Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 4186243, 19 pages
https://doi.org/10.1155/2018/4186243

Research Article

Dynamic Outsourced Proofs of Retrievability Enabling Auditing Migration for Remote Storage Security

Lu Rao, Tengfei Tu, Hua Zhang, Qiaoyan Wen, and Jia Xiao

State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China

Correspondence should be addressed to Hua Zhang; zhanghua_288@bupt.edu.cn

Received 26 December 2017; Accepted 18 February 2018; Published 9 May 2018

Academic Editor: Kok-Seng Wong

Remote data auditing service is important for mobile clients to guarantee the intactness of their outsourced data stored at cloud side. To relieve mobile client from the nonnegligible burden incurred by performing the frequent data auditing, more and more literatures propose that the execution of such data auditing should be migrated from mobile client to third-party auditor (TPA). However, existing public auditing schemes always assume that TPA is reliable, which is the potential risk for outsourced data security. Although Outsourced Proofs of Retrievability (OPOR) have been proposed to further protect against the malicious TPA and collusion among any two entities, the original OPOR scheme applies only to the static data, which is the limitation that should be solved for enabling data dynamics. In this paper, we design a novel authenticated data structure called bv23Tree, which enables client to batch-verify the indices and values of any number of appointed leaves all at once for efficiency. By utilizing bv23Tree and a hierarchical storage structure, we present the first solution for Dynamic OPOR (DOPOR), which extends the OPOR model to support dynamic updates of the outsourced data. Extensive security and performance analyses show the reliability and effectiveness of our proposed scheme.

1. Introduction

In today's information era of data explosion, it is an inevitable trend for most people to have the ever-increasing big data storage demands. Storage outsourcing through the cloud has become a promising technology paradigm that populates the recent literatures [1–3] and has been regarded as a faster profit growth point [4] by various IT industry giants (e.g., Google Drive, Microsoft OneDrive, and Amazon EC2 and S3). Cloud storage not only allows mobile clients to access their outsourced data from anywhere at any time, but also provides mobile clients with many benefits such as the inexpensive storage cost and elastic configuration of the storage capacity, which attract more and more mobile clients (e.g., smartphones and laptops) to join the cloud for the convenient lifestyle.

However, at the side of the cloud storage server (CSS), there still exist all kinds of internal and external threats against the storage security of outsourced data, such as Byzantine failures, the monetary reasons, and the hacker attacks [5, 6]. So, it is well known that CSS would be considered as a malicious entity that might try to hide the accident when data loss occurs, or even deliberately delete client's data for saving storage cost. In this case, for the mobile client who has not actually possessed her data after storage outsourcing, an urgent requirement is how to guarantee the *correctness* and *retrievability* of the outsourced data. Here, correctness guarantee means that any appointed data returned from CSS should be the latest version of the authentic data, and retrievability guarantee means that the whole outsourced data can be correctly retrieved by the client without any data loss.

Based on the application of erasure code and the periodic auditing against CSS, the security model Proof of Retrievability (POR) [7, 8] is defined to offer client-side devices the above two guarantees in the context of malicious CSS. However, given the fact that most mobile clients only have a limited capacity so that these clients are unlikely to keep online all the time to perform the frequent auditing, various public auditing schemes are proposed [5, 6, 9–11] for auditing migration, which enables mobile client to free herself by moving the heavy auditing tasks to a third-party auditor (TPA). But existing public schemes rely on the hypothesis

that TPA is trusted to complete the migrated auditing tasks, meaning that these schemes do not provide any security guarantee to resist a malicious TPA that might break the auditing protocols, which is exactly the potential risk that has not been covered by current public auditing schemes [12].

The first Outsourced Proof of Retrievability (OPOR) solution, proposed by Armknecht et al. and called Fortress [12], is a stronger security model to protect against any malicious entity (e.g., malicious TPA) and against collusion among any two entities. Fortress enables auditing migration, but it is just a static scheme. Supporting dynamic updates is an essential requirement for numerous practical cloud storage applications [13]. Although all kinds of authenticated data structures [6, 9, 13–16] have been proposed to support data dynamics, there still exist research gaps in these structures. For one thing, most existing dynamic authenticated data structures [9, 14–16] are designed based on the Merkle Hash Tree (MHT). Unfortunately, the use of MHT is not efficient in some cases since MHT is an unbalanced tree. For example, the height of MHT will increase linearly when many new data blocks are continuously inserted in the same leaf position, so in this worst case the expected $O(\log n)$ performance cannot be ensured for authenticating the index and value of any appointed leaf node via MHT. For another, although some other authenticated structures such as rb23Tree [6] and Skip List [13] are proposed for dynamism, when there is a need to verify multiple leaf nodes of different data blocks (i.e., authenticate the indices and values of these leaf nodes), all above-mentioned dynamic methods merely adopt the straightforward way of verifying these different leaf nodes one by one with their respective proof paths. Since the verification upon a single proof path is of $O(\log n)$ bandwidth and computation costs, as the number of verified leaf nodes increases, it is clearly not an efficient way to separately verify that many leaf nodes one by one. Although a balanced authenticated data structure (e.g., the rb23Tree [6]) constructed upon the balanced tree can avoid the worst case of MHT as discussed above, however, to the best of our knowledge, there is no such balanced authenticated structure that can deal with the problem of how to efficiently batch-verify any number of appointed leaf nodes altogether, which is a limitation that should be further addressed in this paper.

