International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2025, 15(4): 465–494, doi: 10.21655/ijsi.1673-7288.00361

Research Article

Quantum-resistant and Efficient Blockchain Authentication Storage Scheme

Chuan Zhang (张川) [1], Xuhao Ren (任旭豪) [1], Haotian Deng (邓淏天)[1], Yajie Wang (王亚杰)[1], Chunhai Li (李春海)[3], Tong Wu (吴桐)[2], Licheng Wang (王励成)[1]

[1] (School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China)
[2] (School of Computer & Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China)
[3] (School of Information and Communication, Guilin University of Electronic Technology, Guilin 541004, China)
Corresponding author: Tong Wu, tongw@ustb.edu.cn

Abstract With the widespread application of blockchain technology, authenticated storage, as a core component, plays a crucial role in ensuring data integrity and consistency. In traditional blockchain systems, authenticated storage is maintained through a series of cryptographic algorithms, which verify transactions and preserve the integrity of ledger states. However, the advent of quantum computers has introduced a significant threat to existing blockchain authentication storage technologies, raising the risk of data breaches and compromised integrity. The most advanced authenticated storage schemes primarily rely on the bilinear Diffie-Hellman assumption, which is susceptible to quantum attacks. To enhance the security and efficiency of authenticated storage, this paper introduces a stateless hash signature mechanism and proposes the quantum-resistant blockchain authenticated storage scheme EQAS. The proposed scheme decouples data storage from data authentication, utilizes random forest chains to efficiently generate commitment proofs, and employs a hyper tree structure to perform efficient authentication. Security analyses show that EQAS is resistant to quantum algorithm attacks. Comparative experiments with other authenticated storage schemes demonstrate the superior efficiency and performance of EQAS in handling blockchain authentication storage tasks.

Keywords blockchain; quantum-resistant; authenticated storage; stateless hash signature

Citation Zhang C, Ren XH, Deng HT, Wang YJ, Li CH, Wu T, Wang LC. Quantum-resistant and Efficient Blockchain Authentication Storage Scheme, *International Journal of Software and Informatics*, 2025, 15(4): 465–494. http://www.ijsi.org/1673-7288/361.htm

In the digital age, blockchain technology has shown its revolutionary potential in financial services, supply chain management, smart contracts, and other fields with its unique decentralized characteristics and immutability[1, 2]. As an important part of the blockchain architecture, authenticated storage technology plays a key role in ensuring data integrity and

This is the English version of the Chinese article “抗量子的高效区块链认证存储方案. 软件学报, 2025, 36(10): 4483–4506. doi: 10.13328/j.cnki.jos.007394”
Funding items: National Key Research and Development Program (2022YFB2702700); Collaborative Innovation Fund of Shanghai (XTCX-KJ-2023-54); China Association for Science and Technology Youth Talent Support Project (2023QNRC001); Young Teachers Academic Launch Plan of Beijing Institute of Technology
Received 2024-07-01; Revised 2024-09-05; Accepted 2024-12-30; IJSI published online 2025-12-30

consistency. In traditional blockchains, authenticated storage relies on a series of cryptographic algorithms, which are responsible for verifying the validity of each transaction, ensuring the tamper-proof state of the ledger, and providing transparent data access for each participant in the network[3, 4].

Although blockchain authenticated storage technology has made remarkable achievements in ensuring data security and trust, the rapid development of quantum computers poses a threat to existing blockchain authenticated storage technologies[5–7]. The current mainstream authenticated storage technologies, such as signature algorithms based on RSA or elliptic curve cryptosystems, rely on hard problems such as integer factorization and discrete logarithms. However, with the progress of quantum computing technology, quantum algorithms such as Shor's algorithm[8, 9] can efficiently factor large integers and thus break encryption mechanisms based on these mathematical assumptions. Specifically, existing signature-based authenticated storage technologies become fragile under quantum attacks, meaning that attackers can forge authentication information and insert malicious data, while the system cannot detect tampering. This ultimately results in the loss of data integrity and consistency in blockchain systems. In addition, hash-based authenticated storage technologies also face threats. For example, quantum computing can use Grover's algorithm[10, 11] to accelerate hash collision searches, reducing the collision resistance of hash functions, thus making data authentication in blockchains unreliable[12]. Therefore, to maintain the long-term security and reliability of blockchain systems, existing authenticated storage technologies must be improved to ensure resistance to security threats in the quantum computing era.

There are several blockchain authenticated storage approaches. For instance, simplified payment verification (SPV)[13, 14] allows Bitcoin light nodes to verify transactions by downloading block headers and using Merkle tree proofs without downloading the entire blockchain. SPV light nodes rely on full nodes to provide block headers and transaction proofs to save storage and bandwidth, making them suitable for resource-limited devices. In Ethereum, the Merkle Patricia Trie (MPT) structure stores a value in each leaf node, while internal nodes store the cryptographic hash values of their children. The root hash serves as a commitment to the blockchain state, ensuring data integrity and verifiability. Li *et al.*[3, 4] proposed the efficient blockchain authenticated storage system (LVMT) that adopts a multi-layer design to support unlimited key-value pairs and stores version numbers instead of hash values, avoiding costly elliptic-curve multiplication operations and significantly reducing I/O amplification. Zhang *et al.*[15] proposed a new column-based learning storage mechanism for blockchain systems to reduce storage size and improve system throughput. However, the above works are based on hard problems such as discrete logarithms and the bilinear Diffie-Hellman assumption, which can be solved by quantum computers, leading to potential security risks for authenticated storage mechanisms[16, 17].

In the field of cryptography, many quantum-resistant schemes have been proposed, including cryptographic systems based on lattice theory, code theory, and multivariate equations. Although these schemes theoretically offer strong quantum resistance, they share some limitations. For example, lattice-based encryption schemes, while widely considered strong candidates for post-quantum cryptography, have high computational complexity and relatively large key and signature sizes, making them less ideal for blockchain scenarios that require efficient storage and fast verification. Cryptographic schemes based on codes and multivariate equations face similar challenges, typically requiring significant computational and storage overhead, resulting in low efficiency in practical applications. Compared with these complex schemes, hash-based signature schemes have become a major focus of quantum-resistant cryptography research due to their simple structure, high efficiency, and ease of implementation. This makes them particularly suitable for blockchain environments that require rapid verification of large numbers of transactions and state data. In addition, the stateless design of hash-based signature schemes adapts well to blockchain systems, reducing the storage burden and computational complexity of full nodes, especially in authenticated storage scenarios involving frequent interactions between light nodes and full nodes. This significantly improves scalability and performance. Currently, hash-based signature schemes are the first quantum-resistant signature schemes formally defined in two RFCs[18–20]. In this paper, an optimized version of SPHINCS$^+$, one of the nine signature proposals in the second round of the NIST Post-Quantum Cryptography Standardization Project, is adopted[21]. In view of this, we propose a quantum-resistant and efficient authenticated storage scheme called EQAS, whose main contributions are as follows:

(1) By combining stateless hash signatures with blockchain authenticated storage, in this paper we propose the first quantum-resistant blockchain authenticated storage mechanism, which achieves efficient authenticated storage while resisting quantum-era security threats.

(2) The random forest algorithm is dynamically extended to design the dynamic forest of random chains (DFORC), optimizing its performance in blockchain scenarios. EQAS decouples data storage from data authentication, efficiently generates commitment proofs through DFORC, and employs a hypertree structure for efficient authentication.

(3) Theoretical analysis and experiments are conducted on the security and efficiency of EQAS, showing that EQAS effectively resists quantum attacks in terms of security and demonstrates high performance in terms of efficiency.

In Section 1, we introduce related work on blockchain authenticated storage and quantum attack resistance. In Section 2 the necessary background is presented, including the concepts of authenticated storage and hash-based signatures. We provide in Section 3 an overview of the EQAS scheme, including its system model, workflow, and threat model. We detail the design of EQAS in Section 4. In Section 5, we analyze its quantum security, and we verify the effectiveness of the proposed scheme through experiments in Section 6. Finally, the paper is concluded with a summary.

1 Related Work

1.1 Authenticated storage

Initially, in the context of distributed system outsourced storage, authenticated storage mainly focuses on how to ensure the integrity and availability of stored data when the server is untrusted. With the rise of cloud computing, cloud storage technology has become a research hotspot, meeting the storage demands of the big data era by providing dynamic and scalable storage resources. The key technologies of cloud storage include data encryption, access control, and data deduplication to ensure data security and privacy[22]. The introduction of blockchain technology has brought new changes to authenticated storage[23]. The immutability and decentralization of blockchain make it show great potential in identity authentication and data integrity verification. For example, a self-sovereign authentication scheme based on blockchain and cryptographic accumulators improves system performance and scalability by simplifying blockchain data storage and reducing the frequency of on-chain data interactions, while also re-concealing disclosed user data to enhance system security.

In blockchain systems, authenticated storage usually adopts the Merkle Patricia Trie (MPT) structure[24], which is a specific form of the Merkle tree. Each leaf node in MPT stores a value, and the path from the root node to the leaf node corresponds to the key of the stored value. Each internal node stores the cryptographic hash values of all its children. The root hash of the MPT serves as a commitment to the blockchain state. However, this kind of authentication comes with a heavy performance cost. Modifying key-value pairs in the state requires updating the hash values of all nodes along the path from the corresponding leaf node to the root. Without caching, each state update operation expands to O(log(n)) I/O operations, where n is the storage size. Even a simple payment transaction involves at least two ledger state updates: deducting and increasing the balance of the sender and the receiver respectively. To improve the MPT structure, mLSM proposes a multi-layer MPT[25]. The most recent updates are stored at the lowest layer, and key-value pairs at lower layers are periodically merged into higher layers. LMPT maintains three MPTs: one large MPT containing old states and two small MPTs containing recent state changes[26]. LMPT periodically merges the small MPTs into the large one. For both mLSM and LMPT, the concatenation of the Merkle roots of all MPTs forms the commitment to the ledger state. These techniques reduce disk I/O operations on the critical path because the most recently accessed states are stored in MPTs with smaller heights, and MPT merging can be performed in background threads. However, the mLSM paper only presents a conceptual design without implementation or evaluation[25].

In addition, RainBlock, an authenticated storage model based on pre-stored nodes, introduces three types of nodes in blockchain systems to accelerate transaction execution: storage prefetchers, miners who execute transactions, and storage nodes[27]. During transaction execution, miners obtain the required data from multiple prefetchers and send updates to multiple storage nodes, each maintaining an MPT shard in memory. RainBlock replaces local storage I/O with network-based distributed storage I/O. To reduce network read latency, RainBlock introduces an I/O prefetcher and requires miners to attach all accessed key-value pairs and witnesses (MPT nodes) when broadcasting a block. RainBlock reports an average proof size of 4 KB per transaction and achieves a 95% reduction through optimization, resulting in an additional network storage cost of about 200 B per transaction, roughly twice the size of a normal transaction. However, inefficient network usage creates bottlenecks in high-performance blockchain systems[25]. Moreover, RainBlock is vulnerable to data availability attacks. Due to the high cost of memory storage, the number of replicas in RainBlock is much smaller than in Ethereum.

However, all these schemes are built upon mathematical hard problems such as the discrete logarithm and integer factorization. With the rapid advancement of quantum computing, these problems may become solvable, causing existing authenticated storage mechanisms to face significant security risks that threaten data integrity and authenticity. Therefore, it is urgent to develop blockchain authenticated storage schemes capable of resisting quantum computing attacks to ensure system security and robustness.

