Sphinx-in-the-Head: Group Signatures from Symmetric Primitives

Liqun Chen* Changyu Dong† Christopher J. P. Newton*
Yalan Wang*

Abstract

Group signatures and their variants have been widely used in privacy-sensitive scenarios such as anonymous authentication and attestation. In this paper, we present a new post-quantum group signature scheme from symmetric primitives. Using only symmetric primitives makes the scheme less prone to unknown attacks than basing the design on newly proposed hard problems whose security is less well-understood. However, symmetric primitives do not have rich algebraic properties, and this makes it extremely challenging to design a group signature scheme on top of them. It is even more challenging if we want a group signature scheme suitable for real-world applications, one that can support large groups and require few trust assumptions. Our scheme is based on MPC-in-the-head non-interactive zero-knowledge proofs, and we specifically design a novel hash-based group credential scheme, which is rooted in the SPHINCS+ signature scheme but with various modifications to make it MPC (multi-party computation) friendly. The security of the scheme has been proved under the fully dynamic group signature model. We provide an implementation of the scheme and demonstrate the feasibility of handling a group size as large as 2^{60}. This is the first group signature scheme from symmetric primitives that supports such a large group size and meets all the security requirements.

1 Introduction

Group signatures are a fundamental cryptographic primitive proposed by Chaum and van Heyst in 1991 [20]. A group signature scheme allows one to generate a signature anonymously on behalf of a group. A verifier can determine whether or not the signature was generated by a legitimate member of the group, but cannot identify who generated it. The ability to conceal the signer's identity without hurting the authenticity of the signature makes group signatures an attractive building block in privacy-sensitive applications such as anonymous

*University of Surrey, {liqun.chen, c.newton, yalan.wang}@surrey.ac.uk
†Guangzhou University, changyu.dong@gzhu.edu.cn

credential [17], trusted computing using Direct Anonymous Attestation (DAA) [14] or Enhanced Privacy Identification (EPID) [15], and digital rights management [39].

In light of the threat quantum computing poses to current public-key algorithms based on hard problems such as RSA or discrete logarithm, at the moment the design of group signature schemes is undergoing a transition to post-quantum security. There have been proposals for new group signature schemes based on lattice problems [7, 11, 12, 13, 23, 25, 30, 36, 37, 42, 45, 46, 47, 48, 49, 50], isogenies [7, 21, 43], code [1, 26, 53], multivariate [56, 60], and symmetric key primitives [3, 8, 16, 38, 59, 61, 62]. Each approach for obtaining quantum-resistant signatures has its pros and cons. Among all post-quantum approaches, the symmetric key approach is considered the most conservative approach. The security of symmetric primitives is the most well-understood and easiest to evaluate, hence it serves as a safety net if the security of other approaches is endangered by newly discovered threats. It is therefore the focus of this paper.

The multifaceted security and functional requirements make it difficult to design a group signature scheme, it is even more so when we have to restrict ourselves to only symmetric primitives. The foremost security requirement of a group signature scheme is anonymity. Currently, there are two pathways for achieving anonymity. The first is to use a zero-knowledge proof ([8, 38]). The main part of the group signature is a Non-Interaction Zero-Knowledge (NIZK) proof that asserts two things: the signer possesses a secret signing key, and the key is certified with a group credential from an entity who manages the group membership (the group manager). The main challenge in this pathway is that the group credential needs to be verified with zero-knowledge. A group credential is essentially another signature generated by the manager when the user joins the group. It is bound to the user's identifier and is fixed after generation, so the user cannot simply include it in every group signature they generate because it will destroy anonymity and make the group signatures linkable. Running the credential verification algorithm with zero-knowledge is possible, but not always feasible. This is especially true for signatures based on symmetric primitives, which do not have rich algebraic properties that can be utilized for constructing zero-knowledge proofs. The other pathway is to use One-Time Signatures (OTS) ([3, 16, 59, 61, 62]). Each time when a user needs to generate a group signature, they have to obtain from the manager a randomly generated one-time credential that is used as a part of a one-time signing key. This however requires excessive communication and interaction between the users and the manager and an unrealistic assumption that the manager is always online. Although a user can request a batch of credentials rather than just one in each interaction, it only alleviates the problem, not solving it.

Another important security property is non-frameability. It means that even if an adversary fully corrupts the rest of the group as well as the group manager, the adversary cannot falsely attribute a signature to an honest member who did not produce it. Up to now, all group signature schemes based on symmetric primitives do not support non-frameability. In existing schemes using NIZK,

a tracing key for each user is shared with the group manager, which allows the manager to trace a given signature back to this signer. Although each signer has another key that is not known by the manager, this will not stop a malicious group manager from forging a group signature under an honest user u_h's tracing key along with an arbitrary key generated by the manager. This forged signature will be traced back to u_h. For schemes based on OTS, there are two cases. In Case 1, the group manager knows the user's signing key and can generate a signature on the user's behalf. In Case 2, a user generates their own secret key and the manager adds the corresponding public key into a Merkle tree (either a single tree or multiple trees). The manager maintains the state of the key use. A malicious manager can let an honest user u_h's public key be associated with multiple states and allow further signatures to be generated once u_h's secret key is revealed. This occurs when u_h uses the key to generate a signature. After that, the manager can create another valid signature with a different state. This forged signature will be traced back to u_h. No-one can tell which of these signatures was the one generated by u_h.

In practice, we often need group signature schemes that can support large group sizes. For example, Direct Anonymous Attestation (DAA) [14], which is implemented in every Trusted Platform Module (TPM) since 2003 and distributed with every PC produced since 2006, is a variant of a group signature. Another example is Enhanced Privacy Identification (EPID) [15] by Intel, which is also a variant of a group signature. EPID has been included in many Intel processors since 2008, and in 2016 it was announced that Intel has distributed over 4.5 billion EPID keys since 2008[1] [32]. However, for existing group signature schemes based on symmetric primitives, the group size that they can support is often small, less than 2^{20}.

There are two approaches for membership management. In the first approach, adopted by the majority of the existing schemes, group membership is managed through a single Merkle tree such that each leaf corresponds to a group member. The group public key is dependent on all leaves. For a large group, the time for setting up the group and generating the group public key is extremely long and the space for storing the Merkle tree is also prohibitively large. The second approach, adopted by some OTS-based schemes [61, 62], uses multiple trees, which can provide a large number of leaves and each leaf corresponds to a one-time group membership credential. Due to the nature of an OTS, each credential can only be used to generate one signature. Therefore, a large number of credentials does not automatically translate into a large group size. If each member needs to sign many signatures (e.g. as in DAA and EPID), then the group size cannot be very large: let N be the total number of credentials, and B be the number of group signatures that each group member can make, then the group size is $\lfloor N/B \rfloor$.

Also, a fully dynamic group signature scheme is often preferred, i.e. the

[1]Intel does not put all processors into one group, so in practice, groups are smaller than 4.5 billion. There is a trade-off between strong anonymity and group size. A large group size is beneficial if we want strong anonymity. Ideally, applications like these should have a group size of 2^{40} or above.

Table 1: A comparison of hash-based group signature schemes

schemes	underlying signatures	group credentials	group types	implemented group size[a]	non-frameability
G-Merkle [3]	OTS	Merkle signature	static	2^6	no
DGM[b] [16]	OTS	Merkle signature	dynamic	–	no
DGM$^+$ [62]	OTS	XMSS-T	dynamic	–	no
GM^{MT} [61]	OTS	XMSS-T	dynamic	2^{16}	no
SE[c] [59]	OTS	hash pool	static	–	no
KKW [38]	NIZK	Merkle signature	static	2^{13}	no
BEF [8]	NIZK	Merkle or Goldreich signature	static	–	no
this work	NIZK	F-SPHINCS+	dynamic	2^{60}	yes

[a] It refers to the maximum group size that has been implemented and reported in the paper; "–" indicates that no implementation has been reported.

[b] In this scheme, the group issuer needs to be involved in signature verification.

[c] The security of this scheme is held under the condition of non-colluding members.

group membership is not decided and fixed at the setup phase, and the users can join and leave at any time. In the symmetric setting, for those schemes that rely on a single Merkle tree, the membership is static and fixed when generating the Merkle tree at the setup stage; for those OTS-based schemes, although they appear to be dynamic, they lack support for persistent group membership: each user joins the group on the fly when signing and leaves the group after signing (because the group credential is for one-time only).

Contributions of this paper In this work, we have responded to the above challenges by designing a new NIZK-based group signature scheme from symmetric primitives. Our scheme supports the security notion of fully dynamic group signatures [10] and can handle a large group size. This is the first symmetric setting group signature scheme to meet all these requirements. We also implemented a proof of concept to show the feasibility of the scheme. We have made the following choices in our design:

- *Underlying signatures.* As mentioned before, two types of underlying signatures have been used in group signatures from symmetric primitives: an OTS and a NIZK proof of the knowledge. Although signing and verification are efficient, the OTS approach has a major drawback: the communications between the issuer and the group members are heavy and the issuer's workload is high. Hence in this work, we have chosen the NIZK approach.

- *Group credentials.* To support large group size while retaining reasonable efficiency, we have designed a new variant of the SPHINCS+ signature scheme [6] allowing efficient NIZK (based on MPC-in-the-Head), which provides us with a new group membership credential. We name this new hash-based signature scheme, F-SPHINCS+ (see Subsection 3.1). F-SPHINCS+ is constructed on top of M-FORS, which is a modification of the FORS signature [6]. This variant of SPHINCS+ may have its independent interest.

- *Group types.* Based on F-SPHINCS+, we designed protocols to allow group members to join and leave the group at any time, i.e. we support fully dynamic groups.

- *Non-frameability.* Non-frameability is a desirable security property, however, due to the lack of rich algebraic properties in symmetric primitives, achieving it is challenging. In this work, we decided to design a scheme with non-frameability. To achieve this, we split the group management function into two parts and assign roles to a group issuer and a group tracer. Assuming that there is no collusion between them, we can prove that non-frameability holds in our scheme. Note that non-frameability is not supported in KKW [38] and BEF [8], but potentially it could be achieved using our technique (splitting the group manager into an issuer and a tracer). However this will require significant changes to these two schemes thus is not trivial.

- *Group size and implementation.* The majority of the existing group signature schemes from symmetric primitives are without implementation. For those schemes with implementation, they have only reported the implementation with small group sizes. We have implemented our designed scheme, and our implementation demonstrates that our scheme can handle a very large group size, 2^{60}. This size can meet real-world requirements, such as those for TPM DAA and EPID.

Comparison with the related work As mentioned before, in the literature, there are two types of group signatures derived from symmetric primitives. In Table 1, we compare our proposed scheme with the existing ones. Below We organize the discussion by dividing the schemes into two types based on the underlying signatures.

In the first type, including the schemes from [3, 16, 59, 61, 62], a group signature consists of an OTS and a group membership credential. In [3, 16, 59], such a credential is a Merkle signature [51], while in [61, 62], it is an XMSS-T signature [31]. In [59], group members' signing keys are organized as a hash pool. As a result, the security of this scheme is under the condition that group members will not collude with each other.

In the second type, including the schemes from [8, 38], a group signature uses a NIZK proof of the knowledge of a group membership credential. Each credential is a hash-based signature. In [38], the credential is a Merkle signature. A user has a key pair that forms a Merkle tree leaf and one of them is used as a tracing key. The group master secret key gmsk is all of the users' tracing keys. An adversary controlling gmsk can create a different Merkle tree leaf using an honest user u_h's tracing key and then use this leaf to forge a signature, which will be traced back to u_h. In [8], the authors state that in their scheme either a Merkle signature or a Goldreich signature [29] can be used to create credentials. For traceability, the group manager gives each group member a signed secret token as a tracing key. An adversary controlling the manager can create a

signature using an honest user u_h's tracing key, again this signature will be traced back to u_h.

Construction of the paper In Section 2, we introduce the preliminary material: several signature schemes based on symmetric primitives, some of which directly influenced our proposed schemes, and the concept and security properties of a group signature scheme. In Section 3, we present the constructions of our new schemes, starting from the main building blocks F-SPHINCS+ and M-FORS, then the group signature scheme and its underlying NIZK proof. In Section 4, we provide the security analysis and proofs. In Section 5, we provide a summary of our implementation and performance figures. To make the paper more readable, we put low-level details in Appendices (supplementary online material): (A) a review of hash-based signature schemes; (B) the detail of M-FORS algorithms; (C) tweakable hash functions; (D) the soundness analysis of its underlying NIZK proof; and finally (E) more information of our implementation.

2 Preliminaries

2.1 Hash-based Signature

It has long been known that signature schemes can be constructed purely on top of cryptographic hash functions. Without relying on number theoretical hard problems, hash-based signatures are believed to be secure against a cryptanalytic attack by a quantum computer. Hence they have attracted more and more attention in cryptography research in recent years. Here we briefly introduce some basic ideas in hash-based signatures. A more detailed review can be found in Appendix A.

A hash-based signature scheme is a public key scheme such that a public key is publicized to everyone and a private key is known only to the signer. Usually the private key is a set of randomly generated strings and the public key is derived by applying hash functions on the private key. Early hash-based signatures [44, 52] are one-time signatures (OTS), which means each key pair can be used to sign only one message. Examples include the Lamport signature scheme [44] and the Winternitz one-time signature (WOTS) scheme [52]. A simple strategy used first in the Merkle signature scheme [52] to extend the signing capability to multiple messages (few-time signatures, FTS) is to generate multiple OTS key pairs and aggregate the OTS public keys using a Merkle tree. The Merkle tree root is released as the overall public key. Each signature will consume one OTS secret private key. The signature consist of a OTS and the Merkle tree authentication path for the OTS public key, so that the verifier can verify the signature with only the Merkle tree root. More recent FTS schemes (e.g. FORS [6]) can be more efficient. The strategy is to have a large set of secret random strings, which can be derived using a pseudorandom function from the private key, then the signature is generated by selecting some elements from the set which is determined by the message to be signed. Although each

signature reveals some secret strings in the set, the set is large so that as long as the number of signatures is controlled below a threshold, forging a signature by mix-and-match secret strings from previously generated signatures is infeasible.

All previously mentioned multi-time signature schemes are stateful, meaning that the signer needs to keep a state (e.g. how many messages have been signed and which keys have been used). SPHINCS+ [6] is a stateless hash-based signature scheme and is one of the three digital signature schemes selected by NIST to become part of its post-quantum cryptographic standard [55]. Technically, SPHINCS+ still has an upper limit on how many signatures can be generated per key pair, it is just that the number can be made very large (e.g. 2^{60}) so that it is unlikely to be reached in practice. SPHINCS+ uses a hyper-tree, i.e. a tree of trees, to organize OTS and FTS key pairs. Each SPHINCS+ is a chain of signatures, such that the first signature σ_0 in the chain is a signature generated from the message, and each of the subsequent signature σ_i is a signature of the public key that verifies σ_{i-1}. With the root public key, the verifier can verify the authticity of the signature chain. It is conceptually similar to PKI: the root CA and intermediate CAs signs the public keys of CAs one level below them in the hierarchy, and the CAs at the lowest level signs the user's public key. A signature generated by an unknown user can be verified by verifying the signature itself and all the signatures along the path leading to a trusted root CA.

2.2 PICNIC Signature Scheme

Another signature scheme that relies on only symmetric primitives is PICNIC [19]. This scheme relies on a zero-knowledge proof technique called MPC-in-the-head. The scheme we propose in this paper works with any PICNIC style signature. When necessary, in our description, we use PICNIC as an example, although we do not recommend using the PICNIC scheme in real usage.

MPC-in-the-head This is a paradigm for zero-knowledge proofs introduced by Ishai et. al. [33]. Roughly speaking, given a public value x, the prover needs to prove knowing a witness w such that $f(w) = x$. To do so, the prover simulates, by itself, an MPC (multi-party computation) protocol between m parties that realizes f, in which w is secretly shared as an input to the parties. After simulation, the prover commits to the views and internal state of each individual party. Next, the verifier challenges the prover to open a subset of these commitments, checks them, and decides whether to accept or not. If the MPC realizes f properly, then obviously this protocol is complete, meaning a valid statement will always be accepted. The protocol is also zero-knowledge because only the views and internal states of a subset of the parties are available to the verifier, and by the privacy guarantee of the underlying MPC protocol, no information about w can be leaked. For soundness, if the prover tries to prove a false statement, then the joint views of some of the parties must be inconsistent, and with some probability, the verifier can detect that. The soundness error of a single MPC run can be high, but by repeating this process independently enough times, the soundness error can be made negligible. The interactive ZK

proofs can be made non-interactive through techniques such as the Fiat-Shamir transformation.

There are multiple frameworks for constructing MPC-in-the-head ZK proofs, eg. IKOS [33], ZKBoo [28], ZKB++ [19], KKW [38], BN [4] and Limbo [22]. They follow the same paradigm, but are different in the underlying MPC protocols and have different concrete/asymptotic efficiency. There are also MPCitH frameworks, e.g. BN++ [35], Rainer [24] and AIMer [41], focusing on proofs of AES (or its variants) encryption, that are useful in Picnic style signatures. In this paper, to describe our scheme, we do not need to touch the low-level details, hence we will use MPC-in-the-head (for Boolean circuits) in an abstract way. We will use the following syntax to describe a ZK proof:

$$\pi = \mathcal{P}\{(\texttt{public params}); (\texttt{witness}) | \texttt{relation to be proved}\}$$

For example, to prove the same key sk is used in two different instantiations of a pseudorandom function F with different data inputs, we write:

$$\pi_1 = \mathcal{P}\{(C_1, P_1), (C_2, P_2)); (sk) | C_1 = F(sk, P_1) \wedge C_2 = F(sk, P_2)\}$$

PICNIC Signature Many signature schemes (e.g. Schnorr [58]) boil down to a non-interactive zero-knowledge proof of knowing the signing key used in generating the signatures. PICNIC is in the same direction. In PICNIC, the public key is a pair (C, p) such that $C = E(k, p)$ where E is a block cipher, k is a secret key, and p is a plaintext block. The private key is k. Signing essentially is to generate a non-interactive MPC-in-the-head proof of knowing/possessing the private key:

$$\pi = \mathcal{P}\{(C, p)); (k) | C = E(k, p)\}$$

such that π is parsed in (r, s), and the internal challenge used in the above proof is generated by $H(r, pk||m)$ where H is a hash function and m is the message to be signed. The signature is π. Verification of the signature is then verifying π with regenerated challenges $H(r, pk||m)$.

2.3 Group Signatures

A group signature scheme [20] allows users in a group to sign messages such that the signatures can be verified using a group public key, and the actual signers' identities are not revealed (beyond the fact that they belong to the group). A typical group signature scheme involves the following players:

- **A manager** manages the group membership and performs the related functionalities. In our scheme, we split the manager role into two:
  - **An issuer** decides who can be a group member and issues group credentials.
  - **A tracer** can trace a group signature back to its signer when needed.
- **Group members** create group signatures.
- **Verifiers** verify group signatures.

- **A revocation authority** decides which group member should be removed from the group.
- **A judge** verifies tracing results.

Following the definition of a fully dynamic group signature scheme [10], the group signature scheme proposed in this paper consists of the following algorithms/protocols:

- $\mathsf{Init}(n)$: In the initialization algorithm, the issuer takes a security parameter n as the input, and outputs a master (group) key pair (mpk, msk). The master public key mpk is made public and the master secret key is stored privately by the group issuer. In all other group signature algorithms/protocols, we will assume mpk as an implicit input for all parties. The issuer, tracer, and revocation authority also initialize their internal states.
- $\mathsf{Join}(msk, n)$: the group-joining protocol is an interactive protocol between the issuer, the tracer, and the user who wants to join the group. The issuer has a private input msk and the other parties do not have input. There is a public input that is the security parameter n. At the end of the protocol, the issuer outputs a decision: `accept` or `reject`. If `reject`, then stop. If `accept`, then the user obtains their signing key $gsk_u = (sk_u, tk_u, cred_u)$ where sk_u is a secret key, tk_u is a tracing key, and $cred_u$ is a group credential. Both sk_u and tk_u are chosen by the user, and $cred_u$ is generated by the issuer. The issuer and the tracer also update their internal states.
- $\mathsf{GSig}(gsk_u, msg)$: the group signature generation algorithm allows a group member to produce a signature Σ on a message $msg \in \{0,1\}^*$ using its signing key gsk_u.
- $\mathsf{GVf}(msg, \Sigma, \mathbf{RL})$: the group signature verification algorithm allows anyone who has access to the group public key mpk (as an implicit input) to verify whether a signature Σ is a valid signature of msg and whether the group signing key has been revoked (with a revocation list $\mathbf{RL}$).
- $\mathsf{Trace}(msg, \Sigma)$: In the tracing algorithm, the tracer outputs either a pair (id_u, π_t) or an error symbol $\perp$, based on a valid signature Σ and its internal state, where id_u is the identifier of the group member who produces Σ, and π_t is a proof of this claim.
- $\mathsf{TVf}(id_u, \pi_t, \Sigma, msg)$: Given $(id_u, \pi_t, \Sigma, msg)$, the judge verifies π_t. If this proof is valid, the judge outputs `accept`; otherwise outputs `reject`.
- $\mathsf{Revoke}(tk_u)$: The revocation authority maintains and publishes a revocation list $\mathbf{RL}$. The user can be removed from the group by the authority adding tk_u to the revocation list (as a consequence, this revokes the group signing key).

A group signature scheme needs to satisfy multiple security requirements [10], including:

- **Correctness** Correctness covers two aspects: (1) an honest user can successfully join the group, despite the existence of other malicious users; (2) a signature generated by an honest group member should always be valid when being verified (if the member has not been revoked).
- **Anonymity** Anonymity means that a group signature does not reveal the identity of its signer, i.e., the adversary cannot distinguish which one of the two honest signers has signed a targeted message while both signers and the message are at the adversary's choice.
- **Traceability** Traceability ensures that a group member (even malicious) can be traced by the group tracer through a valid signature, i.e. the tracer can output a convincing proof showing that the signature was signed by the group member.
- **Non-frameability** Non-frameability means that even if the rest of the group as well as the issuer/revocation authority are fully corrupted, they cannot falsely attribute a signature to an honest member who did not produce it.
- **Tracing Binding** Tracing binding [57] guarantees that even if all authorities and users collude, they should not be able to produce a valid signature that can be selectively attributed to a different member, no matter whether this member is honest or one that already colluded.
- **Tracing Soundness** Tracing soundness guarantees that even if all authorities are corrupted, they cannot attribute a signature generated by an honest user to a corrupted user.