Furthermore, to support data dynamics when applying erasure code, the scheme of [6] is based on the way of local coding, that is, encoding each raw data block individually to ensure that an update upon any raw block only affects a small amount of the encoded blocks. However, such local coding solution is vulnerable to the selective deletion attack from malicious CSS [14], because once a targeted raw block has been updated in the case of local coding, CSS can learn which congenetic encoded blocks correspond to this targeted block. Then, CSS can selectively delete these congenetic encoded blocks to actually cause the loss of the targeted block, and simultaneously CSS can pass the data auditing with significant probability, since the auditing relies on the sampling technology that can hardly cover these selectively deleted encoded blocks if the size of deleted data is tiny. Although both the private and the public Dynamic POR (DPOR) schemes [14] are proposed to resist the above selective deletion attack, the direct application of these two DPOR schemes into OPOR model will result in security or efficiency problems. On the one hand, within the private DPOR of [14], the auditing can only be executed using client's secret key. But TPA is prohibited from obtaining such secret in OPOR, or otherwise the malicious TPA might share client's secret key with malicious CSS [12] so that CSS can break the auditing protocols without actually holding the outsourced data. On the other hand, in order to support public auditing, the public DPOR of [14] cannot apply the blockless verification technique [8, 9, 13, 17] that combines multiple challenged blocks into a single aggregated block for efficiency, so it has to use the straightforward way of requiring TPA to retrieve all randomly challenged actual blocks during each POR audit. As shown in [9], this straightforward way could lead to a large communication overhead and thus is inefficient and should be avoided.

From the above, there will be various problems if the current dynamic schemes are directly ported to the OPOR model. In this paper, to solve these mentioned problems, we propose a concrete Dynamic Outsourced Proofs of Retrievability (DOPOR) scheme enabling auditing migration. DOPOR not only can defend against the malicious TPA and collusion, but also can enable efficient data dynamics under the setting of erasure code. Specifically, our contributions are summarized as follows.

(1) Different from traditional authenticated structures that only have the ability to verify different leaves one by one, we propose the novel authenticated data structure called bv23Tree, which is based on the balanced 2-3 tree to ensure the logarithmic complexity in any case of updates and simultaneously enables the verifier to batch-verify the indices and values of multiple appointed leaves all at once for efficiency.

(2) To defend against the selective deletion attack, we utilize a hierarchical storage structure with the same-sized levels for the unified management of outsourced encoded data and encoded update operations. According to this hierarchical structure and the bv23Tree, we resolve the open questions of [12] by transforming another secure public POR into the OPOR model and designing an appropriate dynamic scheme to efficiently integrate dynamic updates with OPOR.

(3) We analyze the security of our solution and conduct an extensive experimental study. The experimental results demonstrate the effectiveness of our scheme.

The rest of this paper is organized as follows: Section 2 states the background and introduces the architecture and system model of DOPOR. Section 3 shows the novel dynamic structure bv23Tree in detail, based on which we present detailed DOPOR solution in Section 4. Section 5 provides the security analysis, and Section 6 evaluates the experimental performance. Section 7 overviews the related work. Finally, this paper is concluded in Section 8.

2. Background and System Architecture

2.1. Problem Statement. We begin with the background of OPOR, as shown in [12]. There are three entities involved

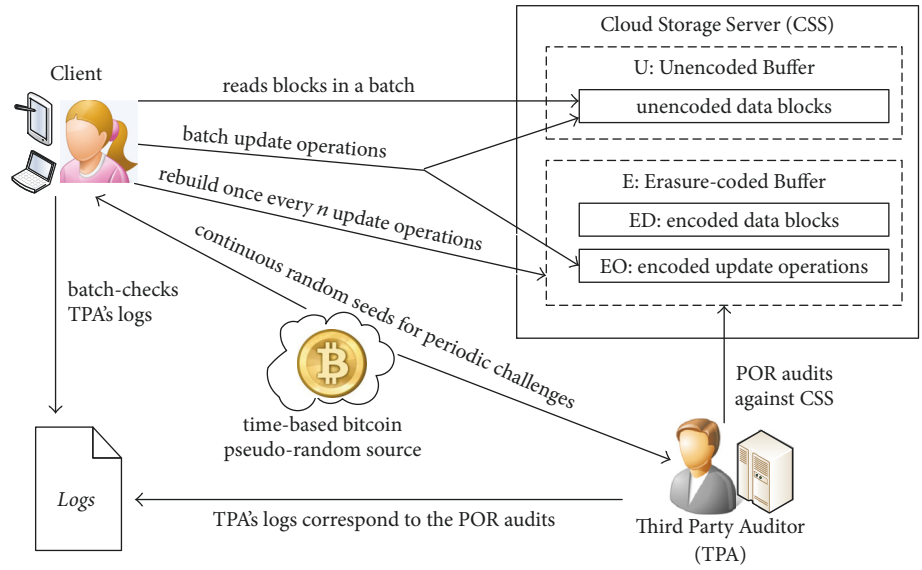

Figure 1: Dynamic OPOR (DOPOR) architecture.

in an outsourced auditing environment: mobile client (i.e., data owner), cloud storage server (CSS), and third-party auditor (TPA). Because of their limited local storage capacity, mobile clients are motivated to outsource their large data files to CSS and then can make use of various on-demand cloud storage services. However, because CSS might be misbehaving, it is very important to design the periodic remote data auditing mechanism against CSS, which enables the mobile client to have the assurance that her outsourced data is always available and can be completely retrieved from CSS if necessary. Further, to liberate the mobile client from struggling with the endless online data auditing, OPOR also introduces TPA that has the expertise and ability to perform the above frequent auditing tasks on behalf of the mobile client, and then the mobile client can be offline to rest most of the time.