1.2 Quantum-resistant signature

This section mainly discusses the hash-based digital signature algorithm SPHINCS, which is designed to maintain security in the post-quantum era. With the development of cryptography, SPHINCS has undergone several iterations, including SPHINCS$^+$ and the recently proposed SPHINCS-α.

SPHINCS was first proposed at EUROCRYPT 2015[29]. It uses the quantum-resistant properties of hash functions to construct a stateless digital signature scheme. This stateless design means that the signer does not need to maintain any state information about previous signatures, which is very useful in some application scenarios. SPHINCS$^+$[19] is an improved version of SPHINCS that preserves the security properties of the original design while improving efficiency and robustness by introducing new techniques and methods. SPHINCS$^+$ defines three different signature schemes: SPHINCS$^+$-SHAKE256, SPHINCS$^+$-SHA-256, and SPHINCS$^+$-Haraka, each implemented with a different hash function. The goal of SPHINCS$^+$ is to reduce signature size while maintaining high security. In its second-round submission, SPHINCS$^+$ introduced two variants: a simple variant and a robust variant. The simple variant, instantiated under the pure random oracle model, achieves about three times faster performance than the robust variant but with a potential security trade-off. SPHINCS-α[21] is a further optimized signature scheme based on SPHINCS$^+$. It improves signature efficiency by enhancing the Winternitz one-time signature scheme and introducing a new few-time signature scheme FORC. Compared with SPHINCS$^+$, SPHINCS-α reduces both signature size and signing time, especially under the parameter settings aiming for smaller signatures, achieving around 8% reduction. Moreover, SPHINCS-α demonstrates better performance under fast signing parameter configurations.

2 Basic Knowledge

The method proposed in this paper mainly combines blockchain authenticated storage with hash-based signatures (including balanced WOTS+, FORC, and the hypertree). The related concepts and basic knowledge are introduced below.

2.1 Authenticated storage in blockchains

In standard public blockchain systems, nodes are divided into two types: full nodes and light nodes. Full nodes synchronize and execute all transactions and maintain the blockchain ledger state accordingly. A light node (client) only synchronizes block headers and does not synchronize transactions or the ledger state. When a new block is proposed by a full node, it executes all transactions in the block and places a commitment to the resulting ledger state in the block header. Therefore, full nodes maintain a write-back cache during transaction execution and flush storage after all transactions in the block are executed. Accordingly, authenticated storage needs to provide two algorithms:

- *Get*$(k) \to v$: get the value v for the given key k.
- *Set*$(\{(k, v)_i\}, e) \to$ *comm*: flush the key-value pairs (k, v) into storage at block number e and obtain the updated ledger-state commitment *comm*.

When a light node wants the value corresponding to a key, it queries a full node, which returns the ledger commitment and a proof associated with that value. The light node then checks whether the commitment is valid and verifies the correctness of the proof. Therefore, authenticated storage needs to provide two algorithms for proving and verifying:

- *Respond*$(k) \to (v, \pi, \textit{comm})$: given the latest commitment, answer with the value v for key k and a proof π.
- *Verify*$(k, v, \pi, \textit{comm}) \to 0/1$: verify the response from the full node.

2.2 Hash functions

Tweakable hash function: A tweakable hash function takes the public parameter P, the tweak T, and the message m as inputs. The public parameter P can be regarded as keying material, and the tweak T can be regarded as a salt or a random number. Based on SPHINCS$^+$[19], we adopt the subsequent simplified representation in this work. For a tweakable hash function with an input length of $k\lambda$, it is $\textit{Th}_k\colon \{0,1\}^\lambda \times \{0,1\}^{256} \times \{0,1\}^{k\lambda} \to \{0,1\}^\lambda$. The extended hash function is defined as $F \triangleq \textit{Th}_1$, $H \triangleq \textit{Th}_2$.

Pseudo-random functions and message digest: In this paper, the pseudo-random function *PRF* is used to generate a pseudo-random key, which is defined as $\textit{PRF}\colon \{0,1\}^\lambda \times \{0,1\}^{256} \to \{0,1\}^\lambda$, while another pseudo-random function $\textit{PRF}_{\text{msg}}$ is used to generate the randomness for message compression, which is defined as $\textit{PRF}_{\text{msg}}\colon \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$. In addition, we use a keyed hash function H_{msg} that can handle messages of arbitrary length to compress the message to be signed, which is defined as $H_{\text{msg}}\colon \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$.

2.3 Balanced WOTS+

The WOTS+ scheme introduces a checksum in the encoding process to prevent attackers from effectively forging signatures given valid message-signature pairs. Given (σ, m), an attacker can forge any signature m' satisfying $m_i \leq m'_i$ for each i by computing $H^{m'_i}(sk_i) = F^{m'_i - m_i}(H^{m_i}(sk_i))$, where H is the hash function. The checksum solves this problem: any change in m_i will result in a change in the checksum C. Therefore, the adversary cannot forge (m', C').

Size-optimal encoding[21]: In WOTS+, the encoding function appends the checksum to the end of the original message. The message size l is fixed, and the total encoded length is minimized. However, the WOTS+ encoding method has a problem of low message-space utilization because the checksum occupies part of the encoding space. To maximize message-space utilization, balanced WOTS+ first fixes the encoding size l. To accommodate the largest possible message space, this encoding method ensures that the sum of all codewords in the encoding process is constant, thus eliminating the need for a checksum. To realize balanced WOTS+, let

$$D_{n,m} = \left| \{v \in [w]^n : \sum_{i=1}^{n} v_i = m\} \right| \tag{1}$$

where $D_{n,m}$ represents the number of possible codewords whose sum of the first n elements equals m. The initial value is defined as

$$\begin{cases} D_{1,m} = 1,\ m \in \{0, 1, \ldots, w-1\} \\ D_{n,m} = 0,\ 2 \leq n \in \mathbb{Z},\ m \in \mathbb{Z}^- \end{cases} \tag{2}$$

and the recursive relation is

$$D_{n,m} = \sum_{i=0}^{w-1} D_{n-1,m-i},\ 2 \leq n \in \mathbb{Z},\ m \in \{0, 1, \ldots, l(w-1)\} \tag{3}$$

Next, the recursive relation in Equation (3) is explained. To compute $D_{n,m}$, first consider the last summand, which can take any value in $\{0, 1, \ldots, w-1\}$. Assume this value is i, then the sum of the previous $n-1$ elements must be $m-i$. Therefore, the problem "the sum of n elements equals m" can be reduced to "the sum of $n-1$ elements equals $m-i$". By this deduction, $D_{n,m}$ can be computed by simply summing up $D_{n-1,m-i}$. According to this method, the specific balanced WOTS+ signature algorithm is as follows.

In Algorithm 1, the private key *sk* consists of l random data blocks, namely $sk = (sk_1, sk_2, \ldots, sk_l)$, $sk_i \in \{0,1\}^\lambda$. The public key $\{pk_i\}_{i=1}^{l}$ is generated by iteratively applying the hash function to each block in the private key $w-1$ times, i.e., $pk_i = F^{w-1}(sk_i)$. Finally, these blocks are merged into the public key *pk* through the tweakable hash function Th_i (see Section 2.2 for details), that is $pk = Th_l(pk_1, pk_2, \ldots, pk_l)$. For a signature consisting of l blocks SIG_{bw}, the verifier applies the hash function to each block $w-1-d[i]$ times, that is $pk'_i = F^{w-1-d[i]}(SIG_{bw}[i])$. Finally, the verifier uses the tweakable hash function Th_i again to merge these blocks into pk', that is $pk' = Th_l(pk'_1, pk'_2, \ldots, pk'_l)$. If $pk = pk'$, the verifier accepts the signature.

2.4 Hypertree

Before introducing the hypertree, the Merkle tree is first described. To sign $2^{h'}$ messages, the signer generates $2^{h'}$ WOTS+ key pairs and uses these $2^{h'}$ public keys as the leaf nodes to build a binary tree. Specifically, for each pair of leaf nodes, the signer repeatedly applies the hash function to generate the corresponding parent node until reaching the root node of the tree, which serves as the public key of the tree. When signing, the signer selects a WOTS+ leaf node and publishes the WOTS+ signature together with the sibling nodes of all nodes along the path from the leaf node to the root node, which constitutes an authentication path (see Figure 1 for details). The verifier first obtains the WOTS+ public key from the signature and then reconstructs the root node using the nodes on the authentication path and their siblings.

Algorithm 1. *bw_sig*.

Input: l, w;
Output: SIG_{bw}.

Let d be an array of size l;
$m = \lfloor l(w-1)/2 \rfloor$;
for $i = l$ **to** 1 **do**
    for $j = 0$ **to** $\min\{w-1, m\}$ **do**
        if $x \geq D_{i-1,s-j}$ **then**
            $x = x - D_{i-1,s-j}$;
        else
            $d[i] = j$;
            break;
        end if
    end for
    $m = m - d[i]$;
end for
$SIG_{bw}[i] = F^{d[i]}(sk_i)$;
return SIG_{bw};

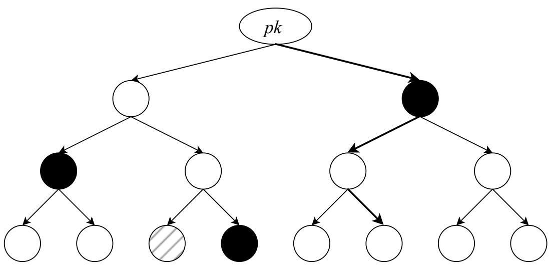


Figure 1 Authentication path (the third leaf node is shown in black)

Note that to generate a root node that can verify $2^{h'}$ messages, all leaf nodes need to be hashed. Signing all messages in a single Merkle tree is infeasible in practice. Therefore, this problem is solved by using a hypertree, as shown in Figure 2. It can be used as an authentication tree, consisting of d layers, where the leaf nodes at the bottom layer are dedicated to signing messages (in this paper, the signed message is the DFORC public key). In contrast, the leaf nodes at higher layers are used to sign the root nodes of the lower-level subtrees. From the lowest leaf node of the hypertree to the topmost root node, the WOTS+ signatures and authentication paths together form a complete authentication path. Importantly, all the leaf nodes of each intermediate tree are deterministically generated WOTS+ public keys, independent of any trees below. This means that the hypertree exists virtually, and the complete hypertree does not need to be generated during computation. In the key generation process, only the topmost subtree is computed to generate the public key. The total height of the hypertree is denoted as h, the number of intermediate layers as d, and $h' = h/d$.

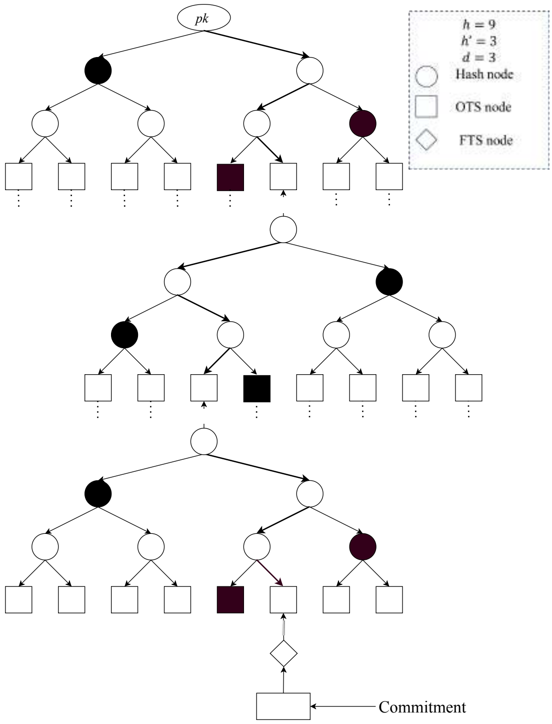


Figure 2 Hypertree structure ($h = 9$, $h' = 3$, $d = 3$)

2.5 FORC

FORC (forest of random chains) is a variant of few-time signature schemes. The Winternitz method can use a hash function to encode multi-bit messages into a single block. Based on this idea, we integrate the hash chain into the leaf nodes of the FORS signature[29]. The parameter w denotes the Winternitz parameter, which determines the length of the hash chain. The structure shown in Figure 3 is composed of parameters k, a, and w'.

