

Hash-based Direct Anonymous Attestation

Liqun Chen¹[0000-0003-2680-4907], Changyu Dong²[0000-0002-8625-0275], Nada
El Kassem¹[0000-0002-2827-6493], Christopher J.P.
Newton¹[0000-0003-1262-2192], and Yalan Wang¹[0000-0002-6963-7582]

¹ University of Surrey, UK

² Guangzhou University, China

Abstract. Direct Anonymous Attestation (DAA) was designed for the Trusted Platform Module (TPM) and versions using RSA and elliptic curve cryptography have been included in the TPM specifications and in ISO/IEC standards. These standardised DAA schemes have their security based on the factoring or discrete logarithm problems and are therefore insecure against quantum attackers. Research into quantum-resistant DAA has resulted in several lattice-based schemes. Now in this paper, we propose the first post-quantum DAA scheme from symmetric primitives. We make use of a hash-based signature scheme, which is a slight modification of SPHINCS+, as a DAA credential. A DAA signature, proving the possession of such a credential, is a multiparty computation-based non-interactive zero-knowledge proof. The security of our scheme is proved under the Universal Composability (UC) model. While maintaining all the security properties required for a DAA scheme, we try to make the TPM’s workload as low as possible. Our DAA scheme can handle a large group size (up to 2^{60} group members), which meets the requirements of rapidly developing TPM applications.

Keywords: Hash-based signatures · Direct anonymous attestation.

1 Introduction

Direct Anonymous Attestation (DAA) [7] is a group type of anonymous signature scheme, which 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). Unlike group signatures [21], DAA signatures are not traceable, there is no group tracer who can find out which signer created a given signature. However, DAA has two properties that aim to stop a malicious signer from abusing anonymity: rogue key-based revocation and user-controlled linkability. These two properties were designed for using DAA in a remote attestation service that allows a Trusted Platform Module (TPM) to serve as a root of trust for attesting to the host platform that it is embedded in. The first property guarantees that a TPM whose key has been revealed will not be allowed to make any attestation reports. The second property allows a user to include a *basename* in the signature. If the same *basename* is used for two signatures then they can be linked, even though the

anonymity of the signer is maintained. This property allows a verifier to build a revocation list based on a link token which is a deterministic function of the TPM's key and a basename.

When using a TPM in a platform's attestation service, the group signer's role is split into two with a principal signer (the TPM) and an assistant signer (the host). They jointly create attestation reports on the state of the platform. These reports include information on the boot sequence and the software running in the host. These attestation reports convince a remote verifier that the computer platform it is communicating with is running on top of the trusted computing technology and using the correct software and hardware. Using DAA allows such attestations to be made in a privacy-preserving manner. That is, the verifier can check that an attestation report originates from a legitimate TPM, but it does not learn the identity of the particular TPM that generated the DAA signature.

The first RSA-based DAA scheme was standardised as part of the Trusted Computing Group's TPM 1.2 specification [53] published in 2004. The TPM specification was updated in 2014 and this newer TPM 2.0 specification [54] supports elliptic curve based DAA (EC-DAA) and an Intel variant called Enhanced Privacy ID (EPID) [11]. All of these versions of DAA (RSA-DAA, EC-DAA and EPID) have also been standardised by ISO/IEC as standard ISO/IEC 20008-2 [42]. Since the first proposal of DAA, many extensions and works to improve security and efficiency have been proposed [8,9,10,13,15,17,18,23,26,27,28,29,30,38,59]. Researchers have also paid attention to studying the security model and proofs of DAA, e.g. [16,27,56,58].

As reported by the Trusted Computing Group (TCG), which is the industry standards body that develops the TPM specifications, more than a billion devices include TPM technology; in particular almost all enterprise PCs, many servers and embedded systems make use of the TPM as trusted hardware anchors.

Authentication and attestation are important mechanisms used to protect computer systems and with increasing attention and awareness being given to privacy concerns, practical interest in DAA is growing. An anonymous attestation service is particularly important in automotive applications such as vehicle-to-vehicle communication, where the tracking of drivers should be prevented but the authenticity of the communication must also be guaranteed [39,57]. A DAA protocol has also been integrated into the Fast IDentity Online (FIDO) authentication framework [14]. Another DAA-based application is a privacy-enhancing cloud service architecture to protect user's data, using DAA to let users control the extent of data sharing among their service accounts [55].

DAA schemes that are currently supported by the TPM are based on either the factorization problem (for RSA-DAA) or the discrete logarithm problem (for EC-DAA and EPID). Since the factorization and discrete logarithm problems are known to be vulnerable to quantum computer attacks, all standardised DAA schemes are not post-quantum secure, i.e. an adversary with a powerful quantum computer could break the TPM's security and privacy. There is therefore a need to update the standard DAA schemes to be quantum resistant. Many proposed post-quantum cryptographic primitives are built on the top of code-, hash-,

lattice-, isogeny- and multivariate-based problems, and could possibly be used as the basis for the development of post-quantum DAA schemes. Recently, El Bansarkhani et al. [1], El Kassem et al. [34,35,44], Chen et al. [24], and Chen et al. [22] proposed several post-quantum DAA schemes from lattice assumptions. Due to their expensive storage and computational cost, research in lattice-based DAA is still ongoing.

Among all post-quantum approaches, the symmetric key approach is considered as the most conservative approach. The security of symmetric primitives is the most well-understood and easier to evaluate, hence it serves as a safety net if the security of other approaches were endangered by newly discovered threats. Symmetric primitives have been used to build several variants of anonymous signature schemes, such as group signatures [12,33,45,52,60,61], ring signatures [36,45] and EPID [5]. However, due to the use of a single Merkle tree for membership credentials in a group, these group signature and EPID schemes can only handle a small group size, which is not suitable for TPM use.

Our contribution. In this paper, we propose the first DAA scheme from symmetric primitives, which meets all the requirements on DAA, particularly:

- *Signer splitting:* To allow the DAA signer role to be split between a TPM and its host, we introduce a novel approach to splitting an MPC-in-the-Head scheme into two portions. The TPM keeps the key material secure and performs a small part of the work. Most of the work necessary is done by the host. The TPM and the host’s contributions work together seamlessly to form a DAA signature.
- *Support a large group size:* our DAA scheme can support a large group size (up to 2^{60}). To achieve this, we make use of a slightly modified SPHINCS+ signature rather than a Merkle signature as a group membership credential.
- *Security proof:* the security of the proposed DAA scheme is proved under the Universal Composability (UC) model [16].

The remaining part of this paper is arranged as follows: Section 2 describes relevant preliminaries, Section 3 presents the proposed DAA construction, Sections 4 to 7 provide security notions and proofs, and finally Section 8 concludes this paper.

2 Preliminaries

2.1 Hash-based signatures

Digital signature schemes can be built exclusively using cryptographic hash functions. In a hash-based signature scheme, a private key is composed of a series of randomly generated strings, while the corresponding public key is obtained by applying hash functions to the private key. Early hash-based signature schemes, such as the Lamport scheme [47] and the Winternitz scheme [48], were one-time signatures (OTS), meaning that each key pair can only be used to sign a single message. The Merkle signature scheme [48] is the first hash-based few-time signatures (FTS). It generates several OTS key pairs and aggregates their public

keys using a Merkle tree. The root of the tree serves as the overall public key. Every signature uses one OTS private key, and it is comprised of the corresponding OTS and the Merkle tree authentication path for the OTS public key. As a result, the verifier can authenticate the signature using only the Merkle tree root. More recent FTS schemes, such as FORS [3], can be more efficient, as they utilize a large set of secret random strings that can be obtained from a pseudorandom function applied to the private key. Signatures are then generated by selecting elements from the set based on the message to be signed. While each signature discloses some secret strings in the set, the set size is large, and the number of signatures can be controlled to make it infeasible to forge a signature by mixing and matching secret strings from previously generated signatures.

All previously discussed multi-time signature schemes are characterized as stateful, as the signer is required to maintain a state containing information such as the number of signed messages and the keys utilized. In comparison, SPHINCS+ [3] is a stateless hash-based signature scheme. It employs a hypertree, i.e., a tree of trees, to organize OTS and FTS key pairs. Each SPHINCS+ signature constitutes a chain of signatures, with the initial signature Σ_0 being generated from the message, and each subsequent signature Σ_i being a signature of the public key that verifies the preceding signature Σ_{i-1} . By using the root public key, the authenticity of the signature chain can be verified. Although SPHINCS+ also has an upper limit on the number of signatures that can be generated per key pair, it can be set to an extremely large value (e.g. 2^{60}), making it highly unlikely to reach this limit in practical scenarios. SPHINCS+ has been chosen as one of the three digital signature schemes by the National Institute of Standards and Technology (NIST) to become a part of its post-quantum cryptographic standard [49].

2.2 MPC-in-the-head and Picnic-style signatures

This is a paradigm for zero-knowledge proofs introduced by Ishai et. al. [40]. 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 Fiat-Shamir transformation.

There are multiple frameworks for constructing MPC-in-the-head ZK proofs, e.g., IKOS [40], ZKBoo [37], ZKB++ [20], KKW [45], Liger++ [4], Limbo [51], BBQ [50], Banquet [2], BN++ [43], Rainer [31] and AIMer [46]. They follow the same paradigm, but are different in the underlying MPC protocols and have different concrete/asymptotic efficiency. 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:

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

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:

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

MPC-in-the-head has been used to generate signature schemes from a symmetric key setting. As the first scheme is named Picnic [19,20,62], this type of signature is called a Picnic-style signature, in which the secret signing key is k and the public verification key is a pair (c, p) , and the key pair satisfy the equation $c = E(k, p)$ where E is a block cipher, k is a secret key, and p and c are respectively a plaintext and ciphertext block. Signing a message m essentially is to generate a non-interactive MPC-in-the-head proof of knowing the private key:

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

Note that this signature is based on the Fiat-Shamir transformation. The message m is included as a part of the input for the challenge hash in the transformation. Again, to describe our scheme, we do not need to explain the details of the E algorithm, and any secure Picnic-style signature scheme can be used.

2.3 DAA concept

A DAA scheme involves the following players:

- **An issuer** manages the group membership, decides who can be a group member, and issues group membership credentials.
- **Group members** create DAA signatures. Each member is formed by two entities: the TPM serves as a principal signer and the host an assistant signer.
- **Verifiers** verify DAA signatures. A verifier also has two other roles: as a **linker** to check whether two given signatures using the same basename were created by the same signer or not; as a **revocation authority** to decide whether a group member should be removed from the group based on the verifier local revocation.

A DAA scheme consists of the following algorithms/protocols:

- $\text{Init}(n)$: 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 msk is stored privately by the issuer. In all other algorithms and protocols, we will assume mpk along with the security parameter n as an implicit input for all parties. The issuer also initializes its internal states.
- $\text{Join}(msk)$: the joining protocol is an interactive protocol between the issuer and the user (a TPM and its host) who wants to join the group. The issuer has a private input msk and the user does not have input. At the end of the protocol, the issuer outputs a decision: **accept** or **reject**. If **reject**, then stop. If **accept**, the user obtains its signing key $gsk_u = (sk_u, cred_u)$ where sk_u is a secret key, and $cred_u$ is a group membership credential. sk_u is chosen and held by the TPM, and $cred_u$ is generated by the issuer and is given to the host. The issuer also updates its internal states.
- $\text{Sign}(gsk_u, msg, bsn)$: the signing algorithm allows a TPM and its host to produce a signature Σ on a message $msg \in \{0, 1\}^*$ using its signing key gsk_u . If a basename $bsn \neq \perp$, Σ will include a link token.
- $\text{Verify}(msg, bsn, \Sigma, \mathbf{keyRL}, \mathbf{linkRL})$: the verification algorithm allows a verifier to verify whether a signature Σ is a valid signature of msg/bsn and whether the signing key has been listed on a rogue key list **keyRL** or whether a link token in the signature has been listed on a link revocation list **linkRL**.
- $\text{Link}(msg_1, \Sigma_1, msg_2, \Sigma_2, bsn)$: the linking algorithm allows a verifier to check whether given two DAA signatures Σ_1 and Σ_2 with the same bsn value are signed using the same gsk_u or not.
- **Revocation** the revocation algorithm allows a verifier to add a revealed signing key in **keyRL** and to add a link token from a signature generated by a revoked signer in **linkRL**.