Although OPOR model has the same three entities as existing public auditing model [9, 10, 15, 16], one of the main differences between the two is that *TPA might be also malicious within OPOR* [12]. In other words, TPA might violate the auditing protocols, for example, by claiming that he has honestly performed all past auditing, but actually he tells a lie. *Furthermore, any two entities might be in collusion within OPOR* [12]. For example, firstly, malicious CSS might collude with malicious TPA to deceive the honest client when outsourced data has been lost. Secondly, malicious client might collude with malicious CSS to frame the honest TPA, by asserting that TPA did not correctly perform the required auditing work to bring TPA into the compensation lawsuit. Since any entity might be malicious, in order to solve the problem of securely sampling the periodic challenges for frequent auditing, the time-based bitcoin pseudorandom source is introduced into the OPOR model. As demonstrated in [12], due to the fact that the bitcoin pseudorandom source cannot be manipulated by any entity, it is secure to be used for generating the continuous random seeds for periodic challenges.

OPOR inherits the retrievability guarantee of POR [8] by applying erasure code, meaning that the client can retrieve the whole outsourced data in case of minor data corruption. However, when both of the data dynamics and erasure code are considered, the problem of how to efficiently perform the updates is intractable. Within the first OPOR scheme Fortress [12], client's original data file (including n raw data blocks) is entirely encoded before outsourcing, so an update operation upon any single raw data block will affect the whole outsourced encoded data. In this case, the only way for client is to download and decode the whole outsourced encoded data and then encode and upload all the data again after performing the updates, which means unbearable bandwidth and computation costs. This is why Fortress is just a static scheme that cannot support efficient dynamic updates.

2.2. Dynamic OPOR (DOPOR) Architecture. The representative DOPOR architecture is presented in Figure 1. Based on the bitcoin source that controls the random sampling of periodic challenges, once TPA accepts the migrated auditing tasks from the client, TPA must generate the corresponding logs after he completes each specified POR audit against CSS. In this case, the client is able to check TPA's work at any point in time by verifying TPA's logs, and then she can judge that if TPA did his auditing work correctly in the past. As shown in [12], such client's checking against TPA can be much less frequent than the TPA's POR audits against CSS, since the client can batch-check a number of accumulated TPA logs all at once.

To support dynamic updates, we apply a similar idea to [14] within our solution, which is to place all accumulated update operations into an erasure-coded buffer at CSS side, rather than immediately executing these update operations upon the outsourced encoded data. As shown in Figure 1, the CSS-side storage is organized in two different buffers

denoted with U (i.e., unencoded buffer) and E (i.e., erasure-coded buffer). Buffer U will independently store an up-to-date copy of all the raw data blocks, which are organized by our proposed bv23Tree, to support the efficient batch reads from cloud storage without struggling with the erasure code. On the other side, buffer E is further divided into two parts, ED and EO, which store the whole outsourced original encoded data blocks and all the accumulated encoded update operations, respectively. In case of data loss, client can recover the up-to-date copy of the whole outsourced raw data by decoding the entire buffer E and combining both of ED and EO, so the periodic POR audits only need to be performed upon buffer E for the retrievability guarantee.

As will be described in Section 4.1, both ED and EO constitute a complete hierarchical storage structure where all levels have the same size, which are different from the levels of exponentially growing capacity in [14]. After the client performs a batch of update operations upon buffer U, benefitting from the same-sized levels of DOPOR, this batch of update operations can be wholly encoded and then directly placed into EO to fill up the corresponding level for improved computation cost, without executing the rebuilding of a level as in [14] that incurs $O(\log n)$ amortized cost for each update operation. More importantly, based on the same-sized levels, our DOPOR solution can build upon the public verification POR scheme of [8], the aggregation technique of which provides the support for the client's checking against malicious TPA.

Finally, after every n update operations, when the size of EO grows to the same size as ED, buffer E will be rebuilt. The rebuilding of E will rewrite the whole ED with the encoded version of all the up-to-date raw data blocks and meanwhile empty the whole EO. Since E is only rebuilt once in every n update operations, the amortized complexity of rebuilding E will be $O(1)$ per update operation. However, different from the existing schemes [14, 18, 19] that require $O(n)$ client-side temporary memory for such rebuilding, as shown in Section 4.3, based on the ability of batch reads and the same-sized levels, the client of DOPOR only requires $O(\lambda)$ client-side memory to gradually rebuild E, where λ is the security parameter that is independent of the data size n. So, DOPOR further reduces the required client's memory when rebuilding and thus is suitable for the client-side mobile devices.

2.3. System Model. Formally, the complete definition of DOPOR system can be described by the following ten protocols:

(i) **GenKey**$(1^{\mathscr{k}}) \rightarrow \{\mathrm{sk}_{\mathbb{E}}, \mathrm{pk}_{\mathbb{E}}\}$: when inputting the security parameter $\mathscr{k}$, this protocol outputs a pair of public-private keys for each entity $\mathbb{E} \in \{\text{client}, \text{CSS}, \text{TPA}\}$.

(ii) **GenTags**$(\mathrm{sk}_{\text{client}}, F) \rightarrow \{B, \Phi\}$: when inputting the client's secret key $\mathrm{sk}_{\text{client}}$ and the original file F that is an ordered set of raw data blocks $\{m_i\}$, this protocol encodes F into the encoded file B and outputs B as an ordered set of codeword blocks $\{b_i\}$. It also outputs the tags set $\Phi = \{\sigma_i\}$, where each σ_i is computed based on $\mathrm{sk}_{\text{client}}$ and b_i.

(iii) **OutsourceData**$(F, B, \Phi) \rightarrow \{\Psi, \mathbb{P}\}$: when inputting the original file F, the encoded file B, and tags Φ, it outputs the bv23Tree Ψ that is constructed based on F. After outsourcing all the input data and Ψ to CSS, this protocol also outputs the public parameters set $\mathbb{P}$ for POR audits.