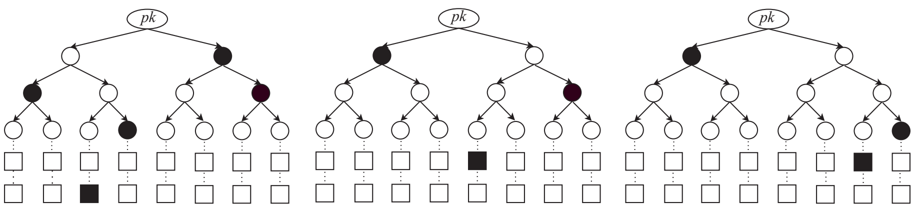


Figure 3 Illustration of a FORC signature for message 010101100110011 ($k = 3$, $a = 3$, and $w' = 2$)

- Key generation: the public key is generated from the roots of k trees using the hash function Th_k, with each tree containing $t = 2^a$ chains attached to its leaf nodes. First, t blocks are randomly generated, and then the hash function F (as defined in Section 2.2) is iteratively applied w' times to each block to construct the root nodes. Finally, the endpoint blocks are treated as leaf nodes and are hashed upward step by step to the root nodes of the Merkle trees.
- Signature and verification: given a message of $k(a + \log w')$ bits, it is divided into k blocks, each consisting of $(a + \log w')$ bits. Each block indicates the index of a leaf node and its position within the corresponding Merkle tree. The signature consists of the

selected nodes and their respective authentication paths (see Figure 1 for details). The verifier uses the authentication paths to reconstruct each root node and then uses Th_k to rebuild the public key.

3 Overview of the EQAS Scheme

This section introduces EQAS from three aspects: system model, workflow, and threat model. The system model describes the functions of light nodes and full nodes. The workflow explains the complete process of authenticated storage through examples. The threat model defines the behavior and potential attack methods of each entity. For clarity, the meanings of related symbols are given in Table 1.

Table 1 Symbols

Symbol	Meaning
n	Security parameter in bytes
h	Height of the hypertree
k	Number of trees in DFORC
w'	Number of nodes in a DFORC hash chain
w	Winternitz parameter
d	Number of layers in the hypertree
t	Number of leaf nodes in a DFORC tree

3.1 System model

The architecture of EQAS mainly consists of two entities: light nodes and full nodes.

(1) Light nodes: light nodes are responsible for verifying the correctness of transactions and states on the blockchain. They only store block headers, which greatly reduces the demand for storage space and computational resources. To verify data authenticity, a light node requests transaction proofs and state commitments from a full node and independently verifies the returned results. This verification process reduces the storage burden on light nodes, enables them to participate efficiently in the system, and improves overall scalability.

(2) Full nodes: full nodes store the complete blockchain data, handle the verification and recording of all transactions, generate and maintain the latest blockchain state, and provide data verification services for light nodes. When a light node requests a transaction proof or a state commitment for a block header, the full node returns the corresponding proof data. Full nodes ensure data integrity and consistency by generating transaction proofs, Merkle root hashes, and related commitments.

Between these entities, after a light node sends a request, the full node generates the proof and sends it back to the light node. The light node then independently verifies the proof and updates its local state information.

3.2 Workflow

In the complete authenticated storage workflow of the blockchain, a user first initiates and signs a transaction, then broadcasts it to the network. Full nodes receive and verify the transaction and temporarily store valid transactions in the transaction pool in memory. Subsequently, a full node creates a new block that packages these transactions, generates the block hash, and submits the new block through the consensus mechanism. After the new block is verified and accepted by other full nodes in the network, the block's transaction data and state changes are written into the blockchain database, and the current ledger-state commitment (such as the Merkle root hash) is generated to ensure data immutability and verifiability.

At this stage, the light node requests a proof to verify a transaction or a state. The specific interaction process is as follows:

(1) Light node sends a request. The light node does not store the full blockchain data. When it needs to verify a transaction or a state, it sends a data request (such as a transaction existence proof or the latest state commitment) to the full node, including the transaction identifier or the hash of the state.

(2) Full node generates the proof. Upon receiving the request, the full node generates the corresponding transaction proof or state commitment proof based on the light node's request. This process includes retrieving the corresponding transaction or state data from the blockchain, constructing the Merkle-tree verification path, and providing the latest block-header information as part of the proof.

(3) Full node returns the proof. The full node returns the generated proof data to the light node. These proofs include: the existence proof of a transaction (the Merkle-tree verification path), which allows the light node to verify whether the transaction truly exists in a specific block; the state commitment proof, in which the full node returns the latest state commitment information to prove that the state has not been tampered with since the last submission.

(4) Light node verifies the proof. After receiving the proof, the light node uses the latest block-header information returned by the full node to verify the integrity of the transaction or state. By verifying the Merkle path or state commitment, the light node can independently confirm whether the requested data is authentic and reliable, then update its local blockchain state.

(5) Verification results. After verification, if the proof is valid, the light node updates its local state and regards the transaction or state as valid. If the verification fails, the light node discards the returned result and may attempt to obtain a correct proof from other full nodes.

Through this interaction mechanism, light nodes can ensure the authenticity and integrity of transactions and states without downloading the entire blockchain data.

3.3 Threat model

In constructing the threat model for a quantum-resistant blockchain authenticated storage system, we assume that light nodes are honest verifiers and specifically consider full nodes as potential attackers. This assumption is based on the following considerations:

(1) Differences in resource capacity between light nodes and full nodes. Light nodes are typically resource-constrained devices. They do not store complete blockchain data but only keep block headers and rely on full nodes to provide transaction proofs and state commitments. Because the main task of light nodes is to verify the proofs provided by full nodes, and they lack the ability to tamper with transactions or generate forged data, their only goal is to ensure the authenticity and integrity of the received proof data. Therefore, light nodes are considered honest verifiers.

(2) The key role of full nodes in the network. Full nodes store all blockchain transaction data and state information, and are responsible for generating and maintaining them. In this process, full nodes can influence the ledger state and provide data services to light nodes. This role makes full nodes potential attack targets. For example, a dishonest full node may forge transaction proofs, tamper with state data, or return incorrect information to light nodes through quantum attacks, thus compromising system integrity and security. Therefore, full nodes are modeled as potential attackers.

Because full nodes may possess quantum computing capabilities, they can directly threaten blockchain authenticated storage. A quantum adversary may exploit quantum algorithms to undermine blockchain security. Based on the above analysis, a full node adversary may employ the following attack methods:

(1) Shor's algorithm[8, 13]: this algorithm efficiently factors large integers, thus breaking RSA and other public-key cryptosystems based on integer factorization. An attacker could use this algorithm to forge digital signatures generated by full nodes and insert malicious data, making it difficult for light nodes to detect tampering.

(2) Grover's algorithm[10]: Grover's algorithm accelerates hash-collision searches and weakens the security of hash-based authenticated storage mechanisms. Such quantum-accelerated attacks may cause hash commitment mechanisms to fail, thus affecting the verification results of light nodes.

(3) Data forgery and tampering: as attackers, untrusted full nodes can forge signatures, fabricate authentication information, or insert malicious data, making the system unable to detect tampering and ultimately compromising the integrity and authenticity of the blockchain.

Following the security definition in Ref. [19], we formally define the quantum-resistant security of this paper as follows:

Definition 1 (PQ-EU-CMA). Let $\mathit{SIG} = (kg, sign, vf)$ be a digital signature scheme. The success probability of a quantum adversary A is defined as

$$\begin{aligned} &\mathit{Succ}_{\text{SIGN}}^{\text{EU-CMA}}(\mathcal{A}) = \Pr[vf(pk, M^*, \sigma^*) \\ &= 1 \wedge M^* \notin \{M_i\}_{i=1}^{q_s} | (sk, pk) \leftarrow kg(); (M^*, \sigma^*) \leftarrow \mathcal{A}^{sign(sk;)}] \end{aligned} \tag{4}$$

where $M_1, \dots, M_{q_s}$ are messages submitted by A to the signing oracle. The PQ-EU-CMA insecurity of *SIGN* is defined as the maximum success probability among all quantum adversaries with computation time ξ and query complexity q_s

$$\mathit{InSec}^{\text{PQ-EU-CMA}}(\mathit{SIGN}, \xi, q_s) = \max_{\mathcal{A}} \left\{ \mathit{Succ}_{\text{SIGN}}^{\text{EU-CMA}}(\mathcal{A}) \right\} \tag{5}$$

4 Quantum-resistant and Efficient Blockchain Authenticated Storage Scheme — EQAS

In this section, EQAS uses a key-value pair database as the backend and maintains a key-value mapping tuple (*KV, VM, ID*), where *KV* stores key-value pairs, *VM* stores version numbers of the keys, and *ID* stores the index corresponding to each key. When the blockchain processes a block, the block contains key-value pairs, each with a version number starting from 0. Next, the variant DFORC of FORC is introduced, followed by the design details of EQAS. The EQAS framework is shown in Figure 4.

4.1 DFORC

The typical FORC structure is static. Once the FORC is constructed, the private key is not updated until the signature limit is reached, and the FORC structure remains unchanged. In addition, because the FORC structure depends on the construction of the private key, if the private key remains the same, the structure also stays unchanged. However, in a public blockchain, such a static FORC structure cannot be used to generate commitments, since commitments change as key-value pairs are updated. Therefore, dynamically generating FORC commitments becomes a challenge. To address this issue, we propose DFORC in this work. In DFORC, only the nodes on the tree are updated, while the private key stored in the hash chain remains unchanged. When a new key-value pair selects the same index on the same tree, the leaf node corresponding to that index is processed and hashed. The number of hash iterations depends on how many times that index has been selected. Then, the nodes on the path from the leaf node to the root node are updated step by step, eventually producing a new root node, which serves as the commitment.

