BAHS: a Blockchain-Aided Hash-based Signature Scheme

Yalan Wang[0000−0002−6963−7582], Liqun Chen*[0000−0003−2680−4907], Long Meng[0000−0002−5648−5049], and Yangguang Tian[0000−0002−6624−5380]

University of Surrey, Guildford, UK
{liqun.chen}@surrey.ac.uk

Abstract. Hash-based one-time signatures are becoming increasingly important as they are post-quantum safe and have been used in multi-cast communication and other applications. However, managing the state of such signatures can present a significant challenge, as signers are typically responsible for ensuring that the state cannot be reused. Recently, blockchain, as a public platform, is used to design revocation management and status verification systems. While blockchain revocation is attractive, many well-known blockchains make use of ECDSA as their underlying signature scheme, and this is not post-quantum safe. Researchers have been working on replacing ECDSA with post-quantum signature schemes but they are much more costly. In this paper, we introduce a new one-time signature scheme, called Blockchain-Aided Hash-based Signature (BAHS), in which a hash-based commitment scheme acts as the building block, and signers' commitments and opened commitments are publicly accessible via a distributed blockchain. A signature is formed from the commitment/opened commitment and blockchain. Unlike existing blockchain systems, the commitment in BAHS is simpler than that in most existing hash-based one-time signature schemes or other post-quantum signature schemes. We provide a formal security model for the BAHS scheme and give the security proof. Finally, we have implemented our BAHS scheme and the result shows its practicality.

Keywords: Digital signature · Hash function · Blockchain · Cryptographic protocols.

1 Introduction

Digital signatures are a cryptographic primitive for verifying the authenticity of digital data. A one-time hash-based signature, as proposed by Lamport in [21], is a special type of digital signature, in which each signing key can be used only once and one-way functions without trapdoors are applied. One-time hash-based signatures can be used in multi-cast communications, such as wireless sensor networks [26] and smart grids [24]. In these applications, signatures are used to achieve demand response, operation and control. The deployment of hash-based one-time signature schemes faces a significant challenge, i.e., state management. This refers to the process of ensuring that a signature cannot be reused.

The problem of state management was discussed in [15]. A comprehensive assessment of the security impact of reusing a one-time signature private key was provided in [9]. Based on the research on this topic, ISO/IEC 14888-4 2nd Committee Draft [2] provides the following recommendations to implement robust state management mechanisms:

- The state used in hash-based one-time signatures is a piece of information, which should be stored, maintained, and updated for the whole lifespan of the private key.
- One way to reduce the chances of state reuse is to prevent the copy or extraction of the private key from the signing module. Assuming this way can be guaranteed, the issue of state management is simplified to a single signing environment, rather than having to manage multiple signing environments. Consequently, the problem of state management is replaced by a more intricate issue: the state synchronization problem.
- During the signing, the signer will first update the state and then start the signing procedure. If this process was done in a reverse order, there is a risk that the signature is produced but the state remains in its previous value.

In accordance with these recommendations, a signer is responsible for ensuring state management. Nevertheless, the signer may either lack the ability or may not be trustworthy enough to assume full responsibility.

Recently, blockchain is used to design revocation management and status verification systems [3, 13]. In these blockchain-based systems, the blockchain acts as data storage. During signature verification, verifiers must examine the key status to determine if the signer has been revoked or not. Obviously, a risk is that a malicious signer will not revoke their key. As a result, these blockchain-based schemes are still unable to effectively implement state management. Recently, in [23], they took the public ledger to assist the threshold signature scheme and the state management, but the underlying signature scheme is based on classical signature algorithms, which is complicated and not post-quantum secure. Generally, most existing blockchains make use of a traditional signature scheme, ECDSA, or its variants, e.g., [8, 25], as an underlying signature scheme. This type of signature scheme is not post-quantum secure. Recently, NIST has announced to standardize three post-quantum signatures, Dilithium [20], Falcon [7] and SPHINCS+ [11]. They are being considered to replace the traditional RSA- and EC-based signatures. There has been some research on considering the use of post-quantum secure signatures in blockchains (cryptocurrency) [7, 11, 12, 14]. However, based on the result of Holmes' work [17], all the well-known post-quantum signatures are quite expensive to be implemented in blockchains.

Now, the question is whether we can use the state verification capability in a blockchain to create a simpler one-time signature scheme. In this paper, by leveraging the Merkle tree in the blockchain to generate commitments, we enable state management through the public accessibility of keys, commitments/opened commitments. Therefore, we develop a straightforward post-quantum one-time signature scheme. We call this new scheme *blockchain-aided hash-based signature* (BAHS). In the BAHS scheme, there are three types of entities, a set of signers,

a blockchain, and a set of verifiers. To sign a message m using the signing key sk, the signer creates a commitment input $cInput$ ($cInput = H(m,\ sk)||H(sk)$) and sends it to a blockchain. Before accepting this input, the blockchain checks whether the signing key has been used before. If not, $cInput$ will be appended to the blockchain to be time-stamped and the commitment com is formed. At a later time (block), the signer needs to open the commitment by sending the input $oInput$, $oInput = (m, sk, cInput)$ to the blockchain. And the opened commitment on the blockchain is denoted by $\tilde{com}$. Finally, com and $\tilde{com}$ form a signature. During verification, the verifier retrieves the signature from the blockchain and checks whether the signature is valid or not. The aid of the blockchain guarantees the key state management in the scheme.

Our contributions can be summarized as follows:

- We propose a BAHS scheme, which is the first hash-based one-time signature scheme that achieves key state management without entirely relying on the signer.
- Our BAHS scheme only makes use of hash functions and blockchain (Merkle tree) to generate the commitment, which is post-quantum secure and more efficient and simpler than traditional signatures and other post-quantum signatures.
- We introduce a formal definition of the security model for our proposed BAHS scheme and provide concrete security proof. This security analysis indicates that the BAHS scheme holds the properties of correctness and unforgeability.
- We provide a proof of concept implementation of the BAHS scheme. The implementation and evaluation confirm its practicality.

The rest of this paper is structured as follows. We introduce preliminaries in Section 2. We present the syntax for a generic BAHS scheme in Section 3. Based on the generic BAHS definition, we present our BAHS scheme in detail in Section 4. In Section 5, we introduce the security model and provide the security proof. In Section 6, implementation results are given. Finally, in Section 7, we present the conclusion.

2 Preliminaries

2.1 Hash functions

Definition 1. *A secure hash function maps a string of bits of variable length (but usually upper bounded) to a fixed-length string of bits.*

The properties of hash functions are one-wayness, second preimage-resistance and collision-resistance [1], which are described as follows:

- **One-wayness**. Given a hash function H and a hash value $H(m)$, it is computationally infeasible to get the input message m.

- **Second Preimage-resistance**. Given a hash value $H(m)$ of a message m, it is computationally infeasible to find a second input m' which maps to the same output $H(m)$.
- **Collision-resistance**. Given a Hash function H, it is computationally infeasible to find a pair of messages m and m' to make $H(m) = H(m')$.