(iv) **ReadBlocks**$(x_{\text{root}}, \Pi, \mathscr{st}) \rightarrow \{\mathscr{M}, \text{false}\}$: when inputting the root hash x_{root} of the bv23Tree, the set of any k blocks indices $\Pi = \{a_1, a_2, \ldots, a_k\}$, and the CSS state $\mathscr{st}$, this protocol outputs the appointed data blocks set $\mathscr{M} = \{m_{a_1}, m_{a_2}, \ldots, m_{a_k}\}$, or false otherwise.

(v) **PerformUpdates**$(x_{\text{root}}, \mathcal{SO}, \mathrm{sk}_{\text{client}}, \mathscr{st}, \mathscr{p}) \rightarrow \{(x_{\text{root}^*}, \mathscr{st}^*, \mathscr{p}^*), \text{false}\}$: when inputting the tree root hash x_{root}, the set of update operations $\mathcal{SO}$, client's secret key $\mathrm{sk}_{\text{client}}$, the CSS state $\mathscr{st}$, and the state pointer $\mathscr{p}$ that is related to $\mathscr{st}$, it outputs a new root hash x_{root^*}, a new CSS state $\mathscr{st}^*$, and a new pointer $\mathscr{p}^*$ showing that all the operations in $\mathcal{SO}$ are correctly executed in a batch, or false otherwise.

(vi) **Rebuild**$(x_{\text{root}}, \mathrm{sk}_{\text{client}}, \mathscr{st}, \mathscr{p}) \rightarrow \{(\mathscr{st}^*, \mathscr{p}^*), \text{false}\}$: when inputting the tree root hash x_{root}, client's secret key $\mathrm{sk}_{\text{client}}$, CSS state $\mathscr{st}$, and state pointer $\mathscr{p}$, it outputs a new CSS state $\mathscr{st}^*$ and a new pointer $\mathscr{p}^*$ showing that the rebuilding is completed, or false otherwise.

(vii) **RandomChal**$(\mathbb{B}, t, \mathbb{P}) \rightarrow \{\mathrm{Q}^{(t)}\}$: when inputting the bitcoin source $\mathbb{B}$, the time t, and the public parameters $\mathbb{P}$, it outputs a bitcoin-based challenge $Q^{(t)}$ for POR audit.

(viii) **GenProof**$(Q^{(t)}, \mathscr{st}) \rightarrow \{\rho^{(t)}\}$: when inputting the challenge $Q^{(t)}$ and the CSS state $\mathscr{st}$, it outputs CSS's proof $\rho^{(t)}$ to enable TPA to perform a POR audit.

(ix) **PORAudit**$(\mathbb{P}, Q^{(t)}, \rho^{(t)}) \rightarrow \{\Lambda^{(t)}, \mathfrak{D}_{\text{TPA}}\}$: when inputting public parameters set $\mathbb{P}$, challenge $Q^{(t)}$, and CSS's proof $\rho^{(t)}$, it outputs TPA's log $\Lambda^{(t)}$ and decision $\mathfrak{D}_{\text{TPA}}$. $\mathfrak{D}_{\text{TPA}}$ is true if TPA audit passes, or false otherwise.

(x) **CheckLogs**$(\mathbb{B}, T, \Lambda^{(T)}, \mathbb{P}, \mathrm{sk}_{\text{client}}) \rightarrow \{\mathfrak{D}_{\text{client}}\}$: when inputting the bitcoin source $\mathbb{B}$, a point-in-time set $T = \{t_1, t_2, \ldots, t_k\}$, the corresponding TPA's logs set $\Lambda^{(T)} = \{\Lambda^{(t)}\}_{t \in T}$, the public parameters $\mathbb{P}$, and client's secret key $\mathrm{sk}_{\text{client}}$, this protocol outputs a client's decision $\mathfrak{D}_{\text{client}}$. $\mathfrak{D}_{\text{TPA}}$ is true if client's batch-checking upon $\Lambda^{(T)}$ can pass, or false otherwise.

3. Balanced Authenticated Data Structure

Within the rb23Tree of [6], for verifying a leaf, the client must retrieve from the adversary a corresponding proof path that consists of $O(\log n)$ *marks*. However, the limitation of rb23Tree method is that a lot of duplicated information will exist in the proof paths of different leaves, which will waste too much communication cost when the client verifies many leaves one after another by retrieving different proof paths. To solve this problem, we propose the *batch-verifications 2-3 Tree*, called *bv23Tree*, for efficiency.

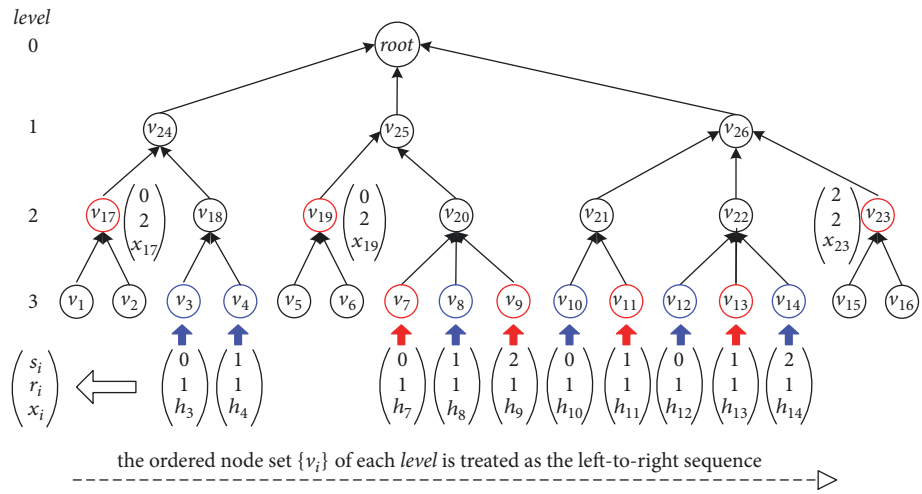

FIGURE 2: An example of bv23Tree. Besides the verified ordered leaves set $\mathscr{L} = \{v_3, v_4, v_8, v_{10}, v_{12}, v_{14}\}$, the verifier only needs to retrieve the necessary auxiliary tree nodes (colored in red), which are organized into a special table structure as will be shown later. Then, the verifier can batch-verify the indices and values of these leaves in $\mathscr{L}$ all together.