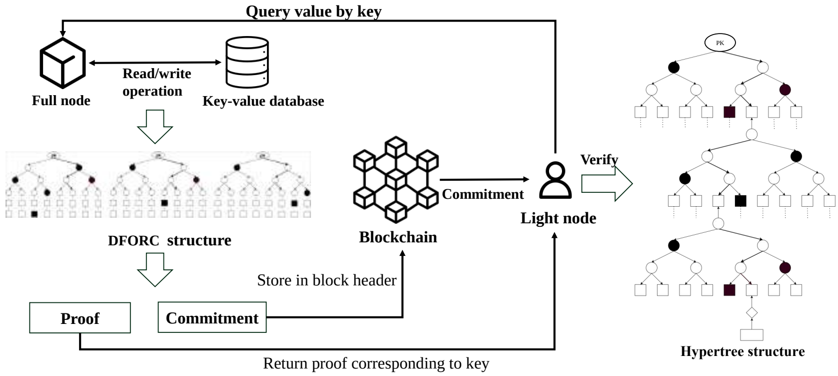


Figure 4 EQAS framework

In traditional blockchain authenticated storage, MPT is a commonly used data structure for generating transaction existence proofs and state commitments. However, a key bottleneck of the MPT structure is that each time the state is updated, not only the hash values of the involved path nodes should be updated, but also a large number of disk I/O operations are required. This is because the data storage of MPT usually involves disk read and write operations, and updating each node along the path results in frequent I/O access. The DFORC proposed in this paper also needs to update the nodes along the path, but unlike the traditional MPT structure, DFORC is designed to avoid additional disk read/write operations. DFORC optimizes the state storage structure into multiple independent hash chains that together form a random forest structure, allowing each state update to be limited to the relevant hash chain without involving nodes on other unrelated chains. Although this design still requires updates to the nodes along the hash path, DFORC greatly reduces the frequency of disk reads and writes through caching mechanisms or in-memory node management. Specifically, when a state update occurs, DFORC only needs to update the hash chain involved in the current state change, without frequently accessing the disk. By avoiding extensive read and write operations on unrelated nodes, DFORC effectively mitigates the disk I/O bottleneck. This not only significantly improves update efficiency but also reduces the storage pressure on full nodes.

4.2 Initialization

First, compute the randomizer R, the message digest *md'*, and its corresponding index in DFORC, so that they can be used to generate commitments and proofs later. See Algorithm 2 for the specific process. To generate EQAS, the random number generator is called three times to produce *SK.seed*, *PK.seed*, and *SK.prf* (line 1 of Algorithm 2). Then, R is generated based on *key* and *SK.prf*, and defined as

$$R = \mathit{PRF}(\mathit{SK.prf}, \mathit{Optrand.key}) \tag{6}$$

where *Optrand* is an n-byte string, initialized to 0, and can be optionally set by randomly overwriting with 1 s. By adding extra randomness, R becomes non-deterministic, which helps to resist side-channel attacks. Then, we derive the index *idx* used, and *md*:

$$(md\|idx) = H_{\text{msg}}(R, \mathit{PK.seed}, \mathit{PK.root}, \mathit{key}) \tag{7}$$

where H_{msg} from Section 2.2 is used, *PK.root* comes from the top root node of the hypertree (see lines 1 to 14 in the *hit_sig* Algorithm), and md has a length of $m = \lfloor(ka+7)\rfloor + \lfloor(h-h/d+7)\rfloor + \lfloor(h/d+7)\rfloor$ bits. According to the selected parameters (lines 5 to 7 of Algorithm 2), $(md||idx)$ is divided into md', the tree address *idx_tree*, and leaf index *idx_leaf*, where m_1 bytes are needed for the message digest md', m_2 bytes for the tree address, and m_3 bytes for the leaf index, so that $m = m_1 + m_2 + m_3$. Then, md' is split into k characters of $\log(t)$ bits to generate the index array *idx*$[k]$, where k is the number of DFORC trees and t is the number of leaf nodes. For example, *idx*$[k] = [2, 5, 7, \ldots]$. Each of these k indices corresponds to each *sk* in DFORC. Specifically, the first index selects the second *sk* in the first tree; the second selects the fifth *sk* in the second tree; and so on, the last index selects the *idx*$[k-1]$-th *sk* in the last tree. Subsequently, the index corresponding to the *key* is added to *ID*.

Algorithm 2. Initialization.

Input: *key*;
Output: md', *idx_tree*, *idx_leaf*.
1. *SK.seed*, *SK.prf*, *PK.seed* $\leftarrow$ *Random*(n);
2. *PK.root* $\leftarrow$ *ht_sig*() (See lines 1 to 14 in Algorithm 7);
3. Generate R according to Eq. (6);
4. Generate $(md||idx)$ according to Eq. (7);
5. md' is front $\lfloor ka+7 \rfloor$ bits in $(md||idx)$;
6. *idx_tree* is following $\lfloor(h-h/d+7)\rfloor$ bits in $(md||idx)$;
7. *idx_leaf* is back $\lfloor h/d+7 \rfloor$ bits in $(md||idx)$;
8. **return** md', *idx_tree*, *idx_leaf*.

4.3 Commitment and proof

4.3.1 *ADRS scheme*

The ADRS scheme adopts a set of address operations, and all the specific operation methods are presented in Ref. [30]. All nodes update their addresses after each hash call. For different use cases, there are five types of addresses: the first type is used for the hash function in the balanced WOTS+ (BW) scheme, the second type is used to compress the BW public key, the third type is used to hash the Merkle tree in the hypertree, the fourth type is used to hash each tree in the DFORC, and the last type is used to compress the root node of the DFORC. As shown in Figure 5, the layer address indicates the height of the node in the hypertree, and for the bottom tree, the layer address is 0. The tree address indicates the position of the node in the hypertree starting from the leftmost tree index 0. *Type* indicates the type of address: BW hash address is 0, BW public key compression is 1, hash tree address is 2, DFORC address is 3, and DFORC root node compression is 4. The hash address represents the address of the key-value pair. The tree height indicates the height of the node in the DFORC. The tree index is calculated in Section 4.3.2.

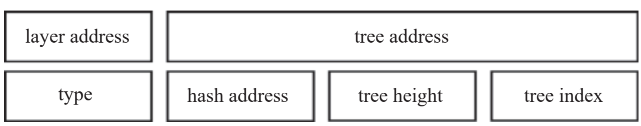


Figure 5 Address structure

To manage types in the pseudocode provided in the rest of this paper, we assign specific constants for easy reference. These constants are: *BW_hash*, *BW_pk*, *DFORC_tree*, and *DFORC_roots*.

4.3.2 *Generate commitment*

dforc_Keygen$(PK.seed, SK.seed, ADRS, idx, s) \rightarrow dforc_{sk}, dforc_{pk}$: This algorithm generates the DFORC public/private key pair. With *PK.seed*, *SK.seed*, *ADRS*, *idx*, and the start index s as inputs, and combined with the tree index and height, the nodes in the tree are generated in turn. Specifically, the tree height and index are set via *ADRS.setTreeHeight*() and *ADRS.setTreeIdx*(). Then, the private key is generated by Eq. (9). Next, by traversing each tree, nodes are generated layer by layer. During this process, the ADRS scheme controls the tree index and height. The nodes at each layer of the tree are generated iteratively, and finally the root node is computed and the key is returned. The complete process is shown in Algorithm 3. Th_k and H are described in Section 2.2. ADRS operations follow Ref. [30].

Algorithm 3. *dforc_Keygen*.

Input: *PK.seed*, *SK.seed*, *ADRS*, *idx_tree*, s;
Output: $dforc_{sk}, dforc_{pk}$.

1. $ADRS.setTreeHeight(0); ADRS.setTreeIdx(idx\_tree)$;
2. Generate the private key *dforc_sk* according to Eq. (9);
3. **for** $i = 0$ **to** $k - 1$ **do**
4.   **for** $q = 0$ **to** $a - 1$ **do**
5.   **for** $j = 0$ **to** w' **do**
6. $leaf[q][j] = F^j(PK.seed, ADRS, sk)$;
7.   **end for**
8. $node[i][q] = leaf[w']$;
9.   **end for**
10. $ADRS.setTreeHeight(1)$;
11. $ADRS.setTreeIdx(s + i)$;
12.   **while** $(Stack.top.height == node.height)$ **do**
13. $ADRS.setTreeIdx((ADRS.getTreeidx() - 1)/2)$;
14. $node = H(PK.seed, ADRS, (Stack.pop() \| node))$;
15. $ADRS.setTreeHeight = (ADRS.getTreeHeight() + 1)$;
16.   **end while**
17. $Stack.push(node)$;
18. $root[i] = Stack.pop()$;
19. **end for**
20. $ADRS.setType(DFORC\_roots)$;
21. $dforc_{pk} = Th_k(PK.seed, ADRS, root[0] \| \ldots \| root[k-1])$;
22. **return** $dforc_{sk}, dforc_{pk}$.

In *ADRS*, the tree index part is computed as

$$tree\_index = i \times t + idx[i] \tag{8}$$

and the private key is generated from *SK.seed* and *ADRS* via *PRF*:

$$dforc_{sk} = PRF(SK.seed, ADRS) \tag{9}$$

Then, we iteratively apply F to each private key w' times and add the result of each hash to $leaf[q][w'+1] = [F^2, F^1, sk[q]]$, which represents the nodes on the hash chain of the q-th leaf node, as described in Section 2.2. Each hash value serves as a node on the hash chain, and the final hashed node is used as the leaf node. Starting from the leaf node, the nodes at each layer are then generated according to lines 3–18 in Algorithm 3 until the root node is obtained.

4.3.3 *Generation proof*

dforc_sig$(PK.seed, ADRS, SK.seed, idx, md') \rightarrow SIG_{dforc}$: The full node generates the corresponding proof via this algorithm. The inputs are *md'*, *SK.seed*, *PK.seed*, *ADRS*, and *idx*. The output is the DFORC signature SIG_{dforc}, i.e., the proof. This proof is a string array of length

$k(\log(t)+1)$, where k is the number of Merkle trees and t is the number of leaf nodes, and it contains the nodes on the authentication path from the leaf to the root. The elements of each authentication path consist of n-byte private keys and the corresponding nodes in the tree. The authentication path is shown in Figure 1. During the actual generation process, each node in the authentication path is computed using *SK.seed* and *ADRS* through the pseudo-random function *PRF*. For each *key*, the algorithm generates the corresponding authentication path by traversing the Merkle tree and combines these paths with the signature to produce the complete proof. For the specific process, see Algorithm 4. The related operations and some functions (e.g., ($SIG_{XMSS}.getAUTH$)) are described in Ref. [30].

Algorithm 4. *dforc_sig*.

Input: *PK.seed*, *SK.seed*, *ADRS*, *idx*, md';
Output: SIG_{dforc}.
for $i = 0$ **to** $k-1$ **do**
  $ADRS.setTreeHeight(0)$;
  $ADRS.setTreeIdx(i \times t + idx[i])$;
  $SIG_{dforc}[i] = PRF(SK.seed, ADRS)$;
  for $j = 0$ **to** $a-1$ **do**
    $s = (idx/2^j) \oplus 1$;
    $AUTH[j] = SIG_{XMSS}.getAUTH(j)$;
  end for
  $SIG_{dforc} = SIG_{dforc}[i] || AUTH[j]$;
end for
return SIG_{dforc};

4.4 Hypertree structure

In this section, we elaborate on the construction process of the hypertree. Since all the leaf nodes in the hypertree are balanced WOTS+ (BW) public keys, BW is introduced first, and then the hypertree is constructed based on BW.

4.4.1 *BW signature*

First, it is necessary to call F $w \times l$ times (see Section 2.2) to generate the corresponding bw_{pk}. When computing $F^w(sk_i)$, the address of the private key is changed in each iteration. During each iteration, *ADRS* and *PK.seed* are used together to compute the new hash value. The updated address corresponds to the current node's position in the hash chain and is used as input for the next step. This design ensures the unpredictability of each hash step and enhances the scheme's quantum resistance. This process is described in detail in Algorithm 5. By updating the hash and the private-key address, the hash value of each iteration in the hash chain is output (with i denoting the start index). ADRS operations follow Ref. [30].

