Hindawi
Security and Communication Networks
Volume 2022, Article ID 9473246, 13 pages
https://doi.org/10.1155/2022/9473246

Research Article

HB$^+$-MHT: Lightweight and Efficient Data Integrity Verification Scheme for Cloud Virtual Machines

Zhi Yang,[1] **Xiaopeng Li**,[1] **Shuyuan Jin**,[2] **Lei Sun**,[1] **Zhao Zhang**,[1] **Baoshan Yang**,[1] **Xuehui Du**,[1] **and Fan Chao**[1]

[1]*PLA Information Engineering University, Zhengzhou 450001, China*
[2]*SUN YAT-SEN University, Guangzhou 510006, China*

Correspondence should be addressed to Shuyuan Jin; jinshuyuan@mail.sysu.edu.cn

Received 16 December 2021; Revised 21 February 2022; Accepted 3 March 2022; Published 25 March 2022

Academic Editor: Yunchuan Guo

With the rapid development of cloud computing, cloud storage is widely used. In the cloud environment, users' virtual machine system mirrors and data are stored in the cloud server. The escape of virtual machines and Trojan virus attacks make it challenging to ensure the integrity of virtual machine systems. Trusted computing is expensive to randomly verify data integrity and does not adapt to dynamic data changes. Provable data integrity is a potential solution to this problem. Merkle Hash Tree (MHT) model is widely adopted in provable data integrity. Although MHT requires only a small amount of evidence for verification, the verifier's number of hash calculations and the server's efficiency of evidence query are not optimal. Moreover, the verification frequency of each piece of data is not considered by MHT. Properly handling these factors can improve the actual verification performance. In this paper, a lightweight and efficient data integrity verification approach called HB$^+$-MHT is proposed for the tenant virtual machine (TVM) in cloud computing. In HB$^+$-MHT, the Huffman hash tree scheme is used for small file verification to ensure that the hot file has a shorter path, which reduces the required amount of evidence for verification. Meanwhile, the B$^+$ hash tree scheme is used for big files verification, which can effectively reduce evidence query time and hash calculation times. The experimental results show that the scheme proposed in this paper can perform data integrity verification well, with reduced computing and storage overhead.

1. Introduction

The wide application of cloud computing provides users with convenient and cheap services, which greatly reduces the storage overhead and management burden of users. However, due to the huge scale of cloud computing systems and unprecedented openness and complexity, these systems are faced with security issues more severe than before. The centralized management of cloud computing centers has become the target of attacks, which endangers the data confidentiality, integrity, and availability of cloud platforms [1–3].

When users or enterprises outsource their data to cloud computing service providers or entrust cloud computing service providers to run their applications, cloud computing service providers have priority access to the data or applications. Cloud system tenants usually use encryption to ensure the security of the private data stored on the cloud server [4]. Meanwhile, the development of encryption technology and access control technology provides reliable confidentiality for the tenant's private data [5]. However, due to various risks such as internal personnel management negligence, hacker attack, or server system failure, cloud service providers cannot make users believe that their data has not maliciously tampered. Since all data of the tenant virtual machine are stored in the cloud, security issues often occur, such as the virtual machine escaping and the attack of virtual machines by illegal users outside the system. These issues bring great challenges to the integrity protection of tenant data in the system. In the cloud environment, it is usually assumed that the cloud server is not trusted, and the tenant virtual machine can only establish trust through

online interaction with the cloud server. In addition, there are many dynamic services in the cloud environment, thus effectively ensuring that the data integrity in the dynamic environment is crucial to users' experience of cloud services. Due to the openness and dynamic characteristics of cloud computing, it is difficult to establish trust, which makes the security of dynamic data integrity of tenants a problem.

Therefore, it is urgent to design an efficient integrity verification scheme to ensure the integrity of tenant virtual machine systems and data in the cloud. Building a secure and reliable integrity verification method has become a research hotspot in the field of information security.

The traditional trusted computing method is based on the trusted computing technology proposed by the Trusted Computing Group (TCG) [6], which tries to provide credibility for the endpoints in the distributed computing environment. The Trusted Platform Module (TPM) is introduced into the hardware layer of the computing platform to provide a hardware-based trusted root for the computing platform. Starting from the trusted root, the integrity of the hardware and software of a local platform is measured layer by layer by the trust chain transmission mechanism. The traditional trusted computing methods have a low verification level and coarse granularity, and they are faced with the problem of high computational cost in system dynamic measurement. When verifying the integrity of a file, these methods need to calculate the integrity of all previous files in turn, which requires a lot of computing power to randomly verify data integrity and does not adapt to dynamic data changes.

Provable data integrity (PDI) is considered a potential solution to this problem. According to whether fault-tolerant processing is adopted, PDI can be divided into two types: POR model of data recovery proof and PDP model of data holding proof [7, 8]. PDP judges whether the cloud data is damaged, while POR attempts to recover the data after identifying the damaged data. PDP pays more attention to detection efficiency, while POR pays more attention to data recovery. From the perspective of monitoring Trojans or Viruses destroying cloud data, this paper mainly discusses the PDP mechanism.

Merkle Hash Tree (MHT) is commonly used in PDP for the cloud environment. MHT is a well-studied authentication structure [9] that intends to efficiently prove that a data set is undamaged and unaltered. It is constructed as a binary tree where the leaves are the hashes of authentic data values. Each middle node is the hash of the concatenation of its left and right child nodes, and the root node is signed by the integrity management authority. The research on MHT has greatly improved the efficiency of PDP. However, there are two issues that MHT does not consider. One issue is that the verification frequency of specific data is not considered, and each data has an equal authentication path. It is reasonable that the frequently verified data should have a shorter path. The other issue is that MHT designs the authentication structure for reducing the number of evidence but ignores the impact of the number of hash calculations of the verifiers and the evidence query time of the servers, which may be important performance constraints in the actual environment.

To address the aforementioned challenges, this paper proposes a data integrity verification approach called HB^+-MHT for TVM. HB^+-MHT adopts different verification schemes according to the file size. Small files are loaded into the TVM, and the optimized hash tree is adopted for verification. In this case, compared with the communication overhead of transmitting small files to TVM, the traditional PDP methods with multiple signature operations might take more verification time. For large files, an improved PDP method is adopted because the communication overhead of transmitting large files to TVM may be too high. Our contributions are summarized as follows:

(i) The Huffman Merkel scheme is designed to verify the integrity of small files. According to different integrity verification frequencies or weights, the authentication path of hot files is optimized and shortened effectively.

(ii) The B^+ hash tree scheme is designed to verify the integrity of large files, which optimizes the server evidence query and reduces the number of client hash verifications.

(iii) Experiments show that our scheme can realize data integrity verification well, which reduces the computational overhead effectively and achieves high verification efficiency.

The rest of the paper is arranged as follows. Section 2 discusses the related work, Section 3 describes the detailed design of the scheme, Section 4 evaluates and analyzes the experimental results, and finally, Section 5 concludes the paper.

2. Related Work

TCG introduces the TPM in the hardware layer. Based on the trusted root and the trusted chain transfer mechanism, the integrity measurement can be implemented for the hardware and software layers of the local platform. However, the integrity measurement mechanism proposed by TCG has low verification levels. It can only verify the credibility of the operating system of the computing platform and does not specify the verification method for application layer credibility. In the process of verification, the whole measurement list should be sent to the verifier, which will easily cause measurement leakage. Reiner Sailer et al. [10] proposed IMA, a more secure system integrity measurement architecture. The system kernel maintains a list of metrics and hashes the measured values to obtain the aggregation values as a reliable basis for the verification of the metrics. Meanwhile, the system extends the concept of TCG trust metrics to the application layer dynamic system. However, every verification needs to hash all the measurements again to obtain the aggregate value, which greatly reduces the efficiency of the system operation. Also, the privacy protection of the TCG remote authentication mechanism is still unsolved. To solve the shortcomings of IMA, Xu et al. [11] store the measurement with the Merkle tree structure so that only the authentication path related to the measurement

needs to be sent. However, since the Merkle tree is a balanced binary tree, too many middle nodes need to be queried. So, the average search complexity is high, and this method needs to be further optimized.