3 Construction

3.1 F-SPHINCS+ and M-FORS

Design rationale The first design choice we need to make is how to achieve anonymity: by using a one-time key/credential for each signature or by using a long-term key/credential with zero-knowledge proof? The benefit of the one-time key approach is mainly its efficiency in signature generation and verification. However, it also limits itself to application scenarios with small groups, infrequent signatures, and well-connected networks. Hence targeting more practical usage, our group signature opts for the zero-knowledge proof approach.

The second design choice is about group credentials. A group credential essentially is a signature on the user's keys generated by the issuer. Because we use only symmetric primitives, the credential can be in the form of the following: (1) a Merkle signature; (2) a SPHINCS+ style signature; (3) a PICNIC-style signature. The first option is ruled out easily because, as discussed before, it cannot handle a large group size. The last option is ruled out because of practical consideration: we have to create a zero-knowledge proof that another zero-knowledge proof (i.e. the PICNIC signature) is valid. Unfortunately, the circuit

for verifying a PICNIC-style signature is too big, which results in prohibitively high computation cost and/or large proof size. Therefore, we focused on utilizing a SPHINCS+ style signature as the group credential.

In the above, we said "SPHINCS+ style" rather than "SPHINCS+". This is because SPHINCS+ is still too heavy when being verified in zero knowledge. The main problem comes from the WOTS+ signature scheme. In WOTS+, verification involves verifying k blocks of d-bit strings. When verified in the clear, each block requires at most $2^d - 1$ hash operations to verify and the exact number of hash operations required depends on the content of the block. However, in a zero-knowledge proof, we will have to hash each block exactly $2^d - 1$ times and then choose the right hash value in the chain blindly, to ensure the verifier is oblivious about the content of the block. Hence in total, $(2^d - 1) \cdot k$ hashes are required to verify a WOTS+ signature. Plug in concrete parameters, which means 510 hashes at 128-bit security, and 990 at 256-bit security. The circuit implementing the hash function typically has 10^3 AND gate. So verifying one WOTS+ signature requires a circuit with over a million AND gates and in total, we need to verify h WOTS+ signatures, where h is at least 7 in SPHINCS+.

To fix the problem, we propose a new variant of SPHINCS+ called F-SPHINCS+. As depicted in Fig. 1, in F-SPHINCS+ we use a hyper-tree that is a tree of M-FORS trees. The M-FORS signature scheme is depicted in

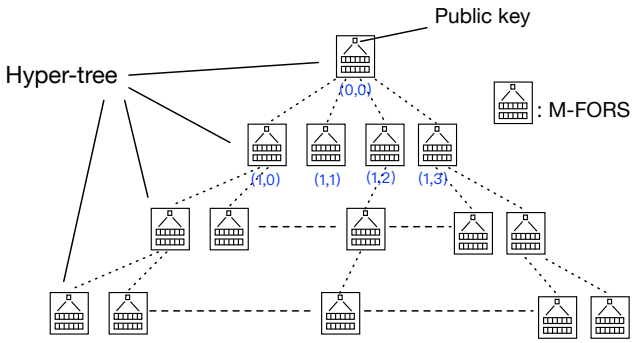


Figure 1: F-SPHINCS+ signatures. The addresses of the nodes in the first two layers are shown in blue text.

Fig. 2 and this is our modification of FORS. Recall that FORS is a few-time signature scheme such that each key pair can be used to sign up to q signatures. M-FORS, short for Merkle FORS, differs from FORS in that the public key is generated as the root of a Merkle tree. The leaf nodes in this Merkle tree are the root nodes of Merkle trees that authenticate each block of the hash value being signed. So with M-FORS, the hyper-tree in F-SPHINCS+ is a q-ary tree such that the public key in a child node is signed by the signing key in the parent node, and the signing key in the leaf node signs the actual message hash. An F-SPHINCS+ signature then contains a list of $h + 1$ signatures, where h is the

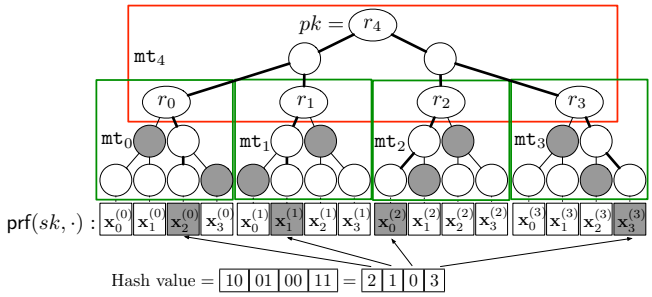


Figure 2: M-FORS signatures for $k = 4$ and $d = 2$.

height of the hyper-tree. The benefit of M-FORS over XMSS that is used in the original SPHINCS+ scheme is the lower verification cost. To verify a message hash that is k blocks of d-bit string, the cost is $d \cdot k + k - 1$ hash operations. This is much less than the $(2^d - 1) \cdot k$ hashes for verifying a WOTS+ signature. On the other hand, the signing time is more than that of WOTS+. However, this is a lesser concern because in our case signing will be done in the clear (while verification needs to be done with zero knowledge).

The schemes We now describe M-FORS and F-SPHINCS+. M-FORS consists of the algorithms below. For readability, we abstract away certain low-level details such as how the Merkle trees are built. A more algorithmic description of M-FORS can be found in Appendix B.

- $\mathsf{keyGen}(\mathtt{seed}, n, d, k, \mathtt{aux})$: it takes as input a random seed $\mathtt{seed}$, a security parameter n, two positive integers d and k, and $\mathtt{aux}$ that is either an empty string or some optional data. If $\mathtt{seed}$ is an empty string, an n-bit random string will be chosen and assigned to it. Then a pseudorandom function prf is used to expand sk into k lists $(\mathbf{x}^{(0)}, \cdots, \mathbf{x}^{(k-1)})$, where each $\mathbf{x}^{(i)}$ contains 2^d distinct n-bit pseudorandom strings.

  Then $k + 1$ Merkle trees $\mathbf{T} = (\mathtt{mt}_0, \cdots, \mathtt{mt}_k)$ are built. In particular, each of $\mathtt{mt}_0, \cdots, \mathtt{mt}_{k-1}$ has 2^d leaf nodes. The jth leaf node in $\mathtt{mt}_i$ is the hash of $\mathbf{x}_j^{(i)}$. The leaf nodes of $\mathtt{mt}_k$ are $r_0, \cdots, r_{k-1}$ that are the roots of $(\mathtt{mt}_0, \cdots, \mathtt{mt}_{k-1})$.

  keyGen outputs $(pk, sk, param)$, such that the public key $pk = r_k$ where r_k is the root of $\mathtt{mt}_k$, the private key $sk = \mathtt{seed}$, and the public parameters $mp = (n, d, k, \mathtt{aux})$.

- $\mathsf{sign}(sk, MD, mp)$: to sign a message hash $MD \in \{0,1\}^{k \cdot d}$, parse it into k blocks, each block is interpreted as a d-bit unsigned integers $(p_0, \cdots, p_{k-1})$. Then for the i-th block p_i, $\mathbf{x}^{(i)}$ and $\mathtt{mt}_i$ (obtained by expanding sk) are used to generate $\mathtt{authpath}^{(i)}$, which is the authentication path of the p_i-th leaf node in the i-th Merkle tree. Then $(\mathbf{x}_{p_i}^{(i)}, \mathtt{authpath}^{(i)})$ is put into the signature. The signature is a list of k pairs $\boldsymbol{\sigma} = \{(\mathbf{x}_{p_0}^{(0)}, \mathtt{authpath}^{(0)}), \cdots, (\mathbf{x}_{p_{k-1}}^{(k-1)}, \mathtt{authpath}^{(k-1)})\}$.

- $\mathsf{recoverPK}(\sigma, MD, mp)$: This algorithm outputs the public key recovered from a signature σ and the message hash MD. First MD is parsed into k blocks $(p'_0, \cdots, p'_{k-1})$. Then for $0 \leq i \leq k-1$, $\boldsymbol{\sigma}_i = (x_i, \mathtt{authpath}^{(i)})$ and p'_i are used to re-generate a Merkle tree root and get the value r'_i (p'_i is used to determine the order of the siblings at each layer). Finally, $r'_0, \cdots, r'_{k-1}$ are used to compute $\mathtt{mt}'_k$ and its root r'_k is returned.

- $\mathsf{verify}(\sigma, pk, MD, mp)$: to verify a signature, call $\mathsf{recoverPK}(\sigma, MD, mp)$. If the recovered public key is the same as pk, accept the signature, otherwise reject.

The hyper-tree nodes in F-SPHINCS+ are addressed by a pair (a, b) where a is its layer and b is its index within the layer. The root node is at layer 0, and the layer number of all other nodes is the layer number of its parent plus 1. All nodes within a layer are viewed as an ordered list, and index each node in the list is from left to right, starting from 0. F-SPHINCS+ consists of the following algorithms:

- $\mathsf{keyGen}(n, q, h)$: This algorithm outputs (sk, pk, fp). It takes as input a security parameter n, the degree of non-leaf nodes in the hyper-tree q, and the height of the hyper-tree h. Then it chooses d, k which are the parameters for the underlying M-FORS signature scheme. The public parameters are $fp = (n, q, h, d, k)$. It also chooses an n-bit random string as the private key sk. It generates the M-FORS key pair for the root node by calling $\mathsf{genNode}((0,0), sk, fp)$, and setting the public key pk to be the M-FORS public key $pk_{0,0}$.

- $\mathsf{genNode}(nodeAdr, sk, fp)$: This algorithm generates a node in the hyper-tree given the address $nodeAdr = (a, b)$. With the private key sk, the algorithm first generates a subseed with a pseudorandom function $\mathtt{seed}_{a,b} = \mathsf{prf}(seed, a||b)$, then it calls M-FORS key generation algorithm $\mathsf{M\text{-}FORS.keyGen}$ $(\mathtt{seed}_{a,b}, n, d, k, a||b)$. The output $(pk_{a,b}, sk_{a,b}, mp_{a,b})$ is the content of the node at (a, b).

- $\mathsf{mHash}(msg, gr)$: This algorithm produces message hash and the leaf node index used in generating the F-SPHINCS+ signature. The input msg is the message to be signed and gr is a random string. The algorithm produces $MD||idx \leftarrow H_{msg}(msg||gr)$, where $H_{msg} : \{0,1\}^* \rightarrow \{0,1\}^{d \cdot k + (\log_2 q) \cdot h}$ is a public hash function, MD is $d \cdot k$ bit long and idx is interpreted as an $(\log_2 q) \cdot h$ bit long unsigned integer.

- $\mathsf{sign}(msg, sk, fp)$: This algorithm produces the F-SPHINCS+ signature as a chain of M-FORS signature along the path from a leaf node to the root node of the hyper-tree. It chooses an n-bit random string gr. Then obtain $MD||idx \leftarrow \mathsf{mHash}(msg, gr)$. A leaf node at (h, idx) is then generated by calling $\mathsf{genNode}((h, idx), sk, fp)$. The M-FORS signing key $sk_{h,idx}$ is used to sign MD and generate σ_0. The parent node of (h, idx) is then generated by calling $\mathsf{genNode}((h-1, b), sk, fp)$ where $(h-1, b)$ is the address of the parent

node. Then the parent secret key $sk_{h-1,b}$ is used to sign the child public key $pk_{h,idx}$, and the signature is σ_1. Repeat the signing process until obtaining σ_h that is signed by $sk_{0,0}$ on $pk_{1,b'}$ for some b'. The F-SPHINCS+ signature is then $\Sigma = (gr, (\sigma_0, \cdots, \sigma_h))$.

- $\mathsf{verify}(msg, \Sigma, pk, fp)$: This algorithm verifies every M-FORS signature chained up in Σ. Given $\Sigma = (gr, (\sigma_0, \cdots, \sigma_h))$, first compute $MD||idx \leftarrow H_{msg}(msg||gr)$. Then obtain $pk_0 \leftarrow \mathsf{recoverPK}(\sigma_0, MD, mp_0)$, $pk_1 \leftarrow \mathsf{recoverPK}(\sigma_1, pk_0, mp_1)$, repeat until $pk_h \leftarrow \mathsf{recoverPK}(\sigma_h, pk_{h-1}, mp_h)$. If $pk = pk_h$, accept the signature, otherwise reject.

Remark 1 In M-FORS algorithms (see Appendix B), we use two tweakable hash functions [6] (see also Appendix C) $H_1 : \{0,1\}^* \to \{0,1\}^n$ and $H_2 : \{0,1\}^* \to \{0,1\}^{d \cdot k}$. Almost all hash operations are done using H_1. H_2 is only used to map the k-th Merkle tree to the $k \cdot d$-bit M-FORS public key, so that when used in F-SPHINCS+ the public key is of the right size to be signed by the parent node. If M-FORS is to be used as a stand-alone signature scheme, the two hash functions can be the same.

Remark 2 The tweakable hash functions follow Construction 7 for tweakable hash functions in [6]. Namely, the hash of an input M is produced by calling a hash function with additional input as $H(\mathtt{P}||\mathtt{ADD}||M)$, where $\mathtt{P}$ is a public hash key and $\mathtt{ADD}$ is the tweak. The tweak is the address where the hash operation takes place within the hyper-tree, and it is a five-part string $a_1||b_1||v||a_2||b_2$:

- (a_1, b_1), where $0 \leq a_1 \leq h, 0 \leq b_1 \leq 2^{a_1} - 1$, is the address of an hyper-tree node. Within the node, an M-FORS key pair that is based on $k + 1$ Merkle trees is stored.

- $0 \leq v \leq k$ is the index of a Merkle tree in the M-FORS key pair stored in the hyper-tree node (a_1, b_1). When $0 \leq v \leq k - 1$, the Merkle tree (of height d) is used to sign the v-th block of the message; when $v = k$, the Merkle tree (of height $\lceil \log_2 k \rceil$) is used to accumulate the roots of all the previous Merkle trees into the public key.

- (a_2, b_2) is the address of a Merkle tree node. When $0 \leq v \leq k-1$, $0 \leq a_2 \leq d$ and $0 \leq b_2 \leq 2^{a_2} - 1$; When $v = k$, $0 \leq a_2 \leq \lceil \log_2 k \rceil - 1$ and $0 \leq b_2 \leq 2^{a_2} - 1$.

We defer the security analysis of F-SPHINCS+ to section 4.1.

3.2 The Group Signature

Design rationale In the previous section, we decided to use a SPHINCS+ style signature for the group credential. Now we need to decide what to be used by the users to sign the actual messages. The first requirement is that the user should be able to sign many signatures with one key pair. This excludes the OTS scheme. Between SPHINCS+ and PICNIC, PICNIC wins because it has no limit on the number of signatures, and it is already based on NIZK so can be integrated easily with the NIZK for proving the group credential.

Then we need to consider other security properties required by group signatures. Anonymity, as mentioned earlier, will be addressed using NIZK. It is trickier for traceability and non-frameability. Since we rely solely on symmetric primitives, we cannot achieve traceability through a trapdoor function, i.e. by revealing to the tracer a piece of data generated from the signing secret key so that the tracer can trace a signature without knowing the signing key. However giving the signing key to the tracer, the non-frameability is violated because now the tracer can sign on the user's behalf. To address this problem, we first split the group manager into two entities, a group issuer and a group tracer, and then let the user to generate two keys when joining the group, a tracing key and a secret signing key. Both keys will be authorized by the group credential and will be used by the user in generating a signature. The tracing key is given to the tracer, which allows the tracer to test whether a signature is generated by a particular user. With only the tracing key, the tracer cannot sign on the user's behalf. Without knowing the tracing key, the issuer cannot create another secret signing key associated with the tracing key, therefore cannot forge a signature that will be traced back to the user.

Overall, the group signature scheme is designed in this way: the group issuer generates an F-SPHINCS+ key pair as the group master key pair. When a user joins the group, it generates a tracing key and a secret signing key. The tracing key is given to the tracer. The issuer and tracer decide together whether the user should be admitted into the group, if so a group credential is generated as an F-SPHINCS+ signature on an entry token (a commitment of the user's tracing and signing keys). When signing a message, the user produces an MPCitH NIZK to show it possesses a group credential and the signature is generated on the hash of the message (sid) under the keys authorized by the group credential. Verifying the group signature involves checking the NIZK so the verifier is convinced of the group membership. Each group signature also includes a tracing token, essentially it is the ciphertext of sid produced using the tracing key. The tracer, when asked, can decrypt the tracing token with a user's tracing key to check whether the result matches sid, if so the signature must have been generated by this user.

The scheme We now give the concrete construction of the group signature scheme.

- **Initialization** $\mathsf{Init}(n)$: Given a security parameter n, the group issuer does the following:
  - Choose a pseudorandom function prf, three hash functions $H_1 : \{0,1\}^* \to \{0,1\}^n$, $H_2 : \{0,1\}^* \to \{0,1\}^{d \cdot k}$, $H_3 : \{0,1\}^* \to \{0,1\}^{d \cdot k + (\log_2 q) \cdot h}$, and a keyed pseudorandom function $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$. H_1 and H_2 are used in the underlying M-FORS signature scheme, and H_3 will be used as H_{msg} in the mHash algorithm.
  - Decide the hyper-tree node degree q and the tree height h, then run the key generation algorithm $(sk, rpk, gp) \leftarrow \mathsf{F\text{-}SPHINCS+.keyGen}(n, q, h)$, where (rpk, sk) is the F-SPHINCS+ key pair, $gp = (n, q, h, d, k)$ are the hyper-tree parameters.

– Publish $mpk = (gp,\ rpk,\ H_1,\ H_2,\ H_3,\ F,\ \mathsf{prf})$ and keep $msk = sk$ private.

In addition, the group issuer initializes a group list **GL**, the group tracer initializes a group tracing list **TL**, and the group revocation authority initializes a revocation list **RL**. These lists are empty when initialized.

- **Group-joining protocol** $\mathsf{Join}(msk, n)$: To join a group, the user, the issuer and the tracer run the following protocol:

  1. A unique session ID u is assigned to the user. For simplicity, we can think of the session ID as a monotonically increasing counter, and each invocation of the Join protocol will increase it by 1.
  2. The user u chooses two random secret keys: $(tk_u,\ sk_u) \xleftarrow{R} \{0,1\}^n \times \{0,1\}^n$, where tk_u is the user's tracing key and sk_u is a secret signing key.
  3. With mpk, the user computes the group identifier $gid = H_1(rpk)$, its user identifier $id_u = F(tk_u,\ gid)$, and an entry token $et_u = F(sk_u,\ id_u)$. The user then chooses an n-bit random string cr and computes a commitment $ct = H_1(et_u||cr)$. The user produces an NIZK π_u that is instantiated with MPCitH 1:

  $$\pi_u : \mathcal{P}\{(gp,\ gid,\ id_u, ct); (tk_u,\ sk_u, cr) | id_u = F(tk_u,\ gid)$$

  $$\wedge\ ct = H_1(F(sk_u,\ id_u)||cr)\}$$

  The user sends $(u,\ tk_u)$ to the group tracer and $(u,\ id_u,\ ct,\ \pi_u)$ to the group issuer to request to join the group.

  Note that in MPCitH 1, $[\![x]\!]$ means that the value x is secret-shared when using an MPC algorithm. The prover knows all the shares but the verifier does not. Therefore the verifier cannot reconstruct x. MPC_X means the MPC subroutine implementing the function X (e.g. MPC_F and MPC_H1 implement F and H_1). These notations will be used throughout the paper.

MPCitH 1: π_u

Public: $gp,\ gid,\ id_u,\ ct$
Private: $[\![tk_u]\!],\ [\![sk_u]\!],\ [\![cr]\!]$
Output: $id'_u,\ ct'$
Check: $id'_u = id_u \wedge ct' = ct$

1 $id'_u = \mathsf{MPC\_F}([\![tk_u]\!],\ gid)$;
2 $[\![et_u]\!] = \mathsf{MPC\_F}([\![sk_u]\!],\ id'_u)$;
3 $ct' = \mathsf{MPC\_H1}([\![et_u]\!]||[\![cr]\!])$

  4. Upon receiving $(u,\ tk_u)$, the tracer first checks whether an entry with the same u or tk_u is in **TL**. If yes, the tracer rejects the user; otherwise, the tracer computes $id_u = F(tk_u,\ gid)$. The tracer waits and if the group issuer sends u, replies with id_u. Then if the issuer acknowledges with `Accept`, add $(u,\ id_u,\ tk_u)$ to **TL**; if `Reject`, discard the entry.
  5. Upon receiving $(u,\ id_u,\ ct,\ \pi_u)$, the issuer:

- Checks whether an entry with the same u or id_u is in $\mathbf{GL}$. If yes, rejects the user.
- Otherwise, verifies the NIZK π_u. If invalid, rejects the user.
- Otherwise, the issuer sends u to the tracer, who then sends back the corresponding id_u from $\mathbf{TL}$. If id_u from the user and the tracer are different, reject the user and send `Reject` to the tracer.
- Otherwise, if the issuer would like to accept the user, the issuer chooses a random string $gr_u \xleftarrow{R} \{0,1\}^n$ and sends it to the user. The user responds by sending (et_u, cr) back. The group issuer verifies $ct = H_1(et_u||cr)$. If so, compute the group credential $\mathbf{S} \leftarrow \mathsf{F\text{-}SPHINCS+.sign}(et_u, gr_u, msk, gp)$. Otherwise, the issuer rejects the user and sends `Reject` to the tracer.
- The group issuer adds $(u,\ id_u,\ et_u, gr_u, \mathbf{S})$ to $\mathbf{GL}$, sends $\mathbf{S}$ to the user, and `Accept` to the tracer.

6. The user, if accepted by the issuer, sets its group membership secret key $gsk_u = (tk_u,\ sk_u,\ gr_u,\ \mathbf{S})$.