3.1. Batch-Verifications 2-3 Tree. Following the definition of 2-3 Tree [20], each nonleaf node of bv23Tree can have two or three children. Let F be an original data file consisting of n raw blocks $F = \{m_1, m_2, \ldots, m_n\}$. With the hash function $h(\cdot)$, the bv23Tree on file$F$ can be constructed by storing at each tree node v a 3-element tuple (s_v, r_v, x_v), defined as follows:

(i) s_v is the *status* of node v. Let $\mathscr{P}$ denote the parent node of v. Let $\mathrm{ch}_1, \mathrm{ch}_2, \mathrm{ch}_3$ be the three left-to-right children of $\mathscr{P}$, respectively. Specifically, if $\mathscr{P}$ only has two children, then $\mathrm{ch}_3 = \text{null}$. So, s_v is defined as

$$s_v = \begin{cases} 0, & \text{if } v \text{ is } \mathrm{ch}_1; \\ 1, & \text{if } v \text{ is } \mathrm{ch}_2; \\ 2, & \text{if } \mathrm{ch}_3 \neq \text{null}, \ v \text{ is } \mathrm{ch}_3. \end{cases} \quad (1)$$

(ii) r_v is the *rank value* of node v, which is similar to the concept defined in existing rank-based MHT scheme [16]. Namely, r_v stores the number of leaf nodes that belong to the subtree with node v as the root. If v is a leaf node, we define $r_v = 1$.

(iii) x_v represents the *authentication hash value* of node v. The value of r_v is defined with different cases.

Case 0. $v = \text{null}$; then

$$x_v = \text{null}. \quad (2)$$

Case 1. v is the ith leaf node; then

$$x_v = h(m_i). \quad (3)$$

Case 2. v is a nonleaf node with children $\mathrm{ch}_1, \mathrm{ch}_2$, and ch_3 as above (sometimes ch_3 will be null):

$$x_v = h\left(r_v \parallel x_{\mathrm{ch}_1} \parallel x_{\mathrm{ch}_2} \parallel x_{\mathrm{ch}_3}\right), \quad (4)$$

where $\parallel$ denotes the concatenation operation.

We show an example of bv23Tree in Figure 2, which is constructed on 16 file blocks $\{m_i\}_{1 \le i \le 16}$. Next, we use the concise shorthand for some symbols. Given a bv23Tree node v_i, we use s_i, r_i, and x_i to denote s_{v_i}, r_{v_i}, and x_{v_i}, respectively, so each tree node v_i and its corresponding 3-element tuple (s_i, r_i, x_i) are not distinguished. And we also use h_i to denote the hash value $h(m_i)$ for convenience.

3.2. Batch Queries. The integrity (i.e., authenticity and freshness) of file blocks can be protected by the corresponding hash values h_i $(1 \le i \le n)$ stored in leaves, while the integrity of the leaves themselves will be protected by bv23Tree. Now, suppose that the whole file blocks $\{m_1, m_2, \ldots, m_n\}$ and a bv23Tree on all blocks have been stored at CSS. Client wants to verify the integrity of any k ordered file blocks $m_{a_1}, m_{a_2}, \ldots, m_{a_k}$ $(1 \le a_1 < a_2 < \cdots < a_k \le n)$ read from CSS and thus batch-queries CSS by issuing the ordered indices set $\Pi = \{a_1, a_2, \ldots, a_k\}$ that appoints k ordered leaves $\{v_{a_1}, v_{a_2}, \ldots, v_{a_k}\}$. Then, CSS calls Algorithm 1 to generate the corresponding *proof table* and responds to client with the k appointed leaves and their proof table.

An example of proof table is shown in Table 1. Without loss of generality, suppose the number of the levels of a bv23Tree is l. Due to the property of balanced tree, the number l must be $O(\log n)$ complexity. Let $T_{k \times l}$ denote the proof table of any k appointed leaves; then $T_{k \times l}$ has the following characteristics:

(i) $T_{k \times l}$ contains k rows and l columns, respectively (i.e., $T_{k \times l}$ consists of $k \times l$ items).

(ii) Each item $T_{i,j}$ $(1 \le i \le k, 1 \le j \le l)$ can have one or two components, and each component can be a tree node v_i or a mark θ $(2 \le \theta \le k)$ or null. Since θ only denotes a pointer that points to the θth row of the table $T_{k \times l}$ itself, the communication cost of a mark θ is little when compared to the cost of a node v_i.

(iii) The more the leaves are batch-verified, the more the null that exists in $T_{k \times l}$. And no matter how many

Proof_Table$(\Psi, \Pi) \rightarrow T_{k\times l}$. With the bv23Tree Ψ of l levels, and the k ordered indices set $\Pi = \{a_1, a_2, \ldots, a_k\}$ that appoint k ordered leaves $\{v_{a_1}, v_{a_2}, \ldots, v_{a_k}\}$, this algorithm generates the corresponding proof table $T_{k\times l}$.