The current PDP mechanisms mainly rely on MAC authentication code, RSA signature or BLS signature, and they can support dynamic operation, multiple copies or privacy protection, etc. Based on the data holding proof of MAC authentication code [12], the MAC value of the message authentication code is used as the verification metadata to verify the integrity of the data stored on the remote server. However, users need to download data for integrity verification, which results in a lot of communication overhead and privacy data leakage. Also, this method does not support dynamic data integrity verification. To solve the problem that dynamic data operation is not supported, Erway et al. [13] introduced the jump table dynamic data structure to support dynamic data operation. This scheme has some problems, such as a long authentication path, requiring a lot of auxiliary information, and high computational and communication overhead. Wang et al. [14] proposed a PDP mechanism based on the Merkle tree, which uses the MHT structure to ensure the correctness of data block location. Meanwhile, they used the BLS signature mechanism to ensure the integrity of data block content. Although the model supports dynamic data operations, the insertion of data is easy to increase the scale of MHT and make it out of balance. Shen et al. [15] proposed a fully dynamic structure combining bidirectional linked list and position array to support data dynamic update more effectively. Tan et al. [16] proposed a lightweight integrity verification scheme based on the jump table and BLS signature mechanism to reduce the overhead of generating verification metadata for verifiers. Sun et al. [17] proposed an adaptive authenticated data structure with privacy-preserving for big data stream in cloud, which can provide real-time authentication of outsourced big data. Jin et al. [18] verified the data that is accessed frequently in the cloud by signature verification. However, this method requires verifiers to download the data locally and verify the data that has not been queried for a long time through PDP.

In general, many methods reduce the amount of verification evidence through MHT and extend the structure of MHT for further performance optimization or security enhancement. However, MHT may not be optimal in some cases. For example, a frequently verified file has the same amount of evidence as an infrequently verified file in the traditional MHT-based scheme. It is more reasonable for hotspot files to have shorter authentication paths. For the verification of large files, because the amount of evidence without transferring the file is significantly reduced in the PDP scheme, how the cloud storage server can quickly find the evidence and how the client can quickly calculate the hash values may be the main bottleneck. To this end, this paper proposes the Huffman hash tree and B^+ hash tree to overcome the bottleneck.

3. Detailed Design

3.1. Overall Framework. As shown in Figure 1, HB^+-MHT consists of TVM, Cloud Storage Server (CSS), and Key Management Center (KMC). KMC securely distributes public and private keys to tenants and transmits the tenants' public keys to CSS. TVM runs the query agent to initiate a data integrity verification challenge to CSS. CSS queries the evidence and generates proof as a response. TVM verifies the data integrity by analyzing the correctness of the proof.

The HB^+-MHT cloud data integrity verification approach consists of two verification schemes. One is the Huffman hash tree (HHT) scheme, which is generally used to verify the integrity of the small file. Compared with the communication overhead of transmitting small files, the traditional PDP methods with more signature operations might take more computing time. Therefore, for small files, this scheme adopts a method that loads these files into TVM and further combines hash tree and Huffman tree to realize HHT, thereby shortening the authentication path and improving the verification efficiency of the files with high verification weight. The other is based on the combination of the B^+ hash tree and BLS aggregate signature. It reduces communication overhead by not loading files into TVM and achieves no-copy verification of large files. Because the amount of evidence is significantly reduced when the PDP scheme is adopted, how the cloud storage server can quickly find the evidence and how the client can quickly calculate the hash values has become the main bottleneck. B^+ hash tree optimizes hash computation and evidence organization, which can effectively reduce the processing time of CSS and TVM.

3.2. Hash Tree Construction

3.2.1. B^+HT: B^+ Hash Tree. B^+HT is an extension of the combination of B^+ tree and MHT, and it is also a balanced multichannel evidence lookup tree. For the B^+HT with an order of m, each node in the tree has at most m subtrees, and all nonleaf nodes except the root have at least $m/2$ subtrees. A middle node consists of a tuple $(C_1, H_1, P_1, C_2, H_2, P_2, \ldots, C_m, H_m, P_m)$, where C_i $(1 \le i \le m)$ is the number of leaf nodes of its i-th subtree; H_i is the hash value of the concatenation of all data items in the root node of its i-th subtree; P_i points to the root node of its i-th subtree. The leaf nodes represent all data blocks of the file. The root node has one more item than the middle node, which is the signature of the root node data.

The data of B^+HT should be usually stored on a disk due to limited memory. Most operating systems read data in blocks and put data written at one time into one disk block or multiple consecutive disk blocks. Therefore, if the size of the B^+HT node is slightly smaller than the disk block size u, the quick query of evidence in nodes can be achieved. Let v be the size of subtree information in a node; then, m should be approximately equal to u/v. To ensure performance and tree balance, the number of subtrees of each middle node cannot be less than $\lceil M/2 \rceil$. Each hash calculation in MHT only takes the data of two subtrees as the input, while each hash calculation in B^+HT takes the data of multiple subtrees as the input. Therefore, B^+HT can obtain the total hash value of all blocks through much fewer hash operations than MHT, which means that the number of hash calculations for

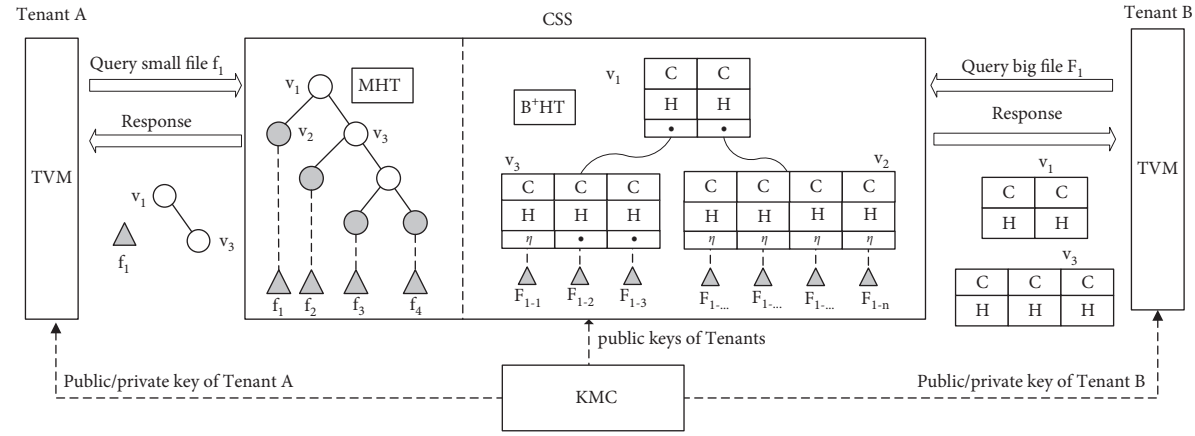

Figure 1: The composition and principles of HB$^+$-MHT.

evidence verification is greatly reduced. Each middle node stores the number of leaves of each subtree, which can realize a fast data block search. As an example, Figure 2 shows looking up the data block d_{11}. According to the root node, it is deduced that d_{11} is the seventh leaf node of V_3's subtree. Similarly, according to the V_3 node, it is known that d_{11} is the fourth leaf node of V_7's subtree.

We define authentication path path$_i$ about data d_i as the set of passing nodes on the path from node i to root node. The set of node hash values of the sibling nodes of all nodes on the authentication path is called auxiliary authentication information, which is denoted as Ω_i. As shown in Figure 2, the authentication path about d_{11} is Path$_{11}$ = {$v1, v3, v7$} and its auxiliary authentication information $\Omega_i = \{v_1\langle h(v_2)\rangle, v_3\langle h(v_6), h(v_8)\rangle, v_7\langle h(d_8), h(d_9), h(d_{10})\rangle\}$. Obviously, the authentication path is much shorter than MHT.

3.2.2. HHT: Huffman Hash Tree. Huffman hash tree (HHT) is an extension of the combination of the Huffman tree and MHT, in which each leaf node is assigned a weight to reflect the frequency of data verification. A middle node is represented as a tuple (LC, LP, H, RC, RP), where H is the hash of the data concatenation of its left child and right child; LP and RP point to the left and right child nodes, respectively; LC and RC are the numbers of leaf nodes contained by the left and right children, respectively. In HHT, the leaf nodes are numbered sequentially from left to right. The path of a leaf can be quickly located according to LC and RC stored in the middle nodes. The root node also adds a signature to the hash value H. For a leaf l, this paper defines l's weighted path length as the product of l's path length and l's. Let AWPL be the average weighted path length of all leaf nodes in the tree. Suppose the weights of n leaf nodes are $(w_1, w_2, w_i, \ldots, w_n)$ $\sum_{i=1}^{n} w_i = 1$; then, the binary hash tree with the smallest AWPL is called the Huffman hash tree.