$bw\_Keygen(PK.seed, ADRS, SK.seed) \rightarrow (bw_{pk}, bw_{sk})$: The full node uses this algorithm to generate BW public/private key pairs. Each n-byte string in the private key is generated from *SK.seed* and the address *ADRS*. The corresponding public key is generated by hashing the private key w times. Specifically, each private key is hashed with the *ADRS* address to generate the corresponding public key. During this process, the private-key address is updated according to the hash-chain structure, and each iteration result is computed via the chain function (Algorithm 5). Algorithm 6 shows the steps for generating BW key pairs. ADRS operations follow Ref. [30]. Using $ADRS.setChainAddr()$ and $ADRS.setHashAddr()$, the corresponding addresses are set, and hashing is performed for each private key to generate l BW key pairs. Then, bw_{pk} is computed as follows:

$$bw_{pk} = Th_l(PK.seed, ADRS, bw_{pk}[0]||\ldots||bw_{pk}[l-1]) \tag{10}$$

Algorithm 5. *chain*.

Input: *PK.seed*, *ADRS*, r, s, i;
Output: Hash value of r after s iterations.
if $s = 0$ **then**
    return r;
end if
if $(i + s) > w - 1$ **then**
    return NULL;
end if
$tmp = chain(PK.seed, ADRS, r, s - 1, i)$;
$ADRS.setHashAddr(i + s - 1)$;
$tmp = F(PK.seed, ADRS, tmp)$;
return *tmp*;

Algorithm 6. *bw_Keygen*.

Input: *PK.seed*, *SK.seed*, *ADRS*;
Output: bw_{sk}, bw_{pk}.
for $i = 0$ **to** l **do**
    $ADRS.setChainAddr(i)$;
    $ADRS.setHashAddr(0)$;
    $bw_{sk}[i] = PRF(SK.seed, ADRS)$;
end for
for $j = 0$ **to** l **do**
    $ADRS.setChainAddr(j)$;
    $ADRS.setHashAddr(0)$;
    $tmp[j] = chain(bw_{sk}[j], 0, w - 1, PK.seed, ADRS)$;
end for
Generate bw_{pk} according to Eq. (10);
return bw_{sk}, bw_{pk};

$bw\_sig(PK.seed, ADRS, SK.seed) \rightarrow SIG_{bw}$: According to the size-optimal encoding (see Section 2.3), we use the BW private key to generate SIG_{bw}. See Algorithm 1 for the specific process (see Section 2.3 for details).

4.4.2 *Hypertree signature*

This section explains how BW is combined with the Merkle tree to generate the hypertree (HT), thus generating a signature scheme with fixed input length. The public key of HT is located at the root node at the top level. The generation process of the hypertree signature is explained below.

$ht\_sig(PK.seed, SK.seed, ADRS, idx\_tree, idx\_leaf) \rightarrow SIG_{ht}$: Light nodes use this algorithm to generate HT signatures. In the *ht_sig* algorithm, the inputs include *SK.seed*, *PK.seed*, *idx,* and *ADRS*, where *idx* selects the hypertree leaf that signs the message. The algorithm first generates the authentication path $AUTH[j]$ by setting relevant parameters, then uses the hash function H (see Section 2.2) to compute node hashes from the bottom leaves upward, and finally outputs the signature SIG_{ht}. Note that *idx* is passed as two separate parameters: one handles the tree index, and the other represents the leaf index in that tree. Specifically, the algorithm calls the index addresses of tree nodes via *ADRS.setTreeIdx* and generates the authentication-path nodes layer by layer. According to adjacent nodes in the authentication path (obtained via $AUTH[j]$), nodes are computed recursively and finally merged to obtain the root *root*. The detailed process is shown in Algorithm 7, where h' denotes the height of the Merkle tree in HT. For ADRS operations and functions such as SIG_{XMSS} and *ADRS.set*, please refer to Ref. [30].

Algorithm 7. *ht_sig*.

Input: *PK.seed*, *SK.seed*, *ADRS*, *idx_tree*, *idx_leaf*;
Output: SIG_{ht}, *root*.
$ADRS.setLayerAddr(0)$;
$ADRS.setTreeAddr(idx\_tree)$;
for $j = 0$ **to** h' **do**
  $AUTH[j] = SIG_{XMSS}.getAUTH(j)$;
  $ADRS.setTreeHeight(j+1)$;
  if $(idx/2^j)\%2 == 0$ **then**
    $ADRS.setTreeIdx(ADRS.getTreeIdx()/2)$;
    $node[1] = H(PK.seed, ADRS, (node[0] \| AUTH[j]))$;
  else
    $ADRS.setTreeIdx(ADRS.getTreeIdx() - 1)/2)$;
    $node[1] = H(PK.seed, ADRS, (AUTH[j] \| node[0]))$;
  end if
end for
$root = node[0]$;
for $j = 1$ **to** $d - 1$ **do**
  front $\lfloor h/d \rfloor$ bits in $idx\_leaf = idx\_tree$;
  back $\lfloor h - (j+1) \times (h/d) \rfloor$ bits in $idx\_tree = idx\_tree$;
  $ADRS.setLayerAddr(j)$;
  $ADRS.setTreeAddr(idx\_tree)$;
  $ADRS.setType(BW\_hash)$;
  $ADRS.setKeypairAddr(idx)$;
  Generate SIG_{bw} according to Algorithm 1;
  $tmp = SIG_{bw}||AUTH$; $SIG_{ht}[] =$ NULL;
  $SIG_{ht} = SIG_{ht}||tmp$;
end for
return SIG_{ht}, *root*;

4.5 Authentication

In this section, we verify the validity of the commitment and the correctness of the proof. Only when both are successfully verified, the corresponding *key* is authenticated. The detailed authentication process is as follows.

$proof\_ver(SIG_{dforc}, PK.seed, ADRS, md', dforc_{pk}) \rightarrow$ (True/False): The algorithm is used to verify the signature provided by *dforc_sig*. The input includes SIG_{dforc}, the public key seed *PK.seed*, the private key seed *SK.seed*, the address *ADRS*, and the message digest *md'*. First, the algorithm computes the leaf node index corresponding to the message digest *md'* and finds the private key $SIG_{dforc}.getsk(i)$ stored in the signature. Then, the node is generated by applying F (see Section 2.2 for details), and according to the authentication path provided by $SIG_{dforc}.getAUTH(i)$, the nodes of each layer are computed successively from the leaf node up to the root node of the tree, denoted as $dforc'_{pk}$. Finally, $dforc'_{\mathrm{pk}}$ is compared with the public key $dforc_{\mathrm{pk}}$. If the two match, the algorithm returns True; otherwise, it returns False, indicating that the signature is invalid. This process is given in Algorithm 8, where $SIG_{dforc}.getsk(i)$ returns the i-th private key stored in the signature, and $SIG_{dforc}.getAUTH(i)$ returns the i-th authentication path. The functions Th_k and H are described in Section 2.2, and the operations related to *ADRS* are given in Ref. [30].

$com\_ver(SIG\_ht, PK.seed, root, ADRS, idx\_tree, idx\_leaf, md') \rightarrow$ (True/False): The algorithm is used to verify whether the root node of the hypertree (HT) matches the provided commitment. First, the algorithm finds the leaf node corresponding to the message through the index *idx_leaf*, and then reconstructs the HT layer by layer using the signature values and hash values in the authentication path provided in *ht_sig*. By recursively hashing the signature values returned by $SIG_{ht}.getsig()$, the algorithm constructs the internal nodes of the tree layer by layer and finally derives the root node $root'$. If the calculated $root'$ matches the provided *root*, the algorithm returns True, indicating successful commitment verification; otherwise, it returns False, indicating verification failure. The detailed process is shown in Algorithm 9, and the operations related to *ADRS* are taken from Ref. [30].

Algorithm 8. *proof_verify*.

Input: SIG_{dforc}, $PK.seed$, $ADRS$, md', $dforc_{pk}$;
Output: True/False.

for $i = 0$ **to** $k-1$ **do**
    $idx[i] = \lfloor i \times \log(t) \rfloor$ to $\lfloor (i+1) \times \log(t) - 1 \rfloor$ bits of md' ;
    $dforc_{sk} = SIG_{dforc}.getsk(i)$;
    $ADRS.setTreeHeight(0)$;
    $ADRS.setTreeidx(i \times t + idx[i])$;
    $node[0] = F(PK.seed, ADRS, dforc_{sk})$;
    $auth[i] = SIG_{dforc}.getAUTH(i)$;
    $ADRS.setTreeIdx(i \times t + idx[i])$;
    for $j = 0$ **to** $a-1$ **do**
        $ADRS.setTreeHeight(j+1)$;
        if $idx[j]\%2 == 0$ **then**
            $ADRS.setTreeIdx(ADRS.getTreeIdx()/2)$;
            $node[1] = H(PK.seed, ADRS, (node[0] \| auth[0]))$;
        else
            $ADRS.setTreeIdx((ADRS.getTreeIdx() - 1)/2)$;
            $node[1] = H(PK.seed, ADRS, (auth[j] \| node[0]))$;
        end if
        $node[0] = node[1]$;
    end for
    $root[i] = node[0]$;
end for
$dforc.setType(DFORC\_roots)$;
$dforc.setKeyPairAddr(ADRS.getKeyPairAddr())$;
$dforc'_{pk} = Th_k(PK.seed, dforc_{pk}, ADRS, root[0] \| \ldots \| root[k-1])$;
if $dforc'_{pk} == dforc_{pk}$ **then**
    return True;
else
    return False;
end if

4.6 Commitment and proof update

When a key-value pair is added, it is first checked whether the key-value pair exists in the tuple *KV*. Now let us discuss two cases.

- **Case 1**: if it does not exist, the commitment value remains unchanged, but the proof changes. In other words, every time a new key-value pair is verified, the commitment value is the same as the previous one, but the generated proof differs because the DFORC node selected by the key-value pair has changed. Subsequently, verification is performed according to Algorithms 1–9.
- **Case 2**: if it exists, both the commitment value and proof are modified, and the version number corresponding to the tuple *VM* is increased by 1. According to the corresponding index in the tuple, the corresponding leaf node is hashed $VM[k]$ times, and the path from the leaf node to the root node is adjusted according to the Merkle tree to generate a new commitment value and proof. Next, re-authentication is performed according to Algorithms 1–9. Please note that when verifying the correctness of the proof, $ID[key]$

also needs to be hashed $VM[k]$ times to obtain the new leaf node of the reconstructed DFORC.

Algorithm 9. *com_ver*.

Input: SIG_{ht}, *PK.seed*, *root*, *ADRS*, *idx_tree*, *idx_leaf*, md';
Output: True/False.

ADRS.setLayerAddr(0);
ADRS.setTreeAddr(*idx_tree*);
for $j = 0$ **to** $d-1$ **do**
    idx_leaf is front $\lfloor h/d \rfloor$ bits in *idx_tree*;
    idx_tree$'$ is back $\lfloor h-(j+1)\times(h/d) \rfloor$ bit in *idx_tree*;
    tmp $=$ SIG_{ht}*.getsig*(j);
    ADRS.setLayerAddr(j);
    Generate *root*$'$ according to Algorithm 7, lines 1 to 14;
end for
if *root*$'$ $==$ *root* **then**
    return True;
end if
else
    return False;
end if

5 Security Analysis

In this section, we prove the quantum security of the scheme through the following theorem. It should be noted that F and H are simply substitute hash functions of *Th*, where F corresponds to message length n, and H corresponds to message length $2n$.

Theorem 1. For the parameters n, w, h, d, m, t, and k in the scheme, EQAS is PQ-EU-CMA secure if the following conditions are satisfied:

- T_h (and thus F and H) has post-quantum single-function multi-target collision resistance under different tweaks;
- F, under different tweaks, has post-quantum single-function multi-target second-preimage resistance;
- *PRF* and PRF_{msg} are families of post-quantum pseudo-random functions;
- H_{msg} has post-quantum interleaved-target subset resistance.