- **Group signature generation** $\mathsf{GSig}(gsk_u, msg)$: To produce a group signature on a message msg, using $gsk_u = (tk_u,\ sk_u, gr_u,\ \mathbf{S})$:

1. The user first computes the signature identifier $sid = H_1(msg||str)$, where msg is the message to be signed and str is an n-bit random string. Then the user produces the signature tracing token $stt = F(tk_u,\ sid)$ and signature signing token $sst = F(sk_u,\ sid)$.
2. The user then computes $com = H_1(sst||pk_h||\cdots||rpk)\}$ where $pk_h, \cdots, rpk$ are the public keys for verifying the signatures in $\mathbf{S}$, from the layer h to layer 0 (the public key at the layer 0 is rpk).
3. The signature $\Sigma = (str,\ stt,\ com,\ \pi_{\mathtt{G}})$, where $\pi_{\mathtt{G}}$ is an NIZK. Roughly, the proof $\pi_{\mathtt{G}}$ asserts that the user indeed knows a valid group signing key and uses that in generating the signature. "Valid" means both the user's tracing key and signing key have been authorized by the group issuer (through the signature chain $\mathbf{S}$).
   To generate $\pi_{\mathtt{G}}$, the user computes $gid = H_1(rpk)$, $id_u = F(tk_u, gid)$, $et_u = F(sk_u,\ id_u)$, $mt_u||idx = \mathsf{F\text{-}SPHINCS+.mHash}(et_u, gr_u)$, then produces $\pi_{\mathtt{G}}$ as (details of $\pi_{\mathtt{G}}$ will follow in Section 3.3):

$$\begin{aligned}\pi_{\mathtt{G}} : \mathcal{P}\{&(gp,\ rpk,\ gid,\ sid,\ stt,\ com); (tk_u,\ sk_u,\ et_u,\ sst,\ gr_u,\ \mathbf{S} = \{\sigma_h, \cdots, \sigma_0\})| \\ & stt = F(tk_u,\ sid) \wedge sst = F(sk_u,\ sid) \wedge et_u = F(sk_u,\ F(tk_u,\ gid)) \\ & \wedge mt_u||idx = H_3(et_u||gr_u) \wedge pk_h = \mathsf{recoverPK}(\sigma_h, mt_u, (n,d,k,(h,idx))) \\ & \wedge pk_{h-1} = \mathsf{recoverPK}(\sigma_{h-1}, pk_h, (n,d,k,(h-1, \lfloor \frac{idx}{q} \rfloor))) \wedge \cdots \\ & \wedge rpk = \mathsf{recoverPK}(\sigma_0, pk_1, (n,d,k,(0,0))) \wedge com = H_1(sst||pk_h||\cdots||rpk)\}\end{aligned}$$

- **Group signature verification** $\mathsf{GVf}(msg, \Sigma, \mathbf{RL})$: Given the message msg, the signature $\Sigma = (str, stt, com, \pi_{\mathtt{G}})$, and the revocation list $\mathbf{RL}$, the verifier

performs the follows: computes $sid = H_1(msg||str)$ and $gid = H_1(rpk)$. If $\exists tk_u \in \mathbf{RL}$ such that $stt = F(tk_u,\ sid)$, then reject. Otherwise, verify $\pi_{\mathtt{G}}$. Accept the signature if $\pi_{\mathtt{G}}$ verification succeeds, otherwise reject.

- **Tracing algorithm** $\mathsf{Trace}(msg, \Sigma)$: The tracer maintains a tracing key list **TL** including all $(u,\ id_u,\ tk_u)$ triples in the group. Given $\Sigma = (str,\ stt,\ com,\ \pi_{\mathtt{G}})$, the tracer computes $sid = H_1(msg||str)$, then for $\exists tk_u \in \mathbf{TL}$ computes $stt' = F(tk_u,\ sid)$. If there is an $stt' = stt$, the tracer outputs the corresponding id_u. If no matching stt' is found, the tracer outputs $\perp$.

  In the case that an id_u is output in the last step, the tracer retrieves the whole $(u,\ id_u,\ tk_u)$ triple from **TL**, also takes sid and stt from the signature, then computes $gid = H_1(rpk)$ and a tracing proof π_t:

$$\pi_t = \mathcal{P}\{(id_u, gid,\ stt,\ sid); (tk_u) | id_u = F(tk_u,\ gid) \wedge stt = F(tk_u,\ sid)\}$$

  The above proof bounds id_u to the signature by showing that id_u and stt (in the signature) were generated using the same tracing key tk_u. This proof can be instantiated with MPCitH 2.

MPCitH 2: π_t

Public: $(id_u,\ gid, stt,\ sid)$
Private: $[\![tk_u]\!]$
Output: $id'_u,\ stt'$
Check: $id'_u = id_u \wedge stt' = stt$

1 $id'_u = \mathsf{MPC\_F}([\![tk_u]\!],\ gid)$;
2 $stt' = \mathsf{MPC\_F}([\![tk_u]\!],\ sid)$

  The tracer then outputs (id_u, π_t) or $\perp$.

- **Tracing proof verification** Given $(id_u,\ \pi_t,\ \Sigma, msg)$, the judge verifies π_t. If this proof is valid, the judge outputs 1 to indicate that id_u is the identifier of the Σ's signer. Otherwise the judge outputs $\perp$.

- **Revocation** To revoke the group membership of the user u, the group revocation authority retrieves the user's tracing key tk_u via the group tracer and then adds tk_u in **RL**.

The security analysis of this group signature scheme, under the security notion [10], is given in Appendix 4.2.

Remark 3 The revocation algorithm in our scheme follows the idea of verifier-local revocation, which was introduced in [9]. It currently does not support *forward anonymity*: when a user is revoked, its tracing key is published in the revocation list and that allows everyone to trace the user's past signatures. If forward anonymity is desirable, we can modify our scheme as follows: when revoking a user u, the revocation authority adds $H(tk_u)$ to the revocation list, where H is a collision-resistant hash function and tk_u is the user's tracing key.

Also, when generating a signature, the signer needs to extend π_G with a non-membership proof such that the hash of the signer's tracing key is not in the revocation list. The complexity of generating this non-membership proof is logarithmic to the size of the revocation list. This revocation mechanism follows Camenisch and Lysyanskaya's approach [18]. Now because the revocation list contains only the hash value of the tracing key, which cannot be used to trace a user, forward anonymity can be achieved. However, this approach shifts the burden of proof of (non-)revocation to the signer and increases the signature size.

3.3 The proof $\pi_{\mathtt{G}}$

The most important part in the group signature $\Sigma = (str,\ stt,\ com,\ \pi_{\mathtt{G}})$ is the proof $\pi_{\mathtt{G}}$. In this section we dissect it to show the design rationale and explain a change we made to MPC-in-the-Head, which greatly improves the efficiency and may be of independent interest.

As Σ is a signature of a message msg, the foremost thing $\pi_{\mathtt{G}}$ needs to prove is that the signer knows a group signing key $gsk_u = (tk_u,\ sk_u,\ gr_u,\ \mathbf{S})$ and it was used to sign msg. Besides that, $\pi_{\mathtt{G}}$ also needs to prove that gsk_u is authorized by the group issuer. To do that, in $\pi_{\mathtt{G}}$ the following is done:

1. It proves that stt, sst and com in the signature are produced from some tk_u, sk_u, and the message msg (in the form of $sid = H_1(msg||str)$). The commitment com also bounds Σ to all public keys used to blindly verify the signatures in $\mathbf{S}$.

2. It proves that the same sk_u and tk_u are used to generate mt_u from gid. Essentially, mt_u is a commitment of both sk_u and tk_u.

3. It proves that mt_u is signed under a private key in a leaf node of the hyper-tree generated by the group issuer. This is done by verifying all the signatures in $\mathbf{S}$ such that mt_u and σ_h produce the leaf public key pk_h, which in turn with σ_{h-1} produces pk_{h-1}, and so on until reaching the root. The last public key produced is rpk which is published by the group issuer. All public keys recovered in this process match those committed in the commitment com.

The challenge for implementing $\pi_{\mathtt{G}}$ with MPCitH comes from the cost of $h+1$ M-FORS signature verifications required by the proof. Recall that in an M-FORS signature (Section 3.1, also the example in Figure 2), the message hash to be signed is broken into k blocks, and each block is authenticated with a Merkle-tree of height d. Then the k Merkle tree roots are organized into a new Merkle tree whose root is the public key. Verifying the full signature means checking whether the public key can be recovered from the message hash, the secret strings corresponding to the hash blocks ($\mathbf{x}_{p_i}^{(i)}$), and the hashes along the Merkle tree authentication paths. In total, to verify a single M-FORS signature, $k \cdot (d+1) + (k-1) = kd + 2k - 1$ hashes are needed, which is in the order of 10^2 for a practical setting. The $h+1$ factor means that if implemented naively, the

MPC used in $\pi_{\mathtt{G}}$ would need to call thousands of times the sub-procedure that implements the hash function, and the size of the circuit for the whole MPC can go easily above million-gate. Even worse, to reduce the soundness error, the same circuit needs to be executed tens to hundreds of times in an MPCitH proof. Thus, a naive implementation of $\pi_{\mathtt{G}}$ will result in a very large signature size and a high computational cost.

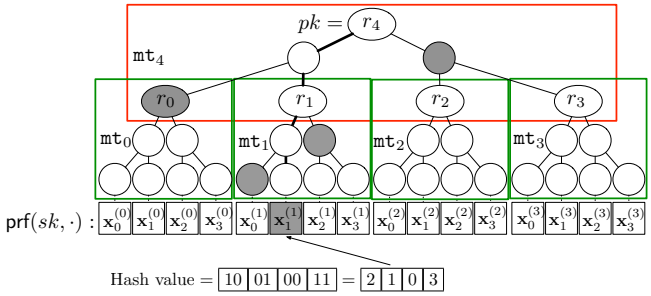


Figure 3: M-FORS Partial Verification using mt_1 (with $k = 4$ and $d = 2$)

Our more efficient strategy for implementing $\pi_{\mathtt{G}}$ is: in MPCitH, rather than repeating t times an MPC procedure in which the M-FORS signatures are fully verified, we run $t' \geq k$ MPC procedures in which the M-FORS signatures are partially verified, one block in each run (see the example of partial verification in Figure 3). More precisely, we extend the M-FORS with the following algorithms:

- $\mathsf{partial\text{-}sig}(\sigma, MD, i, mp)$: to extract a partial signature of the i-th block of MD from a full signature $\sigma = \{(x_0, \mathbf{authpath}^{(0)}), \cdots, (x_{k-1}, \mathbf{authpath}^{(k-1)})\}$. The Merkle tree $\mathtt{mt}_k$ can be recomputed from σ. The partial signature is $\partial_{\sigma,i} = (x_i, \mathbf{authpath}^{(i)}, \mathbf{authpath}^{(k,i)})$ where $(x_i, \mathbf{authpath}^{(i)})$ is a copy of the i-th pair in σ, and $\mathbf{authpath}^{(k,i)}$ is the authentication path of r_i (the root of the i-th Merkle tree) in $\mathtt{mt}_k$.

- $\mathsf{partial\text{-}rec}(\partial_{\sigma,i}, p_i, i, mp)$: This algorithm recovers the public key from $\partial_{\sigma,i}$ and p_i. Given $\partial_{\sigma,i} = (x, \mathbf{authpath}, \mathbf{authpath}')$, first compute the Merkle tree root r_1 from $(x, \mathbf{authpath}, p_i)$, then compute the Merkle tree root r_2 from $(r_1, \mathbf{authpath}', i)$. Output r_2.

With $\mathsf{partial\text{-}rec}$, only one path is used to recover the M-FORS public key instead of k paths.

The MPC procedure for proving the v-th block in $\pi_{\mathtt{G}}$ is shown in MPCitH 3. The signer needs to use partial signatures in the MPC. Recall that in the group signing key gsk_u, a list $\mathbf{S} = \{\boldsymbol{\sigma}_h, \cdots, \boldsymbol{\sigma}_0\}$ of $h+1$ signatures are stored, one for each layer in the hyper-tree of F-SPHINCS+. The signer can extract a partial signature for the v-th block from each signature, i.e. $\{\partial_{\sigma_h,v}, \cdots, \partial_{\sigma_0,v}\}$. In Line 10, an MPC subroutine $\mathsf{MPC\_pRec}$ that implements $\mathsf{partial\text{-}rec}$ is used. This subroutine uses the input to compute the corresponding public key at the l-th layer in the hyper-tree (stored in $[\![M]\!]$ and also appended to $[\![COM]\!]$). After

the last iteration, $[\![COM]\!]$ is hashed and $[\![M]\!]$ is revealed. The results will be checked by the verifier to see whether they match com and rpk. If so, the signer is likely to possess valid partial signatures along the path from the idx-th leaf node to the root node in the hyper-tree.

Why does this strategy make sense? In an MPCitH proof, the same procedure is run multiple times. Each run has a soundness ϵ that a cheating prover can succeed without being detected. Thus t runs are needed so that ϵ^t is negligibly small. In our case, the main cost of the MPC procedure comes from verifying all the M-FORS signatures. The full verification requires every block of the message digest or the child public key to be verified. Our observation is that if a cheating prover is to succeed, then they have to succeed with a high probability in more than 1 block. If the prover has to cheat in λ out of k blocks, then using partial verification with t', such that $t' \cdot \lambda/k \geq t$, ensures that the prover has to cheat in more than t runs, and hence with a negligible success probability. As we analyzed, the implementation with full signature verification requires $t \cdot (h+1) \cdot (k \cdot d + 2k - 1)$ calls to the MPC hash procedure. The partial verification-based implementation, on the other hand, requires only $t' \cdot (h+1) \cdot (d + 1 + \lceil \log k \rceil)$ MPC hash calls. The improvement is roughly $\frac{tk}{t'}$ times.

MPCitH 3: MPC instance for the v-th block in $\pi_{\mathtt{G}}$

Public: $gp = (n,\ q,\ h,\ d,\ k),\ rpk,\ sid,\ gid,\ stt,\ com,\ v$
Private: $[\![tk_u]\!],\ [\![sk_u]\!],\ [\![gr_u]\!],\ [\![\partial_{\sigma_h,v}]\!],\ \cdots,\ [\![\partial_{\sigma_0,v}]\!]$
Output: $pk_0,\ stt',\ com'$
Check: $pk_0 = rpk\ \wedge\ stt' = stt\ \wedge\ com' = com$

$stt' = \mathsf{MPC\_F}([\![tk_u]\!],\ sid)$;
$[\![sst]\!] = \mathsf{MPC\_F}([\![sk_u]\!],\ sid)$;
$[\![id_u]\!] = \mathsf{MPC\_F}([\![tk_u]\!],\ gid)$;
$[\![et_u]\!] = \mathsf{MPC\_F}([\![sk_u]\!],\ [\![id_u]\!])$;
$[\![mt_u]\!]||[\![idx]\!] = \mathsf{MPC\_H3}([\![et_u]\!]||[\![gr_u]\!])$;
$[\![M]\!] = [\![mt_u]\!]$;
$[\![COM]\!] = [\![sst]\!]$;
for $l = h;\ l \geq 0;\ l--$ **do**
  parse $[\![M]\!]$ into k blocks $[\![p_0]\!], \cdots, [\![p_{k-1}]\!]$, each block is d-bit;
  $[\![M]\!] = \mathsf{MPC\_pRec}([\![\partial_{\sigma_l,v}]\!],\ [\![p_v]\!],\ [\![idx]\!],\ gp,\ l,\ v)$;
  $[\![COM]\!] = \mathsf{MPC\_H1}([\![COM]\!]||[\![M]\!])$;
  $[\![idx]\!] = [\![\lfloor idx/q \rfloor]\!]$;
end
$com' = [\![COM]\!]$;
$pk_0 = \mathsf{Reveal}([\![M]\!])$;

The soundness analysis of π_G is given in Appendix D.

Experiment $\text{Exp}^{q\text{-EU-CMA}}_{\text{SIG},A}(n)$

- $(sk, pk) \leftarrow kg$
- $(M^*, \sigma^*) \leftarrow A^{sign(sk,\cdot)}(pk)$, and A can query the $sign$ Oracle at most q times.
- Return 1 iff $vf(pk, M^*, \sigma^*) = 1 \wedge M^* \notin \{M_i\}_{i=1}^{q}$

Figure 4: q-EU-CMA game.

4 Security Analysis

4.1 Security Analysis: F-SPHINCS+

The standard security definition for digital signature schemes is existential unforgeability under adaptive chosen-message attacks (EU-CMA). It can be extended to a few-time signature by limiting the adversary's call to the signing oracle to q times where q is the maximum number of signatures that the few-time signature scheme is allowed to generate for each signing key. Let $SIG = (kg, sign, vf)$ be a q-time signature scheme, Figure 4 shows the q-EU-CMA game.

Definition 1 (q-EU-CMA)**.** *Let SIG be a digital signature scheme. It is said to be q-EU-CMA secure, if for any adversary A, the following holds:*

$$Succ^{q\text{-}EU\text{-}CMA}_{SIG}(A(n)) = Pr\left[\text{Exp}^{q\text{-}EU\text{-}CMA}_{SIG,A}(n) = 1\right] \leq negl(n)$$

Theorem 1. *For suitable parameters, n, d, k, h, q, the F-SPHINCS+ signature is q^h-EU-CMA secure if:*

- *H_1 is SM-TCR and SM-DSPR secure;*
- *H_2 is TSR secure with at most q queries;*
- *H_{msg} is ITSR secure with at most q^h queries;*
- *prf is a secure pseudorandom function.*

Proof. To successfully forge a group issuer's signature on a message M chosen by the adversary, there are the following mutually exclusive cases:

1 Let $MD||idx =_{msg} (M||gr)$ for some gr. In the forged signature, All secret strings corresponding to $MD = p_0||\cdots||p_{k-1}$, i.e. $\{\mathbf{x}^{(i)}_{p_i}\}_{i=0}^{k-1}$, are the same as generated from $\texttt{leaf}_{idx}$'s secret key. This case consists of the following sub-cases:

1.1 The adversary learns all secret strings from signatures obtained in the query phase.

1.2 Some secret strings are not leaked from previous signatures, and for each of them, the adversary either:

1.2.1 learns it by breaking the pseudorandom function that is used to expand the secret key into $\mathbf{x}_i$;

1.2.2 or learns it by looking at their H_1 hash values and finding the pre-images.

2 Let $MD||idx = H_{msg}(M||gr)$ for some gr. In the forged signature, some secret strings corresponding to $MD = p_0||\cdots||p_{k-1}$, i.e. $\{\mathbf{x}_{p_i}^{(i)}\}_{i=0}^{k-1}$, are NOT the same as generated from $\texttt{leaf}_{idx}$'s secret key. Then let $\mathbf{S}$ be the list of $h+1$ M-FORS signatures in the forged signature, we can find i such that when verifying the i-th signature ($0 \leq i \leq h$), we obtain the same public key as would be generated by the signer, but for all $0 \leq j < i$, we obtain a different public key as would be generated by the signer. This means:

2.1 The adversary has found at least one second-preimages of H_1 so that some Merkle trees in the ith signature are computed with the second-preimages. They end up having the same roots as the trees computed by the group issuer.

2.2 The adversary knows all secret strings corresponding to the public key produced from verifying the $(i-1)$th signature. This public key is different from the public key at the same location generated by the group issuer. This can be done by either:

2.2.1 learning all from previous signature queries;

2.2.2 or breaking the pseudorandom function;

2.2.3 or finding some pre-images of H_1.

Given the above, we analyze the F-SPHINCS+ signature scheme through a series of games:

Game 0: The original EU-CMA game in which the adversary needs to forge a valid group issuer's signature after q_s queries.

Game 1: Exactly as Game 0 except all output of prf are replaced by truly random n-bit strings. We eliminate from the above list Case 1.2.1 and 2.2.2 by this modification. Since each call to prf uses a secret key and a distinct value as input, assuming prf is a pseudorandom function, we have:

$$|\text{Succ}^{Game0}(A(n)) - \text{Succ}^{Game1}(A(n))| \leq negl(n)$$

Game 2: Game 2 differs from Game 1 in that we consider the adversary lost if the adversary outputs a forgery by breaking the ITSR security of H_{msg}. This modification eliminates from the above list Case 1.1. The winning condition in Figure 4 is changed to:

- Return 1 iff $ITSR(H_{msg}, M^*) = 0 \wedge vf(pk, M^*, \sigma^*) = 1 \wedge M^* \notin \{M_i\}_{i=1}^{q^h}$.

The predicate $ITSR$ is defined as the following:

- Let M^* be the message that the adversary chooses to generate the forgery on, and gr^* the random string used by the adversary to compute

$$MD^*||idx^* = H_{msg}(M^*||gr^*).$$

- Parse $MD^* = p_0^*||\cdots||p_{k-1}^*$ where each $p_j^* \in [0, 2^d - 1]$. From the above we obtain a set $C^* = ((idx^*, 0, p_0^*), \cdots, (idx^*, k-1, p_{k-1}^*))$.

- For each message queried in the query phase M_i $(1 \leq i \leq q^h)$, and gr_i the random string, compute $MD_i||idx_i = H_{msg}(M_i||gr_i)$ and obtain

$$C_i = ((idx_i, 0, p_{i,0}), \cdots, (idx_i, k-1, p_{i,k-1})).$$

- Return 1 iff $C^* \subseteq \bigcup_{i=1}^{q^h} C_i$.

We can see that $ITSR(H_{msg}, M^*) = 0$ iff the adversary can break the ITSR security of H_{msg}. Hence, we have:

$$|\text{Succ}^{Game1}(A(n)) - \text{Succ}^{Game2}(A(n))| \leq \text{Succ}^{ITSR}_{H_{msg},q^h}(A) \leq negl(n)$$

Game 3: Game 3 differs from Game 2 in that we consider the adversary lost if the forgery contains a second-preimage for an input to H_1 that was part of a signature returned as a signing-query response. Here the second-preimage can be included explicitly in the signature, or implicitly observed when verifying the signature. This eliminates from the above list Case 2.1. Then we have:

$$|\text{Succ}^{Game2}(A(n)) - \text{Succ}^{Game3}(A(n))| \leq \text{Succ}^{SM-TCR}_{H_1,q}(A) \leq negl(n)$$

Game 4: Game 4 differs from Game 3 in that we consider the adversary lost if the adversary outputs a forgery by breaking the TSR security of H_2, which allows the adversary to forge an intermediate signature in $\mathbf{S}$, and then any signature earlier in the chain. This eliminates from the above list Case 2.2.1. The winning condition in Figure 4 is changed to:

- Return 1 iff $TSR(H_2, M^*) = 0 \wedge ITSR(H_{msg}, M^*) = 0 \wedge vf(pk, M^*, \sigma^*) = 1 \wedge M^* \notin \{M_i\}_{i=1}^{q^h}$.

The predicate TSR is defined as the following:

- The adversary chooses an intermediate node in the hyper-tree at address (a, b), and two n-bit string L^*, R^*.