For example, in Figure 3, there are four files a, b, c, and d with verification probabilities of (0.6, 0.2, 0.1, 0.1), respectively. Three binary hash trees are constructed with (I) AWPL = 2, (II) AWPL = 2.8, and (III) AWPL = 1.6. It can be seen that AWPL of the tree in (III) is the smallest. It can be analyzed and verified that the tree in (III) is exactly a Huffman hash tree.

The general process of generating HHT is as follows. (1) According to the given weights $(w_1, w_2, w_i \ldots w_n)$, $\sum_{i=1}^{n} w_i = 1$, the set $F = \{T_1, T_2, \ldots, T_M\}$ of n binary trees are formed. Each T_i has only one node, and its left and right subtrees are empty. (2) Select two trees T_{s1} and T_{s2} with the smallest root node weight from F and build a new binary tree T_{new} with T_{s1} and T_{s2} as its left and right subtrees. The weight of T_{new}'s root node is the sum of the weights of T_{s1} and T_{s2}, and LC, LP, RC, and RP of T_{new}'s root node are assigned corresponding values according to T_{s1} and T_{s2}. (3) Remove T_{s1} and T_{s2} and add T_{new} to the set. (4) Repeat Steps 2 and 3 until there is only one tree in F. This tree is a Huffman hash tree.

3.3. Data Integrity Verification Process

3.3.1. HHT Scheme for Small Files. (1) Initialization Phase

Step 1. The KMC uses the key generation algorithm. Let $\mathbb{G}_1$ be the multiplicative cyclic group of a large prime. p and g are the generators of $\mathbb{G}_1$. Randomly select a SK$\in_R \mathbb{Z}_p$, and calculate PK = g^{ke}. Then, the public parameter is PK, and the system master private key is MSK. KMC generates the tenant's private key sk and the tenant's public key according to the tenant's user $i\ d$. The generated keys are sent to TVM.

Step 2. TVM establishes a hash tree HHT of integrity measurements. The generation of HHT is shown in Algorithm 1. The data stored in the leaf node of HHT is the hash value of the initial small files of the system. The internal nodes are constructed according to the properties of the hash tree and generated by hash operations. TVM sings the HHT root node with the tenant private key, and a timestamp [19] is added to the signature. $\mathrm{Sig}_{sk}(\mathrm{Nodehash}_{\mathrm{rootnode}}) = H(\mathrm{Nodehash}_{\mathrm{root}}\ node\ node \| \mathrm{time}\ stamp\ stamp)^{sk}$.

Step 3. TVM sends the file set that needs to be verified, its HHT and $\mathrm{sig}_{sk}(\mathrm{Nodehash}_{\mathrm{rootnode}})$, to CSS. The data are saved and maintained by CSS.

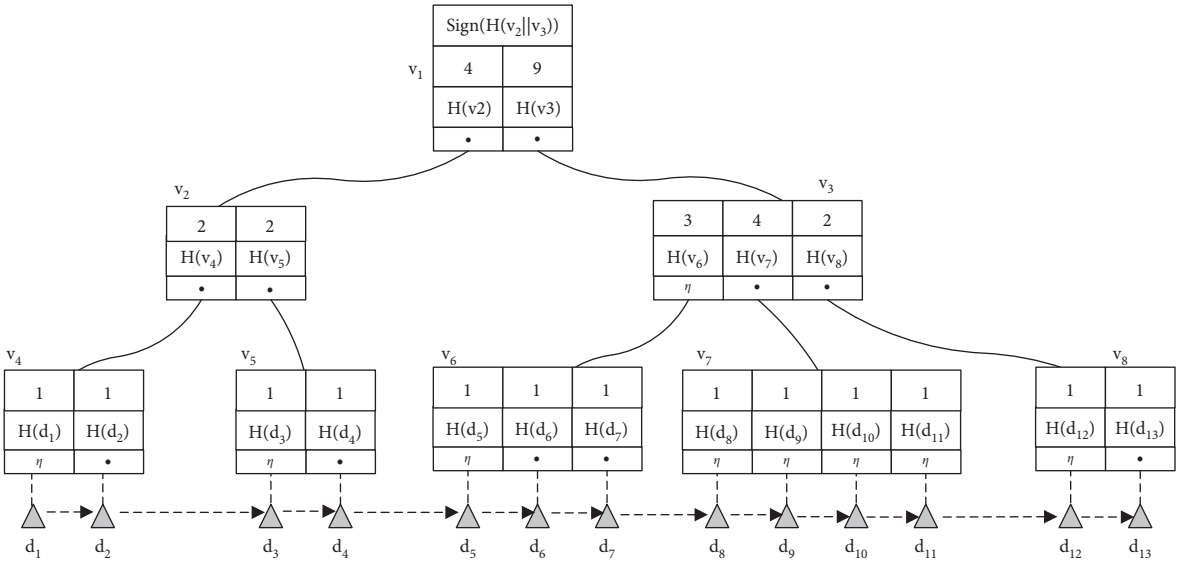

Figure 2: B$^+$HT data structure.

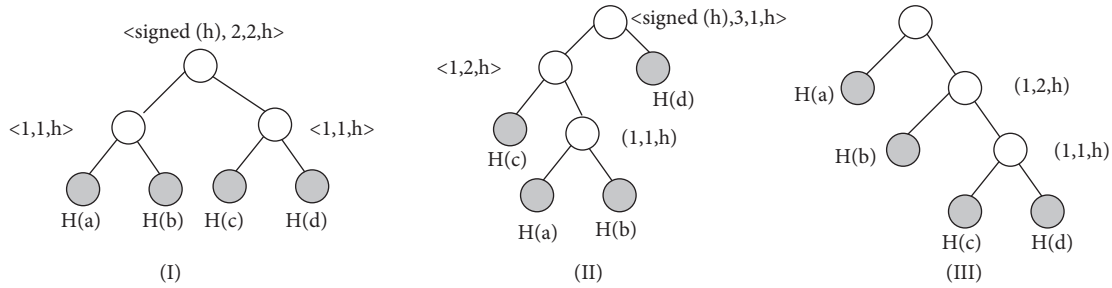

Figure 3: HHTs with different AWPL.

(2) Challenge Verification Phase. Challenge verification is initiated by TVM or a third-party auditor (TPA) on behalf of the tenant. TVM generates a query challenge about the file f_i and sets it to CSS. Then, CSS generates the authentication path Ω_i according to HHT and sends a proof $\text{pro} = \{\Omega_i, \text{sig}_{sk}(\text{Nodehash}_{\text{rootnode}})\}$ to TVM. After receiving the proof, TVM calculates the hash value of the file and uses Ω_i to calculate $\text{Nodehash}'_{\text{rootnode}}$. The verification process can be expressed as $e(\text{Nodehash}_{\text{rootnode}} \| \text{timestamp})\, e(\text{sig}(\text{Nodehash}_{\text{rootnode}}), g)$.

If the verification is successful, the integrity of the file is not damaged.

(3) Dynamic Data Update Phase. The HHT integrity verification scheme supports file addition, deletion, and modification. The operation flow of file-level data modification is shown in Figure 4. First, TVM sends a request of modifying the file f_i to CSS and then uses the hash value of the original file f_i to verify the correctness of received Ω_i. If it passes the verification, TVM uses the updated file f'_i and Ω_i to calculate the new root node $\text{Nodehash}'_{\text{rootnode}}$ of HHT and then uploads the new signature with a timestamp to CSS.

For the file insertion operation, since a newly added file f_{new} has not been verified and has the lowest weight, a leaf node f_{low} with the lowest weight is selected, and f_{low} is expanded into a subtree f_{subtree} with only three nodes. F_{subtree}'s left child is $f_{\text{new,}}$ and the right child is f_{low}. Accordingly, all nodes on the path from f_{subtree}'s root node to the root node of the tree need to be updated, so the complexity of the insertion update operation is the height of the tree.