(1) initialize node array $A[1 \cdots k]$ by $A[i] \leftarrow v_{a_i}$ $(i = 1, 2, \ldots, k)$;
(2) {$A[i]$ tracks the parent of the current node v as below}
(3) $j \leftarrow 1$; {j tracks the column of the proof table $T_{k\times l}$}
(4) **while** $j \leq l$ **do**
(5) **for** $i = 1, 2, \ldots, k$ **do**
(6) **if** $A[i]$ = null **then**
(7) $T_{i,j} \leftarrow$ null;
(8) **else** {$A[i] \neq$ null}
(9) {let v denote the current tree node stored in $A[i]$}
(10) **if** node v only has one sibling node, which is denoted by sv **then**
(11) **if** sv exists in $A[\theta]$, in this case $i < \theta \leq k$ **then**
(12) $T_{i,j} \leftarrow \theta$; $A[\theta] \leftarrow$ null;
(13) **else** {sv does not exist in current array $A[\ ]$}
(14) $T_{i,j} \leftarrow sv$;
(15) **end if**
(16) **else** {node v has two sibling nodes, which are denoted by (sv_1, sv_2)}
(17) {note that the sequence of (sv_1, sv_2) must follow the left-to-right principle as in Figure 2, e.g., two siblings of node v_8 must be denoted by (v_7, v_9)}
(18) **if** only sv_1 (or sv_2) exists in $A[\theta]$ **then**
(19) $T_{i,j} \leftarrow (\theta, sv_2)$ (or $T_{i,j} \leftarrow (sv_1, \theta)$);
(20) $A[\theta] \leftarrow$ null;
(21) **else if** both sv_1 and sv_2 exist in $A[\theta_1]$ and $A[\theta_2]$, respectively, **then**
(22) $T_{i,j} \leftarrow (\theta_1, \theta_2)$; $A[\theta_1] \leftarrow$ null; $A[\theta_2] \leftarrow$ null;
(23) **else** both sv_1 and sv_2 do not exist in current array $A[\ ]$
(24) $T_{i,j} \leftarrow (sv_1, sv_2)$;
(25) **end if**
(26) **end if**
(27) $A[i] \leftarrow$ the parent node of the current node v;
(28) **end if**
(29) **end for**
(30) $j \leftarrow j + 1$;
(31) **end while**
(32) **return** the proof table $T_{k\times l}$;

Algorithm 1: Algorithm for generating proof table $T_{k\times l}$.

Table 1: An example of proof table.

Table item $T_{i,j}$	$j = 1$	$j = 2$	$j = 3$
$i = 1$ (row 1)	2	v_{17}	$(3, 4)$
$i = 2$ (row 2)	Null	Null	Null
$i = 3$ (row 3)	(v_7, v_9)	v_{19}	Null
$i = 4$ (row 4)	v_{11}	$(5, v_{23})$	Null
$i = 5$ (row 5)	$(v_{13}, 6)$	Null	Null
$i = 6$ (row 6)	Null	Null	Null

The corresponding proof table $T_{6\times 3}$ is generated based on the bv23Tree and the appointed ordered leaves set $\mathscr{L} = \{v_3, v_4, v_8, v_{10}, v_{12}, v_{14}\}$ of Figure 2.

leaves are batch-verified, each necessary auxiliary tree node v_i only appears once in $T_{k\times l}$.

In the context of the batch verifications upon k appointed leaves, compared to the proof path method as in [6], the communication cost of the proof table $T_{k\times l}$ is much less than that of transferring different proof paths of k leaves, respectively. This is because the proof table avoids the limitation of proof paths that the repetitive node information (e.g., node hash values) will exhibit in the proof paths of different leaves with high probability, so there is a plenty of null in the proof table to save the communication cost. Furthermore, compared to the 8-element tuple *mark* in proof path of [6], each tree node v_i included in the proof table is only related to a 3-element tuple, which further reduces the communication cost.

3.3. Batch Verifications. Upon receiving from CSS the k appointed leaves set $\mathscr{L} = \{v_{a_1}, v_{a_2}, \ldots, v_{a_k}\}$ and the required proof table $T_{k\times l}$, client can run Algorithm 2 to batch-verify the indices and values of these k leaves in $\mathscr{L}$ all at once, by using her local metadata x_{root}. An example of batch verifications upon multiple leaves is shown in Table 2.

Within Algorithm 2, the function Append(·) is to merge two different sets of numbers while preserving the order of these numbers. For example, given two sets $\mathscr{IS}_1 = \{4\}$ and $\mathscr{IS}_2 = \{6, 8, 10\}$, then Append$(\mathscr{IS}_1, \mathscr{IS}_2) = \{4, 6, 8, 10\}$. In addition, the operator notation "$\oplus$" is to add a number to every element of a set. For example, let $\mathscr{IS}_3 = \{4, 6, 8, 10\}$;

Batch_Verify($x_{\text{root}}, \Pi, \mathscr{L}, T_{k\times l}$) $\rightarrow$ {true, false}. This algorithm can batch-verify not only the hash values of all k ordered leaves $\mathscr{L} = \{v_{a_1}, v_{a_2}, \ldots, v_{a_k}\}$ provided by CSS, but also that the indices of these k leaves are exactly matched with the appointed indices set $\Pi = \{a_1, a_2, \ldots, a_k\}$.