- For each signature obtained in the query phase, if $\mathbf{S}_i$ includes a signature generated using the secret key in node (a, b) over the public key in one of its child nodes, parse this public key into k blocks, each of d-bit $pk_i = p_{i,0}||\cdots||p_{i,k-1}$, and generate a set $C_i = \{(j, p_{i,j})\}_{j=0}^{k-1}$.

- Compute $pk^* = H_2(\texttt{aux}||k||0||0||L^*||R^*)$, parse pk^* into $p_0^*||\cdots||p_{k-1}^*$, and generate a set $C^* = \{(j, p_j^*)\}_{j=0}^{k-1}$.

- Return 1 iff $C^* \subseteq \bigcup_{i=1}^{q} C_i$.

Note that each M-FORS public key is the root of a Merkle tree generated from pseudorandom strings. Also for each intermediate node in a hyper-tree, it has at most q children, hence no more than q signatures signed by the secret key in this intermediate node can be obtained by the adversary. So $TSR(H_2, M^*) = 0$ iff the adversary can break the TSR security of H_2. Hence, we have:

$$|\text{Succ}^{Game3}(A(n)) - \text{Succ}^{Game4}(A(n))| \leq \text{Succ}^{TSR}_{H_2,q}(A) \leq negl(n)$$

Now the cases in which the adversary can forge a signature are all eliminated except Case 1.2.2 and 2.2.3, which requires the adversary to find a pre-image of at least one hash value produced by H_1. The success probability of finding a pre-image is as analyzed in [6]:

$$Succ^{Game4}(A) \leq 3 \cdot Succ^{SM-TCR}_{H_1,p}(A) + \text{Adv}^{SM-DSPR}_{H_1,p}(A) \leq negl(n)$$

So overall, the advantage of the adversary is negligible. □

4.1.1 TSR security of H_2

In any case, q signatures can be generated under the secret key of a non-leaf node in the hyper-tree. Assuming the adversary knows all of them, then for each block of the chosen pk^*, the probability of the secret string has been leaked is $1 - (1 - \frac{1}{2^d})^q$, so all secret strings have been leaked is $(1 - (1 - \frac{1}{2^d})^q)^k$. For $d = 16, q = 1024, k = 68$, this probability is $2^{-468.87}$, if $k = 35$, this probability is $2^{-210.39}$.

4.1.2 ITSR security of H_3

For a leaf node of the hyper-tree, it may have been used to sign γ signatures out of the total q_s signature queries. So the probability that all secret strings of a chosen message M being leaked through a query is:

$$\sum_{\gamma}(1 - (1 - \frac{1}{2^d})^{\gamma})^k \binom{q_s}{\gamma} (1 - \frac{1}{q^h})^{q_s - \gamma} \frac{1}{q^{h\gamma}}$$

For $d = 16, q = 1024, k = 68, h = 6, q_s = 2^{60}$, this probability is $2^{-407.32}$, if $k = 35$, this probability is $2^{-208.95}$.

4.2 Security Analysis: the Group Signature

We follow the security model for fully dynamic group signature schemes by Bootle et.al. [10]. This model is intended to be general enough to accommodate many possible designs and is defined with the flexibility to allow certain aspects to be governed by policies defined out of the model (e.g. whether the group manager is one or two separate entities, how the user is activated and revoked). We adapt the model by concretizing the policies with our design choices.

Experiment $\mathbf{Exp}^{\text{corr}}_{FDGS,A}(n)$

- $h = \perp; K = 0; N = 0; \tau_{\text{Current}} = 0; \tau_{\text{Join}} = \infty; \tau_{\text{Revoke}} = \infty$.
- $(mpk, msk) \leftarrow \text{Init}(n)$; $\mathtt{Reg} = \emptyset$; $\mathbf{RL}_0 = \emptyset$.
- $(msg, \tau) \leftarrow A^{\mathbf{AddHU},\ \mathbf{AddCU},\ \mathbf{Revoke},\ \mathbf{ReadReg}}(mpk, \mathbf{RL}_0)$.
- If $K = k(n)$ and $\tau_{\text{Revoke}} = \infty$ and $\tau_{\text{Join}} = \infty$ return 0.
- If $h = \perp$ or $\tau > \tau_{\text{Current}}$ return 1.
- If $\tau_{\text{Join}} < \tau < \tau_{\text{Revoke}}$ and $\mathbf{IsActive}(h, \tau) = 0$, return 0.
- If $\mathbf{IsActive}(h, \tau) = 0$ return 1.
- $\Sigma \leftarrow \mathsf{GSig}(gsk_h, msg)$.
- Return $\mathsf{GVf}(msg, \Sigma, \mathbf{RL}_\tau)$.

Figure 5: Correctness game

In our scheme, group membership is maintained through 3 lists: $\mathbf{GL}, \mathbf{TL}, \mathbf{RL}$. In particular, $\mathbf{GL}$ and $\mathbf{TL}$ record information about users who have joined the group, and $\mathbf{RL}$ records information about users who have been revoked. In the model, we abstract $\mathbf{GL}$ and $\mathbf{TL}$ into a registry $\mathtt{Reg}$ that is a table storing various data of a group member. For (u, id_u, tk_u) in $\mathbf{TL}$ and $(u, id_u, et_u, gr_u, \mathbf{S})$ in $\mathbf{GL}$, the entry $\mathtt{Reg}_u = (u, id_u, tk_u, et_u, gr_u, \mathbf{S})$. The revocation list $\mathbf{RL}$ is maintained by the group revocation authority and is made public. In the model, $\mathbf{RL}$ is modeled as a list with versions. The lifetime of the group is divided into epochs. Each update on $\mathbf{RL}$ triggers a new epoch. We use $\mathbf{RL}_\tau$ to denote the version of $\mathbf{RL}$ at epoch τ. The following algorithm/oracle can be used to check whether a user has been revoked:

- $\mathbf{IsActive}(u, \tau)$: Given $u \in \mathbb{N}$, if $\mathtt{Reg}_u \neq \perp$, and $tk_u \notin \mathbf{RL}_\tau$, return 1. Otherwise, return 0.

In this model, the security of a dynamic group signature scheme is captured through security properties, including correctness, anonymity, traceability, non-frameability, tracing binding, and tracing soundness. In the following, we will go through the properties and show why they hold in our scheme.

Correctness Informally, correctness covers two aspects: (1) an honest user can successfully join the group, despite the existence of other malicious users; (2) a signature generated by an honest group member should always be valid when being verified (if the member has not been revoked). More formally, correctness is defined as an experiment in Figure 5. In the experiment, several global variables are defined: h tracks the honest user, K tracks the number of attempts that the honest user tries to join the group, N tracks the number of users who invoked Join protocol, $\tau_{\text{Current}}, \tau_{\text{Join}}, \tau_{\text{Revoke}}$ track the current epoch as well as the epoch in which the honest user joins and is revoked. The adversary has access to the following oracles:

- $\mathbf{AddHU}()$: This oracle adds a single honest user to the group. In each call, the oracle executes the Join protocol by simulating the honest user and the

honest group issuer/tracer. The oracle can be called at most $k(n)$ times where $k(\cdot)$ is any polynomial. Once the user is admitted into the group, further calls will be ignored. It returns the honest user's group membership secret key gsk_h as well as all parties' views to the adversary. Each view records the messages received by the party in the Join protocol.

- **AddCU(i)**: This oracle allows an adversary to add a corrupt user i to the group and get the corrupted user's view and output. The oracle plays both roles of honest group issuer and tracer in the Join protocol. The adversary can deviate from the Join protocol by sending arbitrary messages to the oracle.
- **Update($\mathcal{R}$)**: This oracle allows the adversary to update the revocation list **RL**, to remove the set of users $\mathcal{R}$ from the group. If h is revoked in this update, set τ_{revoke} to $\tau_{current}$.
- **ReadReg(u)**: Given $u \in \mathbb{N}$, return the entry $\mathtt{Reg}_u$, or $\perp$ if no such entry for u.

Oracles in Experiment $\mathbf{Exp}^{\text{corr}}_{FDGS,A}(n)$

AddHU()

- If $K = k(n)$ return $\perp$
- K=K+1
- If $h = \perp$:
  - $N = N + 1$; $h = N + 1$
- $(gsk_h, \text{View}_h, \text{View}_{GI}, \text{View}_{TR}) \leftarrow \mathsf{Join}(msk, n)$
- If $gsk_h \neq \perp$:
  - $\tau_{\text{Join}} = \tau_{\text{Current}}$
  - $K = k(n)$ // maximal number of rounds
- Return $(gsk_h, \text{View}_h, \text{View}_{GI}, \text{View}_{TR})$

AddCU(i)

- If $i \notin [N+1] \vee i = h$ return $\perp$
- If $i = N + 1$:
  - $N = N + 1$

// $\mathsf{Join}_i^{GI,TR}$ means i is corrupted, GI and TR are honest.

- $(gsk_i, \text{View}_i, \text{View}_{GI}, \text{View}_{TR}) \leftarrow \mathsf{Join}_i^{GI,TR}(msk, n)$
- Return (gsk_i, View_i)

Update($\mathcal{R}$)

- If $\mathcal{R} \not\subseteq [N]$ return $\perp$
- For each $u \in \mathcal{R}$ do
  - Retrieve $\mathtt{Reg}_u$
  - get tk_u from $\mathtt{Reg}_u$
  - $\mathsf{Revoke}(tk_u)$
- $\tau_{\text{Current}} = \tau_{\text{Current}} + 1$
- If $h \in \mathcal{R}$ and $\tau_{\text{Revoke}} = \infty$ set $\tau_{\text{Revoke}} = \tau_{\text{Current}}$
- Return $\mathbf{RL}_{\tau_{\text{Current}}} = \mathbf{RL}$

ReadReg(u)

- If $u \notin [N]$ return $\perp$
- Return $\mathtt{Reg}_u$

Note that the correctness of the tracing algorithm and tracing proof verification algorithm is not involved in this experiment. It will be covered in the traceability experiment as shown in Figure 7, which guarantees that if a group signature scheme holds traceability a signature generated by an honest or corrupted signer should always be traced to the correct signer.

Theorem 2. *Our group signature scheme is correct, that is for any PPT adversary A:*

$$\Pr\left[\mathrm{Exp}^{corr}_{FDGS,A}(n) = 1\right] = 1 - \mathit{negl}(n)$$

assuming the correctness of the signature $\boldsymbol{S}$ *generated by the group issuer and the proof system* Π*, and the collision-resistance of the function* F*.*

Proof. Following the correctness experiment, the adversary A only wins the game in any of the following three cases: (1) The join process (via **AddHU**) fails to let the honest user h join the group (i.e., $K = k(n)$ and $\tau_{Revoke} = \infty$ and $\tau_{Join} = \infty$); (2) The joined and unrevoked user h is considered to be inactive (i.e., $\tau_{Join} < \tau < \tau_{\tau_{Revoke}}$ and $\mathbf{IsActive}(h, \tau) = 0$); (3) The produced signature Σ created under gsk_h fails to verify (i.e., $\mathbf{GVf}(msg, \Sigma, \mathbf{RL}_\tau) = 0$).

Case 1 can happen if A can predict the honest user's tracing key tk_h and successfully registered with the same $id_h = F(tk_h, gid)$ in a previous session (via $\mathbf{AddCU}(i)$). In **AddHU**, the Join protocol is executed between the honest user h and the honest group issuer/tracer, so tk_h must be selected at random, and the probability of A can pick the same key i.e. $tk_i = tk_h$ in **AddCU** is negligible in the security parameter. Apart from this, the only possibility is that $tk_h \neq tk_i$ but $id_h = id_i$. This means $F(tk_h, gid) = F(tk_i, gid)$ - a collision of F is now found, which contradicts the assumption that the function F is collision-resistant. Therefore, the probability of Case 1 happening is negligible.

Case 2 only happens if there is another user i created by A (via $\mathbf{AddCU}(i)$) with $tk_i = tk_h$ and this user i is revoked when h is active (joined but not revoked). If the user i joined the group via **AddCU** before the user h, this is discussed in Case 1, and the probability is negligible. If the adversary tries to add i with $tk_i = tk_h$ (the adversary can get gsk_h that include tk_h when calling **AddHU** so does not need to guess) to the group via **AddCU** after the user h, it will be detected by the honest group issuer/tracer because, in the step 4 of the Join protocol, the group tracer checks whether tk is in **TL** and the protocol terminates if yes. Then the adversary's attempt will always fail.

It remains only Case 3. In this case, Σ is generated and verified in the epoch τ when $\mathbf{IsActive}(h, RL_\tau) = 1$, that indicates h is not revoked. Following the description of our group signature scheme, Σ will verify, given that the group issuer's signature used as the user h's credential and the proof system Π used in the group signature are both correct. Therefore the probability of Case 3 happening is also negligible.

Throughout these three cases, this theorem is held. □

Anonymity This property means that a group signature does not reveal the identity of its signer, i.e., the adversary cannot distinguish which one of the two

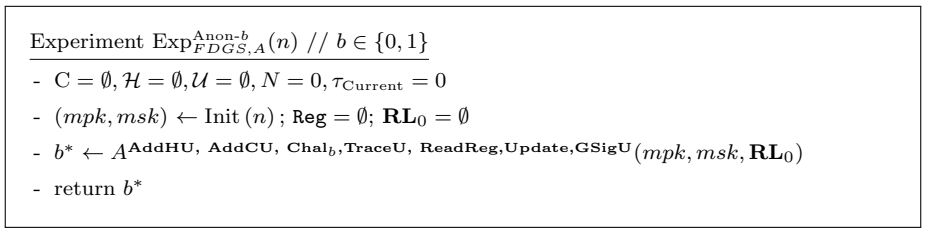


Figure 6: Anonymity Game

honest signers has signed a targeted message while both signers and the message are at the adversary's choice. In particular, as long as the signer's signing key is not leaked and the group tracer is not corrupted by the adversary, the adversary should not be able to breach anonymity even if it fully controls the group issuer and other users. Anonymity is captured by a game in Figure 6. In the game, a global variable C is used to track the challenge signature chosen by the oracle, $\mathcal{H}$ maintains the set of honest users, $\mathcal{U}$ tracks the users chosen for the challenge, and N tracks the number that the Join protocol has been invoked.

The adversary has access to the following oracles:

- **AddHU**(): This oracle allows the adversary to add multiple honest users, one each time, to the group. The Join protocol is executed by the oracle, and the adversary gets the honest user's and group issuer's views.
- **AddCU**(i): This oracle allows an adversary to add a corrupt user i to the group. In this version, the user, group issuer and tracer are all corrupted. The adversary can deviate from the Join protocol arbitrarily, and get the corrupted parties' outputs and views.
- **Chal**$_b$($\mathbf{RL}, msg, i_0, i_1$): This oracle returns a challenge signature and can be called only once. The oracle is given a message msg and two honest users i_0, i_1. Then for a $b \xleftarrow{R} \{0,1\}$, user i_b's group signing key gsk_{i_b} is used to generate a challenge signature. Both users must not be revoked with respect to $\mathbf{RL}$.
- **TraceU**($\mathbf{RL}, msg, \Sigma$): Returns the ID id_u of the user who produced a valid signature Σ on msg, and the corresponding NIZK π_t. This oracle cannot be invoked with the signature generated from **Chal**$_b$.
- **ReadReg**(u): Given u, if $u \in \mathcal{U}$ return the entry in group issuer's $\mathbf{GL}$ list; if $u \notin \mathcal{U}$, return the whole $\mathtt{Reg}_u$ ($\mathbf{GL}$+$\mathbf{TL}$) and also remove u form $\mathcal{H}$ if $u \in \mathcal{H}$.
- **Update**($\mathcal{R}$): This oracle allows the adversary to update the revocation list $\mathbf{RL}$, to remove the set of users $\mathcal{R}$ from the group.
- **GSigU**(msg, u): This oracle allows the adversary to obtain a signature of msg, under an honest user u's group signing key.

Oracles in Experiment $\text{Exp}^{\text{Anon-}b}_{FDGS,A}(n)$

AddHU()

- $h = N + 1$; $N = N + 1$; $\mathcal{H} = \mathcal{H} \cup \{h\}$
- $(gsk_h, \text{View}_h, \text{View}_{GI}, \text{View}_{TR}) \leftarrow \mathsf{Join}(msk, n)$
- Return $(\text{View}_h, \text{View}_{GI})$

AddCU(i)

- If $i \notin [N+1] \vee i \in \mathcal{H}$ return $\perp$
- If $i = N + 1$, $N = N + 1$
- $(gsk_i, \text{View}_i, \text{View}_{GI}, \text{View}_{TR}) \leftarrow \mathsf{Join}_{i,GI,TR}(msk, n)$
- Return $(gsk_i, \text{View}_i, \text{View}_{GI}, \text{View}_{TR})$

$\text{Chal}_b(\mathbf{RL}_{\tau_{\text{Current}}}, msg, i_0, i_1)$

- if $\{i_0, i_1\} \not\subseteq \mathcal{H}$ return $\perp$
- $\Sigma_0 \leftarrow \mathsf{GSig}(gsk_{i_0}, msg)$
- $\Sigma_1 \leftarrow \mathsf{GSig}(gsk_{i_1}, msg)$
- If $\mathsf{GVf}(msg, \Sigma_0, \mathbf{RL}_{\tau_{\text{Current}}}) = 0$ return $\perp$
- If $\mathsf{GVf}(msg, \Sigma_1, \mathbf{RL}_{\tau_{\text{Current}}}) = 0$ return $\perp$
- $\text{C} = \{msg, \Sigma_b\}, \mathcal{U} = \{i_0, i_1\}$
- Return Σ_b

TraceU($\mathbf{RL}, msg, \Sigma$)

- If $(msg, \Sigma) \in \text{C}$ return $\perp$
- If $\mathsf{GVf}(msg, \Sigma, \mathbf{RL}) = 0$ return $\perp$
- return $\mathsf{Trace}(msg, \Sigma)$

GSigU(msg, u)

- If $u \notin [N]$ return $\perp$
- return $\mathsf{GSig}(gsk_u, msg)$

ReadReg(u)

- If $u \notin [N]$ return $\perp$
- Retrieve $\mathtt{Reg}_u = (u, id_u, tk_u, et_u, gr_u, \mathbf{S})$
- If $u \in \mathcal{U}$
  - Return $(u, id_u, et_u, gr_u, \mathbf{S})$
- Else
  - If $u \in \mathcal{H}, \mathcal{H} = \mathcal{H} - u$
  - Return $(u, id_u, tk_u, et_u, gr_u, \mathbf{S})$

Update($\mathcal{R}$)

- If $\mathcal{R} \not\subseteq [N] \vee \mathcal{R} \cap \mathcal{U} \neq \emptyset$ return $\perp$
- For each $u \in \mathcal{R}$ do
  - Retrieve $\mathtt{Reg}_u$
  - get tk_u from $\mathtt{Reg}_u$
  - $\mathsf{Revoke}(tk_u)$
- $\tau_{\text{Current}} = \tau_{\text{Current}} + 1$
- Return $\mathbf{RL}_{\tau_{\text{Current}}} = \mathbf{RL}$

Theorem 3. *Our group signature scheme is anonymous, that is for any PPT adversary A,*

$$\left|\Pr\left[\text{Exp}^{Anon\text{-}1}_{FDGS,A}(n) = 1\right] - \Pr\left[\mathbf{Exp}^{Anon\text{-}0}_{FDGS,A}(n) = 1\right]\right| \leq negl(n)$$

if the function F is a PRF, the function H_1 is a prefix-free PRF, and the proof system Π is a zero-knowledge proof system.

Proof. The adversary A enters either $\text{Exp}^{\text{Anon-1}}_{FDGS,A}$ or $\text{Exp}^{\text{Anon-0}}_{FDGS,A}$, and needs to figure out which experiment it is in to win the anonymity game. In the experiments, an oracle interacts with the adversary and answers any oracle queries from the adversary. At the start of the experiment, the oracle chooses i_0, i_1

uniformly at random from $[N]$. If $\mathbf{Chal}_b$ is invoked with users other than i_0, i_1, the oracle rewinds the adversary to the start of the experiment. The probability that an experiment proceeds without rewinding is $\frac{1}{N^2}$, which is polynomial in n and computable. In either experiment, the only piece of data that contains information about b is Σ_b returned by the oracle $\mathbf{Chal}_b$. Hence in the cases that $\mathbf{Chal}_b$ returns $\perp$, no information about b could possibly be given to the adversary A, so A cannot do better than random guess and its advantage is 0:

$$\left|\Pr\left[\mathrm{Exp}^{\text{Anon-1}}_{FDGS,A}(n)=1\right]-\Pr\left[\mathbf{Exp}^{\text{Anon-0}}_{FDGS,A}(n)=1\right]\right|=0$$

Hence in the following, we only consider the case that $\mathbf{Chal}_b$ returns Σ_b. We process this proof by using a series of games.

$Game_0$: This game is the original experiment $\mathrm{Exp}^{\text{Anon-}b}_{FDGS,A}(n)$.

$Game_1$: This game differs from $Game_0$ only in that, when calling the oracle $\mathbf{Chal}_b(\mathbf{RL}, msg, i_0, i_1)$, the oracle replaces π_G with the output of the simulator of π_G, and returns $(str_b, stt_b, com_b, Sim_{\pi_\mathsf{G}})$ instead of Σ_b. The advantage that the adversary can distinguish whether it is in $Game_0$ or $Game_1$ is negligible, otherwise, it contradicts the assumption that π_G is zero-knowledge.

Now let us move on to other parts in Σ_b. Recall that $stt_b = F(tk_{i_b},\ sid)$ and $sid = H_1(msg||str)$, where tk_{i_b} is a uniformly random secret key held by the user i_b. We need to ensure the adversary knows nothing about tk_{i_b}. From the pseudocode of the oracles, we can conclude that the adversary cannot obtain tk_{i_0} or tk_{i_1} directly. In particular:

- By calling **AddHU**, the returned views do not contain tk of the honest user directly;
- By calling **AddCU**, the adversary obtains tk of the corrupted user because gsk is given to the adversary, but the adversary cannot nominate this user in the challenge because it is not in $\mathcal{H}$ so $\mathbf{Chal}_b$ will return $\perp$;
- The output of **TraceU** and **GSig** does not contain tk of any user;
- By calling **ReadReg**, the adversary can get tk for any user u except $\{i_0, i_1\}$;
- Lastly, the adversary cannot add tk of either i_0 or i_1 into **RL** by calling **Update**, because if i_0 or i_1 has ever been revoked, GVF will output 0 and the oracle $\mathbf{Chal}_b$ will return $\perp$.