For the file deletion operation, if the nodes with high weight are deleted, the nodes with low weight need to move up as a whole, which requires a lot of calculation. Meanwhile, CSS needs to interact with TVM for a lot of information. Considering the overhead, this paper keeps the leaf nodes corresponding to the deleted file and does not move them, except that the hash values stored in them are cleared. Accordingly, all nodes on the path need to be updated, so the complexity of the deleting update is the height of the tree. In the subsequent addition of files, these dummy node locations are preferred.

(4) Verification Frequency Change. Tenants' personalized verification requirements may change dynamically and often show local characteristics; i.e., some files are frequently verified in a certain period. The static HHT generated at one time may not provide good verification services under dynamic changes. Also, regenerating the whole HHT might

Input: *Fileinfoarray* satisfying $\sum_{i=1}^{m}$ Fileinfoarray$[i] \cdot w = 1$
Output: *Nodelist*
Begin
For each item f in *Fileinfoarray* {
//Initialization, each tree has only one root node
HHTNode n = new *HHTnode* ();
$n.h$ = hash ($f.data$);
n.weight = f.weight;
$n.lc = 0$; $n.rc = 0$;
$n.lp$ = null; $n.rp$ = null;
Nodelist.equeue (n)
}
While (*NodeList.size*! = 1)
//Select two trees with the smallest AWPL from the queue
nl, $nr \leftarrow$ min (*NodeList*);
//Merge the two trees
HHTNode n = new *HHTnode*();
$n.h = hash$ ($nl.h \parallel nr.h$);
$n.weight = nl.weight + nr.weight$;
$n.lc = nl.lc + nl.rc$; $n.rc = nr.lc + nr.rc$;
$n.lp$ = nl; $n.rp = nr$;
//Add a new binary tree to the queue
Nodelist.equeue (n);
//Delete the analyzed two subtrees
Nodelist.de*equeue* (n_1);
Nodelist.de*equeue* (n_2);
end while
return *Nodelist*
end

Algorithm 1: Generation of HHT.

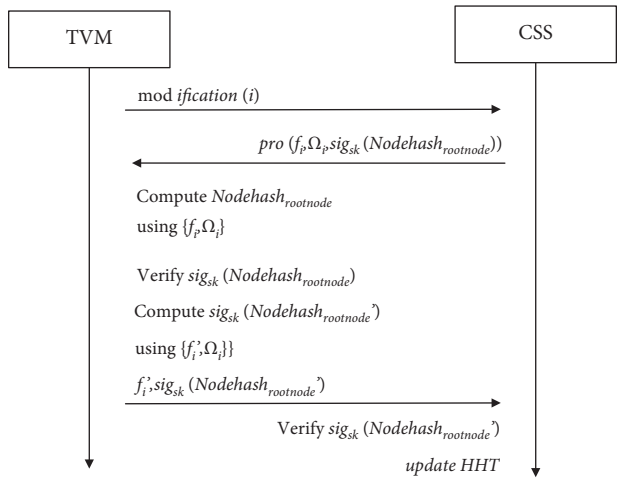

Figure 4: File modification process of HHT.

bring inconvenience and cost to tenants. Thus, this paper proposes the VFC mechanism that combines local updates and global updates to deal with the change of verification frequency. VFC records the verification history and counts the verification frequency of each file in the latest period into a vector W_{cur}. Meanwhile, it maintains a vector W_{hht} that denotes the verification probability of each file used in the current HHT. The elements in W_{cur} and W_{hht} are arranged in descending order. In the local update mode, when a file f under verification is ranked in the top k of W_{cur}, that is, it is a hot file, CSS will check whether f is ranked in the top k of W_{hht}. If not, CSS will seek a file f' and exchange the positions of f and f' in HHT. f' satisfies the following conditions: (1) in the top k of W_{hht}, (2) not in the top k of W_{cur}, and (3) f' has the lowest weight among the files satisfying conditions 1 and 2. After CSS exchanges f and f', it updates their values in

W_{hht} and requests the tenant to resign the relevant path. This operation occurs when TVM verifies f. Therefore, the evidence information related to f can be reused and verified at one time. In the global update mode, the Euclidean distance between the vectors W_{cur} and W_{hht} is calculated regularly. When the distance is greater than the threshold, HHT will be regenerated and resigned. Considering the high overhead of this method, the update interval is long, and the threshold is high. Local update follows the locality principle so that the hot files can always be verified quickly with a shorter authentication path. Moreover, since the update occurs when the tenant verifies the hot file, the reuse of evidence saves communication bandwidth and computational overhead. The global update ensures that HHT will not fall into local optimization and can maintain the adaptation of updated HHT.

3.3.2. B^+HT Scheme for Big Files

(1) Initialization Phase. Firstly, TVM divides the file into fixed-size blocks to obtain the data block set $D = (d_1, d_2, d_3 \ldots d_n)$ and hashes all blocks to obtain the hash set $\mu = \{H(d_i)\}_{1 \le i \le n}$. Then, TVM randomly generates $\varphi \leftarrow \mathbb{Z}_p$ and uses the tenant private key sk to sign each data block d_n, which obtains the signature set $\Phi = \{\sigma_i\}_{1 \le i \le n}$ through $\sigma_i = (H(d_i) \cdot \varphi^{d_i})^{sk}$. Then, TVM uses the elements in μ as leaf node to construct the B^+HT structure C and signs C with sk, i.e., $\text{sig}_{sk}(\text{Nodehash}_{\text{rootnode}}) = H(\text{Nodehash}_{\text{rootnode}} \| \text{timestamp})^{sk}$, where a timestamp is added to the signature. Finally, TVM sends $\{D, \Phi, C, \text{sig}_{sk}(\text{Nodehash}_{\text{rootnode}})\}$ to CSS for saving.

(2) Challenge Verification Phase. Challenge verification is initiated by TVM. TVM randomly selects the location set I of data blocks to be verified. Then, it randomly generates $\varepsilon_i \leftarrow \mathbb{Z}_p$ and sends a challenge message $\{i, \varepsilon_i\}_{i \in I}$ to CSS. CSS calculates $\mu = \sum_{i \in I} \varepsilon_i d_i \in \mathbb{Z}_p$ and $\omega = \prod_{i \in I} \sigma_i^{\varepsilon_i}$. Next, it sends μ, ω, and $H(m_i)$ of the target data block and auxiliary authentication information Ω_i of the data block as the proof pro to TVM.

After receiving the verification, TVM uses $\{\Omega_i, H(d_i)\}_{i \in I}$ to calculate $\text{Nodehash}'_{\text{rootnode}}$ and performs the verification:

$$e\left(h\left(\text{Nodehash}'_{\text{rootnode}} \| \text{timestamp}, g^{sk}\right) \stackrel{?}{=} e\left(\text{sig}\left(\text{Nodehash}_{\text{rootnode}}\right), g\right)\right). \tag{1}$$

If the verification is correct, continue to judge:

$$e(\omega, g) \stackrel{?}{=} e\left(\prod_{i \in I} \left(H(d_i)^{\varepsilon_i} \cdot \varphi^{\mu}\right), pk\right). \tag{2}$$

If the verification is successful, the integrity of the file is not damaged. The whole process of challenge verification is shown in Figure 5.

Correctness:

$$\begin{aligned} e(\omega, g) &= e\left(\coprod_{i \in I} \left(H(d_i) \cdot \varphi^{d_i}\right)^{sk \cdot \varepsilon_i}, g\right) \\ &= e\left(\prod_{i \in I} H(d_i)^{\varepsilon_i} \cdot \prod_{i \in I} \varphi^{d_i \varepsilon_i}, pk\right), \\ &= e\left(\prod_{i \in I} \left(H(d_i)^{\varepsilon_i} \cdot \varphi^{\mu}\right), pk\right) \\ &= e\left(\coprod_{i \in I} \left(H(d_i)^{\varepsilon_i} \cdot \varphi^{\mu}\right), pk\right). \end{aligned} \tag{3}$$

Through formula (1), TVM can verify the integrity of the data block d_i of a file F stored in CSS without downloading d_i. However, verifying the integrity of multiple data blocks in F does not guarantee that F is not destroyed, even if TVM has verified all data blocks. This is because formula (1) does not bind the location of the data block d_i in F and untrusted CSS can use another data block d_j in other locations. If the data of d_j is correct, this fake can pass the verification of (1). Formula (2) limits the position of each data block with the hash tree. Although the hash tree can verify the content, the verification can only be conducted when the verified data block d_i is loaded into TVM. Therefore, only the combination of formulas (1) and (2) can realize the PDP verification of the content and location.