(1) **for** each leaf $v_{a_i} \coloneqq (s_{a_i}, r_{a_i}, x_{a_i})$, $i = 1, 2, \ldots, k$, **do**
(2) initialize $\mathscr{R}_i \leftarrow r_{a_i}$; $\mathscr{H}_i \leftarrow x_{a_i}$; $\mathscr{IS}_i \leftarrow \{1\}$;
(3) **end for**
(4) initialize *Row_Set* $\leftarrow \{1, 2, \ldots, k\}$;
(5) **for** $j = 1, 2, \ldots, l$ **do**
(6) **for** each i in *Row_Set* **do**
(7) **if** item $T_{i,j}$ only has one component $\mathscr{C}$ **then**
(8) **call** *Deal_One_Component* ($\mathscr{C}$);
(9) **else** {$T_{i,j}$ has two components ($\mathscr{C}_1, \mathscr{C}_2$)}
(10) **call** *Deal_Two_Components* ($\mathscr{C}_1, \mathscr{C}_2$);
(11) **end if**
(12) **end for**
(13) **end for**
(14) **if** $\mathscr{H}_1 = x_{\text{root}}$ and $\mathscr{IS}_1 = \{a_1, a_2, \ldots, a_k\}$ **then**
(15) **return** true;
(16) **else**
(17) **return** false;
(18) **end if**

Function *Deal_One_Component* ($\mathscr{C}$)

(1) **if** $\mathscr{C}$ is null **then**
(2) **continue**;
(3) **else if** $\mathscr{C}$ is a node $v_t \coloneqq (s_t, r_t, x_t)$ **then**
(4) $\mathscr{R}_i \leftarrow \mathscr{R}_i + r_t$;
(5) **if** $s_t = 0$ **then**
(6) $\mathscr{IS}_i \leftarrow \mathscr{IS}_i \oplus r_t$;
(7) $\mathscr{H}_i \leftarrow h(\mathscr{R}_i \parallel x_t \parallel \mathscr{H}_i)$;
(8) **else** {$s_t = 1$}
(9) $\mathscr{H}_i \leftarrow h(\mathscr{R}_i \parallel \mathscr{H}_i \parallel x_t)$;
(10) **end** if
(11) **else** {$\mathscr{C}$ is a mark θ}
(12) $\mathscr{IS}_\theta \leftarrow \mathscr{IS}_\theta \oplus \mathscr{R}_i$;
(13) $\mathscr{R}_i \leftarrow \mathscr{R}_i + \mathscr{R}_\theta$;
(14) $\mathscr{H}_i \leftarrow h(\mathscr{R}_i \parallel \mathscr{H}_i \parallel \mathscr{H}_\theta)$;
(15) $\mathscr{IS}_i \leftarrow \text{Append}(\mathscr{IS}_i, \mathscr{IS}_\theta)$;
(16) **delete** θ from the *Row_Set*;
(17) **end if**

Function *Deal_Two_Components* ($\mathscr{C}_1, \mathscr{C}_2$)

(1) **if** $\mathscr{C}_1$ and $\mathscr{C}_2$ are two nodes v_t and v_τ, respectively, **then**
(2) {note that $v_t \coloneqq (s_t, r_t, x_t)$, $v_\tau \coloneqq (s_\tau, r_\tau, x_\tau)$}
(3) assign $\mathbb{S} \leftarrow \mathbb{S} \in \{0, 1, 2\}$ and $\mathbb{S} \notin \{s_t, s_\tau\}$;
(4) $\mathscr{R}_i \leftarrow \mathscr{R}_i + r_t + r_\tau$;
(5) **if** $\mathbb{S} = 0$ **then**
(6) $\mathscr{H}_i \leftarrow h(\mathscr{R}_i \parallel \mathscr{H}_i \parallel x_t \parallel x_\tau)$;
(7) **else if** $\mathbb{S} = 1$ **then**
(8) $\mathscr{IS}_i \leftarrow \mathscr{IS}_i \oplus r_t$; $\mathscr{H}_i \leftarrow h(\mathscr{R}_i \parallel x_t \parallel \mathscr{H}_i \parallel x_\tau)$;
(9) **else** {$\mathbb{S} = 2$}
(10) $\mathscr{IS}_i \leftarrow \mathscr{IS}_i \oplus (r_t + r_\tau)$; $\mathscr{H}_i \leftarrow h(\mathscr{R}_i \parallel x_t \parallel x_\tau \parallel \mathscr{H}_i)$;
(11) **end if**
(12) **else if** $\mathscr{C}_1$ is a node v_t, and $\mathscr{C}_2$ is a mark θ, **then**
(13) {in terms of the left-to-right principle, the value of s_t can only be 0 or 1 in this case}
(14) **if** $s_t = 0$ **then**
(15) $\mathscr{IS}_\theta \leftarrow \mathscr{IS}_\theta \oplus \mathscr{R}_i$;
(16) $\mathscr{IS}_i \leftarrow \text{Append}(\mathscr{IS}_i, \mathscr{IS}_\theta)$;
(17) $\mathscr{IS}_i \leftarrow \mathscr{IS}_i \oplus r_t$;
(18) $\mathscr{R}_i \leftarrow \mathscr{R}_i + r_t + \mathscr{R}_\theta$;
(19) $\mathscr{H}_i \leftarrow h(\mathscr{R}_i \parallel x_t \parallel \mathscr{H}_i \parallel \mathscr{H}_\theta)$;
(20) **else** {$s_t = 1$}

Algorithm 2: Continued.