While tk_{i_0} and tk_{i_1} are not given directly to the adversary, $id_{i_b} = F(tk_{i_b},\ gid)$ can be seen by the adversary. We then have the next game:

$Game_2$: This game is modified from $Game_1$. In this game id_{i_b} is replaced by the output of a random function $f : \{0,1\}^n \to \{0,1\}^n$, also π_u and π_t are replaced by the output of their simulator. We can say $Game_2$ and $Game_1$ are indistinguishable, otherwise, it contradicts either the fact that F is a pseudo-random function, or that π_u and π_t are zero-knowledge.

In $Game_2$, the adversary gets absolutely no information about tk_{i_b}, directly or indirectly. We now proceed to the next game:

$Game_3$: This game is modified from $Game_2$. In this game, stt_b is replaced by the output of a random function $f : \{0,1\}^n \to \{0,1\}^n$. $Game_3$ and $Game_2$ are indistinguishable, otherwise it contradicts to the fact that F is a pseudorandom function.

Next, let us move to $com_b = H_1(sst_b||pk_h||\cdots||rpk)\}$. We first argue that the adversary knows nothing about $sst_b = F(sk_{i_b},\ sid)$. We start by using a similar argument as above that sk_{i_b} has never been leaked directly, and then the following game that deals with indirect information of sk_{i_b}:

$Game_4$: This game is modified from $Game_3$. In this game, $et_{ib} = F(sk_{i_b,id_u})$ is replaced by the output of a random function $f : \{0,1\}^n \to \{0,1\}^n$. $Game_4$ and $Game_3$ are indistinguishable, otherwise it contradicts to the fact that F is a pseudorandom function. Note that sk_{i_b} is also used as a private input in π_u, and since π_u has already been replaced in $Game_2$ by the output of its simulator, we do not need to do anything in $Game_4$.

Then we have a game to deal with sst_b:

$Game_5$: This game is modified from $Game_4$. In this game, $sst_b = F(sk_{i_b},\ sid)$ is replaced by the output of a random function $f : \{0,1\}^n \to \{0,1\}^n$. $Game_5$ and $Game_4$ are indistinguishable, otherwise it contradicts to the fact that F is a pseudorandom function.

Now we can deal with $com_b = H_1(sst_b||pk_h||\cdots||rpk)\}$. In $Game_5$, sst_b is already a random string, so we have the following game:

$Game_6$: This game is modified from $Game_5$. In this game, com_b is replaced by a random n-bit string. Now we discuss the fact that H_1 in the com_b computation can be treated as a PRF. Let $H : \{0,1\}^n \times \{0,1\}^c \to \{0,1\}^n$ denote the underlying compression function that is a PRF. H_1 is a cascade construction of H. Based on [5], a cascade construction with its underlying compression function being a PRF is a prefix-free PRF, meaning that it is a PRF as long as no input is a prefix of another input. In our group signature scheme, $com_b = H_1(sst_b||pk_h||\cdots||rpk)\}$ is implemented by using H as follows: Let $k_0 = sst_b$ and $pk_0 = rpk$,

$$k_i = H(k_{i-1}, pk_{h+1-i}),\ \text{for } i = 1, ..., h+1,$$

where h is the number of M-FORS layers in the Group-tree, and finally $com_b = k_{h+1}$. Here, sst_b serves as an n-bit key that is not known by the adversary A, and all inputs $pk_h, pk_{h-1}, ..., rpk$ are of the same length, so no query of the adversary to H is a prefix of another query. Therefore, in this case, H_1 is a PRF. This concludes that $Game_5$ and $Game_6$ are indistinguishable otherwise it contradicts the fact that H_1 in the com_b computation is a PRF.

Now if the adversary enters the 1 version or the 0 version of $Game_6$, in which $\mathbf{Chal}_1$ or $\mathbf{Chal}_0$ is executed respectively, and the output of $\mathbf{Chal}_b$ is not $\perp$, then the output is $str_b, r1_b, r2_b, Sim_{\pi_G}$ where $r1_b$ is the random string from random function f replacing stt_b and $r2_b$ is the random string replacing com_b. The four parts in the output are all independent of b (str_b is also a random string). Hence given the output,

$$\left|\Pr\left[\mathrm{Exp}^{\text{Anon-1}}_{FDGS,A}(n) = 1\right] - \Pr\left[\mathbf{Exp}^{\text{Anon-0}}_{FDGS,A}(n) = 1\right]\right| = 0$$

Experiment $\mathrm{Exp}^{Trace}_{FDGS,A}(n)$

- $N = 0$; $\tau_{\text{Current}} = 0$
- $(mpk, msk) \leftarrow \text{Init}\,(1^n)$;$\mathtt{Reg} = \emptyset$; $\mathbf{RL}_0 = \emptyset$
- $(msg, \Sigma, \tau) \leftarrow A^{\mathbf{AddCU,Update,ReadReg,WriteReg}}(mpk, \mathbf{RL}_0)$
- If $\mathsf{GVf}(msg, \Sigma, RL_\tau) = 0$ Return 0
- $(id_u, \pi_t) \leftarrow \mathsf{Trace}\,(msg, \Sigma)$
- If $\mathbf{IsActive}\,(u, \tau) = 0$ Return 1
- If $\mathsf{TVf}\,(id_u, \pi_t, \Sigma, msg) = 0$ Return 1
- Return 0

Figure 7: Trace game

All games from $Game_6$ to $Game_0$ can only be distinguished with negligible probability with the previous one, so we have

$$\left|\Pr\left[\mathrm{Exp}^{\text{Anon-1}}_{FDGS,A}(n) = 1\right] - \Pr\left[\mathbf{Exp}^{\text{Anon-0}}_{FDGS,A}(n) = 1\right]\right| \leq negl(n)$$

□

Traceability Traceability ensures that a group member (even malicious) can be traced by the group tracer through a valid signature, i.e. the tracer can output a convincing proof showing that the signature was signed by the group member. The game is as shown in Figure 7. In the game, a counter N is maintained to track the number that the Join protocol has been invoked.

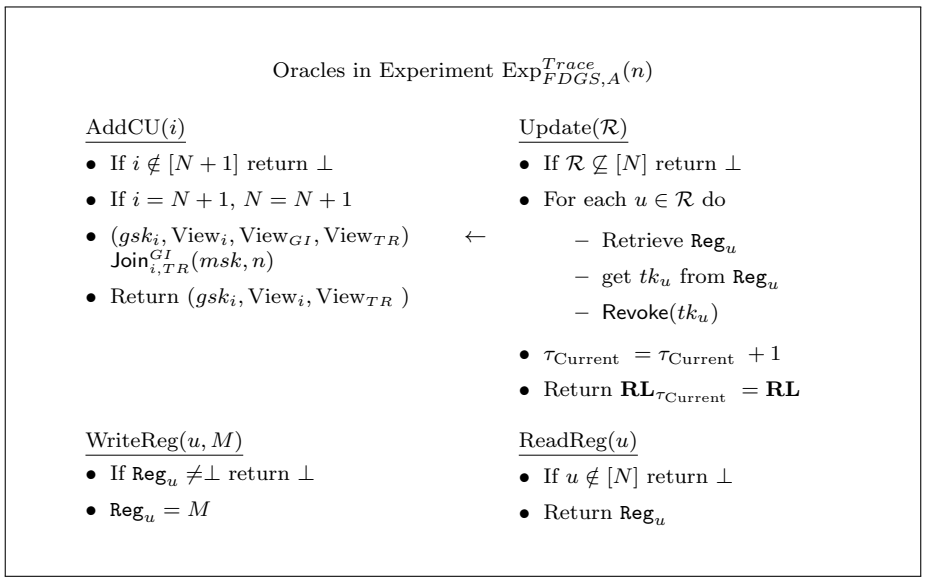
The adversary has access to the following oracles:

- **AddCU**(i): This oracle allows an adversary to add a corrupt user i to the group. In the Join protocol, the group issuer is honest, and the user and the tracer are corrupted.
- **Update**($\mathcal{R}$): This oracle allows the adversary to update the revocation list **RL**, to remove the set of users $\mathcal{R}$ from the group.
- **WriteReg**(u, M): Given u, if $\mathtt{Reg}_u$ is not empty, set $\mathtt{Reg}_u = M$.
- **ReadReg**(u): Given u, return the whole $\mathtt{Reg}_u$ (**GL**+**TL**).

Theorem 4. *Our group signature scheme is traceable, that is for any PPT adversary A*

$$\Pr\left[\boldsymbol{Exp}^{Trace}_{FDGS,A}(n) = 1\right] \leq negl\,(n)$$

if the group issuer's signature S is unforgeable, the function F is collision-resistant, and the proof system Π is a zero-knowledge proof system with simulation-sound extractability.



Proof. Recall that the adversary A wins the traceability experiment if it generates a valid signature that either:

- Case 1: it traces to an inactive user, i.e., $\mathbf{IsActive}(u, \tau) = 0$;
- Case 2: it traces to an active user but an associated proof π_t is invalid, i.e., $\mathsf{TVf}(id_u, \pi_t, \Sigma, msg) = 0$.

In the following, we will prove that the probability for these two cases is negligible. Let (msg, Σ, τ) be the output of A in the experiment $Exp^{\mathrm{Trace}}_{FDGS,A}(n)$. Parse Σ as $(str, stt, com, \pi_{\mathtt{G}})$. When Case 1 occurs, the signature is traced to an inactive user u, which means either $\mathtt{Reg}_u = \perp$ or $tk_u \in \mathbf{RL}_\tau$. It then leads to two sub-cases.

- Case 1_a: $\mathtt{Reg}_u = \perp$ means that A has successfully created a group signature without successfully joining the group. In other words, the adversary can obtain an unauthorized group signing key and generates Σ with it. A group signing key is $gsk_u = (tk_u,\ sk_u, gr_u,\ \mathbf{S})$. While it is easy for A to generate $tk_u,\ sk_u, gr_u$ by itself, $\mathbf{S}$ is a chain of signatures generated by the group issuer. Obtaining an unauthorized group signing key hence can be reduced to forging the signatures in $\mathbf{S}$. Assuming the group issuer's signatures are unforgeable, the probability of this case is negligible.
- Case 1_b: In this case $tk_u \in \mathbf{RL}_\tau$, but also note that this case also requires $\mathsf{GVf}(msg, \Sigma, RL_\tau)$ to output 1, otherwise $Exp^{\mathrm{Trace}}_{FDGS,A}$ returns 0. This can happen if the adversary can find two distinct tracing keys tk_u and tk'_u, such that $id_u = F(tk_u, gid)$, $id'_u = F(tk'_u, gid)$, and $et_u = F(sk_u, id_u) = F(sk'_u, id'_u)$. The adversary A computes $id_u = F(tk_u, gid)$ and $et_u = F(sk_u, id_u)$

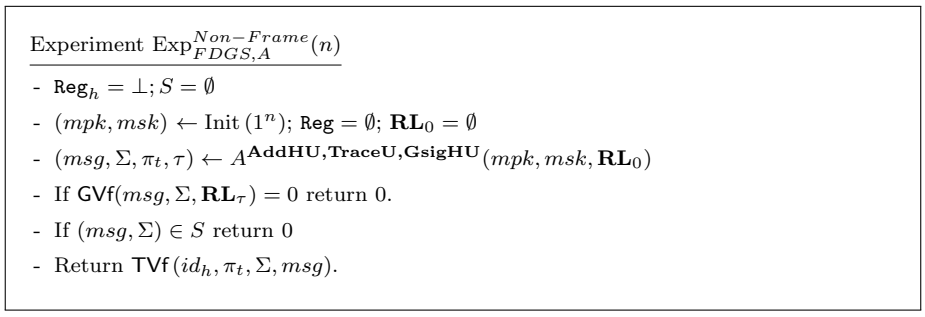


Figure 8: Non-frameability game.

in the Join process, but computes $stt = F(tk'_u, sid)$, $id'_u = F(tk'_u, gid)$ and $et_u = F(sk'_u, id'_u)$ in the signing process. Then tk_u can be revoked but $\mathsf{GVf}(msg, \Sigma, RL_\tau) = 1$. This case requires the adversary to find a collision of F, and due to the property of simulation-sound extractability of Π, this collision can be extracted. Since F is collision-resistant, Case 1_b can happen only with a negligible probability.

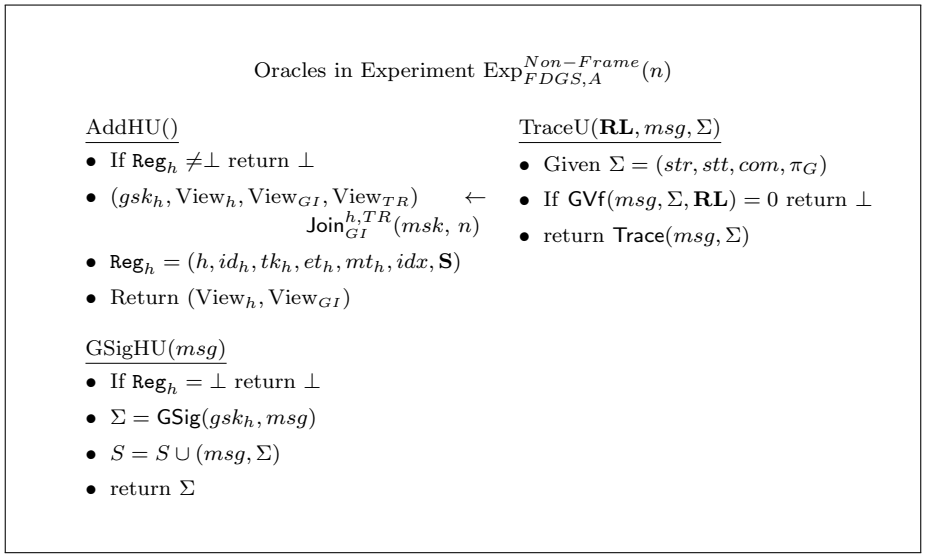
When Case 2 occurs, the signature is traced to an active user i through a legitimate tk_u that has been recorded by the group tracer. In this case, the tracer should also have recorded $id_u = F(tk_u, gid)$, so assuming π_t is complete, the probability of this case is also negligible.

Throughout these two cases, this theorem is proved. □

Non-frameability Non-frameability is a security notion that even if the rest of the group as well as the group issuer/revocation authority (but not the tracer) are fully corrupted, they cannot falsely attribute a signature to an honest member who did not produce it. In the game $\mathbf{Exp}^{\text{Non-Frame}}_{FDGS,A}(n)$ (see Figure 8), the adversary can add at most one honest user. We note that the adversary controls the group issuer and hence a session identifier no longer carries much meaning the adversary can pretend the user has any session identifier. Instead, without loss of generality, we simply identify the honest user h with a generic record $\mathtt{Reg}_h$ and require that this is written once when created and cannot be modified afterward. We also maintain a global list S of signatures produced by the honest user. We grant the adversary access to the oracles described below. Note that since the group issuer and revocation authority are corrupted, they can manipulate the revocation list and create corrupted users. The adversary also obtains the content of $\mathtt{Reg}_h$ except tk_h when adding the honest user.

- **AddHU**(): This oracle allows the adversary to add a single honest user to the group. In the Join protocol, the user and the tracer are honest, and the adversary controls the group issuer and also gets the honest user's and group issuer's views.

- **TraceU(RL**, msg, Σ**)**: This oracle allows the adversary to obtain the tracing result of a valid signature which could be either a forgery or an output of the **GSigHU** oracle. The tracing result includes the id of the user who generated this signature and the corresponding π_t. Note for signatures generated by the adversary on behalf of corrupted users, the adversary can generate the tracing result locally since it knows tk_u.
- **GSigHU**(msg): This oracle allows the adversary to obtain a signature of msg, under the honest user's group membership secret key gsk_h. The tracing token of the signature is recorded in S.



Theorem 5. *If the proof system* Π *(including* (π_u, π_G, π_t)*) is a zero-knowledge proof system with simulation-sound extractability, the function* F *is a PRF, additionally collision-resistant, then our group signature scheme offers non-frameability. That is, for any PPT adversary* A*,*

$$\Pr\left[\boldsymbol{Exp}^{Non\text{-}Frame}_{FDGS,A}(n) = 1\right] \leq negl\,(n)$$

Proof. In the experiment, the adversary wins if it can generate $(msg, \Sigma, \pi_t, \tau)$, such that the Σ is a valid signature of msg at epoch τ, and this signature was not generated by the honest user but the signature can be traced to the honest user. We process this proof by using a series of games.

$Game_0$: This game is the original experiment $\mathrm{Exp}^{Non-Frame}_{FDGS,A}(n)$.

$Game_1$: This game differs from $Game_0$ only in that, when calling the oracles **AddHU**, **GsigHU**, and **TraceU**, the zero-knowledge proof π_u, π_G and π_t are replaced by the output of their simulators. The advantage that the adversary can distinguish whether it is in $Game_0$ or $Game_1$ is negligible, otherwise, it contradicts the assumption that π_u, π_G, and π_t are zero-knowledge.

$Game_2$: This game differs from $Game_1$ only in that, id_h is replaced by an n-bit random string. Recall that $id_h = F(tk_h, gid)$ where tk_h is a uniformly random key not disclosed to the adversary in any way. Hence the advantage of the adversary can distinguish $Game_2$ and $Game_1$ is negligible, otherwise, it contradicts the assumption that F is a PRF.

Then by transitivity, we have:

$$\left|\Pr\left[\boldsymbol{Exp}^{\text{Non-Frame}}_{FDGS,A}(n) = 1\right] - \Pr\left[Game_{2\ A}(n) = 1\right]\right| \leq negl\,(n)$$

In $Game_2$, id_h is chosen independently of tk_h (or any tk chosen by the adversary), hence intuitively the adversary should not be able to attribute a signature that is not generated by **GSigHU** to id_h. Next, we formally prove this. The adversary outputs $(msg, \Sigma, \pi_t, \tau)$, where $\Sigma = (str, stt, com, \pi_G)$, and there are three cases in which $\mathsf{TVf}(id_h,\ \pi_t, \Sigma, msg) = 1$ (hence the output of $Game_2$ is 1):

1. The adversary added a corrupted user u, such that $F(tk_u, gid) = id_h$. Then Σ and π_t are generated using u's group membership secret key gsk_u. Given that id_h is uniform, the probability of this case is negligible. If this case happens, due to the property of simulation-sound extractability of π_G and π_t, the value tk_u can be extracted, therefore, it contradicts the assumption that F is a pseudorandom function.

2. The adversary generates Σ using gsk_u in which $id_u = F(tk_u, gid) \neq id_h$, but in the signature Σ, $stt = F(tk_u, sid) = F(tk_h, sid)$ for $tk_u \neq tk_h$. Σ can be traced to id_h through the oracle **TraceU**. This case indicates that the adversary has found a collision of F. When this case happens, due to the property of simulation-sound extractability of π_G, the value tk_u can be extracted. As tk_h was recorded in Reg_h, then the collision pair (tk_h, tk_u) can be extracted. The probability of this case is negligible, otherwise, it contradicts the assumption that the function F is collision-resistant.

3. The adversary generates Σ using gsk_u in which $id_u = F(tk_u, gid) \neq id_h$ and the adversary also generates π_t. For $\mathsf{TVf}(id_h, \pi_t, \Sigma,\ msg) = 1$ to hold, verification of π_t must be successful. Therefore, the probability of this case is negligible, otherwise, it contradicts the assumption that π_t is a zero-knowledge proof with soundness because the relation $F(tk_u, gid) = id_h$ must be proved in π_t but it does not hold.

□

Tracing Binding The tracing binding property [57] guarantees that even if all authorities and users collude, they should not be able to produce a valid signature that can be selectively attributed to different members. Traceability concerns about the adversary producing a signature and attributing it to an honest user, while tracing binding concerns about an already generated signature, and the goal of the adversary is to create a signature and two distinct attributions to who signed it. It considers a strongly adversarial setting, where

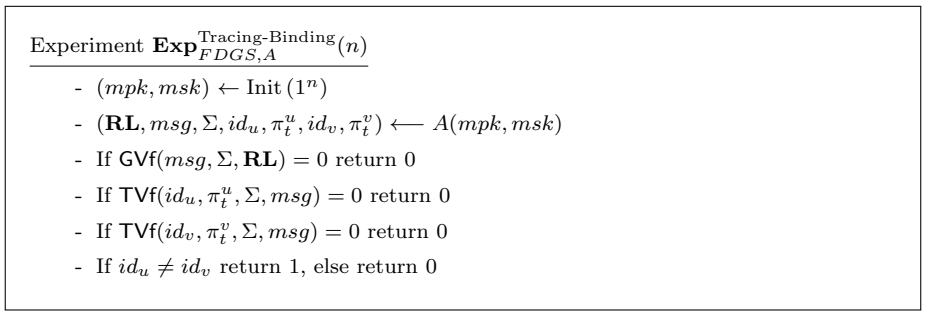


Figure 9: Binding Tracing Game

both the authorities and users may be adversarial but wants the guarantee that each signature must be attributed to a unique record in the registry. We describe the tracing binding game in Figure 9.

Theorem 6. *Our group signature scheme offers tracing-binding, that is for any PPT adversary*

$$\Pr\left[\text{Exp}^{\textit{Tracing-Binding}}_{FDGS,A}(n) = 1\right] \leq negl(n)$$

if the proof π_t is a zero-knowledge proof system with simulation-sound extractability, and the function F is collision-resistant.

Proof. For $\mathbf{Exp}^{\text{Tracing-Binding}}_{FDGS,A}(n)$ to output 1, it is necessary that given a single signature Σ, for two user identities $id_u \neq id_v$, both $\mathsf{TVf}(id_u, \pi_t^u, \Sigma, msg)$ and $\mathsf{TVf}(id_v, \pi_t^v, \Sigma, msg)$ output 1. In π_t^i $(i \in \{u, v\})$, the relation $id_i = F(tk_i, gid)$ and $stt = F(tk_i, sid)$ is proved. If $\mathsf{TVf}(id_u, \pi_t^u, \Sigma, msg)$ and $\mathsf{TVf}(id_v, \pi_t^v, \Sigma, msg)$ both output 1, then the aforementioned relation must hold except a negligible probability, otherwise it contradicts to the assumption that π_t is sound. However, this also means $stt = F(tk_u, sid) = F(tk_v, sid)$, i.e. the adversary finds a collision of the function F. Since π_t also has the property of extractability, these two collision values tk_i $(i \in \{u, v\})$ can be extracted from π_t^i. The probability of this case is negligible, assuming F is a collision-resistant function. □

Tracing Soundness The tracing soundness property guarantees that even if all authorities are corrupted, they cannot attribute an honest user's signature to a corrupted user. This property differs from traceability in that traceability concerns attributing a signature generated by a corrupted user to an honest user. The tracing soundness experiment is defined in Figure 10. The following oracles can be accessed by the adversary:

- **AddHU**(): This oracle allows the adversary to add a single honest user to the group. In the Join protocol, the user is honest, and the adversary controls the group issuer and tracer. The group membership secret key of the honest user and the views of all parties are given to the adversary.