(3) Dynamic Data Update Phase. To make the B^+HT scheme more suitable for the cloud environment, the integrity verification scheme must support the dynamic update of data. The block update operations include modification, insertion, and deletion.

If the verification is correct, auxiliary information Ω_i and $h(d')$ are used to calculate $\text{sig}_{sk}(\text{Nodehash}'_{\text{rootnode}})$, and CSS updates $\text{sig}_{sk}(\text{Nodehash}'_{\text{rootnode}})$.

(b) Data insertion

Data insertion operation inserts a new data block. Besides changing the physical structure of the file, this operation may cause node splitting that does not meet the structural properties of the B^+ hash tree and change the authentication structure C. We use the B^+HT tree in Figure 2 to illustrate the insertion operation.

Suppose TVM wants to insert a block d_{new} after a block d_i. First, TVM generates signature σ' corresponding to d_{new} and then sends the insertion request message to CSS. After CSS receives the message, it first finds the node $\text{node}_{\text{involve}-i}$ that involves $\text{hash}(d_i)$. Assume that the order of the B^+HT tree is k. There are two situations in the current node, as shown in Figure 7.

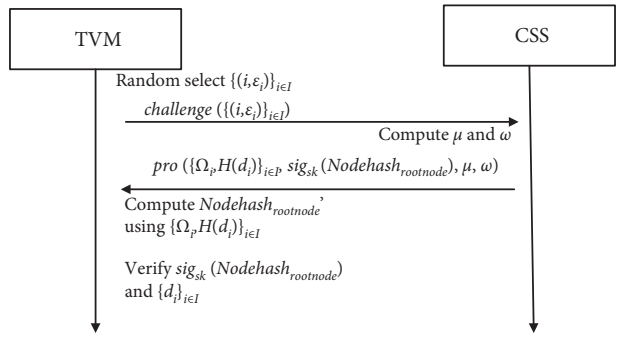

Figure 5: Challenge verification process of B$^+$HT.

(1) The number of subtrees of $\text{node}_{\text{involve}-i}$ is less than k. In this case, $H(d_{\text{new}})$ is added to $\text{node}_{\text{involve}-i}$, and the involved nodes on the authentication path are updated. Figure 7(a) shows the case of inserting a new node m after the block d_1 in Figure 2. Then, CSS will directly generate a proof $\text{pro}(\Omega_{\text{new}}, H(d_{\text{new}}))$.

(2) The number of subtrees of $\text{node}_{\text{involve}-i}$ is equal to k. In this case, $\text{node}_{\text{involve}-i}$ needs to be split, and its parent node $\text{node}_{\text{parent}-\text{involve}-i}$ whose subtrees number is equal to k also needs to be split, etc. Specifically, CSS splits $\text{node}_{\text{involve}-i}$ into $\text{node}_{\text{involve}-i-1}$ and $\text{node}_{\text{involve}-i-2}$. $\text{node}_{\text{involve}-i-1}$ involves the information of all subnodes before (including) $\text{hash}(d_i)$ in $\text{node}_{\text{involve}-i}$, and $\text{node}_{\text{involve}-i-2}$ involves the information of all subnodes after $\text{hash}(d_i)$ in $\text{node}_{\text{involve}-i}$. The information of d_{new} is appended to the end of $\text{node}_{\text{involve}-i-1}$. Then, CSS updates the information of $\text{node}_{\text{parent}-\text{involve}-i}$. The item about the subnode $\text{node}_{\text{involve}-i}$ in $\text{node}_{\text{parent}-\text{involve}-i}$ is divided into two items. The hash values in the two items are, respectively, $\text{hash}(\text{node}_{\text{involve}-i-1})$ and $\text{hash}(\text{node}_{\text{involve}-i-2})$, and their number of leaf nodes and child pointers are assigned according to $\text{node}_{\text{parent}-\text{involve}-i}$ and $\text{node}_{\text{involve}-i-2}$, respectively. Figure 7(b) shows the case of inserting a new node n after block d_9 in Figure 2. CSS will generate proof $\text{pro}(\Omega_{\text{new}}, H(d_{\text{new}}))$ according to the new structure of B$^+$HT.

After receiving pro, TVM continues to calculate the value of the nodes in the authentication path until $\text{Nodehash}_{\text{rootnode}}'$ is obtained. If the verification passes, TVM updates the root signature and sends it to CSS.

(c) Data deletion

Data deletion removes a leaf node, and it is the opposite of data insertion. Assume the order of the B$^+$HT tree is k. If a node d_{del} is deleted, a simple case is that the number of subtrees of the node $d_{\text{hash-del}}$ involving hash (d_{del}) is greater than $\lceil k/2 \rceil - 1$. In this case, the item of d_{del} in $d_{\text{hash-del}}$ is deleted simply. Taking the B$^+$HT tree in Figure 2 as an example, Figure 8(a) shows the processing of deleting node d_5. If the number of subtrees of $d_{\text{hash-del}}$ is less than $\lceil k/2 \rceil - 1$. The rest of $d_{\text{hash-del}}$ must regenerate new nodes with its brother nodes to satisfy the properties of the B$^+$HT tree. Similarly, the related parent node changes are processed. For example, Figure 8(b) shows the process of deleting node d_1 and merging its parent node with its peer neighbor node.

4. Analysis and Evaluation

4.1. Security Analysis. The security of HB$^+$ MHT is analyzed from the aspects of integrity verification and data update, including whether it can prevent untrusted CSS from cheating tenants with incorrect data, and whether it can prevent untrusted third parties from maliciously updating authentication data in CSS.

Theorem 1. *If the hash algorithm is collision-resistant and the signature algorithm is unforgeable, no adversary against the B$^+$HT scheme could make the verifier accept in a data verification protocol instance with a nonnegligible probability, unless it responds by correctly calculating the value.*

Proof. The BLS signature scheme is adopted, whose unforgeability is proved in [14]. The security of HHT and B$^+$HT trees is mainly analyzed. It will be proven that if there is an adversary A who can break the scheme with nonnegligible probability advantage, then A has an algorithm B to find a pair of hash collisions with a nonnegligible probability advantage.

Given a block d_i to be verified with its correct authentication path path_i, and correct auxiliary authentication information Ω_i, TVM's verification process can be denoted as a Boolean function verify $(i, H(d_i), \Omega_i)$ with parameters $(i, H(d_i), \Omega_i)$, and its calculation of path nodes starts from the leaf node. The analysis of the j^{th} node on the path $path_i$ yields

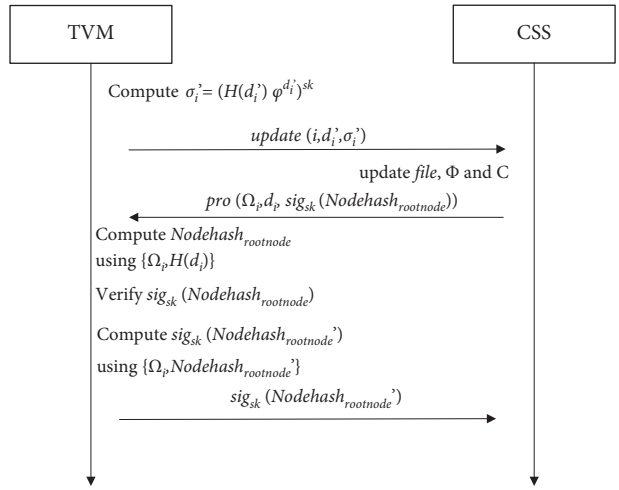

Figure 6: Data modification process of B$^+$HT.