A DAA scheme needs to satisfy multiple security requirements, including:

- **Correctness** covers three aspects: (1) an honest user can successfully join the group, despite the existence of malicious users; (2) a signature generated by an honest and not revoked group member should always be valid when being verified; (3) user-controlled linkability, i.e., two valid signatures with the same bsn values and signed under the same gsk_u should be linked to each other.
- **Anonymity** means that a DAA signature does not reveal the identity of its signer, i.e., an 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. Furthermore, given two signatures, w.r.t. two different basenames, the adversary cannot distinguish whether both signatures were created by one honest signer or two different signers.
- **Non-frameability** means that even if the rest of the group, as well as the issuer and the host of an honest TPM, are corrupted, they cannot falsely attribute a signature to the TPM who did not produce it. This property covers three special cases: (1) no adversary can create a signature w.r.t. a basename that links to another signature created by an honest TPM for the

same basename; (2) when the issuer and all TPMs are honest, no adversary can provide a signature on a message msg w.r.t. a basename bsn when no TPM signed this (msg, bsn) pair; (3) When the issuer is honest, an adversary can only sign in the name of corrupt TPMs. More precisely, if n TPMs are corrupt, the adversary can create at most n unlinkable signatures for the same basename.

These requirements will be described in detail under the DAA UC model in Section 6. Note that the host in a secure DAA scheme is trusted to correctly execute the protocol and to maintain anonymity. This trust requirement is necessary, as the host is a contributor to a DAA signature, so a malicious host is able to not provide correct input or to break anonymity by demonstrating the connection between a DAA signature and the corresponding TPM’s public key and credential. We assume that the host represents the user so it is interested in creating valid DAA signatures and maintaining user privacy. However, for non-frameability, there is no requirement for the host to be trusted. Without the TPM, the host can neither receive a DAA credential nor generate a DAA signature. Several types of TPM have been considered in applications: (1) concrete hardware TPM, (2) integrated TPM, (3) firmware TPM, (4) virtual TPM, and (5) software TPM. Although the TPM tamper-resistant property level decreases from the highest case (1) to the lowest one (5), the trust requirements on the host are the same.

3 Construction

3.1 F-SPHINCS+ and M-FORS

To construct a DAA scheme from symmetric primitives, the first design choice is to select group membership credentials. A 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 because it cannot handle a large group size. The last option is also ruled out because of practical considerations: we have to create a ZKP on that another ZKP (i.e. the Picnic-style signature) is valid. Unfortunately, the circuit for verifying a Picnic-style signature is too big, which results in prohibitively high computation costs and large proof size. Therefore, we focused on utilizing a SPHINCS+ style signature as the group credential.

In the above descriptions, 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 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, that means 510 hashes at 128-bit security, and 990 hashes at 256-bit security. The circuit implementing the hash function typically has 10^3 AND gates. 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 Fig. 2. Recall that

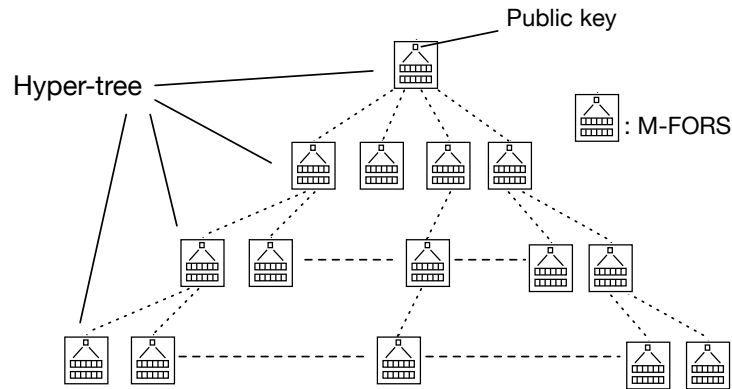


Fig. 1. F-SPHINCS+ signatures.

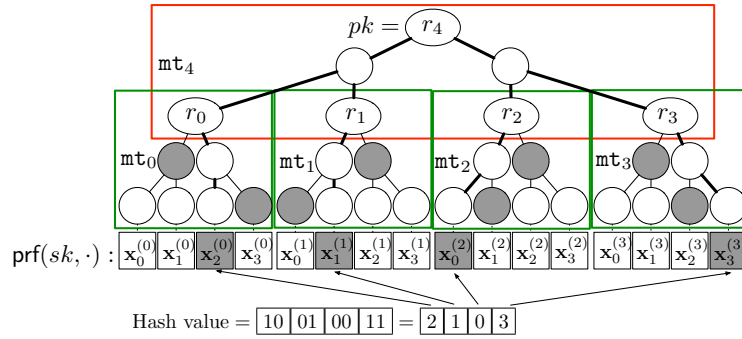


Fig. 2. M-FORS signatures.

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 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).

We now describe M-FORS and F-SPHINCS+. M-FORS consists of the algorithms below. For readability and the page limitation, we abstract away certain low-level details such as how the Merkle trees are built.

- **keyGen**(*seed*, n , d , k , *aux*): it takes as input a random seed *seed*, a security parameter n , two positive integers d and k , and *aux* that is either an empty string or some optional data. If *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 *seed* into k lists $(\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(k-1)})$, where each $\mathbf{x}^{(i)}$ contains 2^d distinct n -bit pseudorandom strings. Then $k + 1$ Merkle trees $\mathbf{T} = (\mathbf{mt}_0, \dots, \mathbf{mt}_k)$ are built. In particular, each of $\mathbf{mt}_0, \dots, \mathbf{mt}_{k-1}$ has 2^d leaf nodes. The j th leaf node in \mathbf{mt}_i is the hash of $\mathbf{x}_j^{(i)}$. The leaf nodes of \mathbf{mt}_k are r_0, \dots, r_{k-1} that are the roots of $(\mathbf{mt}_0, \dots, \mathbf{mt}_{k-1})$. **keyGen** outputs $(pk, sk, param)$, such that the public key $pk = r_k$ where r_k is the root of \mathbf{mt}_k , the private key $sk = \mathbf{seed}$, and the public parameters $mp = (n, d, k, \mathbf{aux})$.
- **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, \dots, p_{k-1}) . Then for the i -th block p_i , $\mathbf{x}^{(i)}$ and \mathbf{mt}_i (obtained by expanding sk) are used to generate **authpath**^(i), which is the authentication path of the p_i -th leaf node in the i -th Merkle tree. Then $(\mathbf{x}_{p_i}^{(i)}, \mathbf{authpath}^{(i)})$ is put into the signature. The signature is a list of k pairs $\sigma = \{(\mathbf{x}_{p_0}^{(0)}, \mathbf{authpath}^{(0)}), \dots, (\mathbf{x}_{p_{k-1}}^{(k-1)}, \mathbf{authpath}^{(k-1)})\}$.
- **recoverPK**(σ, 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, \dots, p'_{k-1}) . Then for $0 \leq i \leq k - 1$, $\sigma_i = (x_i, \mathbf{authpath}^{(i)})$ 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, \dots, r'_{k-1} are used to compute \mathbf{mt}'_k and its root r'_k is returned.
- **verify**(σ, pk, MD, mp): to verify a signature, call **recoverPK**(σ, 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 from left to right, starting from 0. F-SPHINCS+ consists of the following algorithms:

- **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 that 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 **genNode** $((0, 0), sk, fp)$, and set the public key pk to be the M-FORS public key $pk_{0,0}$.
- **genNode**($nodeAdr, sk, fp$): This algorithm generates a node in the hyper-tree given the address $nodeAdr = (a, b)$. With the private key sk used as a *seed*, the algorithm first generates a subseed with a pseudorandom function $seed_{a,b} = \text{prf}(seed, a||b)$, then it calls M-FORS key generation algorithm **M-FORS.keyGen**($seed_{a,b}, n, d, k, a||b$). The output $(pk_{a,b}, sk_{a,b}, mp_{a,b})$ is the content of the node at (a, b) .
- **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, gr is a random string. The algorithm produces $MD||idx \leftarrow H_3(msg||gr)$, where $H_3 : \{0, 1\}^* \rightarrow \{0, 1\}^{d \cdot k + (\log_2 q) \cdot h}$ 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.
- **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 $MD||idx \leftarrow \text{mHash}(msg, gr)$. A leaf node at (h, idx) is then generated by calling **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 **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 $\Sigma = (gr, (\sigma_0, \dots, \sigma_h))$.
- **verify**(msg, Σ, pk, fp): This algorithm verifies every M-FORS signature chained up in Σ . Given $\Sigma = (gr, (\sigma_0, \dots, \sigma_h))$, first compute $MD||idx \leftarrow H_3(msg||gr)$. Then obtain $pk_0 \leftarrow \text{recoverPK}(\sigma_0, MD, mp_0)$, $pk_1 \leftarrow \text{recoverPK}(\sigma_1, pk_0, mp_1)$, repeat until $pk_h \leftarrow \text{recoverPK}(\sigma_h, pk_{h-1}, mp_h)$. If $pk = pk_h$, accept the signature, otherwise reject.

Remark 1 In M-FORS algorithms, we use two tweakable hash functions [3] $H_1 : \{0, 1\}^* \rightarrow \{0, 1\}^n$ and $H_2 : \{0, 1\}^* \rightarrow \{0, 1\}^{d \cdot k}$. 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, these two hash functions can be the same.

Remark 2 The tweakable hash functions follow Construction 7 for tweakable hash functions in [3]. Namely, the hash of an input M is produced by calling a hash function with additional input as $H(\mathsf{P}||\mathsf{ADD}||M)$, where P is a public hash key and ADD acts as 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 are 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 accumulated the roots of all the previous Merkle trees into the public key.
- (a_2, b_2) is the address of an 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 $0 \leq b_2 \leq 2^{a_2} - 1$.

The security analysis of F-SPHINCS+ is given in Section 4.

3.2 The DAA scheme

Overall, the DAA signature scheme is designed in this way: the issuer generates an F-SPHINCS+ key pair as the group master key pair. When a user (including a TPM and its host) joins the group, the TPM generates a secret signing key. The issuer decides 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 signing key). The credential is accessible to the host. When signing a message, the TPM and its host work together to produce an MPC-in-the-head (MPCitH) non-interactive zero-knowledge (NIZK) proof to show it possesses a group credential and the signature is generated on the hash of the message and a random data string under the key authorized by the group credential. We have created a novel approach that allows the TPM and its host each to make a partial signature and a DAA signature is a combination of these two. In particular, the TPM proves its possession of the signing key and the host proves the credential. These two proofs are glued seamlessly in a zero-knowledge manner. Verifying the DAA signature involves checking the NIZK proof so the verifier is convinced of a group membership. Each DAA signature also includes a link token, essentially it is a pseudorandom function output of a basename bsn produced using the signing key as a secret. This link token will be used for user-controlled linkability, key-based revocation and link-based revocation.

We now present the concrete construction of algorithms and protocols.