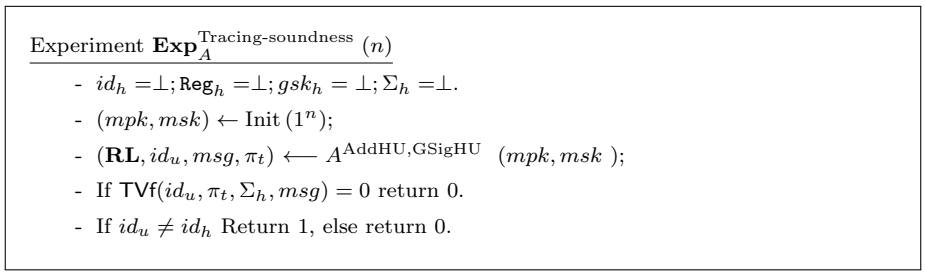
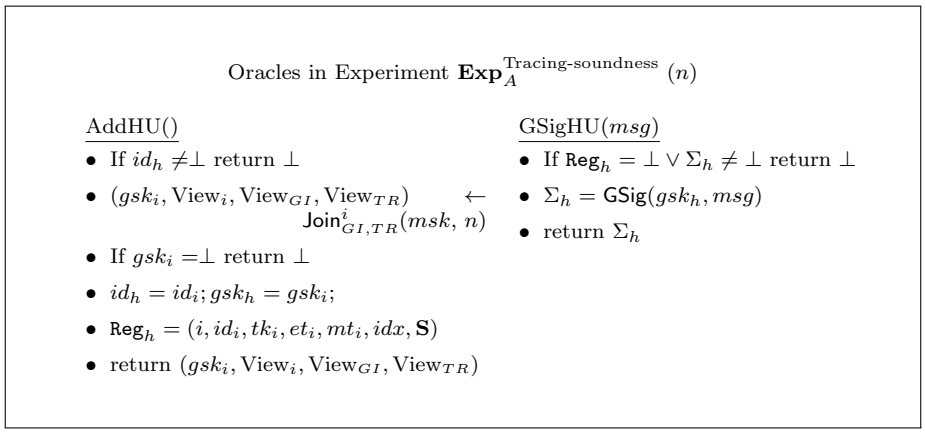


Figure 10: Tracing soundness game

- **GSigHU**(msg): This oracle generates a single challenge signature under the honest user's group membership secret key gsk_h, on a message chosen by the adversary.



Theorem 7. *Our group signature scheme offers or tracing soundness, that is for any PPT adversary A,*

$$\Pr\left[\mathbf{Exp}_{FDGS,A}^{\textit{Tracing-soundness}}(n) = 1\right] \leq negl(n)$$

if the proof π_t is a zero-knowledge proof system with simulation-sound extractability, and the function F is a PRF with the property of second-preimage-resistance.

Proof. $\mathbf{Exp}_{FDGS,A}^{\text{Tracing-soundness}}(n) = 1$ means $\mathsf{TVf}(id_u, \pi_t, \Sigma_h, msg) = 1$ and $id_u \neq id_h$. In π_t, the relation $id_u = F(tk_u, gid)$ and $stt = F(tk_u, sid)$ is proved. Due to the soundness of π_t, if $\mathsf{TVf}(id_u, \pi_t, \Sigma, msg)$ outputs 1, then the same tk_u must be used in generating id_u and stt. However, Σ_h is generated under gsk_h, which means $stt = F(tk_h, sid)$. There are two cases:

- Case 1: $tk_h = tk_u$. This case is possible when the malicious tracer generates a corrupted user u by using the honest user h's tk_h and u's secret

signing key is different from h's, $sk_u \neq sk_h$. This misbehavour should be noticed by the issuer during the join protocol, but since we are assuming that both the tracer and the issuer are controlled by the adversary, this case can happen. However, for a group with the group identifier gid, a user i's identity $id_i = F(tk_i, gid)$, and so $tk_h = tk_u$ means to $id_u = id_h$. Given Σ_h with $stt = F(tk_h, sid)$, to make the attack work, the tracer needs to compute π_t with $id_u \neq id_h = F(tk_h, gid)$ and $stt' = stt = F(tk_h, sid)$. This is impossible because as a PRF, F is deterministic, and so the adversary cannot create two distinct user identifiers from the same tracing key.

- Case 2: $tk_h \neq tk_u$. This means $stt = F(tk_h, sid) = F(tk_u, sid)$, so the adversary has found a second-preimage of F. Since π_t holds the property of simulation-sound extractability, the second-preimage value tk_u can be extracted from π_t. This case can happen with only negligible probability under the assumption that F is second-preimage-resistant.

□

5 Implementation

As a proof of concept, we implemented the generation of an F-SPHINCS+ signature with an M-FORS signature chain, $\mathbf{S}$, and the π_G signing and verification components of the scheme using partial algorithms. These parts of the joining, signing and verification protocols are the most compute intensive and allow us to assess the times involved, we did not implement the full protocols[2]. For ease of implementation, we chose LowMC and KKW, which had readily available implementations that we could use as building blocks. We instantiate the pseudorandom functions prf and F with the LowMC block cipher. The hash functions H_1, H_2, H_3 are also instantiated using LowMC, through the Davies-Meyer transformation (see Appendix E.2). Note that our protocols are not restricted to the use of LowMC and KKW, they could be replaced by more secure/efficient schemes.

The implementation was written in C++ and for some subroutines related to MPC-in-the-head, we used the code from the Picnic KKW scheme (namely `picnic3`) implementation [63] submitted to the NIST Post-Quantum Cryptography Standardization project [54]. The code for LowMC (and its MPC-in-the-head version) was re-written using 64-bit words as this was more efficient than the `picnic3` implementation. In Appendix E, we provide more detailed discussions on our implementation.

In our implementation, we used two different security parameters, i.e. $n = 129$ and $n = 255$. This parameter is determined by the block size and the key size of the LowMC instances, which were chosen by `picnic3`. The case of $n = 129$ can offer 128-bit security against classical attacks while the case

[2]Code is available at: `https://github.com/UoS-SCCS/HBGS`

of $n = 255$ can offer 128-bit security against quantum computer attacks. The other values for the parameters at the corresponding security level that we used are given in Table 2. The parameters k, d, q are for F-SPHINCS+, NR and NU are the total number of MPC instances and the number of unopened MPC instances respectively (KKW uses a cut-and-choose strategy and randomly opens $NR - NU$ MPC instances to detect cheating).

Table 2: The parameters used for testing.

n	k	d	q	NR	NU
129	35	16	1024	560	35
255	70	16	1024	1120	70

We measure the performance of our group signature scheme based on our C++ implementation. The programs were compiled using the GNU GCC compiler [27] version 9.4.0 and executed on a laptop (Intel i7-8850H CPU @2.6GHz : 32Gb RAM) with the Ubuntu operating system. Although the CPU has multiple cores, the timings were obtained using a single core. The performance figures are given in Table 3 and are averages for 6 runs. The timings are in seconds and the sizes in kilo-bytes (KB). For the join protocol, the time measured refers to the time taken for the group issuer to generate the user's membership credential. Computing a membership credential using the F-SPHINCS+ signature scheme is the most time intensive part of the protocol. However, note that this protocol only runs once when a user joins the group. For signing and verification, the running time with large groups is in the order of minutes. The current implementation is not optimized, it was written just using C++ for clarity and not speed. There are a number of possible improvements that we target as the next step: (1) Currently the NIZK proof is obtained by using the Fiat-Shamir transformation from the 3-round KKW interactive protocol. Because the MPC protocol in KKW is with pre-processing and cut-and-choose, an NIZK proof transformed from the 5-round interactive protocol can be more efficient; (2) An optimized implementation can use mutlitple-cores, available SIMD instructions and a fast matrix library (for example, M4RI [2]) to improve the overall performance; However, this will mean that we loose generality and will require different executables for each processor.

The sizes in Table 3 are in KB. For the signatures the size is measured from the raw binary data, while for the credential, this is the output file size. The credential files do contain some extra meta information and are written out as hex strings. A binary format would reduce these figures.

6 Conclusions

Research into post-quantum group signatures based on symmetric primitives has recently drawn a lot of attention from both academia and industry, but it is

Table 3: Test results. Time is measured in seconds, size is measured in KB.

n	group size	join	sign	verify	cred. size	sig. size
129	2^{10}	66.3	8.7	4.4	60.6	571
	2^{20}	99.2	12.9	6.3	90.9	851
	2^{40}	164.7	21.4	10.2	152.5	1419
	2^{60}	230.6	29.4	13.9	214.4	2000
255	2^{10}	274.5	36.4	17.3	229.9	2336
	2^{20}	410.3	53.0	25.6	344.8	3483
	2^{40}	682.8	89.4	43.0	576.8	5784
	2^{60}	952.2	125.1	60.0	809.4	8097

still at an early stage. This paper addresses an open problem: how this type of group signature can handle a large group size while maintaining robust security properties. We propose a new group signature scheme from symmetric primitives, which supports a large group size suitable for real-world use. To do this, we modify the SPHINCS+ signature scheme to create a new group membership credential and use the MPC-in-the-head paradigm to prove the possession of the credential and the signing key in a NIZK manner. The security analysis of our scheme shows that it supports all of the properties required from a state-of-the-art security model for group signatures. We present several optimizations to improve performance and provide a prototype implementation. Our future work on group signatures from symmetric primitives will focus on designing schemes with more robust security properties, such as forward anonymity. While, for the current scheme, we will work on improving performance, obtaining more practical benchmarks.

Acknowledgments

We thank the European Union's Horizon research and innovation program for support under grant agreement numbers: 101069688 (CONNECT), 101070627 (REWIRE), 779391 (FutureTPM), 952697 (ASSURED), 101019645 (SECANT) and 101095634 (ENTRUST). These projects are funded by the UK government's Horizon Europe guarantee and administered by UKRI. We also thank the National Natural Science Foundation of China for support under grant agreement numbers: 62072132 and 62261160651. We also thank the anonymous reviewers for their valuable comments.

References

[1] Quentin Alamélou, Olivier Blazy, Stéphane Cauchie, and Philippe Gaborit. A code-based group signature scheme. *Designs, Codes and Cryptography*, 82(1):469–493, 2017.

[2] Martin Albrecht and Gregory Bard. The M4RI Library, Accessed in Oct 2023.

[3] Rachid El Bansarkhani and Rafael Misoczki. G-Merkle: A hash-based group signature scheme from standard assumptions. In *PQCrypto*, pages 441–463, 2018.

[4] Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. In *PKC*, pages 495–526, 2020.

[5] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *Proceedings of the 37th Symposium on Foundations of Computer Science, IEEE*, 1996.

[6] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS$^+$ signature framework. In *ACM CCS*, pages 2129–2146, 2019.

[7] Ward Beullens, Samuel Dobson, Shuichi Katsumata, Yi-Fu Lai, and Federico Pintore. Group signatures and more from isogenies and lattices: Generic, simple, and efficient. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT*, pages 95–126, 2022.

[8] Dan Boneh, Saba Eskandarian, and Ben Fisch. Post-quantum EPID signatures from symmetric primitives. In *CT-RSA*, pages 251–271, 2019.

[9] Dan Boneh and Hovav Shacham. Group signatures with verifier-local revocation. In *ACM CCS*, pages 168–177, 2004.

[10] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, and Jens Groth. Foundations of fully dynamic group signatures. *Journal of Cryptology*, 33(4):1822–1870, 2020.

[11] Cecilia Boschini, Jan Camenisch, and Gregory Neven. Floppy-sized group signatures from lattices. In *International Conference on Applied Cryptography and Network Security*, pages 163–182. Springer, 2018.

[12] Cecilia Boschini, Jan Camenisch, and Gregory Neven. Relaxed lattice-based signatures with short zero-knowledge proofs. In *International Conference on Information Security*, pages 3–22. Springer, 2018.

[13] Cecilia Boschini, Jan Camenisch, Max Ovsiankin, and Nicholas Spooner. Efficient post-quantum snarks for RSIS and RLWE and their applications to privacy. In Jintai Ding and Jean-Pierre Tillich, editors, *PQCrypto*, pages 247–267, 2020.

[14] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *ACM CCS*, pages 132–145, 2004.

[15] Ernie Brickell and Jiangtao Li. Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. *IEEE Trans. Dependable Secur. Comput.*, 9(3):345–360, 2012.

[16] Maxime Buser, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, and Shifeng Sun. DGM: A dynamic and revocable group Merkle signature. In *ESORICS*, pages 194–214, 2019.

[17] Jan Camenisch and Els Van Herreweghen. Design and implementation of the *idemix* anonymous credential system. In *ACM CCS*, pages 21–30, 2002.

[18] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, pages 61–76, 2002.

[19] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *ACM CCS*, pages 1825–1842, 2017.

[20] David Chaum and Eugène van Heyst. Group signatures. In Donald W. Davies, editor, *EUROCRYPT*, pages 257–265, 1991.

[21] Kai-Min Chung, Yao-Ching Hsieh, Mi-Ying Huang, Yu-Hsuan Huang, Tanja Lange, and Bo-Yin Yang. Group signatures and accountable ring signatures from isogeny-based assumptions. Cryptology ePrint Archive, Paper 2021/1368, 2021. `https://eprint.iacr.org/2021/1368`.

[22] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge mpcith-based arguments. In *ACM CCS*, pages 3022 – 3036, 2021.

[23] Rafaël del Pino, Vadim Lyubashevsky, and Gregor Seiler. Lattice-based group signatures and zero-knowledge proofs of automorphism stability. In *ACM CCS*, pages 574–591, 2018.

[24] Christoph Dobraunig, Daniel Kales, Christian Rechberger, Markus Schofnegger, and Greg Zaverucha. Shorter signatures based on tailor-made minimalist symmetric-key crypto. In *ACM CCS*, pages 843–857, 2022.

[25] Muhammed F. Esgin, Raymond K. Zhao, Ron Steinfeld, Joseph K. Liu, and Dongxi Liu. MatRiCT: Efficient, scalable and post-quantum blockchain confidential transactions protocol. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS*, pages 567–584, 2019.

[26] Martianus Frederic Ezerman, Hyung Tae Lee, San Ling, Khoa Nguyen, and Huaxiong Wang. Provably secure group signature schemes from code-based assumptions. *IEEE Transactions on Information Theory*, 66(9):5754–5773, 2020.

[27] Free Software Foundation, Inc. GCC, the GNU Compiler Collection. `https://gcc.gnu.org`, 2022.

[28] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In *USENIX Security*, pages 1069–1083, 2016.

[29] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[30] Xiuju Huang, Jiashuo Song, and Zichen Li. Dynamic group signature scheme on lattice with verifier-local revocation. Cryptology ePrint Archive, Paper 2022/022, 2022. `https://eprint.iacr.org/2022/022`.

[31] A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen. Xmss: extended merkle signature scheme. RFC 8391, RFC Editor, May 2018.

[32] Intel. Intel enhanced privacy id (epid) security technology. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-enhanced-privacy-id-epid-security-technology.html`, 2021.

[33] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *STOC*, pages 21–30, 2007.

[34] ISO Central Secretary. Information technology — security techniques — hash-functions — part 2: Hash-functions using an n-bit block cipher. Standard ISO/IEC TR 10118-2:2010, International Organization for Standardization, Geneva, CH, 2010.

[35] Daniel Kales and Greg Zaverucha. Efficient lifting for shorter zero-knowledge proofs and post-quantum signatures. Cryptology ePrint Archive, Paper 2022/588, 2022. `https://eprint.iacr.org/2022/588`.

[36] Meenakshi Kansal, Ratna Dutta, and Sourav Mukhopadhyay. Forward secure efficient group signature in dynamic setting using lattices. Cryptology ePrint Archive, Paper 2017/1128, 2017. `https://eprint.iacr.org/2017/1128`.

[37] Shuichi Katsumata and Shota Yamada. Group signatures without NIZK: from lattices in the standard model. In *EUROCRYPT*, pages 312–344, 2019.

[38] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In *ACM CCS*, pages 525–537, 2018.

[39] Aggelos Kiayias and Moti Yung. Extracting group signatures from traitor tracing schemes. In *EUROCRYPT*, pages 630–648, 2003.

[40] Evgeniy O. Kiktenko, Andrey A. Bulychev, P. A. Karagodin, Nikolay O. Pozhar, Maxim N. Anufriev, and Aleksey K. Fedorov. Sphincs$^+$ digital signature scheme with GOST hash functions. *CoRR*, abs/1904.06525, 2019.

[41] Seongkwang Kim, Jincheol Ha, Mincheol Son, Byeonghak Lee, Dukjae Moon, Joohee Lee, Sangyub Lee, Jihoon Kwon, Jihoon Cho, Hyojin Yoon, et al. AIM: Symmetric primitive for shorter signatures with stronger security. *Cryptology ePrint Archive*, 2022/1387.

[42] Fabien Laguillaumie, Adeline Langlois, Benoît Libert, and Damien Stehlé. Lattice-based group signatures with logarithmic signature size. In *International conference on the theory and application of cryptology and information security*, pages 41–61. Springer, 2013.

[43] Yi-Fu Lai and Samuel Dobson. Collusion resistant revocable ring signatures and group signatures from hard homogeneous spaces. Cryptology ePrint Archive, Paper 2021/1365, 2021. `https://eprint.iacr.org/2021/1365`.

[44] Leslie Lamport. Constructing digital signatures from a one-way function. *Tech. Report: SRI International Computer Science Laboratory*, 1979.

[45] Adeline Langlois, San Ling, Khoa Nguyen, and Huaxiong Wang. Lattice-based group signature scheme with verifier-local revocation. In *International workshop on public key cryptography*, pages 345–361. Springer, 2014.

[46] Benoît Libert, San Ling, Fabrice Mouhartem, Khoa Nguyen, and Huaxiong Wang. Signature schemes with efficient protocols and dynamic group signatures from lattice assumptions. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 373–403. Springer, 2016.

[47] San Ling, Khoa Nguyen, and Huaxiong Wang. Group signatures from lattices: simpler, tighter, shorter, ring-based. In *IACR International Workshop on Public Key Cryptography*, pages 427–449. Springer, 2015.

[48] San Ling, Khoa Nguyen, Huaxiong Wang, and Yanhong Xu. Lattice-based group signatures: achieving full dynamicity with ease. In *International Conference on Applied Cryptography and Network Security*, pages 293–312. Springer, 2017.

[49] San Ling, Khoa Nguyen, Huaxiong Wang, and Yanhong Xu. Constant-size group signatures from lattices. In *IACR International Workshop on Public Key Cryptography*, pages 58–88. Springer, 2018.

[50] Vadim Lyubashevsky, Ngoc Khanh Nguyen, Maxime Plançon, and Gregor Seiler. Shorter lattice-based group signatures via "almost free" encryption and other optimizations. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT*, pages 218–248, 2021.

[51] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1987.

[52] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO*, pages 218–238, 1989.

[53] Khoa Nguyen, Hanh Tang, Huaxiong Wang, and Neng Zeng. New code-based privacy-preserving cryptographic constructions. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 25–55. Springer, 2019.

[54] NIST. Post-quantum cryptography standardization. `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization`, 2017-2022.

[55] NIST. NIST announces first four quantum-resistant cryptographic algorithms. `nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms`, 2022.

[56] Satyam Omar and Sahadeo Padhye. Multivariate linkable group signature scheme. In *Proceedings of the International Conference on Computing and Communication Systems*, pages 623–632. Springer, 2021.

[57] Yusuke Sakai, Jacob CN Schuldt, Keita Emura, Goichiro Hanaoka, and Kazuo Ohta. On the security of dynamic group signatures: Preventing signature hijacking. In *International Workshop on Public Key Cryptography*, pages 715–732. Springer, 2012.

[58] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO*, pages 239–252, 1989.

[59] Masoumeh Shafieinejad and Navid Nasr Esfahani. A scalable post-quantum hash-based group signature. *Des. Codes Cryptogr.*, 89(5):1061–1090, 2021.

[60] Guangdong Yang, Shaohua Tang, and Li Yang. A novel group signature scheme based on MPKC. In *International Conference on Information Security Practice and Experience*, pages 181–195. Springer, 2011.

[61] Mahmoud Yehia, Riham AlTawy, and T. Aaron Gulliver. $\mathrm{Gm}^{\mathrm{mt}}$: A revocable group merkle multi-tree signature scheme. In *CANS*, pages 136–157, 2021.

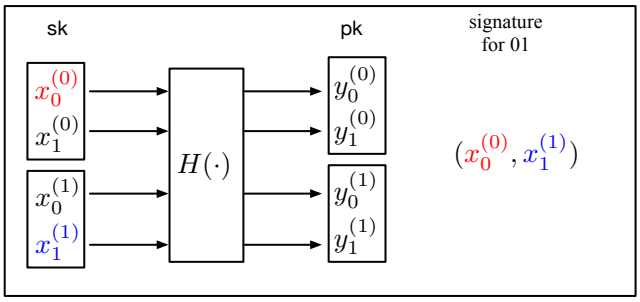
[62] Mahmoud Yehia, Riham AlTawy, and T. Aaron Gulliver. Security analysis of DGM and GM group signature schemes instantiated with XMSS-T. In *Inscrypt*, pages 61–81, 2021.

[63] Zaverucha, Ramacher, Kales, and Goldfeder. Reference implementation of the picnic post-quantum signature scheme. `https://github.com/Microsoft/Picnic`, 2020.

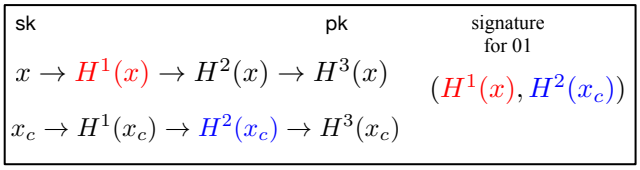
[64] Gerg Zaverucha. The picnic signature algorithm specification. Supporting Documentation in `https://github.com/Microsoft/Picnic`, 2020.