2.2 Commitment scheme

In a commitment scheme [10, 19], there are two phases, i.e., committing and opening. Here we give a definition of these two phases as follows:

- Committing: To commit a data string b, the prover P chooses r randomly $r \leftarrow \{0,1\}^l$ and computes the commitment $com \leftarrow \mathsf{commit}(r, b)$, where commit is a function:$\{0,1\}^l \times \{0,1\}^l \rightarrow \{0,1\}^l$. Then the prover can send com to the verifier V.
- Opening: The prover can reveal r, b to the verifier V. Then the verifier V computes $com' = \mathsf{commit}(r, b)$ and checks whether $com' = com$ or not.

Hash functions can be used to design commitment scheme [4, 6]. In our scheme, we follow the Merkle tree method to compute signers' commitments.

2.3 Blockchains

A blockchain is a distributed digital blockchain of signed transactions that are grouped into blocks. As shown in Fig. 1, a block header contains a block index number bid_i, a nonce non_i, a hash value of the previous block header hbh_{i-1}, a time-stamp ts_i, and a Merkle tree root r_i of all block data. The block data contains a list of transactions along with their corresponding digital signatures. The generation of block b_i and the process of connecting with the block b_{i+1} are described as follows:

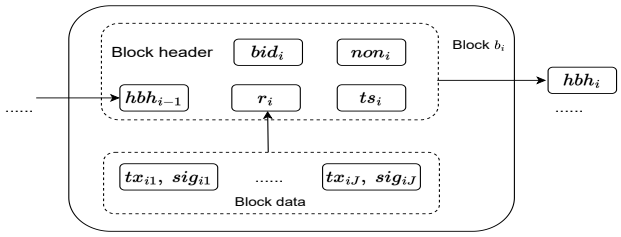

Fig. 1. A general block structure for a blockchain

- Assume there are J transactions in block i, each transaction $tx_{ij}(j \in [1,\ J])$ is signed using a signature scheme SIG, i.e., $sig_{ij} \leftarrow SIG(tx_{ij})$. Then block data bd_i, i.e., $bd_i = ([tx_{i1},\ sig_{i1}],\ ...,\ [tx_{iJ},\ sig_{iJ}])$.

- The consensus nodes aggregate bd_i with a Merkle Tree (MT) by using a hash function H. i.e., $r_i \leftarrow \mathrm{MT}(H;\ bd_i)$. Then consensus nodes calculate the hash value hbh_{i-1} of the previous block header bh_{i-1}, i.e., $hbh_{i-1} = H(bh_{i-1})$.
- The block header bh_i is formed, i.e., $bh_i = (hbh_{i-1},\ r_i,\ bid_i,\ non_i,\ ts_i)$. The block b_i is formed as $b_i = (bd_i,\ bh_i)$.
- The consensus nodes calculate the hash value of the block header of the block b_i, $hbh_i = H(bh_i)$, which is recorded in the next block b_{i+1}.

Then the process of calculating the hash value of the block header hbh_i using a blockchain algorithm Blc is defined as follows:

$$(a_p,\ hbh_i) \leftarrow Blc(..., [tx_{ip},\ sig_{ip}], ...)$$

where $p \in [1,\ J]$ and a_p is the authentication path from the transaction $[tx_{ip},\ sig_{ip}]$ to hbh_i.

Blockchain integrity. Let a blockchain be an entity that maintains an auditable database $\mathbf{L}$. We model the capability of an adversary against a blockchain as corrupted users or power-limited consensus nodes (no more than 49% malicious nodes). A successful adversary could launch the following attacks that bypass the blockchain auditing check:

- **Tampering attack.** The adversary changes, adds, or removes information in the blockchain without being audited.
- **Back-dating attack.** The adversary claims any non-existed information on the blockchain.

Then we define the data integrity property of $\mathbf{L}$ as follows:

Definition 2. *A blockchain $\boldsymbol{L}$ holds data integrity, if for any Probabilistic Polynomial Time (PPT) adversary $\mathcal{A}$, the probability of making either a tampering attack or a back-dating attack is negligible.*

2.4 Quantum random oracle

In this scheme, we are concerned with its post-quantum security. On a classical computer, we can model a hash function as an random oracle F. Based on Boneh's work [5], hash functions used in this work meet the *history free* reduction. Therefore, we can model the hash function as a quantum random oracle. Formally, for the case of an random oracle F, executions of the unitaries describing the adversary are interleaved with executions of an oracle unitary:

$$\mathcal{O}_f : \sum_{x,y} \alpha_{x,y} |x\rangle |y\rangle \rightarrow \sum_{x,y} \alpha_{x,y} |x\rangle |y \oplus f(x)\rangle . \tag{1}$$

For q queries, the adversary is described by a sequence of unitaries U_0,...,U_q and executed as $U_q \mathcal{O}_f U_{q-1} \mathcal{O}_f ... \mathcal{O}_f U_0 |0\rangle$.

3 Generic definitions for a Blockchain-Aided Hash-based Signature Scheme (BAHS)

Let $com = \mathsf{commit}(sk, m)$ be a hash-based commitment scheme. The function of commit can be realized by the Merkle tree in the blockchain. The main concept behind our one-time signature scheme is for a signer to commit to both the private signing key and the message and store the commitment on one block of a blockchain. At a later time, the signer opens the commitment with the message and private key and stores the opened commitment on another block of the blockchain. Each signer's commitment and opened commitment form a signature, which can then be verified by anyone who can access the blockchain. In our BAHS scheme, each block generation happens at a time epoch. Authentication of signers is application-oriented, i.e., some applications only allow legitimate users to submit their keys and signatures; some applications allow any users to do so. The choice of user authentication is out the scope of this paper.

3.1 Notation

The notation is listed in Table 1.

Table 1. Notation used in the BAHS scheme

Notation	Meaning
$\mathcal{S}$	signer space
$i \in \mathcal{S}$	signer identity
sk_i	signer i's signing key
pk_i	signer i's public key
m_i	message to be signed by i
$cInput_i$	signer i's commitment input
$oInput_i$	signer i's opened commitment input
com_i	signer i's commitment
$\tilde{com}_i$	signer's opened commitment
σ_i	signer i's signature
β	epoch index associated with a block in the blockchain
$\mathbf{L}_\beta$	blockchain database from the genesis block to β-th block

BAHS players. A BAHS scheme consists of three types of players: a **blockchain**, a set of **signers**, and a set of **verifiers**.

- By maintaining its database $\mathbf{L}$, the blockchain aids signers by storing their commitments, opened commitments and signatures. The blockchain also maintains the status information of those commitments.

- Let $\mathcal{S}$ be the space of signers. Given a message m_i, a signer $i \in \mathcal{S}$ generates their one-time secret signing key sk_i, input $cInput_i$ to commitment com_i and $oInput_i$ input to the opened commitment $\tilde{com}_i$ and submits $cInput_i$ and $oInput_i$ to the blockchain in two different blocks. The outputs on the blockchain can become com_i and $\tilde{com}_i$, respectively. Finally, signature is $\sigma_i = (cInput_i, com_i, oInput_i, \tilde{com}_i)$.
- A verifier retrieves a signature from the blockchain and verifies the signature.