- **Initialization** $\mathsf{Init}(n)$: Given a security parameter n , the issuer does the following: Choose the hyper-tree node degree q and the tree height h , the values (d, k) for the underlying M-FORS scheme, a pseudorandom function prf , three hash functions $H_1 : \{0, 1\}^* \rightarrow \{0, 1\}^n, H_2 : \{0, 1\}^* \rightarrow \{0, 1\}^{d \cdot k}, H_3 : \{0, 1\}^* \rightarrow \{0, 1\}^{d \cdot k + (\log_2 q) \cdot h}$, and a keyed pseudorandom function $F :$

$\{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}^n$; Run $(sk, rpk, gp) \leftarrow \text{F-SPHINCS+}.\text{keyGen}(n, q, h)$, where (rpk, sk) is the F-SPHINCS+ key pair, $gp = (n, q, h, d, k)$ are the hyper-tree parameters; Publish $mpk = (gp, rpk, H_1, H_2, H_3, F, \text{prf})$ and keep $msk = sk$ private. The issuer provides a non-interactive zero-knowledge (NIZK) proof $\pi_{\mathcal{I}}$ to demonstrate that the key pair is generated correctly, meaning that the secret and public keys are associated with each other. This NIZK proof can be achieved by signing its own public key rpk using F-SPHINCS+.sign, which is similar to the issuer creating a group membership credential in the joining protocol described below. In addition, the issuer initializes a group list **GL**, and each verifier initializes two revocation lists: a key revocation list **keyRL** and a link token revocation list **linkRL**. All these lists are empty when initialized.

- **DAA joining protocol** $\text{Join}(msk, mpk)$: The joining protocol is run between a user (a TPM and its host) and the issuer. Note that this protocol involves the authentication of the TPM by the issuer. The issuer has an authentic copy of the TPM’s endorsement key, which is used to establish a secure and authenticated channel between the TPM and the issuer. In the following protocol description, it is assumed the existence of such a channel, and the reader is recommended to find the detail regarding how to establish such a channel from [25]. The protocol includes the following steps:

1. A unique session ID u is assigned to the user. For simplicity we can think the session ID as a monotonically increasing counter, and each invocation of the joining protocol will increase it by 1. Alternatively, the value u can be computed from the TPM’s endorsement key, which is unique to the TPM.
2. The TPM chooses a random secret key: $sk_u \xleftarrow{R} \{0, 1\}^n$ as its signing key.
3. The host computes the group identifier $gid = H_1(rpk)$ and sends it to its TPM.
4. The TPM then generates and returns its entry token $et_u = F(sk_u, gid)$ together with the NIZK proof π_u :

$$\pi_u : \mathcal{P}\{(gp, gid, et_u); (sk_u) | et_u = F(sk_u, gid)\}$$

5. The host then chooses a random string $cr \xleftarrow{R} \{0, 1\}^n$ and computes a commitment $ct = H_1(et_u || cr)$. The host sends (u, ct) to the issuer to request joining the group.
6. Upon receiving (u, ct) , the issuer checks whether an entry with the same u is in **GL**. If yes, rejects the user. 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 host, who responds by sending (et_u, cr, π_u) back. The issuer verifies $ct = H_1(et_u || cr)$ and the NIZK proof π_u . If both verifications pass, the issuer computes the group credential $(gr_u, \mathbf{S}) \leftarrow \text{F-SPHINCS+}.\text{sign}(et_u || gr_u, msk, gp)$; otherwise the issuer rejects the user. The credential is sent to the TPM through the secure and authenticated channel between the TPM and issuer and then forwarded it to the host. The issuer adds $(u, et_u, gr_u, \mathbf{S})$ to **GL**.
7. The user, if accepted by the issuer, sets its group membership secret key $gsk_u = (sk_u, gr_u, \mathbf{S})$. More specifically, the TPM will record sk_u and the host will record the remaining values.

- **DAA signature generation** $\text{DSig}(gsk_u, msg, bsn)$: To produce a DAA signature on a message msg and a basename bsn , the TPM and its host jointly create a DAA signature using $gsk_u = (sk_u, gr_u, \mathbf{S})$ as follows:

1. The host computes the link identifier $lid = H_1(bsn)$, the signature identifier $sid = H_1(msg||str)$, where $str \xleftarrow{R} \{0,1\}^n$, and the group identifier $gid = H_1(rp_k)$, and sends these three identifier values to the TPM.
2. The TPM computes the group membership entry token $et_u = F(sk_u, gid)$, the signature link token $slt = F(sk_u, lid)$ and the signature signing token $sst = F(sk_u, sid)$ together with the NIZK proof π_{D_T} . The TPM then sends sst and π_{D_T} back to the host.

$$\begin{aligned} \pi_{D_T} : \mathcal{P}\{ & (gp, sid, gid, lid, slt, hk, cet_u); (sk_u, sst, et_u) | \\ & slt = F(sk_u, lid) \wedge sst = F(sk_u, sid) \wedge et_u = F(sk_u, gid) \\ & \wedge hk = H_1(sst) \wedge cet_u = F(sst, et_u) \} \end{aligned}$$

Note that π_{D_T} proves that these three tokens are computed under the same sk_u and also provides a hook (hk, cet_u) , which allows the host to carry on proving the group credential for et_u .

3. The host then computes $mt_u||idx = H_3(et_u||gr_u)$ and $com = H_1(sst||pk_h||\dots||rp_k)$, where pk_h, \dots, rp_k 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 rp_k). Here $H_3(et_u||gr_u)$ is used as F-SPHINCS+.mHash(et_u, gr_u). The host also computes an NIZK proof π_{D_H} :

$$\begin{aligned} \pi_{D_H} : \mathcal{P}\{ & (gp, rp_k, slt, com, hk, cet_u); (et_u, sst, gr_u, \mathbf{S} = \{\sigma_h, \dots, \sigma_0\}) | \\ & hk = H_1(sst) \wedge cet_u = F(sst, et_u) \wedge mt_u||idx = H_3(et_u||gr_u) \\ & \wedge pk_h = \text{recoverPK}(\sigma_h, mt_u, (n, d, k, (h, idx))) \\ & \wedge pk_{h-1} = \text{recoverPK}(\sigma_{h-1}, pk_h, (n, d, k, (h-1, \lfloor \frac{idx}{q} \rfloor))) \wedge \dots \\ & \wedge rp_k = \text{recoverPK}(\sigma_0, pk_1, (n, d, k, (0, 0))) \\ & \wedge com = H_1(sst||pk_h||\dots||rp_k) \} \end{aligned}$$

4. The signature $\Sigma = (str, slt, com, \pi_D)$, where π_D is the combination of π_{D_T} and π_{D_H} , i.e., $\pi_D = (\pi_{D_T}, \pi_{D_H})$. hk and cet_u appearing in both π_{D_T} and π_{D_H} play the role that glues these two MPCitH instances together. From a verifier's point of view, π_D produces the following NIZK proof:

$$\begin{aligned} \pi_D : \mathcal{P}\{ & (gp, rp_k, gid, sid, lid, slt, com); \\ & (sk_u, et_u, sst, gr_u, \mathbf{S} = \{\sigma_h, \dots, \sigma_0\}) | \\ & slt = F(sk_u, lid) \wedge sst = F(sk_u, sid) \wedge et_u = F(sk_u, gid) \\ & \wedge mt_u||idx = H_3(et_u||gr_u) \\ & \wedge pk_h = \text{recoverPK}(\sigma_h, mt_u, (n, d, k, (h, idx))) \\ & \wedge pk_{h-1} = \text{recoverPK}(\sigma_{h-1}, pk_h, (n, d, k, (h-1, \lfloor \frac{idx}{q} \rfloor))) \wedge \dots \\ & \wedge rp_k = \text{recoverPK}(\sigma_0, pk_1, (n, d, k, (0, 0))) \\ & \wedge com = H_1(sst||pk_h||\dots||rp_k) \} \end{aligned}$$

More details of π_D will follow in Section 3.3.

- **DAA signature verification** $DVf(msg, bsn, \Sigma, \mathbf{keyRL}, \mathbf{linkRL})$: Given $\Sigma = (str, slt, com, \pi_D), msg, bsn$, together with two revocation lists \mathbf{keyRL} and \mathbf{linkRL} , the verifier first rejects Σ if $(bsn, slt) \in \mathbf{linkRL}$. Otherwise, the verifier recomputes $lid = H_1(bsn)$, and $\forall sk_u^* \in \mathbf{keyRL}$ computes $slt^* = F(sk_u^*, lid)$. If any $slt^* = slt$, rejects Σ . Otherwise, the verifier verifies π_D . Accept if the verification succeeds; otherwise reject.
- **DAA link algorithm** $\text{Link}(\Sigma, \Sigma')$: Given two valid DAA signatures $\Sigma = (str, slt, com, \pi_D)$ and $\Sigma' = (str', slt', com', \pi_D')$ associated with the same bsn , the verifier checks if $slt = slt'$ holds. If so output linked, otherwise not linked.
- **DAA revocation** There are two cases to revoke the group membership of the user u : (1) Given sk_u , a verifier adds it in \mathbf{keyRL} ³; (2) Given a pair (bsn, slt) associated with a DAA signature signed by the user u to be revoked, a verifier adds this pair in \mathbf{linkRL} .

3.3 The proof π_D

The most important part in the DAA signature $\Sigma = (str, slt, com, \pi_D = (\pi_{D_T}, \pi_{D_H}))$ is the proof π_D . In this section we dissect it to show the design rationale and explain two changes 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 π_D needs to prove is that the signer knows a group signing key $gsk_u = (sk_u, gr_u, \mathbf{S})$ and it was used to sign msg . Besides that, π_D also needs to prove that gsk_u is authorized by the issuer. To do that, in π_D the following is done:

1. It proves that the same signing key sk_u is used to generate three values et_u , slt and sst , where et_u is bound with the group root public key rp_k (as it is computed from $gid = H_1(rp_k)$), slt is bound with the base name bsn (as it is computed from $lid = H_1(bsn)$), and sst is bound with the message msg and random string str (as it is computed from $sid = H_1(msg||str)$). slt is revealed in Σ , and et_u and sst are hidden.
2. It proves that two revealed values slt and com are produced using the same sk_u . It binds com to sst (by using sst in computing com). The commitment com also binds Σ to all public keys used to blindly verify the signatures in \mathbf{S} .
3. It proves that mt_u , which is computed from et_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 rp_k which is published by the group issuer. All public keys recovered in this process match those committed in the commitment com .

The first challenge for implementing π_D with MPCitH comes from splitting the signer role into two parts, the principal signer TPM and the assistant signer

³ It is an open problem for creating a validation check on \mathbf{keyRL} that doesn't take $O(N)$ time, where N is the size of the list.

host, where the TPM holds sk_u and the host holds \mathbf{S} . A straightforward choice is to let the TPM and host be involved in the same MPCitH instance. This will result in a large communication cost between these two entities. Our solution is to split π_D into two MPCinH instances, π_{D_T} and π_{D_H} , each is performed by one entity. The difficulty now is how to glue these two instances together seamlessly in a zero-knowledge manner. We let (sst, et_u) serve as a hidden hook and $hk = H_1(sst)$ and $cet_u = F(sst, et_u)$ as a commitment of sst and et_u . Both π_{D_T} and π_{D_H} include the same MPCitH proofs of hk and cet_u . The collision-resistance property of the functions F and H_1 guarantees that the same pair of (sst, et_u) are in π_{D_T} and π_{D_H} . The preimage resistance property of these two functions guarantees that neither et_u nor sst is revealed. The MPC instance of π_{D_T} is shown in MPCitH 1.

