces the valid message-signature pair $(\mathrm{msg}, \sigma^{\mathrm{DGSP}})$, $\mathcal{A}_{\mathrm{SPHINCS}^+}$ will simply present $(\mathsf{PK}^{\mathrm{WOTS}^+} \| \zeta \| \tau, \sigma^S)$ as its SPHINCS$^+$ forgery. Clearly, this is a perfect simulation for $\mathcal{A}$, and if $\mathcal{A}$ produces a valid message-signature pair $(\mathrm{msg}, \sigma^{\mathrm{DGSP}})$ against SPHINCS$^+$, $\mathcal{A}_{\mathrm{SPHINCS}^+}$ wins its game. Thus, the probability of $\mathcal{A}$ winning the traceability game in this case is at most $\mathbf{Adv}^{\mathrm{sEU-CMA}}_{\mathrm{SPHINCS}^+}(\mathsf{Q}_{\mathsf{SignHU}})$.

We now consider the case when $(\mathsf{PK}^{\mathrm{WOTS}^+}, \zeta, \sigma^S, \tau) = (\mathsf{pk}_{\mathsf{id},j}, \zeta_{\mathsf{id},j}, \sigma^S_{\mathsf{id},j}, \tau_{\mathsf{id},j})$ for some query response from the group manager. Here $\mathsf{DGSP.Open}$ will not output $\perp$, so if the adversary wins the traceability game, then $\mathsf{DGSP.Judge}$ must output 0.

Again, because SPHINCS$^+$ is unforgeable, we know that the signature corresponds to $\mathsf{pk}_{\mathsf{id},j} \| \zeta_{\mathsf{id},j} \| \tau_{\mathsf{id},j}$ from some interaction with an honest user $\mathsf{id} \in \mathcal{H}$. This means $\tau_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \| \pi_{\mathsf{id},j} \| \mathsf{id})$ and $\pi_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{\mathsf{id},j} \| c_{\mathsf{id}})$, where $c_{\mathsf{id}} = \mathsf{H}(\mathsf{DGSP.msk}_1, \mathsf{id})$.

Thus, the challenger will find c_{id} and then $\pi_{\mathsf{id},j}$ by H. They will then be able to supply $\pi_{\mathsf{id},j}$ during the opening.

During $\mathsf{DGSP.Judge}$, the signature must produce $\mathsf{pk}_{\mathsf{id},j}$ that was included in the signature $\sigma^S_{\mathsf{id},j}$ (as the traceability game requires a *valid* signature), and so the judge algorithm obtains $\mathsf{pk}_{\mathsf{id},j}$, $\pi_{\mathsf{id},j}$ and id as they were input into H to compute $\tau_{\mathsf{id},j}$. Since this matches $\tau_{\mathsf{id},j}$, $\mathsf{DGSP.Judge}$ will output 1.

Throughout all of this, the only way to win the traceability game is with a forged signature $\sigma^S_{\mathsf{id},j}$. Hence, the overall advantage is at most $\mathbf{Adv}^{\mathrm{sEU-CMA}}_{\mathrm{SPHINCS}^+}(\mathsf{Q}_{\mathsf{SignHU}})$. □

B.4 Proof of Anonymity 4

This property means that a group signature does not reveal it signer's identity. In other words, if a PPT adversary $\mathcal{A}$ selects a message msg and two honest users id_0 and id_1, each with at least one unused key and its corresponding certificate, then $\mathcal{A}$ has only a negligible advantage in determining which of the two users generated a given signature on msg.

In the anonymity security game $\mathbf{Exp}^{\mathrm{Anon}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$, a PPT adversary $\mathcal{A}$ has access to the oracles AddHU, $\mathsf{CorruptHU}$, RespM, Revoke, SignHU, Ch_b, and Open. The adversary $\mathcal{A}$ is allowed to add honest users to the group and to revoke users from the group. It can also corrupt honest users to obtain their $\mathsf{state}^{\mathcal{U}}_{\mathsf{id}}$s, update it,

and sign messages on their behalf. Additionally, $\mathcal{A}$ can use the oracle SignHU to obtain signatures from honest users on arbitrary messages. Furthermore, $\mathcal{A}$ can use the oracle Open to find the identity of the signer for any message-signature pair.