A Hash-based Signatures

Lamport one-time signature The earliest hash-based signature scheme is by Lamport [44]. An example is depicted in Figure 11(a). Let us start with signing 1 bit. For this bit, a public/private key pair is generated. The private key is a pair of random strings (x_0, x_1) and the public key is (y_0, y_1) such that $y_b = H(x_b)$ is the hash value of x_b. After publishing the public key, the signature of the bit b is then the secret string in the private key corresponding to b, i.e. $\sigma = x_b$. Anyone who wants to verify the signature can compute $y'_b = H(\sigma)$, then compare that to y_b in the public key. If $y'_b = y_b$, then accept, otherwise reject. This can be easily extended to signing an arbitrary message: first hash the message to produce a hash value of l-bit, then generate l key pairs and sign the message hash bit-wisely. The Lamport signature scheme is a one-time signature scheme because each key pair can only be used to sign one message, otherwise, an adversary can forge signatures from the generated ones. For example, if signatures on two message hashes, 00 and 11, have been generated using the same key pair, then the private keys can be fully recovered from the two signatures, and one can easily forge a valid Lamport signature for message hashes 10 and 01.



(a) Lamport one-time signature



(b) Winternitz one-time signature

Figure 11: One-time signatures

Winternitz one-time signature Bit-wise signing can be inefficient, therefore schemes that can sign a block of multiple bits each time have been proposed. The Winternitz one-time signature (WOTS, depicted in Fig. 11(b)) [52] is an example. To sign a d-bit block, let $W = 2^d - 1$, the private key is a pair of two secret strings (x, x_c) and the public key is (y, y_c) such that $y = H^W(x), y_c =$

$H^W(x_c)$ where $H^W(\cdot)$ means apply H repeatedly W times. Then the d-bit block is parsed into an unsigned integer i, the signature $\sigma = (s, s_c)$ where $s = H^i(x), s_c = H^{W-i}(x_c)$. Here i is the data to be signed and $W - i$ serves as the checksum of the data. To verify the block against the signature, compute $(H^{W-i}(s), H^i(s_c))$ and check whether the pair is the same as the public key. Again, a WOTS key pair should be used for signing just one message. For example, if the signature of 00 and 11 are generated using the same key pair, then one can compute $H^1(H^0(x)) = H^1(x)$ from the signature of 00 $(H^0(x), H^3(x_c))$. Also $H^2(H^0(x_c)) = H^2(x_c)$ can be computed from $(H^3(x), H^0(x_c))$ that is the signature of 11. The pair $(H^1(x), H^2(x_c))$ is the signature of 01.

Merkle signature To allow multiple signatures per key pair, one naive method is as the following. To allow 2^a signatures, generate 2^a one-time signature key pairs. Treat the 2^a one-time public keys as one overall public key and the 2^a one-time private keys as one private key. When signing the i-th message, use the i-th private sub-key to generate the signature. This signature can be verified against the i-th sub-public key. Of course, this works but the size of the keys is large. Merkle proposed in [52] a more efficient way to implement this idea, with a much smaller storage overhead. An example is depicted in Fig. 12.

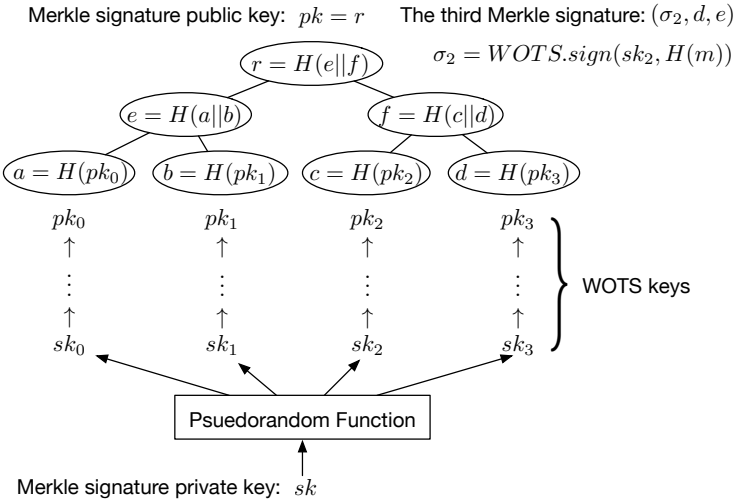


Figure 12: Merkle signatures

There are two main ideas in the proposal. The first idea is to use a pseudorandom function to generate the one-time signature key pairs. Hence the space required to store the 2^a private keys is reduced to store one short pseudorandom function key. The second idea is to compress the 2^a public keys into one short string. This is done with a Merkle tree. More precisely, the Merkle tree is a binary tree with 2^a leaf nodes. The i-th leaf node is the hash of the i-th one-time public key. Then each non-leaf node is the hash of $L||R$, i.e. its two children concatenated. The new overall public key is now the Merkle tree root.

The i-th message's signature is then the one-time signature produced under the i-th one-time signing key, plus the authentication path of the i-th leaf node in the Merkle tree. The authentication path contains all sibling nodes in the Merkle tree along the path from the i-th leaf node to the root. Verification of the signature starts by recovering the i-th one-time public key from the message and the signature, then hashes it to obtain the i-th leaf node of the Merkle tree. Then the leaf node along with its sibling in the authentication path can compute their parent, then the parent of their parent can be computed with the parent's sibling in the authentication path, and so on until reaching the root. If the recovered root is the same as the overall public key, then accept, otherwise reject.

FORS signature Another strategy for a few-time hash-based signature is by sharing a pool of private key components, and randomly disclosing a subset when signing a message. One example is the FORS signature scheme proposed in [6].

FORS is built on top of a base signature scheme that can sign d-bit blocks. To sign a d-bit block, the private key is 2^d random strings $(x_0, \cdots, x_{2^d-1})$. The public key r is the root of a Merkle tree built from the private key such that the i-th leaf node is the hash of the x_i in the private key. The signature of the block is generated by parsing the block into an unsigned integer j, then collecting x_j and the authentication path of the j-th leaf node. To verify the signature, parse the block into j, and check whether the public key (the root of the Merkle tree) can be recovered from x_j and the authentication path in the signature.

FORS uses the above base signature scheme to sign message hash that is k blocks, with each block d bits. The private key in FORS is a pseudorandom function key that is used to derive k base private keys. The public key is the hash of $r_0||\cdots||r_{k-1}$, i.e. the hash of the k base public keys. The signature consists of k base signatures, each for a block in the hash value. Verification is done by verifying every block to derive the base public key, and checking the hash of the concatenation of all recovered base public keys is the same as the overall public key. An example of FORS is depicted in Fig 13.

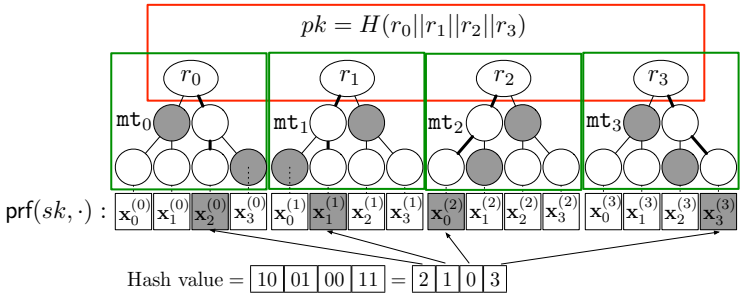


Figure 13: FORS signatures.

A FORS key pair can be used to sign q messages, where q is a function of d and k. The reason why FORS can be used to sign more than one message can be

argued roughly as follows. If we view each block in the hash value as a random number, then after q signatures have been generated, for a particular block, λ out of 2^d private strings may have been revealed by the previously generated signatures. For the message that the adversary wants to forge the signature, the probability the adversary can get the private string for that block in the message hash is the probability that the needed private string has been revealed by previous signatures, which is $\frac{\lambda}{2^d}$, and some negligible probability that the adversary can break the underlying cryptographic primitives. The message has k blocks, so the probability of getting all private strings for the whole message hash can be made negligible.

SPHINCS+ All previously mentioned multi-time signature schemes are stateful, meaning that the signer needs to keep a state (e.g. how many messages have been signed and which keys have been used). SPHINCS+ [6] is a stateless hash-based signature scheme and is one of the three digital signature schemes selected by NIST to become part of its post-quantum cryptographic standard [55]. Technically, SPHINCS+ still has an upper limit on how many signatures can be generated per key pair, it is just that the number can be made very large (e.g. 2^{60}) so that it is unlikely to be reached in practice.

Before presenting the idea in SPHINCS+, let us look back at the Merkle signature. A Merkle signature key pair can be used to sign 2^a messages. If a is large (e.g. 256), then it seems that each time we can randomly pick a leaf signing key to generate the signature, without worrying about accidentally picking the same leaf twice. Then why is not the Merkle signature stateless? The reason is that to generate the public key, all 2^a leaves need to be available in the computation, and to generate a signature, the Merkle tree (that has $2^{a+1}-1$ nodes) needs to be available so that the authentication path can be retrieved from the tree. For large a, it is computationally infeasible.

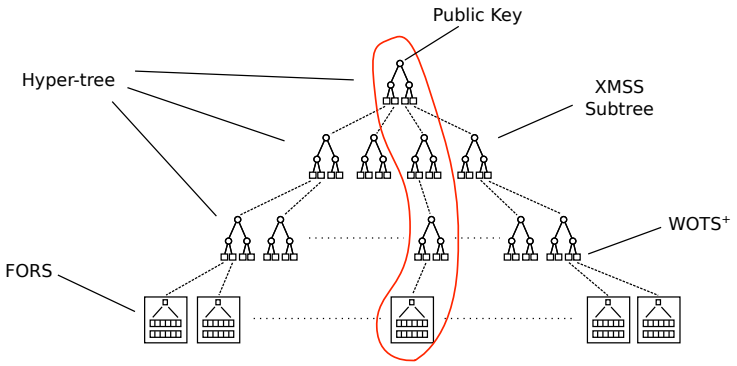


Figure 14: SPHINCS+ signature scheme, a path used in signing is shown within the red line (diagram taken from [40]).

The basic idea in SPHINCS+ that allows many signatures to be generated follows from the above, i.e. to have many signing keys and randomly pick one when signing a message. To make it practical, as shown in Fig 14, there are two

major refinements:

1. Rather than using a single Merkle tree, it uses a hyper-tree (a tree of trees), in which each non-leaf node is an XMSS signature[3] key pair. Each XMSS key pair aggregates 2^b WOTS+ (a more efficient variant of WOTS) key pairs for some small b. The XMSS key pair is used to sign the public keys of its children. Each XMSS key pair can generate 2^b signatures, so each node has 2^b children. If we want 2^a leaf nodes that hold the actual key pairs for signing the message hash in the hyper-tree, we can do so by setting the tree height to $\lceil \frac{a}{b} \rceil$. It is worth noting that each XMSS key pair in a hyper-tree node can be generated independently. Therefore the complexity of generating SPHINCS+ public keys and signatures is much less than the naive Merkle approach. The public key in SPHINCS+ is just the XMSS public key in the hyper-tree root. Hence when computing the public key, in SPHINCS+ only $O(2^b)$ computation is needed. To sign a message, only the path from one leaf node to the root in the hyper-tree is needed. Also, the nodes in the path can be generated on the fly without knowing anything about nodes not in this path.

2. Each leaf node of the hyper-tree holds a FORS key pair. Because FORS is a few-time signature scheme, using FORS in the leaf allows the same signing key to be used more than once. The benefit of doing so is that the scheme now can achieve unforgeability with a smaller tree height/size compared to if the leaf nodes hold one-time signing keys. This in consequence reduces the overhead when generating/verifying the SPHINCS+ signatures.

A SPHINCS+ signature is a list of $h+1$ signatures, where h is the height of the hyper-tree used. To sign a message, the message is hashed into a message hash to be signed and a random index of the leaf node. The FORS key in the chosen leaf node is used to sign the message hash, and the FORS signature is the first in the list. Then the public key in the leaf node is signed by the XMSS key in its parent node, and the parent's public key is signed by the parent's parent, and so on until reaching the root. The h XMSS signatures are appended after the FORS signature, and the list of signatures is the SPHINCS+ signature. Verification starts from using the message and the FORS signature to recover the FORS public key in the leaf node, then using that and the next XMSS signature to recover the parent's public key, iteratively until reaching the root. If the recovered root key is the same as the SPHINCS+ public key, then accept, otherwise reject. The idea is similar to PKI, in which to verify a signature generated by an untrusted CA, we need to find a chain of signatures leading to a trusted CA.

[3]The hypertree might be viewed as a fixed input-length version of multi-tree XMSS (XMSS^{MT}).

B Algorithms in M-FORS

M-FORS consists of the following algorithms:

- $\mathsf{keyGen}(\mathtt{seed}, n, d, k, \mathtt{aux})$: it takes as input a random seed, a security parameter n, positive integers d and k, and `aux` that is either an empty string or some optional data. The algorithm first call *expandSeed*(`aux,n,d,k,seed`) (see Algorithm 1). In the subroutine, the seed is expanded into $\mathbf{X} = (\mathbf{x}^{(0)}, \cdots, \mathbf{x}^{(k-1)})$ using a pseudorandom function, where each $\mathbf{x}^{(i)}$ contains 2^d distinct n-bit pseudorandom strings.

Algorithm 1: expandSeed($\mathtt{aux}, n, d, k, \mathtt{seed}$)

Input: `aux` is the auxiliary data or an empty string, n, d, k are the M-FORS parameters, `seed` is the seed.

Output: $(pk, \mathbf{X}, \mathbf{T})$, where pk is the M-FORS public key, $\mathbf{X}$ is a list of list (containing $k \cdot 2^d$ secret strings), and $\mathbf{T}$ contains k Merkle trees built on the strings in $\mathbf{X}$.

initiate $pk, \mathbf{X}, \mathbf{T}, \mathbf{rs}$;
for $i = 0; i \leq k-1; i++$ **do**
  /* generate the i-th list of the secret strings */
  initiate $\mathbf{x}^{(i)}$;
  for $j = 0; j \leq 2^d - 1; j++$ **do**
    $\mathbf{x}_j^{(i)}$=$\mathsf{prf}(\mathtt{seed}, \mathtt{aux}||i||j)$;
  end
  append $\mathbf{x}^{(i)}$ to $\mathbf{X}$;
  /* generate the i-th Merkle tree */
  $\mathtt{mt}_i$ = genMerkleTree($\mathtt{aux}, i, d, \mathbf{x}^{(i)}, \mathtt{FALSE}$);
  append $\mathtt{mt}_i$ to $\mathbf{T}$;
  /* $\mathtt{mt}_i[0][0]$ stores the root of $\mathtt{mt}_i$ */
  append $\mathtt{mt}_i[0][0]$ to $\mathbf{rs}$;
end
/* compute the last Merkle tree from all roots */
$\mathtt{mt}_k$ = genMerkleTree($\mathtt{aux}, k, \lceil \log_2(k) \rceil, \mathbf{rs}, \mathtt{TRUE}$);
$pk = \mathtt{mt}_k[0][0]$;

While expanding the seed, $k+1$ Merkle trees $\mathbf{T} = (\mathtt{mt}_0, \cdots, \mathtt{mt}_k)$ are built. In particular, each of $\mathtt{mt}_0, \cdots, \mathtt{mt}_{k-1}$ is built using $\mathbf{x}^{(i)}$ as the input, and $\mathtt{mt}_k$ is built using $r_0, \cdots, r_{k-1}$ that are the roots of $(\mathtt{mt}_0, \cdots, \mathtt{mt}_{k-1})$ (see Algorithm 2).

keyGen outputs $(pk, sk, param)$, such that the public key $pk = r_k$ where r_k is the root of $\mathtt{mt}_k$, the private key $sk = \mathtt{seed}$, and the public parameters $mp = (n, d, k, \mathtt{aux})$.

- $\mathsf{sign}(sk, MD, mp)$: to sign a message hash $MD \in \{0,1\}^{k \cdot d}$, parse it into k blocks, each block is interpreted as a d-bit unsigned integer $(p_0, \cdots, p_{k-1})$.

Algorithm 2: genMerkleTree($\mathtt{aux}, i, d, \mathbf{x}, isLast$)

Input: to generate the i-th Merkle tree (of height ht), using $\mathbf{x}$ as input, $\mathtt{aux}$ is the auxiliary data or an empty string, $isLast$ indicates whether this is the last Merkle tree in an M-FORS tree.

Output: $\mathtt{mt}$, a triangle matrix storing the Merkle tree nodes.

initiate $\mathtt{mt}$ with height ht;
/* assign values to leaf nodes */
if $isLast ==$ *TRUE* **then**
  /* the last tree is built from k Merkle tree roots */
  $\mathtt{mt}[ht] = \mathbf{x}$;
else
  /* all other trees are built from hashes of secret key components */
  for $j = 0; j \leq \boldsymbol{x}.\mathit{length} - 1; j++$ **do**
    $\mathtt{mt}[ht][j] = H_1(\mathtt{aux}||i||ht||j||\mathbf{x}_j)$;
  end
end
pad $\mathtt{mt}[ht]$ to 2^{ht} elements with unary strings 0^n;
/* upper layers except for the root */
for $l = ht - 1; l \geq 1; l--$ **do**
  for $j = 0; j \leq 2^l - 1; j++$ **do**
    $L = \mathtt{mt}[l+1][2j]$;
    $R = \mathtt{mt}[l+1][2j+1]$;
    $\mathtt{mt}[l][j] = H_1(\mathtt{aux}||i||l||j||L||R)$
  end
end
/* root node, size is $d \cdot k$ for the last tree, n for the others */
if $isLast$=*TRUE* **then**
  $\mathtt{mt}[0][0] = H_2(\mathtt{aux}||i||0||0||\mathtt{mt}[1][0]||\mathtt{mt}[1][1])$
else
  $\mathtt{mt}[0][0] = H_1(\mathtt{aux}||i||0||0||\mathtt{mt}[1][0]||\mathtt{mt}[1][1])$
end

Then *expandSeed*(`aux,n,d,k,seed`) is called to obtain $\mathbf{X}$ and $\mathbf{T}$.

Next, for $0 \leq i \leq k-1$, the i-th block p_i, $\mathbf{x}^{(i)}$ and $\mathtt{mt}_i$ are used to generate $\mathtt{authpath}_{p_i}^{(i)}$, which is the authentication path of the p_i-th leaf node in the i-th Merkle tree. Then $(\mathbf{x}_{p_i}^{(i)}, \mathtt{authpath}_{p_i}^{(i)})$ is put into the signature. The signature is a list of k pairs $\boldsymbol{\sigma} = \{(\mathbf{x}_{p_0}^{(0)}, \mathtt{authpath}_{p_0}^{(0)}), \cdots, (\mathbf{x}_{p_{k-1}}^{(k-1)}, \mathtt{authpath}_{p_{k-1}}^{(k-1)})\}$. The details of the algorithm can be found in Algorithm 3 and 4.

Algorithm 3: sign$(sk, MD, d, k, \mathtt{aux})$

Input: $sk = \mathtt{seed}$ is the signing key, $MD \in \{0,1\}^{d \cdot k}$ is the message digest to be signed, $d, k, \mathtt{aux}$ are the M-FORS parameters.

Output: $\boldsymbol{\sigma}$, the signature of M under sk.

```
parse MD as k d-bit unsigned integers p_0||...||p_{k-1};
(pk, X, T) = expandSeed(aux,n,d,k,seed);
initiate σ;
for i = 0; i ≤ k-1; i++ do
    /* the list X = {x^(0), ..., x^(k-1)}                          */
    x = x_{p_i}^(i);
    /* get the authentication path for leaf_{p_i} in the i-th tree */
    mt = T[i];
    authPath = getAuthpath(mt, p_i, d);
    append (x, authPath) to σ
end
```

- recoverPK(σ, MD, mp): This algorithm output the public key recovered from a signature σ and the message hash MD. First MD is parsed into k blocks $(p'_0, \cdots, p'_{k-1})$. Then for $0 \leq i \leq k-1$, $\boldsymbol{\sigma}_i = (x_i, \mathtt{authpath}^{(i)})$ and p'_i are used to re-generate a Merkle tree root and get the value r'_i (p'_i is used to determine the order of the siblings at each layer). Finally, $r'_0, \cdots, r'_{k-1}$ are used to compute $\mathtt{mt}'_k$ and its root r'_k is returned. See Algorithm 5 and 6.

- verify(σ, pk, MD, mp): to verify a signature, recoverPK(σ, MD, mp) is called. If the recovered public key is the same as pk, accept the signature, otherwise reject. The details of the algorithm can be found in Algorithm 7.

- partial-sig(σ, MD, i, mp): to extract a partial signature of the i-th block of MD from a full signature $\sigma = \{(x_0, \mathbf{authpath}^{(0)}), \cdots, (x_{k-1}, \mathbf{authpath}^{(k-1)})\}$. The Merkle tree $\mathtt{mt}_k$ can be recomputed from σ. The partial signature is $\partial_{\sigma,i} = (x_i, \mathbf{authpath}^{(i)}, \mathbf{authpath}^{(k,i)})$ where $(x_i, \mathbf{authpath}^{(i)})$ is a copy of the i-th pair in σ, and $\mathbf{authpath}^{(k,i)}$ is the authentication path of r_i (the root of the i-th Merkle tree) in $\mathtt{mt}_k$. The details of the algorithm can be found in Algorithm 8.