More specifically,

$$\begin{aligned} InSec^{\text{PQ-EU-CMA}}(EQAS;\xi,q_s) \le\ & InSec^{\text{PQ-PRF}}(PRF;\xi,q_1) + InSec^{\text{PQ-PRF}}(PRF_{\mathrm{msg}};\xi,q_s) \\ &+ InSec^{\text{PQ-ITSR}}(H_{\mathrm{msg}};\xi,q_s) + InSec^{\text{PQ-SM-TCR}}(T_h;\xi,q_2) \\ &+ 3\cdot InSec^{\text{PQ-SM-TCR}}(F;\xi,q_3) + InSec^{\text{PQ-SM-DSPR}}(F;\xi,q_3) \end{aligned} \tag{11}$$

where the concepts PQ-PRF, PQ-ITSR, and PQ-SM-TCR appearing in the above equation follow the definitions in Ref. [19].

We first estimate the insecurity level using a generic attack model with a fixed number of signature and oracle queries, and then compute the adversary's computational complexity to determine its success probability in the EU-CMA game.

Proof: The proof of security in this section follows the methodology in Ref. [19]. To prove the EU-CMA security of the EQAS scheme, we introduce five games (GAME.0 to GAME.4) to gradually establish the upper bound of the adversary's success probability. Each game represents a different abstraction of the signature scheme, allowing detailed analysis of its security. In

GAME.0, the adversary faces a real signature environment, where it can issue queries, obtain valid signatures, and attempt to find vulnerabilities for forgery. In GAME.1, we replace the output of the pseudo-random function (PRF) used for generating BW and DFORC private keys with uniformly random values. This replacement must be done without the adversary noticing. Specifically, each secret value generated for private keys is replaced by a random value of the same length, while ensuring that the corresponding public key remains consistent, i.e., the public key is derived from these random values so that the signing process still appears valid. The pseudo-signatures generated under this replacement continue to verify correctly for message m, as the substituted values still satisfy the structural requirements of verification. Therefore, the adversary cannot distinguish between GAME.1 and GAME.0, since pseudo-signatures are computationally indistinguishable from real signatures, and the only difference lies in the use of randomly substituted keys. The difference in the adversary's success probability between GAME.0 and GAME.1 is given by

$$|\mathit{Succ}_{\mathrm{GAME.1}}(\mathcal{A}) - \mathit{Succ}_{\mathrm{GAME.0}}(\mathcal{A})| \leq \mathit{InSec}^{\text{PQ-PRF}}(\mathit{PRF}; \xi, q_1) \tag{12}$$

where $\mathit{InSec}^{\text{PQ-PRF}}$ denotes the insecurity measure of the PRF, ξ represents the adversary's query space, and q_1 is denotes the number of queries made by the adversary.

In GAME.2, we replace the message-hash $\mathit{PRF}_{\mathrm{msg}}$ with a truly random function. After a message m is processed by $\mathit{PRF}_{\mathrm{msg}}$, a random value is obtained and used to generate the signature, ensuring consistency with previously derived private keys. The pseudo-signatures are constructed based on these random hash values but are structurally identical to real signatures, so the verification algorithm continues to accept them. Since the outputs of $\mathit{PRF}_{\mathrm{msg}}$ were already replaced by random values in GAME.1, the adversary cannot distinguish these pseudo-signatures by repeated queries, because they are completely consistent with the real signatures. The difference in success probability between GAME.1 and GAME.2 is

$$|\mathit{Succ}_{\mathrm{GAME.2}}(\mathcal{A}) - \mathit{Succ}_{\mathrm{GAME.1}}(\mathcal{A})| \leq \mathit{InSec}^{\text{PQ-PRF}}(\mathit{PRF}_{\mathrm{msg}}; \xi, q_s) \tag{13}$$

where $\mathit{InSec}^{\text{PQ-PRF}}$ represents the insecurity measure of $\mathit{PRF}_{\mathrm{msg}}$ and q_s is the number of message queries.

In GAME.3, the adversary cannot directly find a second preimage of a valid signature and instead tries to exploit weaknesses in the signature structure to forge one. Specifically, it attempts to infer the internal structure of the hash function from legitimate signatures obtained via the signing oracle, and then generate a valid forgery. Here, all keys used in signature generation are randomized. This means the signing process depends not only on random keys but also on random structural components, making it infeasible for the adversary to reproduce it. The pseudo-signatures at this stage are randomized in each component by adding additional random values, which obfuscate their internal structure and prevent inference of the signing process. The forgery at this stage mainly depends on forging the structure of the signature itself so that the forged signature passes verification. If the adversary can distinguish these pseudo-signatures from genuine ones, the key randomization has failed. Ideally, however, pseudo-signatures remain indistinguishable. Therefore, the difference in success probability between GAME.3 and GAME.2 is

$$|\mathit{Succ}_{\mathrm{GAME.3}}(\mathcal{A}) - \mathit{Succ}_{\mathrm{GAME.2}}(\mathcal{A})| \leq \mathit{InSec}^{\text{PQ-ITSR}}(H_{\mathrm{msg}}; \xi, q_s) \tag{14}$$

where $\mathit{InSec}^{\text{PQ-ITSR}}$ represents the resistance measure of the hash function H_{msg}.

In GAME.4, unlike GAME.3, the adversary's goal is to find a second preimage with the same structure as a known signature thus forging a new message-signature pair that can pass the verification process. By replacing the hash function with an ideal random function Oracle and introducing random perturbations during signature generation, it becomes hard for the adversary to find a second preimage. Specifically, a random value r is combined with the message m to generate a pseudo-secret value, which is then hashed to generate the signature, while maintaining consistency with the public key. Ensuring that the pseudo-signature passes both hash and commitment verification prevents the adversary from identifying the forgery. Although the adversary may attempt to exploit the tree construction to infer the private key, the anti-collision (TCR) and anti-collision of the tree structure ensure that such attacks are difficult to succeed. Difference in the success probability of opponents between GAME.3 and GAME.4 is

$$|\mathit{Succ}_{\text{GAME.4}}(\mathcal{A}) - \mathit{Succ}_{\text{GAME.3}}(\mathcal{A})| \leq \mathit{InSec}^{\text{PQ-SM-TCR}}(Th; \xi, q_2) \tag{15}$$

where $\mathit{InSec}^{\text{PQ-SM-TCR}}$ represents the anti-collision measure of the tweakable hash function *Th*.

Finally, the probability that the adversary successfully forges a signature in the whole signature system can be expressed by the following equation, which combines the security of multiple anti-collision and anti-second preimage attacks:

$$\mathit{Succ}_{\text{GAME.4}} \leq 3 \cdot \mathit{InSec}^{\text{PQ-SM-TCR}}(F; \xi, q_3) + \mathit{InSec}^{\text{PQ-SM-DSPR}}(F; \xi, q_3) \tag{16}$$

This shows that, in an ideal security environment, the probability that an adversary can successfully forge a signature is strictly limited.

By gradually transforming the game and designing an effective pseudo-signature scheme, the adversary cannot distinguish whether the signature in the game comes from the real signing process. The security of EQAS is based on the above-mentioned hash algorithms. For example, in the process of proof and commitment generation, the security of this stage mainly depends on *Th*, F, *PRF*, and H_{msg}, while the verification of commitments mainly depends on F and *PRD*. Therefore, the security of this scheme is guaranteed by these hash functions. Through the proof of theorems, we know that the digital signature scheme based on these hash functions is quantum secure. Therefore, this scheme can resist quantum attacks and has quantum security.

6 Experimental Analysis

When designing the EQAS scheme, the trade-off between security and efficiency is the key consideration. The security of the system mainly depends on the selection of security parameters in the hash-based signature scheme, while efficiency involves the optimization of computation, storage, and communication. Next, we explain how to choose appropriate parameters for the experiment, followed by theoretical and performance analyses of the scheme.

6.1 Parameter selection

The security of a hash-based signature directly depends on the output length of the hash function, which determines the security level against quantum attacks. However, although choosing a larger hash output length improves security, it inevitably increases the size of the signature and the computational cost of generation and verification. To reduce the signature length while maintaining security, we introduce the Winternitz parameter w, and reduce the number of hash chains in the signature by increasing the value of w. This can accelerate the speed of signature generation and verification, but it will also make the signature slightly larger, leading to higher communication and storage costs. In addition, a larger Winternitz parameter may reduce the security of the signature to some extent. Therefore, under different security requirements, the value of w and the output length of the hash function are adjusted to balance the relationship between signature size and computational complexity. DFORC is designed to maximize the storage and authentication efficiency of the system without compromising security. The height h and number of layers d of the tree directly affect storage efficiency. By setting reasonable h and d values, we ensure that only a few path nodes need to be updated during the authentication process, which greatly reduces storage read and write overhead. However, although a higher tree height reduces the hash chain length required for updates, it increases the verification path length in each authentication. To strike a balance between reducing storage overhead and accelerating verification, we dynamically adjust the number of layers in the hypertree structure to ensure system efficiency. In addition, DFORC needs to balance the trade-off between authentication speed and storage overhead in its design. By reducing the height of the Merkle tree, we reduce the number of nodes that need to be traversed each time, thus improving authentication speed. However, this also means that more storage space is needed to accommodate additional nodes. Therefore, we adopt a multi-layer structure, where the tree height of each layer is moderate, which not only optimizes authentication speed but also keeps storage overhead within a reasonable range. A lower tree height reduces computational complexity along the authentication path, but the multi-layer design increases implementation complexity. Parameters k and t determine the performance and security of DFORC. In DFORC, the number of leaf nodes of a tree must be a power of 2, and k can be freely chosen. Smaller t values usually lead to smaller and faster signatures. However, for a given security level, a smaller t requires a larger k, which increases the signature size and slows down signing speed. Therefore, it is very important to balance these two parameters. The message digest length *md* is the output length of H_{msg}, measured in bytes, and equals $(k \log t + 7)/8 + (h - h/d + 7)/8 + (h/d + 7)/8$ bytes. The number of hash chains in the BW key pair determines the size of the BW signature.

Some parameter settings of EQAS are shown in Table 2 (the unit of proof size is bit). Importantly, the parameter set proposed in this paper is not obtained simply by searching for the smallest or fastest option in the output. For the 256-bit security level, the signature size is about 15 KB. However, the computation of a single signature takes more than 20 minutes. Although such a trade-off may be attractive for some specific applications, many applications are unlikely to accept such long signing times. In contrast, the parameter set proposed in this paper is considered non-extreme, meaning that in our implementation, the signing time does not exceed a few seconds. Therefore, the SHA2-128f-simple algorithm is adopted in this paper to implement our scheme, which uses SHA-256 with 128-bit security, employs fast parameters (faster computation, larger signature size), and has a simple form (that is, no bitmask is used during hashing). Based on the above settings, we implement EQAS using the Python programming language.

Table 2 Parameter settings of EQAS

n	h	d	$\log t$	k	w	w'	Proof size (bit)
128	66	22	7	23	13	2	17,088
128	64	8	13	11	32	2	7,856
192	66	22	7	37	15	2	35,664
192	64	8	12	19	38	2	16,224
256	64	16	8	48	16	2	49,856
256	63	9	12	27	46	2	29,792

6.2 Theoretical analysis

This section outlines the number of hash function calls required for each operation. Since it can be ignored when estimating speed, we omit the single call to H_{msg} and the use of PRF_{msg} for generating commitments and signatures. Table 3 summarizes the number of calls to F, H, PRF, and Th_k. The sizes of the private key and public key, as well as the signature and authentication path, can be inferred from the algorithmic process (Table 4).