BAHS key management. Let $\mathcal{S}_\beta \subset \mathcal{S}$ be the set of signers whose commitments have appeared in the blockchain's database up to the start of epoch β. The blockchain maintains information about the status of $cInput_i$ (because it is computed by signing key sk_i), $i \in \mathcal{S}_\beta$, for each epoch β, and this information is denoted by info_β^i. We write info_β for the set of all these info_β^i with different i and info^i for the set of all these info_β^i with different β. We give the definition of info_β as follows:

Definition 3. *The key status information info_β can be retrieved from the blockchain. It can be used to obtain the status, status_β^i, of any given $cInput_i$. This status will be as follows :*

$$\mathsf{status}_\beta^i \in \{(cInput_i,\ +),\ (cInput_i,\ -),\ (cInput_i,\ \perp)\},$$

where $(cInput_i,\ +)$ means that $cInput_i$ has been submitted to the blockchain and the signer i is allowed to sign, $(cInput_i,\ -)$ means that $cInput_i$ has been submitted to the blockchain but the signer i is not allowed to sign, $(cInput_i,\ \perp)$ means that $cInput_i$ has not yet been submitted to the blockchain.

3.2 Description of a generic construction of BAHS

A generic construction of BAHS is described in a timeline with multiple epochs and it consists of the following algorithms/protocols. Note that during signing, the signer will generate the signing key and message to be signed, in which both sk and m are random numbers.

- $\mathrm{Setup}(1^\lambda) \to (pp, \mathsf{info}_0, \mathbf{L})$: In epoch 0, the blockchain nodes run the Setup algorithm by taking as input a security parameter λ and outputting the system parameters pp, the initial system information info_0 and database $\mathbf{L}$.
- $\mathrm{Sign}\{\beta, \mathsf{info}_\beta, (sk_i, m_i)_{i\in[Q]}, \mathbf{L}_\beta\} \to (\sigma_i, \mathbf{L}_{\beta+1}, \mathsf{info}_{\beta+1})$: In epoch β, a set of Q users and the blockchain nodes run the Sign protocol as follows. We assume $Q = M + N$, M is the number of users submitting the commitment onto the blockchain, N is the number of users opening their commitment onto the blockchain. The nodes take as inputs system information info_β, and the database $\mathbf{L}_\beta$. A user is in one of the two following stages:
  1. Committing. For a user i_b ($i_b \in [M]$), who wants to commit, given a private key sk_{i_b} and a message m_{i_b}, he/she computes the input to the commitment $cInput_{i_b}$ and submits it to the blockchain. If $cInput_{i_b}$ does not exist on the blockchain database $\mathbf{L}$, the blockchain nodes will record it and the record of this transaction is called commitment com_{i_b}.

2. Opening. To make a signature publicly verifiable, user $j_d \in [N]$ releases sk_{j_d} on the blockchain to open the commitment. Upon receiving sk_{j_d}, the blockchain nodes check the status of $cInput_{j_d}$. If sk_{j_d} has not been used before, the blockchain nodes store $oInput_{j_d} = (cInput_{j_d}, m_{j_d}, sk_{j_d})$ to the blockchain datebase $\mathbf{L}$, and the record of this transaction is called opened commitment $\tilde{com}_{j_d}$.

After the signing protocol, the outputs include a signature $\sigma_i = (cInput, com_i, oInput, \tilde{com}_i)$, the updated database $\mathbf{L}_{\beta+1}$, in which public information for verification forms signer i's public key pk_i, and system information $\mathsf{info}_{\beta+1}$ for the next epoch. From a signer's view, the Committing and Opening stages are run in sequence in two different blocks. From the blockchain's view, these two stages are run simultaneously in every block for different signers.

– Verify$(\sigma_i, \mathsf{info}^i) \to 0/1$: In any epoch after a signature σ_i is generated and available on the blockchain database $\mathbf{L}$, a verifier can retrieve σ_i together with the system information info^i and verify it. The verifier outputs 0 for rejecting the signature and 1 for accepting it.

3.3 Security model for BAHS

We adopt a security model modified based on [18] for the BAHS scheme. The capability of an adversary against the BAHS scheme can be modeled as corrupted signers who can generate signing keys by themselves or outside attackers who cannot obtain signing keys. A successful adversary can launch any one of the following attacks:

– Tampering attack. The adversary changes, adds, or deletes existing records on the blockchain.
– Forging attack. The adversary claims a valid signature that is generated or released by an entity using a signing key more than once or is not generated by an entity.

The security of a BAHS scheme can be captured through two properties: correctness and unforgeability. The unforgeability is defined as an experiment, which is performed between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. Several global variables are used in experiments: h records the honest signer, M is the number of signers who are invoked in the experiment, and K is the number of honest signers who attempt to submit commitments onto the blockchain. β_{Current} and β_{Revoke} denote the current epoch as well as the epoch in which the honest signer is revoked. $\mathcal{R}$ is the set of signers to be revoked. The adversary can access the blockchain database $\mathbf{L}$ and the system information info_β for any epoch β.

Note that we need to model hash functions as oracles. It is common practice to model hash functions as random oracles [22], specifically, with a random value space, and a table T to record values. Furthermore, Boneh *et al.* [5] formalized the notion of quantum-accessible random oracle model (QROM), where the adversary can query the classical random oracle (RO) with quantum states. They introduced a concept called history-free reduction, showing that certain

lattice-based schemes in the random oracle model (ROM) can be proven secure in QROM, such as GPV'08 [16]. Specifically, if a simulator can decide the classical RO answers independently of the history of previous queries, then it implies security in the QROM. Therefore, in our scheme, the hash function can be modeled as a quantum random oracle because it meets the *history-free* reduction requirements and the scheme can be proved post-quantum secure. A simple definition of executions in quantum random oracle is given in Section 2.4.

Correctness. In general, correctness means a signature generated by an honest signer should always be valid (if the signer has not been revoked). We give the definition of correctness as follows:

Definition 4. *A BAHS scheme is correct that we get the result* $1 \leftarrow \text{Verify}(\sigma_i, \mathit{info}^i)$, *where* λ *is the security parameter, if* $(\mathit{info}_{\beta+1}, \boldsymbol{L}_{\beta+1}, \sigma_i) \leftarrow \text{Sign}\left\{\beta, \mathit{info}_\beta, (sk_i, m_i)_{i\in[Q]}, \boldsymbol{L}_\beta\right\}$ *and* $(pp, \mathit{info}_0, \boldsymbol{L}) \leftarrow \text{Setup}(1^\lambda)$.

Unforgeability. It means that the adversary can corrupt any number of signers except for one honest signer h. The adversary can query signatures from h on any messages at the adversary's choice, but can not generate a new valid signature of h. The adversary can generate a valid signature σ_i for a corrupted signer i but this signature generation must be with the assistance of the blockchain. Formally, unforgeability is defined as an experiment in Fig. 2.