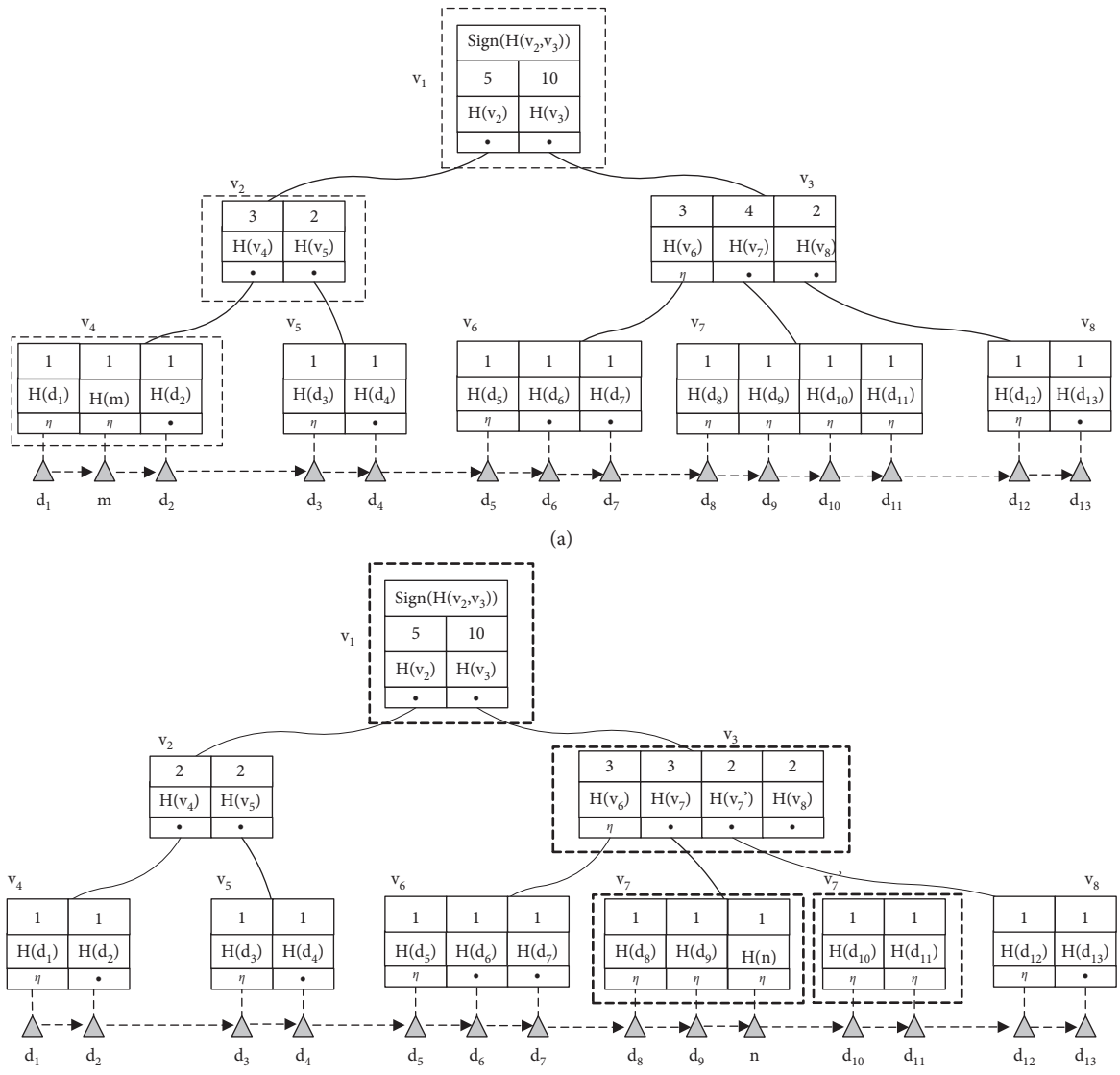

Figure 7: Data insertion in B$^+$HT. (a) Insert new node m after block d_1. (b) Insert new node n after block d_9.

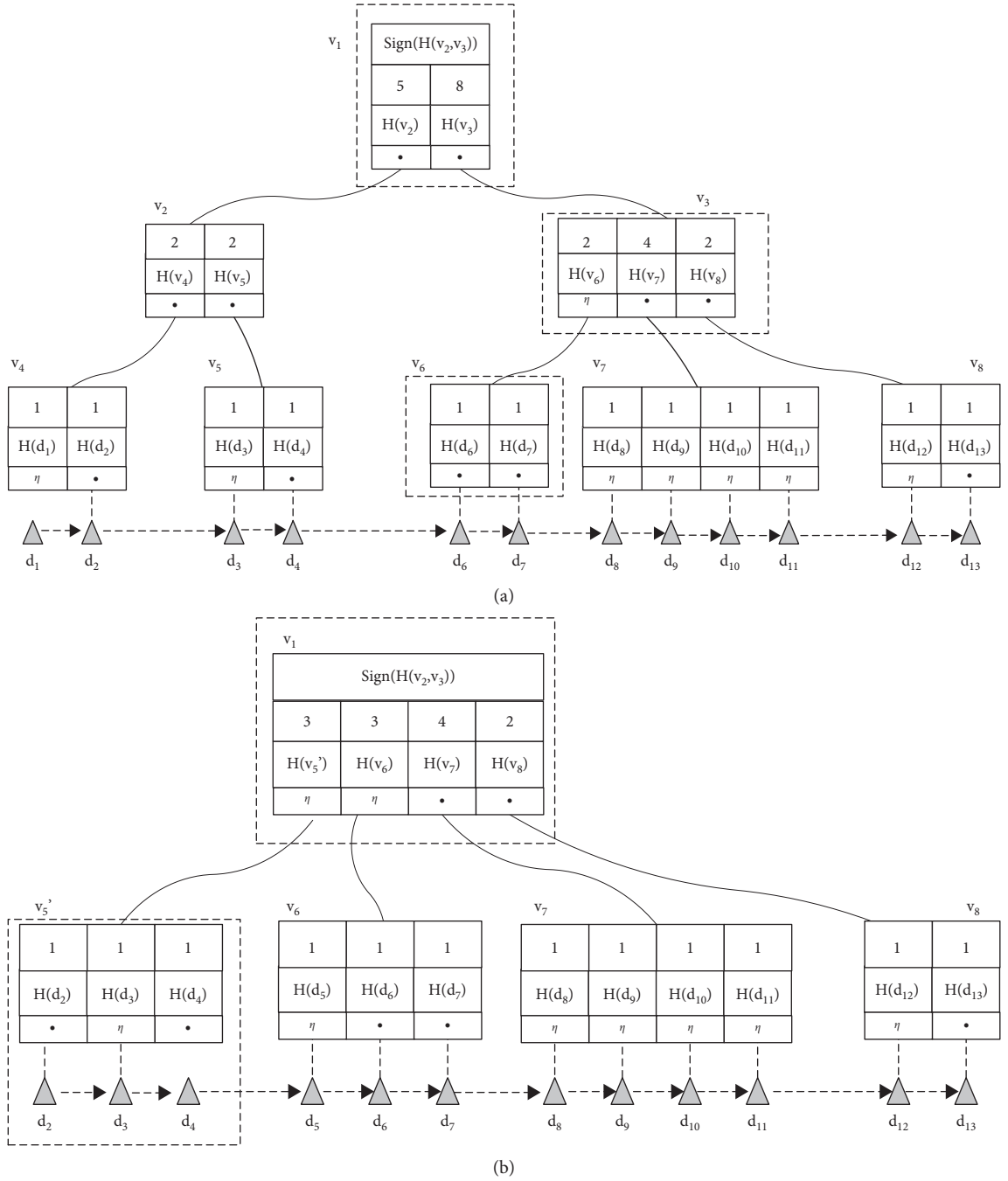

Figure 8: Data deletion in B$^+$HT. (a) Delete node d_5. (b) Delete node d_1.

$$\text{path}_i[j].H_r = \text{H}\left(\Omega_i^{j-1}||\Omega_i^{j-2}||\cdots||\Omega_i^{j-(r-1)}||\text{path}_i[j-1]||\cdots||\Omega_i^{j-K}\right), \tag{5}$$

where the j-1$^{\text{th}}$ node on path$_i$ is the r^{th} child of the j node, K is the order of B$^+$HT, and Ω_i^{j-s} is the hash value of the s^{th} child of the j^{th} node on the path. Finally, the correctness of path$_i$[root]. sign is verified and the verification result is obtained.

If A can construct a tuple $(\mathrm{i}, H(d_i)^*, \Omega_i{}^*)$ with $d_i{}^* \neq d_i$ as a proof to the TVM, and the function verify $(\mathrm{i}, H(d_i)^*, \Omega_i{}^*)$ returns true, A wins the security game.