MPCitH 1: π_{D_T} – MPC instance for the TPM’s part of π_D

Public: $gp = (n, q, h, d, k)$, sid , gid , lid , slt

Private: $\llbracket sk_u \rrbracket$

Output: slt' , hk , cet_u

Check: $slt' = slt \wedge hk' = hk \wedge cet'_u = cet_u$

- 1 $slt' = \text{MPC_F}(\llbracket sk_u \rrbracket, lid)$;
 - 2 $\llbracket sst \rrbracket = \text{MPC_F}(\llbracket sk_u \rrbracket, sid)$;
 - 3 $\llbracket et_u \rrbracket = \text{MPC_F}(\llbracket sk_u \rrbracket, gid)$;
 - 4 $hk' = \text{MPC_H1}(\llbracket sst \rrbracket)$;
 - 5 $cet'_u = \text{MPC_F}(\llbracket sst \rrbracket, \llbracket et_u \rrbracket)$;
-

Let us first introduce the notation used in such an MPCitH algorithm: $\llbracket x \rrbracket$ means that the value x is secret-shared when using an MPC algorithm, meaning that it is known by the prover but not the verifier. MPC_X means the MPC subroutine implementing the function X (e.g. MPC_F , MPC_H1 , MPC_H2 and MPC_H3 implement F , H_1 , H_2 and H_3). This notation will be used throughout the paper. Based on [41], in an implementation MPC_F can be used as a building block for the hash functions that we need.

In MPCitH 1, the TPM performs the MPC_F algorithm four times and the MPC_H1 algorithm once when computing the signature link token $slt = \text{MPC_F}(\llbracket sk_u \rrbracket, lid)$, the signature signing token $\llbracket sst \rrbracket = \text{MPC_F}(\llbracket sk_u \rrbracket, sid)$, the entry token $\llbracket et_u \rrbracket = \text{MPC_F}(\llbracket sk_u \rrbracket, gid)$, the hash value $hk = \text{MPC_H1}(\llbracket sst \rrbracket)$, and the connection entry token $cet_u = \text{MPC_F}(\llbracket sst \rrbracket, \llbracket et_u \rrbracket)$. These five operations are performed in the same MPCinH knowledge-proof routine, where sk_u , sst and et_u are kept secret. The TPM outputs the proof along with slt' , hk , and cet_u . The proof demonstrates that the same sk_u value was used in steps 1) - 3), and steps 4) and 5) are used to pass sst and et_u to the host, which allows the latter to carry on the MPCinH knowledge-proof π_{D_H} for the DAA credential associated with et_u . In an implementation this reduces to 5 calls to MPC_F .

The second challenge for implementing π_D with MPCitH comes from the cost of $h + 1$ M-FORS signature verifications required by the proof in π_{D_H} . 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 to check 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 (with an extra factor of 2 if implementing with MPC_F). The $h+1$ factor means that if implemented naively, the MPC used in π_{Dh} 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 a million-gates. 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 π_{Dh} will result in a very large signature size and a high computational cost.

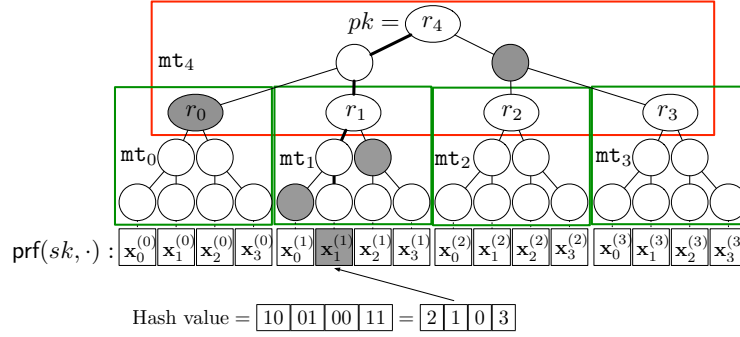


Fig. 3. M-FORS Patial Verification.

Our more efficient strategy for implementing π_{Dh} is: in MPCitH, rather than repeating t times a 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:

- **partial-sig**(σ, MD, i, mp): to extract a partial signature of the i -th block of MD from $\sigma = \{(x_0, \mathbf{authpath}^{(0)}), \dots, (x_{k-1}, \mathbf{authpath}^{(k-1)})\}$. The Merkle tree 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 mt_k .
- **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 tree root r_i from $(x, \mathbf{authpath}, p_i)$, then compute the Merkle tree root pk from $(r_i, \mathbf{authpath}', i)$. Output pk .

With `partial-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 π_{D_H} is shown in MPCitH 2. The first 2 steps of this algorithm are the same as steps 4) and 5) in MPCitH 1. This duplication can glue the TPM part π_{D_T} and the host part π_{D_H} together.

MPCitH 2: π_{D_H} – MPC instance for the v -th block in the host’s part of π_D

Public: $gp = (n, q, h, d, k), rp_k, com, v, hk, cet_u$
Private: $\llbracket sst \rrbracket, \llbracket et_u \rrbracket, \llbracket gr_u \rrbracket, \llbracket \partial_{\sigma_h, v} \rrbracket, \dots, \llbracket \partial_{\sigma_0, v} \rrbracket$
Output: pk_0, hk', cet'_u, com'
Check: $pk_0 = rp_k \wedge hk' = hk \wedge cet'_u = cet_u \wedge com' = com$

- 1 $hk' = \text{MPC_H1}(\llbracket sst \rrbracket);$
- 2 $cet'_u = \text{MPC_F}(\llbracket sst \rrbracket, \llbracket et_u \rrbracket);$
- 3 $\llbracket mt_u \rrbracket \parallel \llbracket idx \rrbracket = \text{MPC_H3}(\llbracket et_u \rrbracket \parallel \llbracket gr_u \rrbracket);$
- 4 $\llbracket M \rrbracket = \llbracket mt_u \rrbracket;$
- 5 $\llbracket COM \rrbracket = \llbracket sst \rrbracket;$
- 6 **for** $l = h; l \geq 0; l --$ **do**
- 7 parse $\llbracket M \rrbracket$ into k blocks $\llbracket p_0 \rrbracket, \dots, \llbracket p_{k-1} \rrbracket$, each block is d -bit;
- 8 $\llbracket M \rrbracket = \text{MPC_pRec}(\llbracket \partial_{\sigma_l, v} \rrbracket, \llbracket p_v \rrbracket, \llbracket idx \rrbracket, gp, l, v);$
- 9 $\llbracket COM \rrbracket = \text{MPC_H1}(\llbracket COM \rrbracket \parallel \llbracket M \rrbracket);$
- 10 $\llbracket idx \rrbracket = \llbracket \lfloor idx/q \rfloor \rrbracket;$
- 11 **end**
- 12 $com' = \llbracket COM \rrbracket;$
- 13 $pk_0 = \text{Reveal}(\llbracket M \rrbracket);$

The host uses partial signatures in the MPC. Recall that in the group signing key gsk_u , a list $\mathbf{S} = \{\sigma_h, \dots, \sigma_0\}$ 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. $\{\partial_{\sigma_h, v}, \dots, \partial_{\sigma_0, v}\}$. In Line 8, an MPC subroutine `MPC_pRec` that implements `partial-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 $\llbracket M \rrbracket$ and also appended to $\llbracket COM \rrbracket$). After the last iteration, $\llbracket COM \rrbracket$ is hashed and $\llbracket M \rrbracket$ is revealed. The results will be checked by the verifier to see whether they match com and rp_k . 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 get away 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 prover has to cheat, then it has to cheat in more than 1 blocks with a high probability.

If the prover has to cheat in n out of k blocks, then using partial verification with t' , such that $t' \cdot n/k \geq t$, ensures that the prover has to cheat in more than t runs, and hence with a negligible success probability. As we analyzed, an 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.

The soundness analysis of π_D is given in Section 5.

4 Security Analysis of 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 few-time signature by limiting the adversary's call to the sign oracle to q_s times where q_s 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_s -time signature scheme, Fig. 4 shows the q_s -EU-CMA game.

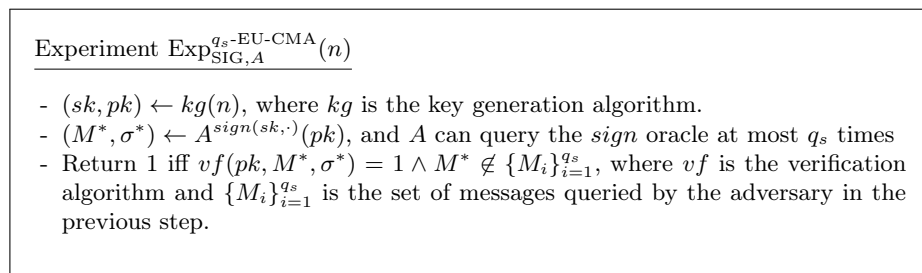


Fig. 4. q_s -EU-CMA game.

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

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

Theorem 1. Following the definitions of SM-TCR (single function, multi-target collision resistance), SM-DSPR (single function, multi-target decisional second-preimage resistance), TSR (target subset resilience), and ITSR (interleaved target subset resilience) given in [3], 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_3 is ITSR secure with at most q^h queries;
- prf is a secure pseudorandom function.

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

- 1 Let $MD||idx = H_3(M||gr)$ for some gr . In the forged signature, all secret strings corresponding to $MD = p_0||\dots||p_{k-1}$, i.e. $\{\mathbf{x}_{p_i}^{(i)}\}_{i=0}^{k-1}$, are the same as generated from 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 find the pre-images.
- 2 Let $MD||idx = H_3(M||gr)$ for some gr . In the forged signature, some secret strings corresponding to $MD = p_0||\dots||p_{k-1}$, i.e. $\{\mathbf{x}_{p_i}^{(i)}\}_{i=0}^{k-1}$, are NOT the same as generated from 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 i th signature are computed with the second-preimages. They end up having the same roots as the trees computed by the 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 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 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:

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

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_3 . This

modification eliminates from the above list Case 1.1. The winning condition in Figure 4 is changed to:

- Return 1 iff $ITSR(H_3, M^*) = 0 \wedge vf(pk, M^*, \sigma^*) = 1 \wedge M^* \notin \{M_i\}_{i=1}^{q^h}$.

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 $MD^* || idx^* = H_3(M^* || gr^*)$.
- Parse $MD^* = p_0^* || \dots || p_{k-1}^*$ where each $p_j^* \in [0, 2^d - 1]$. From the above we obtain a set $C^* = ((idx^*, 0, p_0^*), \dots, (idx^*, k-1, p_{k-1}^*))$.
- For each message queried in the query phase M_i ($1 \leq i \leq q^h$), and gr_i the random string, compute $MD_i || idx_i = H_3(M_i || gr_i)$ and obtain $C_i = ((idx_i, 0, p_{i,0}), \dots, (idx_i, k-1, p_{i,k-1}))$.
- Return 1 iff $C^* \subseteq \bigcup_{i=1}^{q^h} C_i$.

We can see that $ITSR(H_3, M^*) = 0$ iff the adversary can break the ITSR security of H_3 . Hence, we have:

$$|\text{Succ}^{Game1}(A(n)) - \text{Succ}^{Game2}(A(n))| \leq \text{Succ}_{H_3, q^h}^{ITSR}(A) \leq \text{negl}(n)$$

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:

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

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 $TSR(H_2, M^*) = 0 \wedge ITSR(H_3, M^*) = 0 \wedge vf(pk, M^*, \sigma^*) = 1 \wedge M^* \notin \{M_i\}_{i=1}^{q^h}$.