Table 3 Number of calls to F, H, *PRF*, and Th_k

Step	F	H	*PRF*	Th_k
Key generation	$2^{h/d}wl + ktw'$	$2^{h/d} - 1$	$2^{h/d}l$	$2^{h/d}$
Commitment generation	kt	$k\,(t-1)$	kt	$d(2^{h/d})$
Authentication path	$d(2^{h/d})wl$	$d(2^{h/d}-1)$	$d(2^{h/d})l$	—
Authentication	$k + dwl$	$k \log t + h$	—	d

Table 4 Storage sizes of keys, signatures, and authentication paths

Parameter	Size
Private key	$4n$
Public key	$2n$
Signature	$nk\,(\log t + 1)$
Authentication path	$n\,(h + dl + 1)$

6.3 Performance analysis

6.3.1 *Experimental setup and indicators*

The EQAS framework is used to address the security problem of blockchain authenticated storage. To better demonstrate the performance of EQAS, we conduct experiments to show the running time of key algorithms. All operations are carried out on a notebook computer with an Intel® Core™ i5-8300H CPU, NVIDIA GeForce GTX 1050 GPU, and 8 GB memory. The operating system is 64-bit Windows 10, and the Python compiler is Visual Studio Code. EQAS uses RocksDB[31] as the backend key-value database. The security parameter is 128 bits, with $w = 16$, $k = 33$, $t = 64$, and $w' = 2$. In addition, we test the proposed EQAS scheme on the Fisco blockchain platform, deploying it locally through FISCO BCOS technical documentation and deploying smart contracts on the platform for transactions.

When evaluating the time cost of EQAS, the total number of transactions T, the number of light nodes U, and the number of queries q are three key variables. The total number of transactions directly affects the size of the blockchain ledger and the amount of data that full nodes need to process. As the number of transactions increases, full nodes must perform more transaction verification, ledger updates, and block construction, which leads to more computation and storage operations, thus increasing the time overhead. The increase in the number of light nodes means that more network participants need to obtain data and proofs from full nodes. Each light node's query request requires the full node to respond, including data retrieval and proof generation, which may cause additional computational and network communication burdens on the full node, especially under high concurrency, which may lead to increased response delay. The number of queries refers to the frequency at which light nodes request data verification from full nodes. Frequent queries cause the full node to continuously retrieve data and generate proofs, which not only increases the computational load of the full node but may also affect the performance of the storage system because of the increased I/O operations. Under high query load, the time overhead of authenticated storage increases significantly.

The performance of each authenticated storage scheme is evaluated based on proof generation time and verification time. The authenticated storage schemes used for comparison are as follows:

(1) SPHINCS^{+}[19]: this is a stateless hash-based signature framework, which has significant advantages over traditional technologies in terms of speed, signature size, and security.

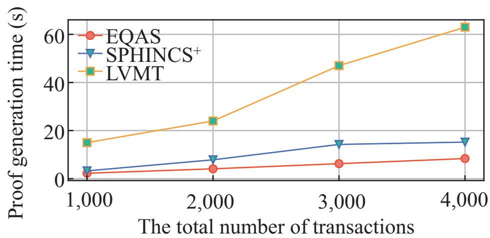
(2) LVMT[4]: this is a novel blockchain storage structure, which aims to improve the performance of blockchain systems by reducing the number of disk I/O operations. LVMT significantly reduces the amplification effect of read/write operations by using multi-level authenticated multi-point evaluation trees and a series of append-only Merkle

trees.

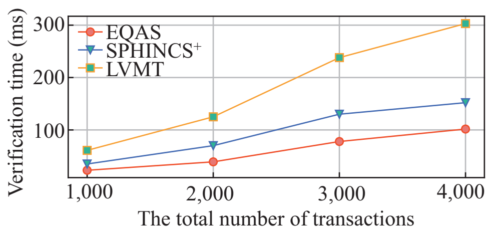
(3) EQAS: refer to Section 4 for a detailed description.

6.3.2 *Experimental results*

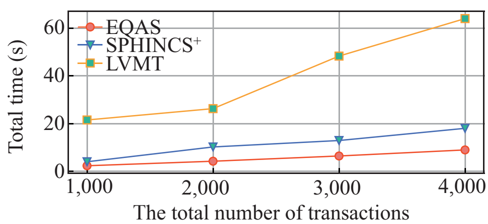
Figure 6 shows the influence of fixed query number $q = 100$ on each key process of authenticated storage when the total number of transactions is fixed in the range of 1,000 to 4,000. Figure 6(a) shows that with the increase in total transaction number, the proof generation time of EQAS is clearly better than that of SPHINCS$^+$ and LVMT. Especially when the total transaction number reaches 4,000, the generation time of LVMT is close to 60 seconds, while EQAS remains at a low level. Figure 6(b) shows that EQAS is also significantly superior to the other schemes in verification time. With the increase in total transaction number, the verification time of EQAS increases slowly, while that of SPHINCS$^+$ and LVMT rises significantly. Figure 6(c) shows the influence of different transaction numbers on the time overhead of authenticated storage under a fixed number of queries, where EQAS exhibits the slowest growth rate in total time overhead, further confirming its efficiency in high-transaction-number scenarios. Overall, there is a positive correlation between time overhead and total transaction number, because as the number of queries increases, full nodes must handle more requests, which may increase CPU and memory utilization and the frequency of I/O operations, thus increasing the overall time overhead.



(a) Proof generation time under different transaction numbers



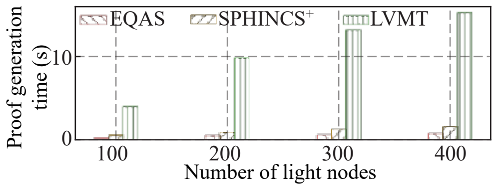
(b) Verification time under different transaction numbers



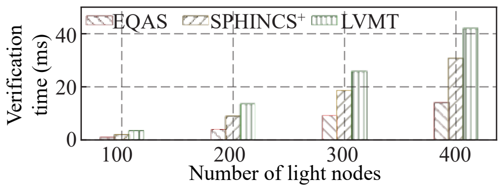
(c) Total time overhead under different transaction numbers

Figure 6 Influence of the total transaction number on the time overhead of authenticated storage under a fixed number of queries

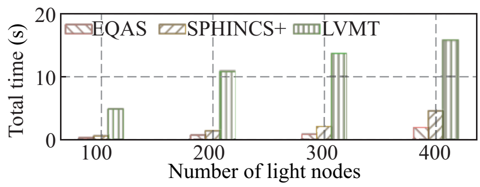
Figure 7 shows the change of signature generation time under different numbers of light nodes ($U = 100$, 200, 300, 400) with fixed query number $q = 100$.



(a) Proof generation time under different numbers of light nodes



(b) Verification time under different numbers of light nodes

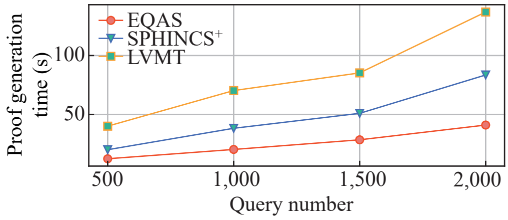


(c) Total time overhead under different numbers of light nodes

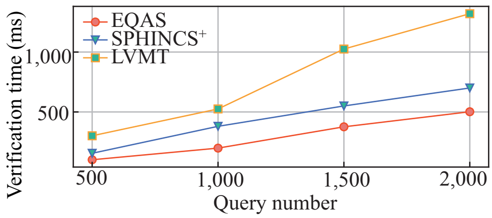
Figure 7 Influence of the number of light nodes on the time overhead of authenticated storage under a fixed number of queries

Figure 7(a) shows the proof generation time under different numbers of light nodes. EQAS shows extremely low generation time as the number of light nodes increases, which is clearly better than SPHINCS$^+$ and LVMT. Especially when the number of light nodes reaches 400, the generation time of LVMT increases greatly, while EQAS is almost unaffected. Figure 7(b) shows the change of verification time. With the increase of light nodes, the verification time of EQAS grows very slowly and always remains low, while the verification time of SPHINCS$^+$ and LVMT shows a significant upward trend. Figure 7(c) reflects the influence of the number of light nodes on the total time overhead under a fixed number of queries. EQAS maintains the smallest total time overhead as light nodes increase, further confirming its efficiency in multi-light-node scenarios. Overall, there is a positive correlation between time overhead and the number of light nodes, because as light nodes increase, full nodes need to generate more proofs to respond to more queries, which increases computational and network communication load, thus increasing time overhead.

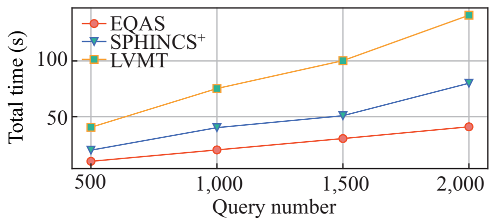
Figure 8 shows the influence on signature generation time under different query numbers ($q = 100$, 200, 300, 400) and total transaction number $T = 1000$. Figure 8(a) shows the proof generation time under different query numbers. As the query number increases, the generation time of EQAS remains at a low level, which is significantly better than SPHINCS$^+$ and LVMT. Especially when the number of queries reaches 2,000, the generation time of LVMT increases sharply, while the growth trend of EQAS is relatively flat. Figure 8(b) shows the change in verification time. As the query number increases, EQAS always keeps ahead in verification time, showing only a slight increase, while SPHINCS$^+$ and LVMT show a more obvious upward trend. Figure 8(c) reflects the influence of query number on total time overhead under different total transaction numbers. The total time overhead of EQAS remains clearly lower than the other two schemes as query number increases, showing its high efficiency in high-query-frequency scenarios. In general, as the query number increases, time overhead also rises. This is because as query frequency grows, full nodes must process more requests, increasing computational and I/O load, which may lead to longer response times. In addition, under higher transaction numbers, the ledger data size becomes larger, and each query may need to retrieve and process more information, increasing both complexity and time required per query.



(a) Proof generation time under different query numbers



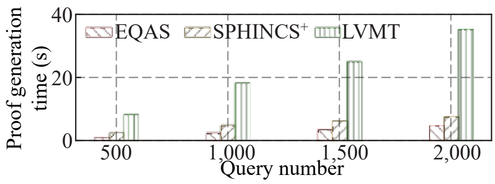
(b) Verification time under different query numbers



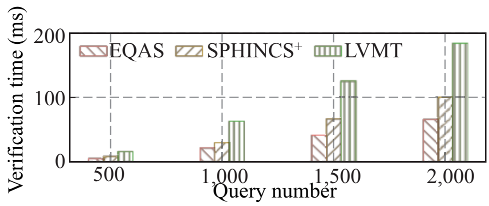
(c) Total time overhead under different query numbers

Figure 8 Influence of the number of queries on the time overhead of authenticated storage under a fixed total transaction number

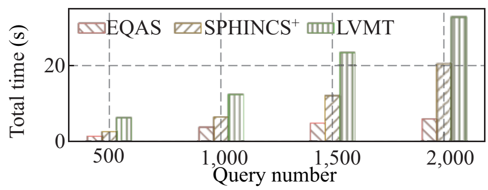
Figure 9 shows the influence of fixed query number $q = 100$ on signature generation time under different numbers of light nodes ($U = 100$, 200, 300, 400). Figure 9(a) shows the proof generation time under different query numbers. As the number of queries increases, the generation time of EQAS remains at a low level, while the generation time of SPHINCS$^+$ and LVMT increases significantly with more queries. Especially when the number of queries in LVMT reaches 2,000, the generation time rises sharply and is clearly higher than that of EQAS. Figure 9(b) shows the change in verification time. EQAS maintains low verification time under different query numbers, showing clear advantages compared with SPHINCS$^+$ and LVMT. Figure 9(c) shows the total time overhead. The total time overhead of EQAS increases slowly under different query numbers and always remains lower than that of the other two schemes, demonstrating its high efficiency in high-query-frequency scenarios. As the number of light nodes increases, each query may need to obtain more data from full nodes. In addition, under a higher number of light nodes, full nodes may need to process more query requests simultaneously. Therefore, the time overhead is proportional to both the number of queries and the number of light nodes.