In this case, d_i and $d_i{}^*$ have the same root node, while $d_i{}^* \neq d_i$. The authentication path of $d_i{}^*$ is recorded as $\mathrm{path}_i{}^*$; then, there must be u and v satisfying

$$\begin{aligned} \mathrm{path}_i[u].H_r &= H\left(\Omega_i^{u-1} \| \Omega_i^{u-2} \| \cdots \left\| \Omega_i^{u-(r-1)} \right\| \mathrm{path}_i[u-1] \| \ldots \| \Omega_i^{u-K}\right), \\ \mathrm{path}_i^* &= [v].H_s H\left(\Omega_i^{s-1} \| \Omega_i^{s-2} \| \cdots \left\| \Omega_i^{s-(r-1)} \right\| \mathrm{path}_i[v-1] \| \ldots \| \Omega_i^{v-K}\right), \\ \mathrm{path}_i[u].H_r &= \mathrm{path}_i^*[v].H_s. \end{aligned} \tag{6}$$

However, since $\mathrm{path}_i[u-1] = \mathrm{path}_i[v-1]$, it is contradictory to the collision-resistant characteristics of the hash function. It can be concluded from the above discussion that the probability the adversary A can destroy the data verification protocol of the B$^+$HT is negligible. Therefore, the data verification protocol of the B$^+$HT is secure. □

Theorem 2. *If the hash algorithm is collision-resistant and the signature algorithm is unforgeable, no adversary against the B$^+$HT scheme could make CSS accept in a data update protocol instance with a nonnegligible probability, unless it responds by correctly calculating the value.*

Proof. When an adversary A submits an update request of inserting a block $d_{\mathrm{malice}}{}'$ after the i^{th} block to CSS, A will receive a message $(i, H(d_i), \Omega_i)$ from CSS. According to $d_{\mathrm{malice}}{}'$, Ω_i and the current time $timestamp_{\mathrm{now}}$, A needs to recalculate the root value $hash_{\mathrm{root}}$ of the tree and sign the hash value of the concatenation of $hash_{\mathrm{root}}$ and $timestamp_{\mathrm{now}}$. Because the signature cannot be forged and A has no private key, A can only reuse a previously signed message $signature_{\mathrm{old}}$. Assume $signature_{\mathrm{old}} = hash(hash_{\mathrm{root\text{-}old}} || timestamp_{\mathrm{old}})$. The current timestamp $timestamp_{\mathrm{new}}$ is greater than $timestamp_{\mathrm{old}}$ and Ω_i cannot be changed. If A can construct $hash_{\mathrm{root\text{-}new}}$ by adjusting $d_{\mathrm{malice}}{}'$ that satisfies $hash(hash_{\mathrm{root\text{-}new}} || timestamp_{\mathrm{new}}) = hash(hash_{\mathrm{root\text{-}old}} || timestamp_{\mathrm{old}})$, then A wins the security game. However, due to $hash_{\mathrm{root\text{-}new}} \neq hash_{\mathrm{root\text{-}old}}$ and $timestamp_{\mathrm{new}} \neq timestamp_{\mathrm{old}}$, it is contradictory to the collision-resistant characteristics of the hash function. It can be concluded from the above discussion that the probability the adversary A can destroy the data insertion protocol of the B$^+$HT is negligible. The analysis of the deletion and modification operations is similar. Therefore, the data update protocol of the B$^+$HT is secure.

From the perspective of security, HHT is a deformation of MHT, and the security of the authentication structure of the binary hash tree has been analyzed by [14]. Therefore, the analysis of HHT is omitted here. □

4.2. Performance Evaluation. Here, representative schemes are compared to better evaluate the performance of HB$^+$ MHT. More specifically, HHT is compared with MHT, and B$^+$HT is compared with Wang's scheme [14] in terms of space and time complexity. The experiment is performed in a simulation environment. TVM and CSS are, respectively, run on two hosts, which are equipped with an Intel(R) Core (TM) i7-8700k@ 3.70 GHz CPU and 16.0 GB RAM and run Windows 10 operating system. The two hosts are directly connected through a gigabit network cable. B$^+$HT is implemented in C++ language, and the development tool is Visual Studio 2019. The hash operation uses SHA1. The signature operation uses BLS. Besides, a pairing-based cryptography library [20] is adopted to implement these cryptographic algorithms.

4.2.1. Space Complexity. In PDP integrity verification schemes, TPA or TVM only stores a small amount of information such as signature and timestamp, which takes up a small storage space. The storage overhead of CSS is analyzed in Table 1, where $l\mathbb{G}_1$ is the signature length, lH is the length of hash, M is the number of files, $item_{\mathrm{hht}}$ is the size of the node in HHT, $item_{\mathrm{mht}}$ is the size of the node in MHT, N is the number of file blocks, and K is the order of B$^+$HT tree.

As shown in Table 1, the storage overhead of CSS is mainly related to the storage of tree nodes. For small file verification, HHT occupies slightly more storage space than MHT because the data of HHT nodes includes the number of leaf nodes in its subtree with a size of 32 bits. However, compared with the 160-bit size of the SHA's hash value stored in a tree node, the increment of storage space is small. For large file verification, the storage overhead of B$^+$HT and Wang's scheme [14] for storing block signature is the same. However, B$^+$HT occupies significantly less storage space than Wang's scheme for storing tree nodes. With the increase of K, B$^+$HT can save storage space logarithmically than Wang's scheme [14]. Of course, K cannot increase infinitely, and it should not exceed the size of the disk block.

4.2.2. Time Complexity

(1) HHT Scheme. Because HHT is obtained by changing MHT, their time complexity is evaluated and compared. Integrity verification services are used more frequently than data update services. Here, the time from the initiation of a user request to the end of verification is analyzed. The main overhead includes CSS's query evidence time T_{query}, the communication time T_{com}, and TVM's verification time T_{ver}. The time cost of each phase of the two schemes is shown in Table 2. It can be seen that whether HHT is superior to MHT

Table 1: Storage complexity comparison.

Entity	Storage cost (bits)	Entity	Storage cost (bits)
Our HHT scheme	$O(l\mathbb{G}_1 + \sum_{i=0}^{\log_2 M} 2^i \cdot \text{itme}_{hht} \cdot lH)$	Our B$^+$HT scheme	$O((N+1)l\mathbb{G}_1 + \sum_{i=0}^{\log_K N} K^i \cdot lH)$
Merkle Hash Tree	$O(l\mathbb{G}_1 + \sum_{i=0}^{\log_2 M} 2^i \cdot \text{item}_{mht} \cdot lH)$	Wang [14]	$O((N+1)l\mathbb{G}_1 + \sum_{i=0}^{\log_2 N} 2^i \cdot lH)$

Table 2: Time complexity comparison between HHT and MHT.

Entity	T_{query}	T_{com}	T_{ver}
Our HHT scheme	O (AWPL)	O (AWPL)	O (AWPL)
Merkle Hash Tree	$O(\log_2 M)$	$O(\log_2 M)$	$O(\log_2 M)$

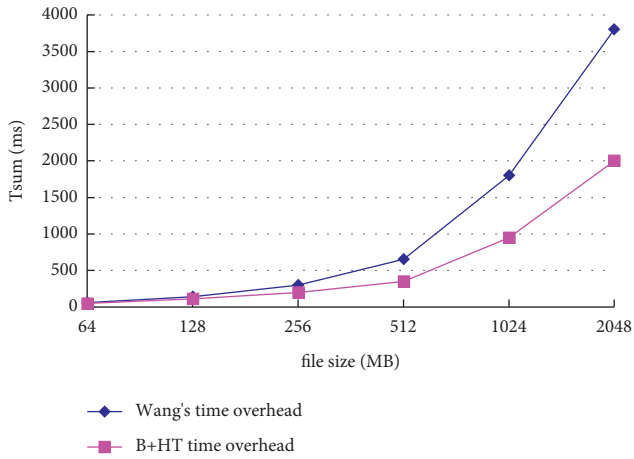

Figure 9: Time overhead under different file sizes.

mainly depends on whether HHT can provide an AWPL lower than $\log_2 M$, where M is the number of the files.