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 node, parse this public key into k blocks, each of d -bit $pk_i = p_{i,0} || \dots || p_{i,k-1}$, and generate a set $C_i = \{(j, p_{i,j})\}_{j=0}^{k-1}$.
- Compute $pk^* = H_2(\text{aux} || k || 0 || 0 || L^* || R^*)$, parse pk^* into $p_0^* || \dots || p_{k-1}^*$, 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:

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

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 [3]:

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

So overall, the advantage of the adversary is negligible.

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 string 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 $2^{-210.39}$.

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 string of a chosen message M being leaked through 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 $2^{-208.95}$.

5 Soundness Analysis of π_D

In π_D , k instances of MPC are run. In the i th 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, the adversary may have learned the secret strings correspond to λ_1 blocks through queries, and has to cheat in all the remaining $k - \lambda_1$ blocks. For each MPC instance, the verifier opens the views of a subset of the MPC parties and a cheat prover can be detected with a probability $1 - \epsilon$. Therefore, if using an MPC protocol without pre-processing, then the soundness error is;

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

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:

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

As a concrete example, let us consider a case in which we implement π_D using KKW [45]. Then we have:

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

$$\text{Succ}^{pre}(\lambda_2, k, M) = \frac{\binom{M-\lambda_2}{M-k}}{\binom{M}{M-k}}, \quad \epsilon = \frac{1}{N}$$

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

6 UC Security Model for DAA

Security in the Universal Composability (UC) framework follows the simulation-based paradigm, where a protocol is secure when it is as secure as an ideal functionality that performs the desired tasks in a way that is secure by design. In this framework, an environment \mathcal{E} passes inputs and outputs to the protocol parties. The network is controlled by an adversary \mathcal{A} that may communicate freely with \mathcal{E} . The framework includes an ideal world and a real world. In the ideal world, the parties forward their inputs to the ideal functionality \mathcal{F} , which then (internally) performs the defined task and creates outputs that are forwarded to \mathcal{E} by the parties. A real-world protocol Π is said to securely realize a functionality \mathcal{F} , if the real world is indistinguishable from the ideal world, meaning that for every adversary performing an attack in the real world, there is an ideal world adversary (often called simulator) \mathcal{S} that performs the same attack in the ideal world. More precisely, a protocol Π is secure if for every adversary \mathcal{A} , there exists a simulator \mathcal{S} such that no environment \mathcal{E} can distinguish executing the real world with Π and \mathcal{A} , and executing the ideal world with \mathcal{F} and \mathcal{S} . Another key point of UC, towards reducing the computational complexity of the specified protocol, is the composition theorem: It guarantees composition with arbitrary sets of parties and executed computational tasks. This ensures that UC-security proofs, for any subroutine of \mathcal{F} , are also transferred to the security model of the entire protocol Π .

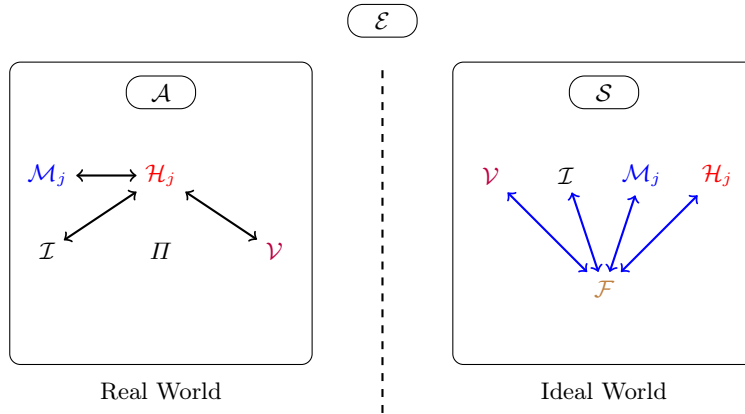


Fig. 5. UC security model for DAA

Now we employ the UC model for the security of our DAA protocol Π . Figure 5 depicts the network topology of the real and ideal worlds. The endmost goal is to prove the completeness and soundness of the DAA protocol by proving that an adversary cannot gain any significant advantage when monitoring the operations and interacting tasks that take place in the real world; i.e., be indistinguishable from the case where all the DAA internal phases are executed in the ideal world. Security of our DAA protocol Π is captured by the fact that every attack \mathcal{A} mounted in the real world, \mathcal{S} carries out in the ideal world. Protocol security is implied since such attacks cannot be mounted in the ideal world. We have then that the output \mathcal{E} retrieved from the execution of Π in the ideal world with \mathcal{S} and from the execution of Π with the real-world entities and \mathcal{A} are indistinguishably distributed. This ensures that a real-world DAA protocol Π *securely realizes* all internal cryptographic tasks (e.g., JOIN, SIGN, VERIFY, and LINK) if for any real-world adversary \mathcal{A} that interacts with the DAA players, running Π , there exists an ideal world simulator \mathcal{S} that interacts with the ideal functionality \mathcal{F} , and the notional entities executing DAA protocol so that no probabilistic polynomial time environment \mathcal{E} can distinguish whether it is interacting with the real world adversary \mathcal{A} or the ideal world adversary \mathcal{S} .

We follow the UC security model for DAA given by Camenisch et al. in [16], where the ideal functionality \mathcal{F} assumes static corruptions, i.e., the adversary decides upfront which parties are corrupt and makes this information known to the functionality. The UC framework allows us to focus the analysis on a single protocol instance with a globally unique session identifier sid . \mathcal{F} uses session identifiers of the form $sid = (\mathcal{I}, sid')$ for some issuer \mathcal{I} and a unique string sid' .

The ideal functionality \mathcal{F} is further parametrized by a leakage function $l : \{0, 1\}^* \rightarrow \{0, 1\}^*$, that models the information leakage occurred in the communication between a host \mathcal{H}_j and its TPM \mathcal{M}_j . We define \mathcal{F} by using two “macros” to determine if a TPM’s signing key sk_u is consistent with the internal

functionality records or not. This is checked at several places in the functionality and also depends on whether the sk_u belongs to an honest or corrupt TPM. The first macro `CheckTtdHonest` is used when the functionality stores a new TPM key sk_u that belongs to an honest TPM, and checks that none of the existing valid signatures is identified as belonging to this TPM key. The second macro `CheckTtdCorrupt` is used when storing a new sk_u that belongs to a corrupt TPM, and checks that the new sk_u does not break the identifiability of signatures, i.e., it checks that there is no other known TPM key sk'_u , unequal to sk_u , such that both keys are identified as the owner of a signature. Both functions output a bit b where $b = 1$ indicates that the new sk_u is consistent with the stored information, whereas $b = 0$ signals an invalid key. We also define the JOIN and SIGN sub-sessions by $jsid$ and $ssid$. In addition \mathcal{F} maintains a group member list `ML`, a key record list `DomainKeys`, a signature record list `Signed`, and a verification result list `VerResults`.

We adopt two sub-functionalities introduced in [16] and they are available to all parties. The first one is a certificate authority functionality \mathcal{F}_{ca} that allows the issuer to register their public key. The second is the common reference string functionality \mathcal{F}_{crs} , which is used to provide all entities with the system parameters comprising the random seed to generate the commitments and the issuer's public key. Note that for the communication between the TPM and issuer (via the host) in the join protocol we adopt the key binding protocol introduced in [25] that provides a secure and authenticated channel between the TPM and the issuer even in the presence of a corrupt host, therefore no need for the semi-authenticated channel \mathcal{F}_{auth}^* in our model. We now define the algorithms that will be used inside the ideal functionality as follows:

- `ukgen(n)`: A probabilistic algorithm that takes a security parameter n as input and generates a key sk_u for a honest TPM.
- `sign(sk_u, msg, bsn)`: A probabilistic algorithm used by a honest TPM; input is a key sk_u , a message msg and a basename bsn , and output is a signature Σ .
- `verify(Σ, msg, bsn)`: A deterministic algorithm that is used in the VERIFY interface. On input of a signature Σ , a message msg and a basename bsn , it outputs $f = 1$ if the signature is valid, and $f = 0$ otherwise.
- `link($\Sigma_1, msg_1, \Sigma_2, msg_2, bsn$)`: A deterministic algorithm that is used in the LINK interface. Given two signatures with the same bsn , it outputs 1 if both Σ_1 and Σ_2 were generated by the same TPM, and outputs 0 otherwise.
- `identify(sk_u, Σ, msg, bsn)`: A deterministic algorithm that is used to ensure consistency with the ideal functionality \mathcal{F} 's internal records. It outputs 1 if a key sk_u was used to produce a signature Σ , and outputs 0 otherwise.

We explain the interfaces of the ideal functionality \mathcal{F} in the UC framework:

Setup

1. **Issuer Setup.** On input `(SETUP, sid)` from issuer \mathcal{I} ,
 - Verify that $sid = (\mathcal{I}, sid')$ and output `(SETUP, sid)` to \mathcal{S} .
2. **Set Algorithms.** On input `(ALG, sid , ukgen, sign, verify, link, identify)` from \mathcal{S} ,

- Check that verify, link and identify are deterministic (i).
- Store $(sid, ukgen, sign, verify, link, identify)$ and output $(SETUPDONE, sid)$ to \mathcal{I} .

Join

3. **Join Request.** On input $(JOIN, sid, jsid, \mathcal{M}_j)$ from host \mathcal{H}_j ,
 - Create a join session record $\langle jsid, \mathcal{M}_j, \mathcal{H}_j, status \rangle$ with $status \leftarrow request$.
 - Output $(JOINSTART, sid, jsid, \mathcal{M}_j, \mathcal{H}_j)$ to \mathcal{S} .
4. **Join Request Delivery.** On input $(JOINSTART, sid, jsid)$ from \mathcal{S} ,
 - Update the session record $\langle jsid, \mathcal{M}_j, \mathcal{H}_j, status \rangle$ to $status \leftarrow delivered$.
 - Output $(JOINPROCEED, sid, jsid, \mathcal{M}_j)$ to \mathcal{I} .
5. **Join Proceed.** On input $(JOINPROCEED, sid, jsid)$ from \mathcal{I} ,
 - Update the session record $\langle jsid, \mathcal{M}_j, \mathcal{H}_j, status \rangle$ to $status \leftarrow complete$.
 - Output $(JOINCOMPLETE, sid, jsid)$ to \mathcal{S} .
6. **Platform Key Generation.** On input $(JOINCOMPLETE, sid, jsid, sk_j)$ from \mathcal{S} ,
 - Look up record $\langle jsid, \mathcal{M}_j, \mathcal{H}_j, status \rangle$ with $status = complete$.
 - Abort if \mathcal{I} or \mathcal{M}_j is honest and a record $\langle \mathcal{M}_j, *, * \rangle \in \text{ML}$ already exists (ii).
 - If \mathcal{M}_j and \mathcal{H}_j are honest, set $sk_j \leftarrow \perp$.
 - Else, verify that the provided sk_j is eligible by checking
 - $\text{CheckTtdHonest}(sk_j) = 1$ if \mathcal{H}_j is corrupt (iii) and \mathcal{M}_j is honest, or
 - $\text{CheckTtdCorrupt}(sk_j) = 1$ if \mathcal{M}_j is corrupt (iv).
 - Insert $\langle \mathcal{M}_j, \mathcal{H}_j, sk_j \rangle$ into ML and output $(JOINED, sid, jsid)$ to \mathcal{H}_j .