- partial-rec$(\partial_{\sigma,i}, p_i, i, mp)$: This algorithm recovers the public key from $\partial_{\sigma,i}$ and p_i. Given $\partial_{\sigma,i} = (x, \mathbf{authpath}, \mathbf{authpath}')$, first compute the Merkle

Algorithm 4: getAuthpath($\texttt{mt}, p, d$)

Input: `mt` is a Merkle tree, p is the index of the leaf node to be authenticated, and d is the height of the tree.
Output: **authPath**, the authentication path.

initiate **authPath**;
$idx = p$;
for $j = d; j \geq 1; j--$ **do**
  if *idx is an even number* **then**
    /* this node is a left child, so add the right child to authPath */
    append $\texttt{mt}[j][idx+1]$ to **authPath**;
  else
    append $\texttt{mt}[j][idx-1]$ to **authPath**;
  end
  /* the index of the parent node */
  $idx = \lfloor idx/2 \rfloor$;
end

Algorithm 5: recoverPK($\sigma, MD, d, k, \texttt{aux}$)

Input: $\sigma = \{(x_0, \textbf{authpath}^{(0)}), \cdots, (x_{k-1}, \textbf{authpath}^{(k-1)})\}$ is the signature, MD is the message digest, k d, and `aux` are the M-FORS parameters.
Output: pk, the last Merkle tree root in M-FORS

parse MD as k d-bit unsigned integers $p_0 || \cdots || p_{k-1}$;
initiate **rs**;
for $i = 0; i \leq k-1; i++$ **do**
  $temp = H_1(\texttt{aux}||i||d||p_i||x_i)$;
  /* recompute the i-th root from a leaf and its authpath */
  $r = \text{reconRoot}(temp, \textbf{authpath}^{(i)}, p_i, d, i, \texttt{aux}, \texttt{FALSE})$;
  append r to **rs**;
end
$\texttt{mt}_k = \text{genMerkleTree}(\texttt{aux}, k, \lceil \log_2(k) \rceil, \textbf{rs}, \texttt{TRUE})$;
$pk = \texttt{mt}_k[0][0]$;

Algorithm 6: reconRoot($leaf, \mathbf{authpath}, p, d, i, \mathtt{aux}, isLast$)

Input: $leaf$ is the leaf node to be verified in the Merkle tree, **authpath** is the authentication path of leaf, p is the index of the leaf node, d is the height of the Merkle tree, i is the index of the Merkle tree, aux is the auxiliary data or an empty string, $isLast$ indicates whether this is the last Merkle tree in M-FORS.

Output: r, the Merkle root recomputed from $leaf$ and **authpath**.

$idx = p, temp = leaf$;
for $l = d - 1; l \geq 1; l - -$ **do**
  $parentIdx = \lfloor idx/2 \rfloor$;
  if *idx is an even number* **then**
    $temp = H_1(\mathtt{aux}||i||l||parentIdx||temp||\mathbf{authpath}_{d-1-l})$
  else
    $temp = H_1(\mathtt{aux}||i||l||parentIdx||\mathbf{authpath}_{d-1-l}||temp)$
  end
  $idx = parentIdx$;
end
if *idx is an even number* **then**
  $L = temp, R = \mathbf{authpath}_{d-1}$;
else
  $R = temp, L = \mathbf{authpath}_{d-1}$;
end
if $isLast == \mathtt{TRUE}$ **then**
  $r = H_2(\mathtt{aux}||i||0||0||L||R)$;
else
  $r = H_1(\mathtt{aux}||i||0||0||L||R)$;
end

Algorithm 7: verify($\sigma, pk, MD, d, k, \mathtt{aux}$)

Input: σ is the signature, pk is the public key, $MD \in \{0,1\}^{d \cdot k}$ is the message digest to be verified, $d, k, \mathtt{aux}$ are the M-FORS parameters.

Output: Accept or Reject.

$pk' =$ recoverPK($\sigma, MD, d, k, \mathtt{aux}$);
if $pk' == pk$ **then**
  output Accept ;
else
  output Reject ;
end

tree root r_1 from $(x, \mathbf{authpath}, p_i)$, then compute the Merkle tree root r_2 from $(r_1, \mathbf{authpath}', i)$. Output r_2. The details of the algorithm can be found in Algorithm 9.

Algorithm 8: partial-sig(σ, MD, i, k, d)

Input: $\sigma = \{(x_0, \mathbf{authpath}^{(0)}), \cdots, (x_{k-1}, \mathbf{authpath}^{(k-1)})\}$ is the signature, M is the message, i is the index of the message block to be authenticated by the partial signature, k and d are the M-FORS parameters.

Output: $\partial_{\sigma,i}$, the partial signature

parse MD as k d-bit unsigned integers $p_0||\cdots||p_{k-1}$;
initiate **rs**;
for $i = 0; i \leq k-1; i++$ **do**
    $temp = H_1(\mathtt{aux}||i||d||p_i||x_i)$;
    /* recompute the i-th root from a leaf and its authpath */
    $r = \text{reconRoot}(temp, \mathbf{authpath}^{(i)}, p_i, d, i, \mathtt{aux}, \mathtt{FALSE})$;
    append r to **rs**;
end
$\mathtt{mt}_k = \text{genMerkleTree}(\mathtt{aux}, k, \lceil \log_2(k) \rceil, \mathbf{rs}, \mathtt{TRUE})$;
$\mathbf{authpath}^{(k,i)} = \text{getAuthpath}(\mathtt{mt}_k, i, \lceil \log_2(k) \rceil)$;
$\partial_{\sigma,i} = (x_i, \mathbf{authpath}^{(i)}, \mathbf{authpath}^{(k,i)})$

Algorithm 9: partial-rec$(\partial_{\sigma,i}, p_i, i, d, k, \mathtt{aux})$

Input: $\partial_{\sigma,i} = (x, \mathtt{authpath}, \mathtt{authpath}')$ is the partial signature, p_i is the message block authenticated by the partial signature, i is the index of the block, $d, k, \mathtt{aux}$ are the M-FORS parameters.

Output: pk the recovered M-FORS public key.

$temp = H_1(\mathtt{aux}||i||d||p_i||x)$;
$r_1 = \text{reconRoot}(temp, \mathbf{authpath}, p_i, d, i, \mathtt{aux}, \mathtt{FALSE})$;
$pk = \text{reconRoot}(r_1, \mathbf{authpath}', i, \lceil \log_2(k) \rceil, k, \mathtt{aux}, \mathtt{TRUE})$;

C Tweakable Hash Functions

Recall the definition of tweakable hash functions from [6]:

Definition 2 (Tweakable Hash Function [6]). *Let $n, a \in \mathbb{N}$, $\mathcal{P}$ the public parameters space and $\mathcal{T}$ the tweak space. A tweakable hash function is an efficient function*

$$Th : \mathcal{P} \times \mathcal{T} \times \{0,1\}^a \to \{0,1\}^n, MD \leftarrow Th(P, T, M)$$

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

In the definition, P is essentially a *public* hash key and T is a nonce that makes hash function calls independent. With the combination, the security of hash-based signatures (using Th) can be reduced to security properties weaker than collision resistance.

We will use two security properties provided by tweakable hash functions: post-quantum single function, multi-target-collision resistance for distinct tweaks (SM-TCR) and post-quantum single function, multi-target decisional second-preimage resistance for distinct tweaks (SM-DSPR).

SM-TCR allows an adversary to choose p targets, each is hashed with a distinct tweak, in a game similar to the usual target collision game. More formally:

Definition 3 (SM-TCR [6]). *In the following let Th be a tweakable hash function as defined above. We define the success probability of any adversary $A = (A_1, A_2)$ against the SM-TCR security of Th. The definition is parameterized by the number of targets p for which it must hold that $p \leq |\mathcal{T}|$. In the definition, A_1 is allowed to make p queries to an oracle $Th(P, \cdot, \cdot)$. We denote the set of A_1's queries by $Q = \{(T_i, M_i)\}_{i=1}^p$ and define the predicate $DIST(\{T_i\}_{i=1}^p) = (\forall i, k \in [1, p], i \neq k) : T_i \neq T_k$, i.e., $DIST(\{T_i\}_{i=1}^p)$ outputs 1 if all tweaks are distinct. A tweakable hash function Th is said to be SM-TCR if for all adversary A, the following holds:*

$$\begin{aligned} Succ_{Th,p}^{SM-TCR}(A(n)) = Pr\Big[& P \xleftarrow{R}; S \leftarrow A_1^{Th(P,\cdot,\cdot)(n)}; \\ & (j, M) \leftarrow A_2(Q, S, P, n) : Th(P, T_j, M_j) = Th(P, T_j, M) \\ & \wedge M \neq M_j \wedge DIST(\{T_i\}_{i=1}^p] \leq negl(n) \end{aligned}$$

SM-DSPR captures the notion that it is hard for an adversary to distinguish whether a certain hash value has more than 1 preimage or just 1. In [6], the intuition behind this property is elaborated in detail. To define SM-DSPR, a second-preimage-exists predicate is defined as the following:

Definition 4 (SPexists for tweakable hash functions [6]). *The second-preimage-exists predicate $SPexists(Th)$ for a tweakable hash function th is the function $SP : \mathcal{P} \times \mathcal{T} \times \{0,1\}^a \to \{0,1\}$ defined as follows:*

$$SP_{P,T}(M) = \begin{cases} 1 & \text{if } \left| Th_{P,T}^{-1}(Th(P, T, M)) \right| \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

where $Th^{-1}_{P,T}$ refers to the inverse of the function obtained by fixing the first two inputs to th to the given values.

The formal definition of SM-DSPR is as follows:

Definition 5 (SM-DSPR [6])**.** *In the following let Th be a tweakable hash function as defined above. We define the success probability of any adversary $A = (A_1, A_2)$ against the SM-DSPR security of Th. The definition is parameterized by the number of targets p for which it must hold that $p \leq |\mathcal{T}|$. In the definition, A_1 is allowed to make p queries to an oracle $Th(P, \cdot, \cdot)$. The query set Q and predicate $DIST(\{T_i\}_{i=1}^p)$, are defined as in Definition 3. A tweakable hash function Th is said to be SM-DSPR if for all adversary A, the following holds:*

$$Adv^{SM-DSPR}_{Th,p}(A(n)) = max(0, succ - triv) \leq negl(n)$$

with

$$\begin{aligned} succ = Pr\Big[P \xleftarrow{R} \mathcal{P}; S \leftarrow A_1^{Th(P,\cdot,\cdot)}(n); (j,b) \leftarrow A_2(Q,S,P,n) : \\ SP_{P,T_j}(M_j) = b \wedge DIST(\{T_i\}_{i=1}^p\Big] \\ triv = Pr\Big[P \xleftarrow{R} \mathcal{P}; S \leftarrow A_1^{Th(P,\cdot,\cdot)}(n); (j,b) \leftarrow A_2(Q,S,P,n) : \\ SP_{P,T_j}(M_j) = 1 \wedge DIST(\{T_i\}_{i=1}^p\Big] \end{aligned}$$

Like several other hash-based signature schemes, we use a hash function to select a random subset of a given set and derive the signature from this subset. The security is based on the notion of target subset resilience of the hash function. In such schemes, the signing key is essentially a set of random strings. Each signature generated by such schemes includes a random subset of the signing key components. Target Subset Resilience requires that the signing of q messages does not leak enough signing key components that allow an adversary to forge a signature of a chosen message.

Below we present the definition of Interleaved Target Subset Resilience (ITSR) from [6]. ITSR further considers that there exist 2^h distinct signing keys and the hash function is also used to randomly choose one from them when signing messages. Target Subset Resilience (TSR) can be defined as a special case of ITSR in which $h = 0$.

Definition 6 (ITSR)**.** *Let $H : \{0,1\}^n \times \{0,1\}^\alpha \to \{0,1\}^{d \cdot k + \ell}$ be a keyed hash function. Further consider the mapping function $MAP_{\ell,k,d} : \{0,1\}^{d \cdot k} \to \{0,1\}^\ell \times [0, 2^d - 1]^k$ which maps a $d \cdot k + \ell$-bit string to a set of k indexes $((I, 0, J_0), \cdots, (I, k-1, J_{k-1}))$ where $I \in [0, 2^\ell - 1]$ and each $J_i \in [0, 2^d - 1]$. Note that the same I is used for all tuples (I, i, J_i). We define the success probability of any adversary A against TSR of H. Let $G = MAP_{\ell,k,d} \circ H$. The definition uses an oracle $O(\cdot)$ which on input an α-bit message M_i samples a key $K_i \xleftarrow{R} \{0,1\}^n$ and returns K_i and $G(K_i, M_i)$. The adversary may query this oracle with messages of its choosing q times. H is said to be ITSR if for any*

adversary A, the following holds:

$$Succ_{H,q}^{ITSR}(A) = Pr\Big[(K,M) \leftarrow A^{O(\cdot)}(n) \\ s.t.\ G(K,M) \subseteq \bigcup_{j=1}^{q} G(K_j,M_j) \wedge (K,M) \notin \{K_j,M_j\}_{j=1}^{q}\Big] \leq negl(n)$$

D Soundness Analysis of π_G

In π_G, k instances of MPC are run. In the ith instance, the partial verification procedure is used to verify every M-FORS signature in $\mathbf{S}$, but only the i-th block of the hash value being signed. Out of the k blocks (each block is d-bit), the adversary may have learned the secret strings correspond to λ_1 blocks through q queries, and for those λ_1 blocks, the adversary can pass the check for sure. Given a particular $\lambda_1 = i$, the probability of the adversary have learned λ_1 out of the k blocks after q queries is:

$$\Pr[\lambda_1 = i] = \binom{k}{i}(1-(1-2^{-d})^q)^i((1-2^{-d})^q)^{k-i} \tag{1}$$

If using an MPC protocol without pre-processing, then in proving all the remaining $k-\lambda_1$ blocks, the adversary must cheat. For each MPC instance, the verifier opens the views of a subset of the MPC parties, and a cheating prover can be detected with a probability $1-\epsilon$. So the overall soundness error is:

$$\sum_{i=0}^{k} \Pr[\lambda_1 = i] \cdot \epsilon^{k-i} \tag{2}$$

If using an MPC protocol with pre-processing, then the adversary can also cheat in the pre-processing phase. If the adversary cheats in λ_2 (out of M) copies of pre-processing data, and not being detected when checking the pre-processing data (the probability is denoted as $\text{Succ}^{pre}(\lambda_2, k, M)$), then it needs to cheat in $k-\lambda_1-\lambda_2$ MPC instances. The soundness error is:

$$\sum_{i=0}^{k} \Pr[\lambda_1 = i] \left(\sum_{\lambda_2=0}^{k-\lambda_1} \text{Succ}^{pre}(\lambda_2, k, M) \cdot \epsilon^{k-\lambda_1-\lambda_2} \right) \tag{3}$$

As a concrete example, let us consider a case in which we implement π_G using KKW [38]. For KKW, $\text{Succ}^{pre}(\lambda_2, k, M) = \frac{\binom{M-\lambda_2}{M-k}}{\binom{M}{M-k}}, \epsilon = \frac{1}{N}$, then plugging them and Equation (1) into Equation (3), the soundness error is:

$$\sum_{i=0}^{k} \binom{k}{i}(1-(1-2^{-d})^q)^i((1-2^{-d})^q)^{k-i} \left(\sum_{\lambda_2=0}^{k-\lambda_1} \frac{\binom{M-\lambda_2}{M-k}}{\binom{M}{M-k}} \cdot (\frac{1}{N})^{k-\lambda_1-\lambda_2} \right) \tag{4}$$

In the above, d, k, q are the parameters for the M-FORS signature, M is the number of pre-processing data generated, and N is the number of MPC parties.

When $d = 16, k = 70, q = 1024, M = 1120, N = 16$, then the soundness error is $2^{-257.769}$; when $d = 16, k = 35, q = 1024, M = 560, N = 16$, then the soundness error is $2^{-128.987}$.

E Implementation Considerations

Now we present some detailed material of our implementation.

E.1 Overview of `picnic3`

We first briefly review the `picnic3` signing and verification processes that we use, for further details please refer to Section 7 of the `picnic` signature algorithm specification [64]. A knowledgeable reader can skip this subsection.
An outline of the signing protocol is:

1. Prepare seeds and tapes, one tape for each party in each round.
2. Generate the `aux` tape, this is used when verifying to enable the simulation to proceed, even when not all of the party's tapes are available. To do this, call `compute_aux` for each MPC round.
3. Commit to the seeds and `aux` data.
4. For each MPC round do the simulation.
5. Commit to the data used in the simulations.
6. Calculate the challenge hash.
7. Use the challenge hash to obtain the list of rounds to remain unopened. For unopened rounds, we also choose the party whose tape will remain unopened.
8. Assemble the proof.
   For each MPC round:

   If (opened)
   reveal seeds to allow the tapes to be generated by the verifier,
   else
   save data to enable the simulation to be done by the verifier.

Note that for the unopened rounds one of the party's tapes is made unavailable, so the masks used cannot be regenerated.
An outline of the verification protocol is:

1. Generate seeds and tapes for the opened rounds. For unopened rounds generate the party's tapes which are un-concealed.
2. For opened rounds generate the `aux` tape. To do this, call `compute_aux` for each opened MPC round. For unopened rounds the `aux` data is read from the signature.

3. Commit to the seeds and `aux` data.
4. For each opened MPC round do the simulation.
5. Commit to simulation data from the signature.
6. Calculate the challenge hash and compare it with the value from the proof.

If (equal)
    accept the signature,
else
    reject the signature.

E.2 Generating the M-FORS trees

In the implementation we start by generating an M-FORS tree. We then generate a set of indices and these are used to calculate the group credential to be used for signing (a set of partial authpaths, one for each base Merkle tree). As we are using MPC for the proof and verification we derive the hash functions that are used to build the tree (H_1 and H_2) from LowMC. The LowMC encryption is:

$$ciphertext = low\_mc(key, plaintext)$$

This is used as a building block for the hash functions we need, based on [34]:

- $H_1 : \{0,1\}^* \to \{0,1\}^n$ – this is the simplest case as n is the same as the LowMC block size. In the implementation there are three cases we need to consider (with 1, 2 or 3 blocks):

$$\begin{aligned} hash\_1(u) &= low\_mc(u, 0^n), \\ hash\_1a(u, v) &= low\_mc(u, v) \oplus v, \quad \text{and} \\ hash\_1b(u, v, w) &= low\_mc(hash\_1a(u, v), w) \oplus w. \end{aligned}$$

- $H_2 : \{0,1\}^* \to \{0,1\}^{d \cdot k}$ – this is used to generate the data to be signed by the M-FORS tree. To do this we calculate a set of hashes

$$hash\_2i(u) = hash\_1a(u, i) \qquad \text{for } i = 0, \ldots, m$$

  where m is chosen so that we get the k sets of d bits that we require. In the implementation $d = 16$ and to avoid too much bit manipulation each $hash\_2i$ is used to provide $\lfloor n/d \rfloor$ indices.

- $H_3 : \{0,1\}^* \to \{0,1\}^{d \cdot k + (\log_2 q) \cdot h}$ – this is used to generate a membership ticket and its corresponding index. This hash function can also be implemented by using low_mc via H_1 in a similar way as H_2 implementation.

E.3 MPC_F and masked hashes

In the Picnic signature scheme [19] an MPC version of LowMC is used. Masks from the tapes are used internally, but outside the routine only the secret key is masked. To maintain anonymity for the group members, the tree locations and the intermediate results as we move up the tree must be masked as well. To do this the starting point is a masked version of LowMC which we refer to as MPC_F. In MPC_F the plaintext and ciphertext can both be masked with the masks being obtained from the tapes.

MPC_F can then be used to define masked versions of the hashes described above. These functions can be chained together by using the same mask for the output of one routine and the input of the next.

As we have more than one function using the tapes we introduce an extra setup stage into the protocols outlined above. This sets up the tape offsets to be used by the functions and returns the total number of bits needed for each tape.

E.4 Node addresses

Each hash used to calculate a node value as we move up the tree has a tweak applied. This tweak is the address of the location of the hash on the F-SPHINCS+ tree. Parts of this node address are also masked to help maintain anonymity. These are the index for the F-SPHINCS+ tree and, when calculating the nodes for a base Merkle tree, the Merkle tree index. For the top Merkle tree there is no need to conceal the index as the verifier knows which base tree is being used.

As we move up from one M-FORS tree to the next, the new M-FORS tree index is obtained by dividing the current index by q (integer division). We choose q to be a power of 2 so that the division is just a right shift. We apply this shift to the index value and the mask, so there is no need for a new mask. The same applies to the base Merkle trees where the index and mask are both divided by 2.

E.5 Handling the top Merkle tree

The base Merkle trees are all complete binary trees, but this is not usually the case for the top Merkle tree. In this case we need to handle a truncated Merkle tree. Missing leaves could be set to zero and we can then proceed from there, but this can be very wasteful. For example, if $k = 35$ the height would be 6 and there would be 29 empty root nodes and further empty nodes as we move up the tree. We opted for a different approach. Figure 15 shows part of a truncated tree. In the diagram the left node, v, at the bottom has no partner right node and so the parent node is given the value v. The value is 'lifted' up to the next level. At this level there is a left and right node and so the parent value is calculated in the usual way.

$$w = H_1(h \parallel v \parallel \mathit{node\ address})$$

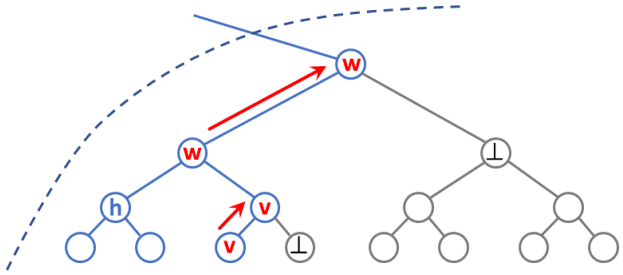


Figure 15: Handling a truncated tree.

In the diagram, this result, w, again has no partner and so this is 'lifted' to the next level. This process carries on until we reach the root of the tree.

E.6 The MPC implementation

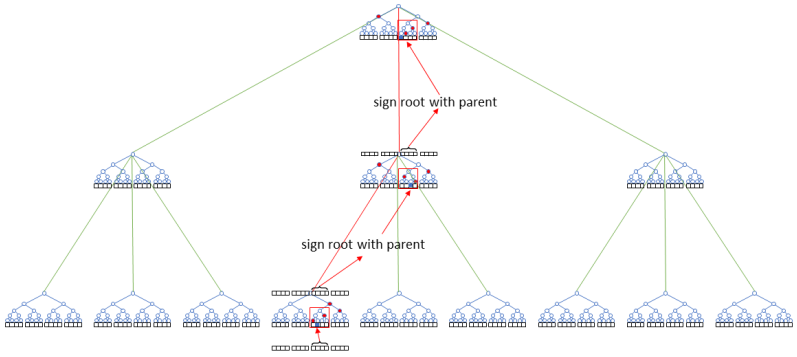


Figure 16: Partial path 2 for a F-SPHINCS+ hyper tree with $k = 4$, $d = 2$, $q = 3$ and $h = 3$.

When signing and verifying we are using partial paths to improve efficiency (see Fig. 16). We need to ensure that we get good coverage for all k partial paths. In the implementation we have $16k$ rounds and so each partial path is used 16 times. When determining which rounds should be unopened we again ensure good coverage, each partial path is used once – which path to leave unopened is chosen at random from the 16 rounds for that partial path. The round numbers to be unopened are obtained from the challenge hash subject to the requirements above.