(a) Proof generation time under different query numbers



(b) Verification time under different query numbers



(c) Total time overhead under different query numbers

Figure 9 Influence of the number of queries on the time overhead of authenticated storage under a fixed number of light nodes

To sum up, the total authenticated storage time overhead of the EQAS scheme proposed in this paper averages about 40 seconds. However, in Ethereum, transactions usually need to wait for about 12 blocks to be confirmed, and the interval between each block is about 15 s, totaling about 180 s. It can be seen that the time overhead of the EQAS scheme is relatively small.

7 Conclusion

We proposed an efficient and quantum-resistant blockchain authenticated storage scheme in this paper, termed EQAS, to address the security threats posed by quantum computing to authenticated storage technology. By introducing stateless hash-based signature technology into the blockchain authenticated storage system, this method effectively enhances the system's quantum resistance. Specifically, we applied stateless hash-based signatures to blockchain authenticated storage for the first time and achieved efficient data authentication by decoupling the data storage and authentication processes, utilizing FORC and hypertree structures. Security analysis and experimental results showed that EQAS can resist quantum attacks and provides an efficient authenticated storage process, offering a secure and high-performance solution for blockchain technology.

Future research can be conducted in the following directions. First, regarding the relatively large signature length of EQAS, further optimization of hash-based signature schemes can be explored to reduce storage and communication costs. Second, with the development of blockchain technology, the scalability and operability of the system are crucial. Future work can investigate how to apply quantum-resistant authenticated storage technology in multi-chain or even cross-chain environments to ensure the security of cross-chain transactions. Finally, with the rapid advancement of distributed computing and privacy-preserving technologies, it is worth exploring how to achieve more efficient data sharing and verification mechanisms while maintaining quantum-resistant security. This will be an important direction for further research.

References

[1] Farah MB, Ahmed Y, Mahmoud H, Shah SA, Al-Kadri MO, Taramonli S, Bellekens X, Abozariba R, Idrissi M, Aneiba A. A survey on blockchain technology in the maritime industry: Challenges and future perspectives. Future Generation Computer Systems, 2024, 157: 618–637. [doi: 10.1016/j.future.2024.03.046]

[2] Chen J, Yang H, He K, Li K, Jia M, Du RY. Current situation and prospect of blockchain scaling technology. Ruan Jian Xue Bao/Journal of Software, 2024, 35(2): 828-851(in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6954.htm [doi: 10.13328/j.cnki.jos.006954]

[3] Li CX, Beillahi SM, Yang G, Wu M, Xu W, Long F. LVMT: An efficient authenticated storage for blockchain. Proc. of the 17th USENIX Symp. on Operating Systems Design and Implementation. Boston: USENIX Association, 2023. 135-153.

[4] Li CX, Beillahi SM, Yang G, Wu M, Xu W, Long F. LVMT: An efficient authenticated storage for blockchain. ACM Trans. on Storage, 2024, 20(3): 17. [doi: 10.1145/3664818]

[5] Yang ZB, Zolanvari M, Jain R. A survey of important issues in quantum computing and communications. IEEE Communications Surveys & Tutorials, 2023, 25(2): 1059–1094. [doi: 10.1109/COMST.2023.3254481]

[6] Lu Y, Sigov A, Ratkin L, Ivanov LA, Zuo M. Quantum computing and industrial information integration: A review. Journal of Industrial Information Integration, 2023, 35: 100511. [doi: 10.1016/J.JII.2023.100511]

[7] "Nezha" flies into the sky and into the sea, the dawn of quantum computing has appeared. 2024 (in Chinese). https://www.cnr.cn/shanghai/tt/20240614/t20240614_526745312.shtml

[8] Monz T, Nigg D, Martinez EA, Brandl MF, Schindler P, Rines R, Wang SX, Chuang IL, Blatt R. Realization of a scalable Shor algorithm. Science, 2016, 351(6277): 1068–1070. [doi: 10.1126/science.aad9480]

[9] Willsch D, Willsch M, Jin FP, De Raedt H, Michielsen K. Large-scale simulation of Shor's quantum factoring algorithm. Mathematics, 2023, 11(19): 4222. [doi: 10.3390/math11194222]

[10] Grover LK. A fast quantum mechanical algorithm for database search. Proc. of the 28th Annual ACM Symp. on Theory of Computing. Philadelphia: ACM, 1996. 212–219. [doi: 10.1145/237814.237866]

[11] Gao F, Hu RZ, Yin LF, Cao HB, Yu J, Shuang F. Quantum Grover search-inspired global maximum power point tracking for photovoltaic systems under partial shading conditions. IEEE Trans. on Sustainable Energy, 2024, 15(3): 1601–1613. [doi: 10.1109/TSTE.2024.3361483]

[12] Hasanova H, Baek UJ, Shin MG, Cho K, Kim MS. A survey on blockchain cybersecurity vulnerabilities and possible countermeasures. Int'l Journal of Network Management, 2019, 29(2): e2060. [doi: 10.1002/NEM.2060]

[13] Faridi A, Siddiqui F. Improving SPV-based cryptocurrency wallet. Proc. of the 2019 Int'l Conf. of Cybernetics, Cognition and Machine Learning Applications. Singapore: Springer, 2020. 127–137. [doi: 10.1007/978-981-15-1632-0_14]

[14] Zhao YL, Niu BN, Li P, Fan X. A novel enhanced lightweight node for blockchain. Proc. of the 1st Int'l Conf. on Blockchain and Trustworthy Systems. Guangzhou: Springer, 2020. 137–149. [doi: 10.1007/978-981-15-2777-7_12]

[15] Zhang C, Xu C, Hu HB, Xu JL. COLE: A column-based learned storage for blockchain systems. Proc.

of the 22nd USENIX Conf. on File and Storage Technologies. Santa Clara: USENIX Association, 2024. 329–345.

[16] Pillai SEVS, Polimetla K. Analyzing the impact of quantum cryptography on network security. Proc. of the 2024 Int'l Conf. on Integrated Circuits and Communication Systems. Raichur: IEEE, 2024. 1–6. [doi: 10.1109/ICICACS60521.2024.10498417]

[17] Truger F, Barzen J, Bechtold M, Beisel M, Leymann F, Mandl A, Yussupov V. Warm-starting and quantum computing: A systematic mapping study. ACM Computing Surveys, 2024, 56(9): 229. [doi: 10.1145/3652510]

[18] Hülsing A, Butin D, Gazdag S, Rijneveld J, Mohaisen A. XMSS: extended Merkle signature scheme. Internet Research Task Force. 2018. [doi: 10.17487/RFC8391]

[19] McGrew D, Curcio M, Fluhrer S. RFC 8554: Leighton-Micali hash-based signatures. 2019. [doi: 10.17487/RFC8554]

[20] Bernstein DJ, Hülsing A, Kölbl S, Niederhagen R, Rijneveld J, Schwabe P. The SPHINCS$^+$ signature framework. Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security. London: ACM, 2019. 2129–2146. [doi: 10.1145/3319535.3363229]

[21] Zhang KY, Cui HR, Yu Y. SPHINCS-α: A compact stateless hash-based signature scheme. Cryptology ePrint Archive, 2022.

[22] Karniavoura F, Magoutis K. Decision-making approaches for performance QoS in distributed storage systems: A survey. IEEE Trans. on Parallel and Distributed Systems, 2019, 30(8): 1906–1919. [doi: 10.1109/TPDS.2019.2893940]

[23] Cai ZH, Lin JY, Liu F. Blockchain storage: Technologies and challenges. Chinese Journal of Network and Information Security, 2020, 6(5): 11–20 (in Chinese with English abstract). [doi: 10.11959/j.issn.2096-109x.2020019]

[24] de Ocáriz Borde HS. An overview of trees in blockchain technology: Merkle trees and Merkle Patricia Tries. 2022. https://www.researchgate.net/publication/358740207

[25] Raju P, Ponnapalli S, Kaminsky E, Oved G, Keener Z, Chidambaram V, Abraham I. mLSM: Making authenticated storage faster in Ethereum. Proc. of the 10th USENIX Conf. on Hot Topics in Storage and File Systems. Boston: USENIX Association, 2018.

[26] Choi JA, Beillahi SM, Li PL, Veneris A, Long F. LMPTs: Eliminating storage bottlenecks for processing blockchain transactions. Proc. of the 2022 IEEE Int'l Conf. on Blockchain and Cryptocurrency. Shanghai: IEEE, 2022. 1–9. [doi: 10.1109/ICBC54727.2022.9805484]

[27] Ponnapalli S, Shah A, Banerjee S, Malkhi D, Tai A, Chidambaram V, Wei M. RainBlock: Faster transaction processing in public blockchains. Proc. of the 2021 USENIX Annual Technical Conf. USENIX Association, 2021. 333–347.

[28] Han YL, Li CX, Li PL, Wu M, Zhou D, Long F. Shrec: Bandwidth-efficient transaction relay in high-throughput blockchain systems. Proc. of the 11th ACM Symp. on Cloud Computing. ACM, 2020. 238–252. [doi: 10.1145/3419111.3421283]

[29] Bernstein DJ, Hopwood D, Hülsing A, Lange T, Niederhagen R, Papachristodoulou L, Schneider M, Schwabe P, Wilcox-O'Hearn Z. SPHINCS: Practical stateless hash-based signatures. Proc. of the 34th Annual Int'l Conf. on the Theory and Applications of Cryptographic Techniques. Sofia: Springer, 2015. 368–397. [doi: 10.1007/978-3-662-46800-5_15]

[30] Jean-Philippe A, Daniel JB, Warb B, et al. SPHINCS$^+$ submission to the NIST post-quantum project, v.3. 2020. https://di-mgt.com.au/pqc-08-sphincs-example.html

[31] Facebook Database Engineering Team. RocksDB: A persistent key-value store for flash and RAM storage. 2022.

Chuan Zhang, Ph.D., associate professor, doctoral supervisor. His research interests include machine learning, blockchain, and applied cryptography.

Chunhai Li, Ph.D., assistant professor. His research interests include Internet of Things security, network security, and blockchain.

Xuhao Ren, master's degree candidate. His research interest is applied cryptography.

Tong Wu, Ph.D., associate professor. Her research interests include blockchain, cryptography, and cloud security.

Haotian Deng, Ph.D. candidate. His research interest is blockchain.

Licheng Wang, Ph.D., professor, doctoral supervisor. His research interests include blockchain, cryptography, and quantum computing.

Yajie Wang, Ph.D. His research interests include machine learning and artificial intelligence.