Sign

7. **Sign Request.** On input $(SIGN, sid, ssid, \mathcal{M}_j, msg, bsn)$ from host \mathcal{H}_j ,
 - If \mathcal{I} is honest and no entry $\langle \mathcal{M}_j, \mathcal{H}_j, * \rangle$ exists in ML, abort.
 - Create a sign session record $\langle ssid, \mathcal{M}_j, \mathcal{H}_j, msg, bsn, status \rangle$ with $status \leftarrow request$.
 - Output $(SIGNSTART, sid, ssid, l(msg, bsn), \mathcal{M}_j, \mathcal{H}_j)$ to \mathcal{S} .
8. **Sign Request Delivery.** On input $(SIGNSTART, sid, ssid)$ from \mathcal{S} ,
 - Update the session record $\langle ssid, \mathcal{M}_j, \mathcal{H}_j, msg, bsn, status \rangle$ to $status \leftarrow delivered$.
 - Output $(SIGNPROCEED, sid, ssid, msg, bsn)$ to \mathcal{M}_j .
9. **Sign Proceed.** On input $(SIGNPROCEED, sid, ssid)$ from \mathcal{M}_j ,
 - Look up record $\langle ssid, \mathcal{M}_j, \mathcal{H}_j, msg, bsn, status \rangle$ with $status = delivered$.
 - Output $(SIGNCOMPLETE, sid, ssid)$ to \mathcal{S} .
10. **Signature Generation.** On input $(SIGNCOMPLETE, sid, ssid, \Sigma)$ from \mathcal{S} ,
 - If \mathcal{M}_j and \mathcal{H}_j are honest, ignore the adversary's signature and internally generate the signature for a fresh or established sk_j :
 - If $bsn \neq \perp$, retrieve sk_j from $\langle \mathcal{M}_j, bsn, sk_j \rangle \in \text{DomainKeys}$ for (\mathcal{M}_j, bsn) . If no such sk_j exists or $bsn = \perp$, set $sk_j \leftarrow ukgen()$. Check $\text{CheckTtdHonest}(sk_j) = 1$ (v) and store $\langle \mathcal{M}_j, bsn, sk_j \rangle$ in DomainKeys .

- Compute signature as $\Sigma \leftarrow \text{sign}(sk_j, msg, bsn)$ and check $\text{verify}(\Sigma, msg, bsn) = 1$ (vi).
- Check $\text{identify}(\Sigma, msg, bsn, sk_j) = 1$ (vii) and check that there is no $\mathcal{M}'_j \neq \mathcal{M}_j$ with key sk'_j registered in **ML** or **DomainKeys** such that $\text{identify}(\Sigma, msg, bsn, sk'_j) = 1$ (viii).
- If \mathcal{M}_j is honest, store $\langle \Sigma, msg, bsn, \mathcal{M}_j \rangle$ in **Signed**.
- Output (**SIGNATURE**, $sid, ssid, \Sigma$) to \mathcal{H}_j .

Verify

11. **Verify**. On input (**VERIFY**, $sid, msg, bsn, \Sigma, \text{keyRL}, \text{linkRL}$) from some party \mathcal{V} ,
 - Retrieve all pairs (sk_j, \mathcal{M}_j) from $\langle \mathcal{M}_j, *, sk_j \rangle \in \text{ML}$ and $\langle \mathcal{M}_j, *, sk_j \rangle \in \text{DomainKeys}$ where $\text{identify}(\Sigma, msg, bsn, sk_j) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions holds:
 - More than one key sk_j was found (ix).
 - \mathcal{I} is honest and no pair (sk_j, \mathcal{M}_j) was found (x).
 - There is an honest \mathcal{M}_j but no entry $\langle *, msg, bsn, \mathcal{M}_j \rangle \in \text{Signed}$ exists (xi).
 - There is a $sk'_u \in \text{keyRL}$ where $\text{identify}(\Sigma, msg, bsn, sk'_u) = 1$ and no pair (sk_j, \mathcal{M}_j) for an honest \mathcal{M}_j was found, or there exists $(slt', msg', bsn') \in \text{linkRL}$ such that $\text{identify}(slt', msg', bsn', sk_j) = 1$. (xii).
 - If $f \neq 0$, set $f \leftarrow \text{verify}(\Sigma, msg, bsn)$ (xiii).
 - Add $\langle \Sigma, msg, bsn, \text{keyRL}, \text{linkRL}, f \rangle$ to **VerResults**, output (**VERIFIED**, sid, f) to \mathcal{V} .

Link

12. **Link**. On input (**LINK**, $sid, \Sigma, msg, \Sigma', msg', bsn \neq \perp$) from some party \mathcal{V} ,
 - Output \perp to \mathcal{V} if at least one signature tuple (Σ, msg, bsn) or (Σ', msg', bsn) is not valid (verified via the **verify** interface with **keyRL** = \emptyset and **linkRL** = \emptyset) (xiv).
 - For each sk_i in **ML** and **DomainKeys** compute $b_i \leftarrow \text{identify}(\Sigma, msg, bsn, sk_i)$ and $b'_i \leftarrow \text{identify}(\Sigma', msg', bsn, sk_i)$ and do the following:
 - Set $f \leftarrow 0$ if $b_i \neq b'_i$ for some i (xv).
 - Set $f \leftarrow 1$ if $b_i = b'_i = 1$ for some i (xvi).
 - If f is not defined yet, set $f \leftarrow \text{link}(\Sigma, msg, \Sigma', msg', bsn)$.
 - Output (**LINK**, sid, f) to \mathcal{V} .

We highlight that our model catches all the security requirements discussed in section 2.3 (correctness, anonymity and non-frameability):

- The *correctness* of our scheme is guaranteed in our model. When an honest signer (including both the TPM and Host) successfully creates a signature, honest Verifiers will always accept this signature. This is due to the checks v, vi, vii, and viii performed by \mathcal{F} in the **Sign** interface.

- The *anonymity* in our scheme is also guaranteed by \mathcal{F} due to the random choice of sk_j that will be later used for the construction of DAA signatures as part of the **Sign** interface. In the case of corrupt devices, the Simulator is allowed to provide a signature that will convey the signer’s identity, as the signing key can be extracted from the respective device key pair. This reflects that the anonymity of the DAA signer is guaranteed if both the TPM and the Host are honest.
- The *non-frameability* property guarantees that a signature created by an adversary cannot be linked to a legitimate signature created by the target device, this is due to the check ix in our model. CheckTtdHonest prevents registering an honest sk_j in the Join interface that matches an existing signature so that conflicts can be avoided and signatures can always be traced back to the original signer. This ensures that honest signers are not revoked due to the identify algorithm being deterministic in our model. Consider an adversary aiming to create a signature on a message that has not been signed by an honest device, checks x and xi in the **Verify** interface ensure the scheme *unforgeability* property, which dictates that it is computationally infeasible to maliciously forge signatures.

7 UC Security Proof of the DAA Scheme

7.1 High-level description of our proof

We start with the real-world protocol execution in Game 1. In the next game, we construct one entity C that runs the real-world protocol for all honest parties. Then we split C into two pieces, an ideal functionality \mathcal{F} and a simulator \mathcal{S} that simulates the real-world parties. Initially, we start with an “empty” functionality \mathcal{F} . With each game, we gradually change \mathcal{F} and update \mathcal{S} accordingly, moving from the real world to the ideal world, and culminating in the full ideal functionality \mathcal{F} being realized as part of the ideal world, thus, proving our proposed security model presented in Section 6. The endmost goal of our proof is to prove the indistinguishability between Game 1 and Game 16, i.e., between the complete real world and the fully functional ideal world. This is done by proving that each game is indistinguishable from the previous one. We use the “ \approx ” sign to express games indistinguishably between games.

The ideal functionality \mathcal{F} is introduced in Game 3; at this stage \mathcal{F} only forwards its inputs to the simulator \mathcal{S} who simulates the real world. From Game 4 onward, \mathcal{F} starts executing the setup interface on behalf of the Issuer. Moving on to Game 5, \mathcal{F} handles simple verification and identification checks without performing any detailed checks at this stage; i.e., it only checks if the signer belongs to a revocation list separately. In Games 6-8, \mathcal{F} executes the Join interface while performing checks to keep the consistency of registered keys. It also adds checks that allow only the signers that have successfully been enrolled to create signatures. Game 9 proves the anonymity of our protocol by letting \mathcal{F} handle the sign queries on behalf of honest signers. To do this, \mathcal{F} creates signatures using

freshly generated random keys instead of running the signing algorithm using the signer’s signing key. At the end of this game, we prove that by relying on the ZKP constructions, an external environment will notice no change from previous games where the real-world Sign algorithm was executed. Now moving to Games 10 - 16, we let \mathcal{F} perform all other checks that are explained in Section 6.

7.2 The DAA scheme proof

Due to the limited space, we provide a sketch of the security proof of the proposed DAA protocol, including a sequence of games based on the model of Camenish et al. in [16]. A detailed proof will be given in the full paper. The proof in [16] is constructed under the Discrete Logarithm (DL) and Decisional Diffie-Hellman (DDH) assumptions and the unforgeability of the Camenisch-Lysyanskaya (CL) signatures. Other DAA signatures such as [24,35] are proved based on lattice hard problems, namely Ring-LWE and Ring-SIS, and the unforgeability is supported on the modified Boyen or Dilithium signature scheme [6,32]. In contrast to the previous DAA schemes, our game indistinguishability is based on the perfect simulation of the MPCitH-based NIZK proofs, the soundness, completeness and zero-knowledge properties of the proofs $\pi_{\mathcal{I}}$ and $\pi_{\mathcal{D}}$, the unforgeability of the F-SPHINCS+ signature scheme, and the security properties of the tweakable hash functions, H_1 , H_2 and H_3 , and the pseudorandom function F . The sequence of games is as follows:

Proof. (sketch)

Game 1: (Real-World execution of the protocol): This is the start.

Game 2: (Introducing C): An entity C is introduced; C receives all inputs from the parties and simulates the real-world protocol for them. This is equivalent to Game 1.

Game 3: (Reconstruction of C): We now split C into two parts, \mathcal{F} and \mathcal{S} , where \mathcal{F} behaves as an ideal functionality. \mathcal{F} receives all the inputs and forwards them to \mathcal{S} , who simulates the real-world protocol for honest parties, and sends the outputs to \mathcal{F} . \mathcal{F} then forwards the outputs to the environment \mathcal{E} . This game is simply Game 2 but with different structure, so Game 3 \approx Game 2.

Game 4: (\mathcal{F} handles the setup queries): \mathcal{F} now behaves differently in the setup interface and stores the algorithms for the issuer \mathcal{I} . \mathcal{F} also does checks to ensure that the structure of sid is correct for an honest \mathcal{I} , and aborts if not. In case \mathcal{I} is corrupt, \mathcal{S} extracts the secret key for \mathcal{I} and proceeds in the setup interface on behalf of \mathcal{I} . Clearly \mathcal{E} will notice no change, so Game 4 \approx Game 3.

Game 5: (\mathcal{F} handles the verification and linking queries): \mathcal{F} now performs the verification and linking checks instead of forwarding them to \mathcal{S} . There are no protocol messages and the outputs are exactly as in the real-world protocol. However, the only difference is that the verification algorithm used by \mathcal{F} does not contain a revocation check. \mathcal{F} performs this check separately thus the outcomes are equal, so Game 5 \approx Game 4.

Game 6: (\mathcal{F} handles the join queries) : The join interface of \mathcal{F} is now changed, and \mathcal{F} stores the joined member information in the Member List ML .