Experiment $Exp^{Unforge}_{BAHS,\mathcal{A}}(\lambda)$

- $h = \perp$; $iS = \emptyset$; $cS = \emptyset$.
- $(pp,\ \mathbf{L},\ \mathsf{info}_0) \leftarrow \text{Setup}(1^\lambda)$, $\mathbf{L} = \emptyset$, $\mathsf{info}_0 = \emptyset$.
- $(\mathbf{L}_{\beta+1}, \sigma_i,\ \mathsf{info}_{\beta+1}) \leftarrow \mathcal{A}^{\mathbf{AddHU},\mathbf{AddCU},\mathbf{Update},\mathbf{Revoke},\mathbf{H}}(pp,\ \mathbf{L},\ \mathsf{info}_0)$
- If $i = h$:
  - If $(cInput_h,\ m) \in iS \wedge (\sigma_h,\ m) \in cS$ return 0.
- Else:
  - If $(\sigma_i,\ m) \in cS$ return 0.
- Return Verify $(\sigma_i,\ \mathsf{info}^i)$.

Fig. 2. The unforgeability experiment for the BAHS scheme

The adversary can have access to the following oracles and the details of oracles are shown in Fig. 6 in Appendix A. We present details of the random oracle **H** and it can be modelled as the corresponding quantum random oracle following the definition in Section 2.4.

- **AddHU()**: This oracle allows the adversary to add a single honest signer in the experiment. In each call, this oracle executes the submission of the commitment onto the blockchain by simulating the honest signer and the

blockchain. This oracle can be called at most $k(\lambda)$ times where $k(\cdot)$ is any polynomial. Once the commitment is submitted successfully, further calls will be ignored. This oracle only returns the honest signer's input to the commitment $cInput_h$.

- **AddCU**(i, $cInput_i$): This oracle allows the adversary to add a corrupt signer i to the system. The adversary can choose the corrupted signer's signing key sk_i and the corresponding input $cInput_i$ to the commitment.
- **Revoke**($\mathcal{R}$): This oracle allows the adversary to update the information list from $\mathsf{info}_{\beta_{\mathsf{Current}}}$ to $\mathsf{info}_{\beta_{\mathsf{Current+1}}}$, by revoking the set of signers $\mathcal{R}$ and keeping the remaining. If h is revoked in this oracle query, set β_{Revoke} to β_{Current}.
- **Update**(): This oracle allows the adversary to query the signature associated with a signer i which is recorded in the list cS. Note that the signer i can be an honest signer h or a corrupted signer $i \neq h$, who was created by the adversary via the **AddCU**(i, $cInput_i$) oracle.
- **H()**: On input a string x, the oracle checks if x has been queried before. If yes, it returns $T[x]$. If no, a random string h can be returned and be recorded as $T[x] = h$.

Based on the above definitions, we define unforgeability as follows:

Definition 5. *A BAHS scheme is unforgeable, if for any p.p.t. (quantum) adversary $\mathcal{A}$, the following condition holds:*

$$\Pr\left[\mathrm{Exp}_{BAHS,\mathcal{A}}^{Unforge}(1^\lambda) = 1\right] \leq \mathrm{negl}(\lambda) \tag{2}$$

4 The Blockchain-aided Hash-based Signature scheme (BAHS)

We now present a concrete BAHS scheme. In this scheme, we need the following three extra hash functions: $H_1 : \{0,1\}^\lambda \to \{0,1\}^l$; $H_2 : \{0,1\}^* \times \{0,1\}^\lambda \to \{0,1\}^l$; $H_3 : \{0,1\}^l \times \{0,1\}^l \times \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^l$, where λ is the system security level and l is the length of hash outputs. It is required that H_1, H_2, and H_3 hold the properties of one-wayness and collision-resistance. Note that in this scheme, the key status information can be instantiated by the status of $H_1(sk)$, which is part of the input $cInput$. In the following proof, we will apply this instantiation.

4.1 BAHS algorithms/protocols

Following the BAHS syntax in Section 3.2, the concrete BAHS algorithms/protocols are instantiated in detail as follows:

protocol 1: Sign protocols for BAHS

Input: β, $\mathbf{L}$, info_β, $cInput_{[M]}$, $oInput_{[N]}$, /* $(M,\ N) \in \mathbb{N} \times \mathbb{N}$, $[M] = \{i_1,\ \ldots,\ i_M\}$, $[N] = \{j_1,\ \ldots,\ j_N\}$, $(i_b,\ j_d) \in \mathcal{S} \times \mathcal{S}$, $[M] \cap [N] = \emptyset$. */
Output: $\mathbf{L}$ (updated), $\mathsf{info}_{\beta+1}$.
initiate $\mathsf{info}_{\beta+1} = \emptyset$, $z_{[M+N]} = \emptyset$; /* A set storing leaf values */
initiate $a_{[M+N]} = \emptyset$; /* A set storing authentication path */
initiate $r_\beta = \emptyset$; /* This is used to store the root value. */, /* $\mathsf{status}^k_\beta \in \{(H_1(sk_k),\ +),\ (H_1(sk_k),\ -),\ (H_1(sk_k,\ \perp)\}$*/.
$\forall k \in \mathcal{S}_\beta$, set $\mathsf{info}^k_{\beta+1} = \mathsf{info}^k_\beta$;
for $b = 1;\ b \le M;\ b++$ **do**
  initiate $\sigma_{i_b} = \emptyset$;
  obtain $\mathsf{status}^{i_b}_\beta$ from $\mathsf{info}^{i_b}_\beta$;/* $A = H_1(sk_{i_b})$ and $B = H_2(m_{i_b},\ sk_{i_b})$. */
  parse $cInput_{i_b} = A||B$;
  if $\mathit{status}^{i_b}_\beta = (A,\ \perp)$ **then**
    set $\mathsf{status}^{i_b}_{\beta+1} = (A,\ +)$, $\sigma_{i_b} = \sigma_{i_b} \cup cInput_{i_b}$; compute $z_b = H_2(cInput_{i_b})$;
  else
    reject this entry;
  end
end
for $d = 1;\ d \le N;\ d++$ **do**
  initiate $\sigma_{j_d} = \emptyset$, $num.\mathsf{H}_1(sk_{j_d}) = 0$, /* $num.\mathsf{H}_1(sk_{j_d})$ is the number of $\mathsf{H}_1(sk_{j_d})$; */; obtain $\mathsf{status}^{j_d}_\beta$ from $\mathsf{info}^{j_d}_\beta$; /* $C = H_1(sk_{j_d})$, $D = H_2(m_{j_d},\ sk_{j_d})$, $E = sk_{j_d}$, and $F = m_{j_d}$; */
  parse $oInput_{j_d} = C||D||E||F$;
  if $\mathit{status}^{j_d}_\beta = (C, \perp) \vee (C, -)$ **then**
    reject this entry;
  else
    if $\mathit{status}^{j_d}_\beta = (C,\ +)$ **then**
      retrieve $cInput_{j_d} = C||D'$ from $\mathbf{L}$;
    else
      if $D = D' \wedge H_1(E) = C \wedge H_2(F,\ E) = D$ **then**
        $num.\mathsf{H}_1(sk_{j_d})++$;
      else
        reject this entry;
      end
    end
    if $num.\mathsf{H}_1(sk_{j_d}) == 1$ **then**
      set $\mathsf{status}^{j_d}_{\beta+1} = (C,\ -)$;
      set $\sigma_{j_d} = \sigma_{j_d} \cup oInput_{j_d}$;
      compute $z_{d+M} = H_3(oInput_{j_d})$;
    else
      reject this entry;
    end
  end
end
compute a block β by using the Blc algorithm: $(a_i,\ hbh_\beta) \leftarrow Blc(z_1,\ ..., z_i, ...,\ z_{M+N})$;
for $b = 1;\ b \le M;\ b++$ **do**
  set $\sigma_{i_b} = \sigma_{i_b} \cup z_b \cup a_b \cup hbh_\beta$; $\mathbf{L}$=$\mathbf{L} \cup \sigma_{i_b}$;
end
for $d = 1;\ d \le N;\ d++$ **do**
  set $\sigma_{j_d} = \sigma_{j_d} \cup z_{d+M} \cup a_{d+M} \cup hbh_\beta$; $\mathbf{L}$=$\mathbf{L} \cup \sigma_{j_d}$;
end