At some stage of the game, $\mathcal{A}$ calls Ch_b and selects a message msg along with two honest users, identified as id_0 and id_1, each possessing at least one unused key and its corresponding certificate. $\mathcal{A}$ then sends identifiers id_0 and id_1 to the challenger, who verifies whether id_0 and id_1 are honest and have at least one unused key with a valid certificate. If these conditions are not met, $\mathcal{A}$ loses the game. The challenger computes

$$(\sigma^{\mathrm{DGSP}}_{\mathsf{id}_i,j}, \mathsf{state}^{\mathcal{U}}_{\mathsf{id}_i}) \leftarrow \mathsf{DGSP.Sign}(\mathsf{state}^{\mathcal{U}}_{\mathsf{id}_i}, \mathrm{msg}),$$

where $\sigma^{\mathrm{DGSP}}_{\mathsf{id}_i,j} = \left(\sigma^{W}_{\mathsf{id}_i,j}, r_{\mathsf{id}_i,j}, \zeta_{\mathsf{id}_i,j}, \sigma^{S}_{\mathsf{id}_i,j}, \tau_{\mathsf{id}_i,j}\right)$, for $i \in \{0,1\}$, and selects a random bit $b \in \{0,1\}$ Finally the challenger sends $\sigma^{\mathrm{DGSP}}_{\mathsf{id}_b,j}$ to $\mathcal{A}$ to determine whose signature it is. $\mathcal{A}$ has still access with oracles AddHU, $\mathsf{CorruptHU}$, RespM, Revoke, SignHU, and Open. $\mathcal{A}$ cannot call the oracle Open on the signatures generated by Ch_b, one of which is stored in the challenge list CL and one of which $\mathcal{A}$ never received. Also, $\mathcal{A}$ cannot invoke the oracle Revoke on users with identifiers id_0 or id_1, as this would allow $\mathcal{A}$ to distinguish the signer by searching $\zeta_{\mathsf{id}_b,j}$ in the revocation list before and after calling the oracle Revoke on one of these identifiers. Finally, $\mathcal{A}$ outputs a bit b' and wins the anonymity experiment if $b' = b$.

In the following, we proceed through a sequence of games to show that $\sigma^{\mathrm{DGSP}}_{\mathsf{id}_b,j}$ reveals only negligible information about the signer's identity.

Game$_0$*:* This game is the original experiment $\mathbf{Exp}^{Anon}_{\mathrm{DGSP},\mathcal{A}}(\lambda)$ that outputs a bit. The adversary $\mathcal{A}$ has advantage

$$\mathbf{Adv}^{\mathrm{G}_0}_{\mathcal{A}}(\lambda) = \mathbf{Adv}^{\mathrm{Anon}}_{\mathrm{DGSP},\mathcal{A}}(\lambda)$$

in this game, which is defined similarly to $\mathbf{Exp}^{\mathrm{Unforg}}_{\mathcal{SFDGS},\mathcal{A}}(\lambda)$

Game$_1$*:* In this game, we sample c_{id} uniformly at random for all users. Treating $\mathsf{H}(k\|x)$ as a PRF, the adversary's ability to distinguish the two games is bounded above by the security of the PRF, via a straightforward hybrid where all calls to $\mathsf{H}(\mathsf{DGSP.msk}_1\|\mathsf{id})$ are replaced by sending id to a PRF oracle. Notice that this function is called both during AddHU and Open, so the probability of distinguishing these games is bounded by $\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, \mathsf{Q}_{\mathsf{AddHU}} + \mathsf{Q}_{\mathsf{Open}})$. To simulate decryption during Open, we maintain a table of the input-output pairs.

Game$_2$: In this game we replace all values of $\zeta_{\mathsf{id},j}$ with uniformly random, non-repeating values, which we select at signing time. As this effectively replaces E with a truly random permutation, Game$_2$ and Game$_1$ are indistinguishable up to $\mathbf{Adv}^{\mathsf{SPRP}}_{\mathsf{E}}(\lambda, \mathsf{Q}_{\mathsf{SignHU}})$.

Game$_3$*:* In this game we sample two random values i and j from 1 to AddHU, and declare the ith and jth users created to be "special" users. The adversary loses if they do not query id_i and id_j for the anonymity challenge. This is a perfect simulation except for the adversary's loss of $\frac{1}{\mathsf{Q}^2_{\mathsf{AddHU}}}$.

Game$_4$: Here we replace $\pi_{\mathsf{id},j}$ with uniformly random values instead of $\pi_{\mathsf{id},j} = \mathsf{H}(\mathsf{pk}_{\mathsf{id},j}\|c_{\mathsf{id}})$, and c^*_{id} with uniformly random values, for the two special users. To

ensure we can still correctly respond to trace queries, we make a table mapping values of $\pi_{\mathsf{id},j}$ to users; however, we do not add $\pi_{\mathsf{id}_b,j}$ to this table during the anonymity challenge.

Here we can prove that this game is indistinguishable by PPF security of $\mathsf{H}(x\|c_{\mathsf{id}})$, where the values of c_{id} are playing the role of the PRF keys. Technically this is two hybrid games: one for special user i, and another for special user j. The adversary's advantage in distinguishing Game_4 is bounded by $2\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, \mathsf{Q}_{\mathsf{SignHU}})$.

Game$_5$: Here we have the special users run $\mathsf{DGSP.RequestU}$, including generating random $r_{\mathsf{id},j}$, only when the adversary calls SignHU. This is indistinguishable to the adversary, since the adversary does not see this interaction and only sees the signature/certificate from one of the two users, and both users delete their certificate after signing during Chal.