If \mathcal{I} is honest, \mathcal{F} stores the secret key sk_u , extracted from \mathcal{S} , for corrupt TPM's. \mathcal{S} always has enough information to simulate the real-world protocol except when the issuer is the only honest party. In this case, \mathcal{S} does not know who initiated the join since the host does not authenticate towards the issuer in the real world, so \mathcal{S} can't make a join query with \mathcal{F} on a corrupt host's behalf. Thus, to deal with this case, \mathcal{F} can safely choose any corrupt host and put it into ML, the identities of hosts are only used to create signatures for platforms with an honest TPM or honest host, so fully corrupted platforms do not matter. In the only case, where the TPM has already been registered in ML, \mathcal{F} may abort the protocol, but \mathcal{I} should have already tested this case before continuing with the query JOINPROCEED, hence \mathcal{F} will not abort. Thus in all cases, \mathcal{F} and \mathcal{S} can interact to simulate the real-world protocol, so Game 6 \approx Game 5.

Game 7: (\mathcal{F} knows bsn and msg to be signed or $l(msg, bsn)$): \mathcal{F} now no longer informs \mathcal{S} about the message and the basename that are being signed. If the whole signer is honest, \mathcal{S} can learn nothing about the message msg and the basename bsn . Instead, \mathcal{S} knows only the leakage $l(msg, bsn)$. To simulate the real world, \mathcal{S} chooses a pair (msg', bsn') such that $l(msg', bsn')=l(msg, bsn)$. Therefore Game 7 \approx Game 6.

Game 8: (\mathcal{F} performs pre-sign checks): If \mathcal{I} is honest, \mathcal{F} only allows the signer that has joined to sign. An honest host will always check whether it has joined with its TPM in the real-world protocol, so no difference for honest hosts. Also, an honest TPM only signs when it has joined with the host before. In the case that an honest \mathcal{M}_i performs a join protocol with a corrupt host \mathcal{H}_j and the honest issuer, the simulator \mathcal{S} will make a join query with \mathcal{F} , to ensure that \mathcal{M}_i and \mathcal{H}_j are in ML. Therefore, Game 8 \approx Game 7.

Game 9: (\mathcal{F} handles the sign queries, i.e., simulating the TPM without knowing its secret): In this game, \mathcal{F} creates anonymous signatures for honest signers by running the algorithms defined in the setup interface. Let us start by defining Game 9. $k.k'$, in this game \mathcal{F} handles the first k' signing inputs of \mathcal{M}_k , and subsequent inputs are then forwarded to \mathcal{S} . For $i < k$, \mathcal{F} handles all the signing queries with \mathcal{M}_i using algorithms. For $i > k$, \mathcal{F} forwards all signing queries with \mathcal{M}_i to \mathcal{S} who creates signatures as before. Now from the definition of Game 9. $k.k'$, we note that Game 9.0.0 = Game 8. For increasing k' , Game 9. $k.k'$ will be at some stage equal to Game 9. $k+1.0$, this is because there can only be a polynomial number of signing queries to be processed. Therefore, for large enough k and k' , \mathcal{F} handles all the signing queries of all TPMs, and Game 9 is indistinguishable from Game 9. $k.k'$. We want to prove now that Game 9. $k.k'+1$ is indistinguishable from Game 9. $k.k'$. Suppose that there exists an environment that can distinguish a signature of an honest party using sk_u from a signature using a different sk'_u , then the environment can break the pseudorandom property of the function F .

The first $j \leq k'$ signing queries on behalf of \mathcal{M}_k are forwarded by \mathcal{F} to \mathcal{S} , which calls the real-world protocol. Now suppose that \mathcal{E} is given tuples $\Sigma = (str, slt, com, \pi_D)$ and it is challenged to decide if $\Sigma = (str, slt, com, \pi_D)$ is calculated from uniform random $r \leftarrow \{0, 1\}^n$ or from a certified TPM secret

key sk_u . In the reduction, we have to be able to simulate the TPM without knowing the secret sk_u . The issuer's zero-knowledge proof $\pi_{\mathcal{I}}$ for the correctness of the master secret and public key pair allows the simulator \mathcal{S} extract the master secret key. Furthermore, the zero-knowledge proof of the group membership credential $\pi_{\mathcal{D}}$ helps \mathcal{S} extract the TPM's secret key sk_u for corrupt TPM and create signatures on behalf of the TPM as in the real world scenario. Let r be a randomly sampled key from $\{0, 1\}^n$ that will be used to generate signatures on behalf of honest TPMs rather than using the real TPM secret key sk_u . Since the issuer's secret key msk can be extracted from $\pi_{\mathcal{I}}$ due to the soundness of the proof $\pi_{\mathcal{I}}$ and getting access to \mathcal{F}_{crs} , then a credential can be created on $et'_u = F(r, gid)$ by running the signing algorithm of F-SPHINCS+, $\text{sign}(et'_u, msk, gp)$. After getting a credential on et'_u , slt and sst are calculated as functions of r , i.e. $slt = F(r, lid)$ and $sst = F(r, sid)$. Then all other parts of the signature follow exactly the same as the real-world protocol (i.e. when using the TPM's sk_u). The commitment com is calculated as our defined sign algorithm and the proof $\pi_{\mathcal{D}}$ can then be perfectly simulated using the random secret r . Due to the zero-knowledge property of the proof $\pi_{\mathcal{D}}$ and the pseudorandom outputs of the function F , we argue that an external environment cannot distinguish between 1) a signature generated using the TPM's (sk_u, et_u). 2) a signature generated by a random (r, et'_u) . Therefore, Game 9 \approx Game 8.

Game 10 (\mathcal{F} performs key consistency checks): When storing a new sk_u , \mathcal{F} checks $\text{CheckTtdHonest}(sk_u) = 1$ or $\text{CheckTtdCorrupt}(sk_u) = 1$. We want to show that these checks will always pass. In fact, valid signatures always satisfy $slt = F(sk_u, lid)$, $et_u = F(sk_u, gid)$, $(gr_u, \mathbf{S}) \leftarrow \text{F-SPHINCS+}.\text{sign}(et_u, msk, gp)$ and $sst = F(sk_u, sid)$. By the soundness property of $\pi_{\mathcal{D}}$, there exists only one secret sk_u satisfying the slt construction, and there exists one sst that matches this signature by the soundness of the $hk = \text{MPC_H1}(\llbracket sst \rrbracket)$. Thus, $\text{CheckTtdCorrupt}(sk_u) = 1$ will always give the correct output. On the other hand, the keys for honest TPMs are chosen uniformly at random from an exponentially large group $\{0, 1\}^n$, due to the large min-entropy of the uniform distribution the probability that sampling a selected sk_u is negligible for large n with probability equal to $1/2^n$, thus with overwhelming probability, there does not exist a signature already using the same sk_u , which implies that $\text{CheckTtdHonest}(sk_u) = 1$ will always give the correct output. Hence, Game 10 \approx Game 9.

Game 11: (\mathcal{F} checks the correctness of the protocol): In this game \mathcal{F} checks that any honestly generated signature $\Sigma = (str, slt, com, \pi_{\mathcal{D}})$ is always valid due to the completeness property of $\pi_{\mathcal{D}}$ and the correctness of the F-SPHINCS+ signature. A valid proof $\pi_{\mathcal{D}}$ on the credential ensures that the credential has the correct structure, follows the correct authentication path, and always leads to the issuer's public key rpk due to the soundness of $\pi_{\mathcal{D}}$ and the correctness of the F-SPHINCS+ signature. Second, \mathcal{F} makes sure $\text{identify}(\Sigma, msg, bsn, sk_u) = 1$, this is also achieved in the real-world protocol due to the soundness of $\pi_{\mathcal{D}}$. \mathcal{F} checks, using its internal records ML and DomainKeys that honest users are not sharing the same secret key sk_u . If there exists a key $sk'_u \neq sk_u$ in DomainKeys

such that $slt = F(sk'_u, lid) = F(sk_u, lid)$, then this breaks the collision resistance property of the function F . Therefore Game 11 \approx Game 10.

Game 12 (\mathcal{F} checks that valid signatures are deterministic): Add Check (ix) to ensure that there are no multiple sk_u values matching to one signature. A signature Σ includes $slt = F(sk_u, lid)$, $com = H_1(F(sk_u, sid)||pk_h|| \dots ||rpk)$ and π_D . Due to the soundness of the function F and the proof π_D , and also due to the collision resistance and second-preimage properties of H_1 , two different keys cannot create the same signature and two different signatures cannot share the same sk_u . Therefore a valid signature should be identified to one sk_u only. Hence, Game 12 \approx Game 11.

Game 13 (\mathcal{F} checks the unforgeability of the credential): To prevent accepting a signature that was not generated by using a group membership credential issued by an honest issuer, \mathcal{F} adds Check (x). A credential is an F-SPHINCS+ signature on $mt_u||idx$, using the tweakable hash functions H_1 , H_2 and H_3 . Following the proof of Theorem 1 in Section 4, the F-SPHINCS+ signature scheme is unforgeable due to the security properties of H_1 , H_2 and H_3 , so this check is always passed and Game 13 \approx Game 12.

Game 14 (\mathcal{F} checks the unforgeability of signatures): Check (xi) is added to \mathcal{F} to prevent an adversary from forging signatures using honest signer's credential key $gsk_u = (sk_u, gr_u, \mathbf{S})$. As discussed before, a DAA signature Σ is proof of the correct construction of slt , com and π_D , which form a NIZK proof of an F-SPHINCS+ signature associated with a single key sk_u . If the signature is verified, due to the unforgeability of F-SPHINCS+, the binding property of the commitment scheme used to generate $com = H_1(F(sk_u, sid)||pk_h|| \dots ||rpk)$, and the soundness of the function F used to compute slt and com , sk_u belonging to an honest TPM must be involved. If the adversary uses a different key sk'_u to create this signature. Due to the soundness of π_D analyzed in Section 5, the proof π_D cannot be simulated with overwhelming probability unless $sk'_u = sk_u$, so Game 14 \approx Game 13.

Game 15 (\mathcal{F} checks the correct revocation): Check (xii) is added to \mathcal{F} to ensure that an honest TPM with sk_u are not being revoked. If there exists a matching revoked key $sk_u^* (\neq sk_u) \in \mathbf{keyRL}$ such that $slt = F(sk_u^*, lid) = F(sk_u, lid)$, then this breaks the collision resistance property of the function F . For the same reason, there does not exist $(slt', msg', bsn') \in \mathbf{linkRL}$ such that $slt' = F(sk'_u, lid') = F(sk_u, lid')$ and $sk'_u \neq sk_u$. Therefore, our protocol ensures the correct revocation. So Game 15 \approx Game 14.

Game 16 (\mathcal{F} checks the linkability): Checks (xv and xvi) of the ideal functionality \mathcal{F} that are related to link queries are now included. The output of \mathcal{F} based on these checks is still consistent with the output which the link algorithm would give: If there is an sk_u that matches two signatures signed under the same bsn , by the soundness of π_D we have that the pseudonyms based on the same sk_u must be equal, resulting in link outputting 1. If there is an sk_u that matches one signature but not the other, by the soundness of π_D we have that the pseudonyms slt that are not generated using sk_u must also differ from

those generated by a different key $sk'_u \neq sk_u$ which results in link outputting 0. Therefore, Game 16 \approx Game 15. This concludes the proof.

8 Conclusion

This paper proposes the first DAA scheme from symmetric primitives and this scheme has some interesting features. We make use of a modified SPHINCS+ signature as a group membership credential and use of a Picnic-style signature to prove the possession of that credential. Our DAA scheme splits the signer role between a TPM and its host and allows the TPM to have a much smaller workload than the host. This scheme can handle a large group size (up to 2^{60}), which is suitable for rapidly increasing trusted computing applications. This research topic is still in its early stage. Improving the performance of this DAA scheme is challenging and it will be possible if either a more efficient stateless hash-based signature scheme than F-SPHINCS+ or an efficient Picnic-style signature scheme is developed.