– Setup$(1^\lambda) \rightarrow (pp,\ \mathsf{info}_0,\ \mathbf{L})$: The blockchain nodes run this algorithm Setup to initiate the system. Given a security parameter λ, choose three hash functions H_1, H_2 and H_3, initiate the system public parameters pp, the beginning epoch as epoch 0, the associated system information info_0 and the database $\mathbf{L}$ to be empty.

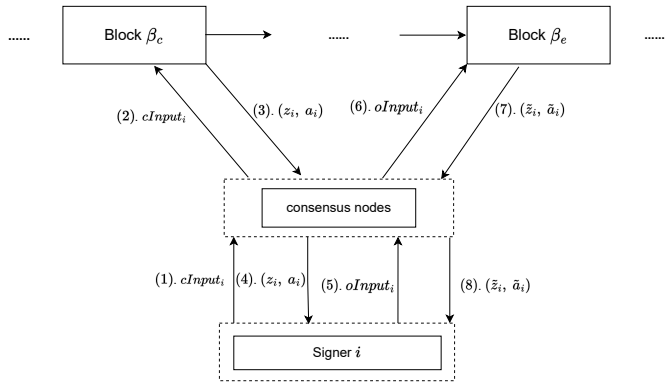

Fig. 3. The signature generation protocol from the signer's view

– Sign($\beta,\ \mathsf{info}_\beta,\ (sk_i, m_i)_{i\in[Q]},\ \mathbf{L}_\beta$) $\rightarrow$ ($\sigma_i,\ \mathbf{L}_{\beta+1},\ \mathsf{info}_{\beta+1}$): From a signer's view, the process of this protocol can be seen in Fig. 3. From s ledger's view, the process of this protocol can be shown in Fig. 4. All steps are also arranged in protocol 1.
  - Committing: For user i_b ($i_b \in [M]$), the commitment input is computed as $cInput_{i_b} = A||B$, $A = H_1(sk_{i_b})$ and $B = H_2(m_{i_b},\ sk_{i_b})$. Then the user sends the $cInput_{i_b}$ to the blockchain nodes. The blockchain nodes need to check the validity of A based on info_β. If the check result is positive, the blockchain nodes add $cInput_{i_b}$ to the corresponding signer's commitment com_{i_b} recorded by the blockchain by time-stamping the input $cInput_{i_b}$. Note that $com_{i_b} = (cInput_{i_b}, z_b, a_b, hbh_\beta)$, where z_b is a leaf value, a_b is the authentication path, hbh_β is the hash value of the block header. Otherwise, reject it.
  - Opening: To make a signature publicly verifiable, user j_d ($j_d \in [N]$) releases $oInput_{j_d} = (cInput_{j_d}, m_{j_d}, sk_{j_d}) = (H_1(sk_{j_d})||H_2(m_{j_d}, sk_{j_d})||sk_{j_d}||$ $m_{j_d}) = C||D||E||F$ on the blockchain. Upon receiving the $oInput_{j_d}$, the blockchain nodes use C to retrieve $cInput_{j_d}$. Then the blockchain nodes check the validity of $oInput_{j_d}$. If the check is positive, the blockchain nodes add $oInput_{j_d}$ to the corresponding signer's opened commitment $\tilde{com}_{j_d}$ recorded by the blockchain by time-stamping the $oInput_{j_d}$. Note that $\tilde{com}_{j_d} = (oInput_{j_d}, \tilde{z}_d, \tilde{a}_d, hbh_\beta)$, where $\tilde{z}_d$ is a leaf value, $\tilde{a}_d$ is the authentication path, hbh_β is the hash value of the block header. Otherwise, reject it.

  For a signer i, the signature $\sigma_i = (cInput_i, com_i, oInput_i, \tilde{com}_i)$ ($\beta < \beta'$). The public key for the signer i is $pk_i = (a_b, hbh_\beta, \tilde{a}_b, hbh_{\beta'})$. Finally, the blockchain outputs the updated database $\mathbf{L}_{\beta+1}$ and information list $\mathsf{info}_{\beta+1}$.
– Verify($\sigma_i,\ \mathsf{info}^i$) $\rightarrow$ 0/1: A verifier runs the algorithm Verify to verify a signature. The verifier works as follows:
  - Parses σ_i as $(com_i, \tilde{com}_i)$, where $\beta' < \beta$.
  - Computes $z_i' = H_2(cInput_i)$ and checks whether z_i' equals z_i or not.

- Use z_i and a_i to recompute the Merkle tree root in the block β', and then compute $hbh'_{\beta'}$. Finally checks whether $hbh'_{\beta'}$ equals $hbh_{\beta'}$ or not.
- Computes $z'_i = H_3(oInput_i)$ and checks whether $\tilde{z}'_i$ equals $\tilde{z}_i$ or not.
- Use $\tilde{a}_i$, $\tilde{z}_i$ to recompute the Merkle tree root and hbh'_{β}. Checks whether hbh'_{β} equals hbh_{β} or not.
- If all previous checks pass, outputs 1 for "accept". Otherwise, outputs 0 for "reject".