Game$_6$: Now we replace $\rho_{\mathsf{id},j} = \mathsf{H}(r_{\mathsf{id},j})$ and the seed input to WOTS$^+$ (equal to $\mathsf{H}(\mathsf{User.seed}_{\mathsf{id}} \parallel r_{\mathsf{id},j})$) with uniformly random values, for all $r_{id,j}$ of the two special users. This is effectively $2\mathsf{SignHU}$ game hops, each reducing to the key-suffix PRF security of H with exactly 2 queries, thus adding a $2\mathsf{SignHU}\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, 2)$ advantage loss.

Game$_7$: In this game we have the two special users use the same random values $\rho_{\mathsf{id},j}$ and the WOTS$^+$ key during signing for the challenge. This is indistinguishable to the adversary: the signature not returned to the adversary is deleted, as is the ephemeral randomness $r_{\mathsf{id},j}$ and $\rho_{\mathsf{id},j}$. This ensures that the WOTS$^+$ key and the randomness $\rho_{\mathsf{id},j}$ are independent of which user is selected.

We now argue that the adversary's advantage in Game_7 is 0. The values $\pi_{\mathsf{id}_b,j}$, $\zeta_{\mathsf{id}_b,j}$, $\rho_{\mathsf{id}_b,j}$ are now selected entirely independently of b and independently of all other user data. The WOTS$^+$ secret keys – and hence $\mathsf{pk}_{\mathsf{id}_b,j}$ and $\sigma^W_{\mathsf{id}_b,j}$ – are also selected independently of b and the user data. Since $\sigma^S_{\mathsf{id}_b,j}$ and $\tau_{\mathsf{id}_b,j}$ are deterministic functions of the other components of the signature, they are also independent of b. Thus, the adversary's view in the game is independent of b and so its advantage is 0.

Once the user deletes $\rho_{\mathsf{id}_b,j}$ and $r_{\mathsf{id}_b,j}$ (which is done after signing), the state of both challenge users is independent of the signatures.

Combining all of the terms through the game hops gives

$$\begin{aligned}\mathbf{Adv}^{\text{Anon}}_{DGSP}(\lambda) \leq\ & \mathbf{Adv}^{\mathsf{SPRP}}_{\mathsf{E}}(\lambda, \mathsf{Q}_{\mathsf{SignHU}}) + \mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, \mathsf{Q}_{\mathsf{AddHU}} + \mathsf{Q}_{\mathsf{Open}}) \\ & + 2\mathsf{Q}^2_{\mathsf{AddHU}}\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, \mathsf{Q}_{\mathsf{SignHU}}) + 2\mathsf{Q}^2_{\mathsf{AddHU}}\mathsf{Q}_{\mathsf{SignHU}}\mathbf{Adv}^{\mathsf{PRF}}_{\mathsf{H}}(\lambda, 2).\end{aligned}$$

□

B.5 Keyed Second Preimage Resistance

In [7], they use a notion of *interleaved target subset resilience*, which plays the same role as keyed second preimage resistance for our scheme: when we hash a message before signing, we do not want collisions in these digests, but there is randomness in the digest to prevent basic collision attacks.

Interleaved target subset resilience is defined for a keyed hash function $\mathsf{H} : \{0,1\}^\lambda \times \{0,1\}^\alpha \to \{0,1\}^m$, and an mapping function MAP, whose outputs we will not be concerned with. The function G is defined as $\mathrm{MAP} \circ H$. An adversary is given access to an oracle which takes messages m as input, randomly samples $k \in \{0,1\}^\lambda$, and returns k and $G(k,m)$. The adversary must produce a pair (k', m') such that, letting all previous queries and responses be (k_i, m_i), we have that $(k', m') \neq (k_i, m_i)$ for any i but

$$G(k', m') \subseteq \bigcup_{i=1}^{q} G(k_i, m_i).$$

We prove the following easy lemma:

Lemma 6. *For any keyed hash function* H *and mapping function MAP,*

$$\mathbf{Adv}^{\mathsf{kSPR}}_{\mathsf{H}}(\lambda, \mathsf{Q}) \leq \mathbf{Adv}^{\mathsf{ITSR}}_{\mathsf{H},MAP}(\lambda, \mathsf{Q}).$$

Proof. Suppose we have a kSPR adversary $\mathcal{A}$. We use it to build a ITSR adversary as follows: whenever $\mathcal{A}$ calls Eval on a message m, we send m to the ITSR oracle. The oracle responds with $(k, G(k,m))$; we can replace $G(k,m)$ with $\mathsf{H}(k,m)$. Eventually $\mathcal{A}$ responds with some (k', m') such that it has received k' as a query response before, but it did not receive k' in response to a query for m', and where $\mathsf{H}(k', m') = \mathsf{H}(k_i, m_i)$ for some previous query. In this case, it is clear that $G(k', m') = G(k_i, m_i)$, so this pair is also a winning response for the ITSR game. □