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 would like to thank Qingju Wang and Scott Fluhrer for helpful discussions. We also thank the anonymous reviewers from PQCrypto for their valuable comments.

References

1. Bansarkhani, R.E., Kaafarani, A.E.: Direct anonymous attestation from lattices. *IACR Cryptol. ePrint Arch.* **2017**, 1022 (2017)
2. Baum, C., de Saint Guilhem, C.D., Kales, D., Orsini, E., Scholl, P., Zaverucha, G.: Banquet: Short and fast signatures from AES. In: *PKC 2021*. pp. 266–297 (2021)
3. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS+ signature framework. In: *ACM CCS*. pp. 2129–2146 (2019)
4. Bhadauria, R., Fang, Z., Hazay, C., Venkatasubramanian, M., Xie, T., Zhang, Y.: Liger++: A new optimized sublinear IOP. In: *ACM CCS*. pp. 2025–2038 (2020)
5. Boneh, D., Eskandarian, S., Fisch, B.: Post-quantum EPID signatures from symmetric primitives. In: *CT-RSA*. pp. 251–271 (2019)
6. Boyen, X.: Lattice mixing and vanishing trapdoors: A framework for fully secure short signatures and more. In: *International Workshop on Public Key Cryptography*. pp. 499–517 (2010)
7. Brickell, E.F., Camenisch, J., Chen, L.: Direct anonymous attestation. In: *ACM CCS*. pp. 132–145 (2004)

8. Brickell, E., Chen, L., Li, J.: A new direct anonymous attestation scheme from bilinear maps. In: International Conference on Trusted Computing. pp. 166–178 (2008)
9. Brickell, E., Chen, L., Li, J.: Simplified security notions of direct anonymous attestation and a concrete scheme from pairings. *International Journal of Information Security* **8**, 315–300 (2009)
10. Brickell, E., Li, J.: A pairing-based DAA scheme further reducing TPM resources. *Trust* **6101**, 181–195 (2010)
11. Brickell, E., Li, J.: Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. *IEEE Trans. Dependable Secur. Comput.* **9**(3), 345–360 (2012)
12. Buser, M., Liu, J.K., Steinfeld, R., Sakzad, A., Sun, S.: DGM: A dynamic and revocable group Merkle signature. In: ESORICS. pp. 194–214 (2019)
13. Camenisch, J., Chen, L., Drijvers, M., Lehmann, A., Novick, D., Urian, R.: One TPM to bind them all: Fixing TPM 2.0 for provably secure anonymous attestation. In: IEEE S&P. pp. 901–920 (2017)
14. Camenisch, J., Drijvers, M., Edgington, A., Lehmann, T.A., Urian, R.: FIDO ECDA algorithm. (2018), <http://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-ecdaa-algorithm-v2.0-id-20180227.html>
15. Camenisch, J., Drijvers, M., Lehmann, A.: Anonymous attestation using the strong Diffie-Hellman assumption revisited. In: TRUST. pp. 1–20 (2016)
16. Camenisch, J., Drijvers, M., Lehmann, A.: Universally composable direct anonymous attestation. In: PKC, Part II. pp. 234–264 (2016)
17. Camenisch, J., Drijvers, M., Lehmann, A.: Anonymous attestation with subverted TPMs. In: CRYPTO. pp. 427–461 (2017)
18. Casey, M., Chen, L., Giannetsos, T., Newton, C., Sasse, R., Whitefield, J.: Direct anonymous attestation in the wild, Presentation at Real World Crypto 2019, <https://rwc.iacr.org/2019/slides/DAA.pdf>
19. Chase, M., Derler, D., Goldfeder, S., Katz, J., Kolesnikov, V., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., Wang, X., Zaverucha, G.: The Picnic signature scheme design document. <https://microsoft.github.io/Picnic/> (2020)
20. Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., Zaverucha, G.: Post-quantum zero-knowledge and signatures from symmetric-key primitives. In: ACM CCS. pp. 1825–1842 (2017)
21. Chaum, D., van Heyst, E.: Group signatures. In: EUROCRYPT. pp. 257–265 (1991)
22. Chen, L., Tu, T., Yu, K., Zhao, M., Wang, Y.: V-LDAA: A new lattice-based direct anonymous attestation scheme for VANETs system. *Security and Communication Networks* **2021**, 1–13 (2021)
23. Chen, L.: A DAA scheme requiring less TPM resources. In: Information Security and Cryptology (Inscrypt 2009). pp. 350–365 (2010)
24. Chen, L., El Kasseem, N., Lehmann, A., Lyubashevsky, V.: A framework for efficient lattice-based DAA. In: Proceedings of the 1st ACM Workshop on Cyber-Security Arms Race. pp. 23–34 (2019)
25. Chen, L., El Kasseem, N., Newton, C.J.: How to bind a TPM’s attestation keys with its endorsement key. *The Computer Journal* p. bxad037 (2023)
26. Chen, L., Li, J.: Flexible and scalable digital signatures in TPM 2.0. In: ACM CCS. pp. 37–48 (2013)
27. Chen, L., Morrissey, P., Smart, N.P.: On proofs of security for DAA schemes. In: International Conference on Provable Security. pp. 156–175 (2008)

28. Chen, L., Page, D., Smart, N.P.: On the design and implementation of an efficient DAA scheme. In: CARDIS. pp. 223–237 (2010)
29. Chen, L., Urian, R.: DAA-A: Direct anonymous attestation with attributes. In: International Conference on Trust and Trustworthy Computing. pp. 228–245 (2015)
30. Dai, Y., Zhang, F., Zhao, C.A.: Fast hashing to G_2 in direct anonymous attestation. Cryptology ePrint Archive (2022/996)
31. Dobraunig, C., Kales, D., Rechberger, C., Schofnegger, M., Zaverucha, G.: Shorter signatures based on tailor-made minimalist symmetric-key crypto. In: ACM CCS. pp. 843–857 (2022)
32. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-Dilithium: A lattice-based digital signature scheme. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 238–268 (2018)
33. El Bansarkhani, R., Misoczki, R.: G-Merkle: A hash-based group signature scheme from standard assumptions. In: PQCrypto. pp. 441–463 (2018)
34. El Kassem, N.: Lattice-based direct anonymous attestation. Ph.D. thesis, University of Surrey (2020)
35. El Kassem, N., Chen, L., El Bansarkhani, R., El Kaafarani, A., Camenisch, J., Hough, P., Martins, P., Sousa, L.: More efficient, provably-secure direct anonymous attestation from lattices. Future Generation Computer Systems **99**, 425–458 (2019)
36. Fujisaki, E., Suzuki, K.: Traceable ring signature. In: PKC. pp. 181–200 (2007)
37. Giacomelli, I., Madsen, J., Orlandi, C.: ZKBoo: Faster zero-knowledge for boolean circuits. In: USENIX Security. pp. 1069–1083 (2016)
38. Greveler, U., Justus, B., Loehr, D.: Direct anonymous attestation: enhancing cloud service user privacy. In: On the Move to Meaningful Internet Systems: OTM 2011: Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2011, Proceedings, Part II. pp. 577–587 (2011)
39. Hicks, C., Garcia, F.D.: A vehicular DAA scheme for unlinkable ecdsa pseudonyms in v2x. In: EuroS&P. pp. 460–473 (2020)
40. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: STOC. pp. 21–30 (2007)
41. ISO/IEC 10118-2:2010: Information technology — Security techniques — Hash-functions — Part 2: Hash-functions using an n -bit block cipher. Standard, International Organization for Standardization (2010)
42. ISO/IEC 20008-2: 2013: Information technology - Security techniques - Anonymous digital signatures - Part 2: Mechanisms using a group public key. Standard, International Organization for Standardization (2013)
43. Kales, D., Zaverucha, G.: Efficient lifting for shorter zero-knowledge proofs and post-quantum signatures. Cryptology ePrint Archive (2022/588)
44. Kassem, N.E., Chen, L., El Bansarkhani, R., El Kaafarani, A., Camenisch, J., Hough, P., Martins, P., Sousa, L.: Lattice-based direct anonymous attestation (LDAA). Cryptology ePrint Archive (2018/401)
45. Katz, J., Kolesnikov, V., Wang, X.: Improved non-interactive zero knowledge with applications to post-quantum signatures. In: ACM CCS. pp. 525–537 (2018)
46. Kim, S., Ha, J., Son, M., Lee, B., Moon, D., Lee, J., Lee, S., Kwon, J., Cho, J., Yoon, H., et al.: AIM: Symmetric primitive for shorter signatures with stronger security. Cryptology ePrint Archive (2022/1387)
47. Lamport, L.: Constructing digital signatures from a one-way function. Tech. Report: SRI International Computer Science Laboratory (1979)
48. Merkle, R.C.: A certified digital signature. In: CRYPTO. pp. 218–238 (1989)

49. NIST: NIST announces first four quantum resistant cryptographic algorithms (2022), <https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms>
50. de Saint Guilhem, C.D., De Meyer, L., Orsini, E., Smart, N.P.: BBQ: Using AES in picnic signatures. In: SAC. pp. 669–692 (2020)
51. de Saint Guilhem, C.D., Orsini, E., Tanguy, T.: Limbo: Efficient zero-knowledge MPCitH-based arguments. In: ACM CCS. pp. 3022–3036 (2021)
52. Shafieinejad, M., Esfahani, N.N.: A scalable post-quantum hash-based group signature. *Designs, Codes and Cryptography* **89**, 1061–1090 (2021)
53. TCG: TPM 1.2 Main Specification. Rev 116, Trusted Computing Group (Mar 2011), <https://trustedcomputinggroup.org/resource/tpm-main-specification/>
54. TCG: Trusted Platform Module 2.0 Library Specification. Rev 01.59, Trusted Computing Group (Nov 2019), <https://trustedcomputinggroup.org/resource/tpm-library-specification/>
55. Wang, H.Z., Huang, L.S.: An improved trusted cloud computing platform model based on DAA and privacy CA scheme. In: 2010 International Conference on Computer Application and System Modeling (ICCSM 2010). pp. 13–33 (2010)
56. Wesemeyer, S., Newton, C.J., Treharne, H., Chen, L., Sasse, R., Whitefield, J.: Formal analysis and implementation of a TPM 2.0-based direct anonymous attestation scheme. In: AsiaCCS. pp. 784–798 (2020)
57. Whitefield, J., Chen, L., Giannetsos, T., Schneider, S., Treharne, H.: Privacy-enhanced capabilities for VANETs using direct anonymous attestation. In: IEEE Vehicular Networking Conference (VNC). pp. 123–130 (2017)
58. Whitefield, J., Chen, L., Sasse, R., Schneider, S., Treharne, H., Wesemeyer, S.: A symbolic analysis of ECC-based direct anonymous attestation. In: EuroS&P. pp. 127–141 (2019)
59. Yang, K., Chen, L., Zhang, Z., Newton, C.J.P., Yang, B., Xi, L.: Direct anonymous attestation with optimal TPM signing efficiency. *IEEE Transactions on Information Forensics and Security* **16**, 2260–2275 (2021). <https://doi.org/10.1109/TIFS.2021.3051801>
60. Yehia, M., AlTawy, R., Gulliver, T.A.: GM^{MT} : A revocable group Merkle multi-tree signature scheme. In: CANS. pp. 136–157 (2021)
61. Yehia, M., AlTawy, R., Gulliver, T.A.: Security analysis of DGM and GM group signature schemes instantiated with XMSS-T. In: Inscrypt. pp. 61–81 (2021)
62. Zaverucha, G.: The Picnic signature algorithm specification. Supporting Documentation in <https://github.com/Microsoft/Picnic> (2020)