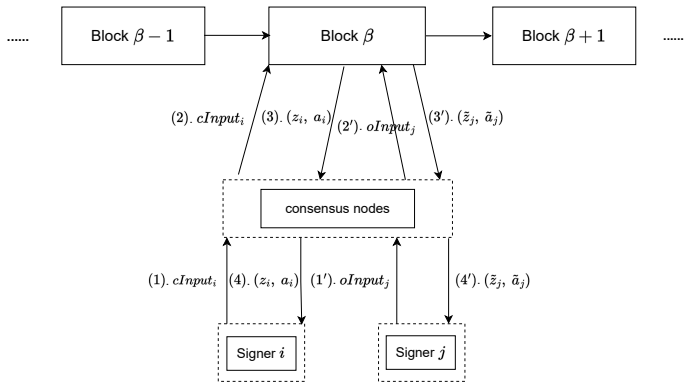

Fig. 4. The signature generation protocol from the blockchain's view

5 Security Analysis

Following the security model defined in Section 3.3, we need to clarify three random oracles $\mathbf{H}_1$, $\mathbf{H}_2$ and $\mathbf{H}_3$ for hash functions H_1, H_2 and H_3, respectively. Details of these oracles are similar to the definition in Section 3.3. Due to the page limit, we omit details here. In the unforgeability experiment, these three oracles can not only be accessed by the adversary but also internally be called by the simulation of the **AddHU** and **Update** oracles. The oracle **AddHU** includes the process of Committing while **Update** includes the process of Sign.

Because the simulator of each RO $\mathbf{H}_i$, $i \in \{1, 2, 3\}$, can decide the RO answers independently of the history of previous queries, hash functions meet the *history-free* reduction requirements. These three ROs can be modeled as QROs.

Theorem 1. *The BAHS scheme is correct, assuming the hash function H_1 is collision-resistant and the blockchain follows the BAHS scheme description correctly.*

Proof. On one hand, if a signer $\mathcal{A}$ is corrupt, there will be two cases. Firstly, he can predict the honest signer's signing key sk_h and the corresponding commitment is successfully submitted with the same $H_1(sk_h)$ in a previous session.

However sk_h must be selected at random, and the probability of $\mathcal{A}$ picking the same signing key, i.e., $sk_i = sk_h$, is negligible in the security parameter. Except this, the only probability is that $sk_h \neq sk_i$ but $H_1(sk_h) = H_1(sk_i)$ – when this happens, the collision of the hash function H_1 is found, which contradicts to the assumption that the function H_1 is collision-resistant. Therefore, the probability of this case happening is negligible. Secondly, there is another signer i created by $\mathcal{A}$ with $sk_i = sk_h$ and this signer i is revoked when h is valid. If the adversary attempts to add i with $sk_i = sk_h$ after the signer h, it will be rejected by the blockchain. Therefore, the adversary's attempt will always fail.

On the other hand, based on $\mathsf{status}_\beta^h = (H_1(sk_h),\ +)$, we can get that at epoch β, the honest signer h has been submitted and is allowed to sign. Following the BAHS scheme description, the signature on the blockchain for this valid signer h can pass the Verify algorithm.

Theorem 2. *The BAHS scheme is unforgeable if the hash function H_1 holds properties of one-wayness and collision-resistance, the hash functions H_2 and H_3 hold properties of collision-resistance, and the blockchain follows the BAHS scheme descriptions correctly and holds integrity.*

Proof. The adversary wins the unforgeability experiment in any one of the two scenarios: (1) The adversary generates $(\sigma_h,\ \mathsf{info}^h)$ for an honest signer h, where com_h and σ_h are respectively a valid signer commitment and signature for m at epoch β, $\mathsf{status}_\beta^h = (H_1(sk_h),\ +)$. (2) The adversary generates $(\sigma_i,\ \mathsf{info}^i)$ for a corrupted signer i, who is controlled by the adversary, and the adversary does not get help from the blockchain. The proof for unforgeability is as follows.

In scenario (1), The adversary outputs $(\sigma_h,\ \mathsf{info}^h)$, which meets following conditions:

- $\mathsf{status}_\beta^h = (H_1(sk_h),\ +) \wedge \mathrm{Verify}(\mathsf{info}^h,\ \sigma_h) = 1$.
- $(cInput_h,\ m) \notin iS \vee (\sigma_h,\ m) \notin cS$.

This may happen in any one of the following cases:

1. The honest signer h generated an input to the commitment $cInput_h$ and $H_1(sk_h)$, which have been recorded on the blockchain. If the adversary wins the game, there are some sub-cases described as follows:
   - **Case 1**. Given certain record $H_1(sk_h)$ on the blockchain, the adversary gets the right signing key sk_h. Using sk_h and a different message m', the adversary can create a valid commitment input $cInput' = H_1(sk_h)||H_2(m',\ sk_h)$ by querying the oracle $\mathbf{H}_2$. Then the commitment input $cInput'$ can pass the blockchain's check and the algorithm Verify. This means the one-wayness of the hash function H_1 is broken which is contradicted with the assumption that the hash function H_1 is one-way. Therefore, the probability of this sub-case happening is negligible.
   - **Case 2**. The adversary can use a different pair of sk' and m' to query oracles H_1 and H_2 to get $H_1(sk_h) = H_1(sk')$, $H_2(m_h,\ sk_h) = H_2(m',\ sk')$. Then the adversary can forge a valid commitment input and open it on

the blockchain before the honest signer h opens it. This means two scenarios happened at the same time: (1) the challenger can find a collision sk' and sk_h in the oracle H_1, which contradicts the assumption that the function H_1 is collision-resistant; (2) the challenger can find a collision $m'||sk'$ and $m_h||sk_h$ in the oracle H_2, which contradicts to the assumption that the function H_2 is collision-resistant; Therefore, the probability of this case happened is negligible.

2. The honest signer h has send the input to the opened commitment $oInput_h = (cInput_h,\ m_h,\ sk_h)$ to the blockchain. In this case, the adversary can get the signing key sk_h directly. Using the signing key sk_h, the adversary can use a different message m' to generate the commitment input $cInput' = H_1(sk_h)||H_2(m',\ sk_h)$, which can be submitted to the blockchain. However, this is contradicted with the assumption the blockchain follows the scheme description. Therefore, the probability of this case happening is negligible.
3. During computing the Merkle tree, the adversary can have access to the blockchain to change some input value of hash function H_2. For example, the adversary changes certain leaf values z_h to be z'. If the final record on (including the authentication path a_i) the blockchain can keep the same, which means the challenger finds a collision for the hash function H_2. This is contradicted to the assumption the hash function H_2 is collision-resistant. Or the final record on the blockchain can be changed. This is contradicted to the assumption that the blockchain is trusted. Therefore, the probability of this case happening is negligible.

In scenario (2) $i \neq h$, the adversary outputs $(\sigma_i,\ \mathsf{info}^i)$ and there are some cases, which meet the following conditions:

- $\text{Verify}(\sigma_i,\ \mathsf{info}^i) = 1$
- $(\sigma_i,\ m) \notin cS$.