Therefore, the value of AWPL is tested. During the initialization stage, given M files with randomly allocated weights, B$^+$HT runs Algorithm 1 to establish an HHT tree. In the verification stage, in order to simulate local selection, we set up a set whose size is equal to 50. Only the files in the set can be selected. After selecting a file in the set to verify at a time, we randomly update a small number of files in the set. According to their verification weights, the files to be verified are selected by roulette-wheel. Meanwhile, a variation factor is established to slightly adjust the weights during verification. According to whether HHT enables the VFC mechanism, HHT is divided into HHT-1 (disabled) and HHT-2 (enabled). Let C be the number of tests and let L be the average authentication path. Then, the value of $L/\log_2 M$ is recorded in Table 3 to intuitively reflect the performance of HHT. The test results show that the HHT scheme is generally better than the MHT scheme. When M is small or C is large, the HHT scheme has a better effect, which is mainly attributed to the sampling variance. HHT optimizes the authentication path of the tree according to the weight. When the number of samples increases, sample indicators can better reflect the real indicators of HHT. In addition, the result that the HHT-2 scheme is better than the HHT-1 scheme shows that the VFC mechanism can better adapt to the change of verification probability.

Table 3: Time overhead comparison between HHT and MHT.

C	HHT-1				HHT-2			
	M							
	64	256	512	1024	64	256	512	1024
100	0.98	1.00	1.02	1.06	0.84	0.92	0.99	1.02
200	0.90	0.95	0.99	1.03	0.75	0.82	0.91	0.99
300	0.77	0.89	0.94	1.00	0.58	0.70	0.83	0.94
400	0.60	0.79	0.90	0.98	0.48	0.60	0.78	0.88
800	0.49	0.65	0.75	0.88	0.32	0.50	0.62	0.79

(2) B$^+$ HT Scheme. B$^+$HT and Wang's scheme [14] adopt the same signature scheme for the content verification of data blocks. The difference between them is the hash tree structure for the location authentication of data blocks. Thus, this paper mainly compares the performance of their hash trees. TVM's verification time T_{ver} is mainly related to the number of hash calculations, which depends on the height of the tree. The communication time T_{com} is mainly related to the product of the node size and the height of the tree. CSS's evidence query time T_{query} mainly depends on the node query times, i.e., the height of the tree. The complexity of location verification for a single data block is shown in Table 4. It can be seen that the verification performance of TVM and CSS in our scheme is $\log_2 K$ times higher than that of Wang's scheme [14]. When the order K of B$^+$HT equals 4, T_{ver} and T_{query} are reduced by two times. Of course, this is accompanied by the increase of T_{com}. Assuming that the number of blocks $M = 1204$, $K = 4$, and the hash length is 20 bytes, B$^+$HT only increases the traffic by 0.6 k more than Wang's scheme [14] for each verified data block. In fact, with the development of communication technology, the increased traffic can be negligible. This scheme is suitable for the case where the computing power of both TVM and CSS is weak or the burden is heavy, and the communication bandwidth is good. This case is common in the cloud environment.

From the perspective of tenant experience, the total verification response time is further evaluated because the tenant may be concerned about how long it can complete verification after it sends a query request. The total time T_{sum} is equal to the sum of T_{query}, T_{com}, and T_{ver}. The experimental parameters are as follows: $K = 4$, BLS signature, a block size of 20 bytes, and 1G bandwidth. Figure 9 shows the time overhead of the two schemes with the same detection

Table 4: Time complexity comparison between B$^+$HT and Wang's scheme about hash trees.

Entity	T_{query}	T_{com}	T_{ver}
Our B$^+$HT scheme	$O(\log_K N)$	$O((K-1)log_K M)$	$O(\log_K N)$
Wang [14]	$O(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N)$

probability of 99% under different file sizes. It can be seen that T_{sum} of our scheme is lower than that of Wang's scheme. When $K = 4$, the overhead of T_{com} increases by 1.5x, while the overhead of T_{query} and T_{ver} decreases by 2x. The test results show that the reduced time of T_{query} and T_{ver} exceeds the increased time of T_{com}. At this time, T_{query} and T_{ver} become the performance bottleneck. Moreover, the larger the file, the higher the height of the tree, and the more complex the system management, resulting in a slower search of disk I/O and a larger T_{query}.

5. Conclusions

This paper proposes a lightweight and efficient integrity verification method for cloud data file systems, which uses different integrity verification schemes for different file types to improve the verification efficiency. The Huffman Merkel tree verification scheme is used for small files, which could shorten the authentication path of small files according to the file verification frequency or user-defined file verification weight. Meanwhile, the B$^+$ hash tree integrity verification scheme is used to verify large files, which can effectively reduce the query response time of nodes. Besides, Bilinear aggregate signature is used to verify the existence of large files, which can effectively reduce the requirements of user computing power and communication bandwidth. The experimental results indicate that our scheme can perform data integrity verification well, with less computation and communication overhead and higher verification efficiency than the existing methods.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research work was supported by the National Natural Science Foundation of China under Grant nos. 62176265and 61972040.

References

[1] P. Mell and T. Grance, "The NIST definition of cloud computing," *Communications of the ACM*, vol. 53, no. 6, p. 50, 2011.

[2] S. Aldossary and W. Allen, "Data security, privacy, availability and integrity in cloud computing: issues and current solutions," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 4, pp. 485–498, 2016.

[3] J. Li and B. Li, "Erasure coding for cloud storage systems: a survey," *Tsinghua Science and Technology*, vol. 18, no. 3, pp. 259–272, 2013.

[4] D.-G. Feng, M. Zhang, Y. Zhang, and Z. Xu, "Study on cloud computing security," *Journal of Software*, vol. 22, no. 1, pp. 71–83, 2011.

[5] Z. Zhang, Z. Yang, X. Du, W. Li, X. Chen, and L. Sun, "Tenant-led ciphertext information flow control for cloud virtual machines," *IEEE Access*, vol. 9, pp. 15156–15169, 2021.

[6] Trusted Computing Group, "TCG specification architecture overview revision 2," 2013, http://www.trustedcomputinggroup.org.

[7] L. Zhou, A. Fu, S. Yu, M. Su, and B. Kuang, "Data integrity verification of the outsourced big data in the cloud environment: a survey," *Journal of Network and Computer Applications*, vol. 122, pp. 1–15, 2018.

[8] S. Tan, Y. Jia, and W. H. Han, "Research and development of provable data integrity in cloud storage," *Chinese Journal of Computers*, vol. 38, no. 1, pp. 164–177, 2015.

[9] R. C. Merkle, "Protocols for public key cryptosystems," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 122–133, Oakland, CA, USA, April 1980.

[10] R. Sailer, X. Zhang, T. Jaeger, and L. V. Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of the USENIX Security symposium*, pp. 223–238, San Diego, CA, USA, August 2004.

[11] Z.-Y. Xu, Y.-P. He, and L.-L. Deng, "Efficient remote attestation mechanism with privacy protection," *Journal of Software*, vol. 22, no. 2, pp. 339–352, 2011.

[12] Y. Deswarte, J. J. Quisquater, and A. Saïdane, "Remote integrity checking," in *Proceedings of the Working Conference on Integrity and Internal Control in Information Systems*, pp. 1–11, Fairfax, VA, USA, November 2003.

[13] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," *ACM Transactions on Information and System Security*, vol. 17, no. 4, pp. 1–29, 2015.

[14] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling public verifiability and data dynamics for storage security in cloud computing," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS 2009)*, pp. 355–370, Saint-Malo, France, September 2009.

[15] J. Shen, J. Shen, X. Chen, X. Huang, and W. Susilo, "An efficient public auditing protocol with novel dynamic structure for cloud data," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 10, pp. 2402–2415, 2017.

[16] S. Tan, L. Tan, X. Li, and Y. Jia, "An efficient method for checking the integrity of data in the cloud," *China Communications*, vol. 11, no. 9, pp. 68–81, 2014.

[17] Y. Sun, Q. Liu, X. Chen, and X. Du, "An adaptive authenticated data structure with privacy-preserving for big data stream in cloud," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3295–3310, 2020.

[18] H. Jin, K. Zhou, H. Jiang, D. Lei, R. Wei, and C. Li, "Full integrity and freshness for cloud data," *Future Generation Computer Systems*, vol. 80, pp. 640–652, 2018.

[19] N. Garg and S. Bawa, "RITS-MHT: relative indexed and time stamped Merkle hash tree based data auditing protocol for cloud computing," *Journal of Network and Computer Applications*, vol. 84, pp. 1–13, 2017.

[20] Crypto.Stanford, "The pairing-based cryptography library," 2021, https://crypto.stanford.edu/pbc/.