This may happen in any one of the following cases:

1. The adversary uses a different signing key sk_i to query oracles $\mathbf{H}_1$ and $\mathbf{H}_2$ to make $H_1(sk_i) = H_1(sk_j)$ and $H_2(m,\ sk_i) = H_2(m,\ sk_j)$, then submits the commitment input $cInput_j$ on the blockchain to claim that this commitment is valid for an uncorrupt signer j. This means the challenger can find a collision in oracles $\mathbf{H}_1$ and $\mathbf{H}_2$, which is contradicted to the assumption that hash functions H_1 and H_2 are collision-resistant. Therefore, the probability of this case happening is negligible.
2. The adversary can use a different pair of sk_i and m_i to query oracles H_1 and H_2 to get $H_1(sk_i) = H_1(sk_j)$, $H_2(m_i,\ sk_i) = H_2(m_j,\ sk_j)$. Then the adversary can forge a valid commitment input and submit it on the blockchain, which is considered as a valid commitment input generated by the signer j. This means two scenarios happened at the same time (1) the challenger can find a collision sk_i and sk_j in the oracle H_1, which contradicts to the assumption that the function H_1 is collision-resistant; (2) the challenger can find a collision $m_i||sk_i$ and $m_j||sk_j$ in the oracle H_2, which contradicts to

the assumption that the function H_2 is collision-resistant; Therefore, the probability of this case happened is negligible.

3. Because the adversary can control a corrupted signer to get the signer's signing key sk_i. The adversary can send a number of commitment inputs $cInput_j$, $j \in [1,\ R]$ with one signing key sk_i and different messages m_j, $j \in [1,\ R]$ to the blockchain in one block. These commitment inputs can be recorded on the blockchain. Then the adversary can try to open these commitments to the blockchain. However, this is contradicted to the assumption that one signing key can be used only once. If there is more than one commitment input using the same signing key recorded on the blockchain, all of these commitment inputs will be rejected. So the probability of this case happening is negligible.
4. Considering a signer i has submitted the input to the opened commitment $(cInput_i,\ m_i,\ sk_i)$ to the blockchain, the adversary uses a different pair of $(m',\ sk')$ to generate the input to the opened commitment $cInput'_i = H_1(sk')||H_2(m',\ sk') = cInput_h$ to make $H_1(sk_h) = H_1(sk')$, $H_3(H_1(sk')||H_2(m', sk')||m'||sk') = H_3(H_1(sk_h)||H_2(m_h, sk_h)||m_h||sk_h)$. This means the challenger finds a collision in H_1, H_2 and H_3, which is contradicted to the assumption that hash functions H_1, H_2 and H_3 are all collision-resistant. Therefore, the probability of this case happening is negligible.
5. During computing the Merkle tree, the adversary can have access to the blockchain to change some input value of hash function H_2. The adversary changes a certain leaf value z_h to be z'. If the final record (including the authentication path a_i) on the blockchain can keep the same, which means the challenger finds a collision for the hash function H_2. This is contradicted to the assumption the hash function H_2 is collision-resistant. Or then the final record on the blockchain can be changed. This is contradicted to the assumption that the blockchain is trusted. Therefore, the probability of this case happening is negligible.

Overall, the BAHS scheme provides unforgeability.

6 Implementations

We have made a prototype implementation, in which we only measure the communication and computational overhead of our commitment scheme rather than the cost or transaction overhead on the blockchain.

Implementation of a specific blockchain. We implement our BAHS scheme in Python. Note that the signing time includes the time for the blockchain to generate the whole Merkle tree in one block and the corresponding hash value of the block header. The programs were compiled using Pycharm and executed on a laptop (processor: 2.6GHz, 6-Core, Intel Core i7; Memory: 16GB 2667 MHz) with the macOS operating system. We set the security level as 256-bit. As shown in Table 2, J is the number of signers in a block. For simplicity, we assume $\frac{J}{2}$ signers to submit commitments and the other $\frac{J}{2}$ signers to open signatures. We

implement two blocks as an example of blockchains. We choose 2^{10},..., 2^{15},...,2^{20} as different parameters for the number of signers, which are larger than that in Bitcoin. According to Table 2, we can see that the signing time is far less than 15 seconds in Ethereum or 10 minutes in Bitcoin. Our scheme is practical.

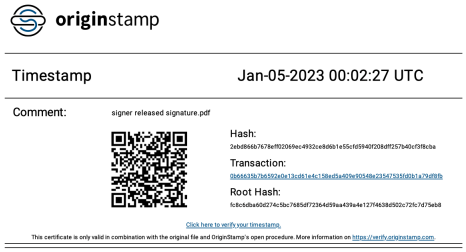

Fig. 5. The returned proof from the OriginStamp service

Implementation based on public blockchains. In this implementation, we want to test performance of the BAHS scheme on known blockchain platforms. Therefore, classic signature algorithms, such as ECDSA, used in existing blockchain does not influence our implementation. Considering existing blockchains, we use the platform "OriginStamp" to publish and timestamp our opened signature, which contains three typical blockchains, i.e., Bitcoin, Ethereum, and Aion. Due to the page limit, we take the Bitcoin as an example. We upload data to Bitcoin and each time the web server calculates the Merkle tree root value and inserts it into a Bitcoin transaction. After the transaction is committed, the web server returned a proof for verification, which is shown in Fig. 5. Also, the information on the certificate can be accessed at the website https://verify.originstamp.com.

Our BAHS scheme is the first blockchain-aided hash-based signature scheme and it is different from any traditional digital signature schemes, so we do not compare the BAHS scheme with other signature schemes.

7 Conclusion

In this paper, we propose a new one-time signature scheme, i.e., BAHS, in which signing keys, commitments and opened commitments are publicly accessible via a distributed blockchain. The BAHS scheme is much simpler than traditional signature schemes and other post-quantum signature schemes. We also provide a formal definition of the security model for the BAHS scheme and security proof. Finally, we implement this scheme and show its practicality.

Table 2. The implementation results for the signature scheme

Parameters	SS(KB)[a]	CIG(ms)[b]	ST(ms) [c]
$J = 2^{10}$	1.69	$5.42 * 10^{-3}$	3.38
$J = 2^{11}$	1.81	$5.36 * 10^{-3}$	5.94
$J = 2^{12}$	1.93	$5.79 * 10^{-3}$	11.55
$J = 2^{13}$	2.06	$6.03 * 10^{-3}$	23.15
$J = 2^{14}$	2.19	$5.61 * 10^{-3}$	53.23
$J = 2^{15}$	2.31	$5.26 * 10^{-3}$	102.04
$J = 2^{16}$	2.43	$2.82 * 10^{-3}$	258.52
$J = 2^{17}$	2.55	$1.3 * 10^{-3}$	499.65
$J = 2^{18}$	2.67	$0.68 * 10^{-3}$	1123.12
$J = 2^{19}$	2.79	$0.32 * 10^{-3}$	2131.42
$J = 2^{20}$	2.91	$0.16 * 10^{-3}$	4337.23

[a] SS stands for the signature size and KB means kilobytes.
[b] CIG stands for the commitment input generation time and ms stands for millisecond.
[c] ST stands for the signing time and ms stands for millisecond.

Acknowledgments

We thank the European Union's Horizon research and innovation program for support under grant agreement numbers: 101069688 (CONNECT), 101070627 (REWIRE), 952697 (ASSURED), 101019645 (SECANT) and 101095634 (ENTRUST). These projects are funded by the UK government's Horizon Europe guarantee and administered by UKRI. The first author thanks the China Scholarship Council (CSC) for providing the research scholarship. We also thank the anonymous reviewers from ISPEC for their valuable comments.

References

1. ISO/IEC 10118-1. Information technology – Security techniques – Hash functions – Part 1: General. Standard, (2016).
2. ISO/IEC CD 14888-4.2. Information technology – Security techniques – Digital signatures with appendix – Part 4: Stateful hash-based mechanisms, (2022).
3. Yakubov A., Shbair W., and Wallbom A. A blockchain-based PKI management framework. In *The First IEEE/IFIP International Workshop on Managing and Managed by Blockchain (Man2Block) colocated with IEEE/IFIP NOMS*, (2018).
4. Georg Becker. Merkle signature schemes, merkle trees and their cryptanalysis. *Ruhr-University Bochum, Tech. Rep*, 12:19, (2008).

5. Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In *ASIACRYPT*, pages 41–69, (2011).
6. Dario Catalano and Dario Fiore. Vector commitments and their applications. In *PKC*, pages 55–72, (2013).
7. Cozzo D. and Smart N. P. Sharing the luov: threshold post-quantum signatures. In *Cryptography and Coding: IMACC*, pages 128–153, (2019).
8. Johnson D., Menezes A., and Vanstone S. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1:36–63, (2001).
9. McGrew D., Kampanakis P., Fluhrer S., Gazdag S. L., Butin D., and Buchmann J. State management for hash-based signatures. In *SSR*, pages 244–260, (2016).
10. Ivan Damgård. Commitment schemes and zero-knowledge protocols. In *School organized by the European Educational Forum*, pages 63–86, (1998).
11. Bernstein D.J. and Hülsing A. The SPHINCS$^+$ signature framework. In *ACM CCS*, pages 2129–2146, (2019).
12. Bernstein D.J., Hopwood D., and Hülsing A. SPHINCS: practical stateless hash-based signatures. In *EUROCRYPT*, pages 368–397, (2015).
13. Adja Y. C. E., Hammi B., Ahmed S., and Zeadally S. A blockchain-based certificate revocation management and status verification system. *Computers & Security*, 104:102209, (2021).
14. Bansarkhani R. E., Mohamed M. S. E., and Petzoldt A. MQSAS-a multivariate sequential aggregate signature scheme. In *Information Security*, pages 426–439, (2016).
15. Bruinderink L. G. and Hülsing A. "Oops, i did it again"–security of one-time signatures under two-message attacks. In *SAC*, pages 299–322, (2017).
16. Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *STOC*, pages 197–206, (2008).
17. Stephen Holmes. Impact of post-quantum signatures on blockchain and DLT systems. In *DLT*, (2023).
18. Bootle J., Cerulli A., Chaidos P., Ghadafi E., and Groth J. Foundations of fully dynamic group signatures. *Journal of Cryptology*, 33(4):1822–1870, (2020).
19. Ari Juels and Martin Wattenberg. A fuzzy commitment scheme. In *ACM CCS*, pages 28–36, (1999).
20. Ducas L., Lepoint T., Lyubashevsky V., Schwabe P., Seiler G., and Stehlé D. Crystals–dilithium: Digital signatures from module lattices. (2018).
21. Lamport L. : Constructing digital signatures from a one way function. (1979).
22. Bellare M. and Rogaway P. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS*, (1993).
23. Laurane Marco, Abdullah Talayhan, and Serge Vaudenay. Making classical (threshold) signatures post-quantum for single use on a public ledger. *Cryptology ePrint Archive*, (2023/420).
24. Li Q. and Cao G. Multicast authentication in the smart grid with one-time signature. *IEEE Transactions on Smart Grid*, 2(4):686–696, (2011).
25. Gennaro R., Goldfeder S., and Narayanan A. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *ACNS*, pages 156–174, (2016).
26. Chang S.M., Shieh S., Lin W. W., and Hsieh C.M. An efficient broadcast authentication scheme in wireless sensor networks. In *ASIACCS*, pages 311–320, (2006).

Appendix

A Oracles for the unforgeability

AddHU()

- If $K = k(\lambda)$ return $\perp$.
- $K = K + 1$.
- If $h = \perp$: $N = N + 1$; $h = N + 1$.
- $com_h \leftarrow \text{Committing}(sk_h,\ m)$.
- If $cInput_h \neq \perp$:
  - $\beta_{\mathsf{Add}} = \beta_{\mathsf{Current}}$.
  - $K = k(\lambda)$.
  - Set $\mathsf{status}^h_{\beta_{\mathsf{Current}}} = (cInput_h,\ +)$ and let $\mathbf{L} = \mathbf{L} \cup com_h$.
- Return $(sk_h, cInput_h)$.

AddCU$(i,\ cInput_i)$

- If $i \notin [N+1] \vee i = h$, return $\perp$.
- If $\mathsf{status}^i_{\beta_{\mathsf{Current}}} \neq (cInput_i,\ \perp)$, return $\perp$.
- If $i = N + 1$: $N = N + 1$.
- Set $\mathsf{status}^i_{\beta_{\mathsf{Current}}} = (cInput_i,\ +)$ and let $\mathbf{L} = \mathbf{L} \cup cInput_i$.

Revoke$(\mathcal{R})$

- If $\mathcal{R} \not\subseteq [N]$ return $\perp$.
- $\beta_{\mathsf{Current}} = \beta_{\mathsf{Current}} + 1$.
- $\forall i \in \mathcal{R}$, set $\mathsf{status}^i_{\beta_{\mathsf{Current}}} = (cInput_i,\ -)$.
- If $h \in \mathcal{R}$ and $\beta_{\mathsf{Revoke}} = \infty$ set $\beta_{\mathsf{Revoke}} = \beta_{\mathsf{Current}}$.

Update $(\beta,\ \mathsf{info}_\beta,\ \mathbf{L},\ ...,\ cInput_{i_b}/oInput_{j_d},\ ...)$

- If inputs=$(\beta,\ \mathsf{info}_\beta,\ \mathbf{L},\ ...,\ cInput_i,\ ...)$
  - $(\sigma_{i_b},\ \mathbf{L}_{\beta+1}, \mathsf{info}_{\beta+1}) \leftarrow \text{Committing}(\beta,\ \mathsf{info}_\beta,\ \mathbf{L},\ ...,\ cInput_{i_b},\ ...)$.
  - Return σ_i.
- If inputs=$(\beta,\ \mathsf{info}_\beta,\ \mathbf{L},\ ...,\ oInput_{j_d},\ ...)$
  - $(\sigma_{j_d},\ \mathbf{L}_{\beta+1}, \mathsf{info}_{\beta+1}) \leftarrow \text{Opening}(\beta,\ \mathsf{info}_\beta,\ \mathbf{L},\ ...,\ oInput_{j_d},\ ...)$.
  - Return σ_{j_d}

Fig. 6. Oracles for the unforgeability