(21) $\mathcal{IS}_\theta \leftarrow \mathcal{IS}_\theta \oplus (\mathcal{R}_i + r_\ell)$;
(22) $\mathcal{IS}_i \leftarrow \text{Append}(\mathcal{IS}_i, \mathcal{IS}_\theta)$;
(23) $\mathcal{R}_i \leftarrow \mathcal{R}_i + r_\ell + \mathcal{R}_\theta$;
(24) $\mathcal{H}_i \leftarrow h(\mathcal{R}_i \parallel \mathcal{H}_i \parallel x_\ell \parallel \mathcal{H}_\theta)$
(25) **end if**
(26) delete θ from the *Row_Set*;
(27) **else if** $\mathcal{C}_1$ is a mark θ, and $\mathcal{C}_2$ is a node v_ℓ, **then**
(28) $\mathcal{IS}_\theta \leftarrow \mathcal{IS}_\theta \oplus \mathcal{R}_i$;
(29) $\mathcal{IS}_i \leftarrow \text{Append}(\mathcal{IS}_i, \mathcal{IS}_\theta)$;
(30) $\mathcal{R}_i \leftarrow \mathcal{R}_i + \mathcal{R}_\theta + r_\ell$;
(31) $\mathcal{H}_i \leftarrow h(\mathcal{R}_i \parallel \mathcal{H}_i \parallel \mathcal{H}_\theta \parallel x_\ell)$;
(32) delete θ from the *Row_Set*;
(33) **else** {$\mathcal{C}_1$ and $\mathcal{C}_2$ are two marks θ_1 and θ_2, respectively}
(34) $\mathcal{IS}_{\theta_2} \leftarrow \mathcal{IS}_{\theta_2} \oplus \mathcal{R}_{\theta_1}$;
(35) $\mathcal{IS}_{\theta_1} \leftarrow \text{Append}(\mathcal{IS}_{\theta_1}, \mathcal{IS}_{\theta_2})$
(36) $\mathcal{IS}_{\theta_1} \leftarrow \mathcal{IS}_{\theta_1} \oplus \mathcal{R}_i$;
(37) $\mathcal{IS}_i \leftarrow \text{Append}(\mathcal{IS}_i, \mathcal{IS}_{\theta_1})$;
(38) $\mathcal{R}_i \leftarrow \mathcal{R}_i + \mathcal{R}_{\theta_1} + \mathcal{R}_{\theta_2}$;
(39) $\mathcal{H}_i \leftarrow h(\mathcal{R}_i \parallel \mathcal{H}_i \parallel \mathcal{H}_{\theta_1} \parallel \mathcal{H}_{\theta_2})$;
(40) delete θ_1 and θ_2 from the *Row_Set*;
(41) **end if**

Algorithm 2: Algorithm for batch-verifying all appointed leaves in $\mathcal{L}$.

Table 2: An example of batch verifications.

Variables	Initialization	$j = 1$	$j = 2$	$j = 3$
$\mathcal{H}_1$	x_3	x_{18}	x_{24}	x_{root}
$\mathcal{IS}_1$	{1}	{1, 2}	{3, 4}	{3, 4, 8, 10, 12, 14}
$\mathcal{H}_2$		x_4		
$\mathcal{IS}_2$	{1}		{2}	
$\mathcal{H}_3$	x_8	x_{20}	x_{25}	
$\mathcal{IS}_3$	{1}	{2}	{4}	{8, 10, 12, 14}
$\mathcal{H}_4$	x_{10}	x_{21}	x_{26}	
$\mathcal{IS}_4$	{1}	{1}	{1, 3, 5}	{6, 8, 10}
$\mathcal{H}_5$	x_{12}		x_{22}	
$\mathcal{IS}_5$	{1}	{1, 3}	{3, 5}	
$\mathcal{H}_6$		x_{14}		
$\mathcal{IS}_6$	{1}		{3}	

The process of batch verifications upon the appointed ordered leaves set $\mathcal{L} = \{v_3, v_4, v_8, v_{10}, v_{12}, v_{14}\}$ of Figure 2, according to Algorithm 2 and the proof table $T_{6\times3}$ of Table 1.

then $\mathcal{IS}_3 \oplus 4 = \{8, 10, 12, 14\}$. Algorithm 2 applies each nonnull item $T_{i,j}$ of $T_{k\times l}$ to iteratively compute tuple ($\mathcal{R}_i$, $\mathcal{H}_i$, $\mathcal{IS}_i$). If the returned leaves set $\mathcal{L}$ and table $T_{k\times l}$ are right, we will get the following results after the outermost for-loop of Algorithm 2 is finished:

(i) Value $\mathcal{H}_1$ is equal to x_{root}, that is, the authentication hash value of the root of bv23Tree.

(ii) Value $\mathcal{IS}_1$ is exactly the same as $\{a_1, a_2, \ldots, a_k\}$, that is, the indices set of k appointed ordered leaves in $\mathcal{L}$.

At a high level, Algorithm 2 is also the way to gradually construct the *partial bv23Tree*, which precisely covers all appointed leaves in $\mathcal{L}$, the paths from these appointed leaves to the root, and all the siblings of the nodes on these paths. Based on this partial bv23Tree, the batch updates upon outsourced original raw blocks $\{m_1, m_2, \ldots, m_n\}$ can be supported, as shown in Algorithms 3 and 4.

3.4. Batch Updates. Three basic types of dynamic update operations are modification (M), insertion (I), and deletion (D) [9]. Any block-level update operation $\mathcal{O}$ can be defined by the form of $\mathcal{O} :=$ "$\mathcal{U}$ a m", where $\mathcal{U} \in \{M, I, D\}$ denotes operation type, a is the index of targeted block, and m is the new data block that will be exactly stored according to the targeted index a (m is null for deletion). For example, "M 2 m" is to modify the 2nd block to m, "I 3 m" is to insert m after the 3rd block, and "D 4 null" is to delete the 4th block.

In the setting of dynamism, the update operations should be performed not only on the data blocks, but also on the bv23Tree. Note that only the insertion and deletion can

Batch_Updates $(\mathcal{SO}, \Psi, F) \rightarrow \{\mathcal{L}, T_{k\times l}, x_{\text{root}^*}\}$. Input parameters $\mathcal{SO} = \{\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_c\}$ are a batch of update operations, Ψ is the bv23Tree, and $F = \{m_1, m_2, \ldots, m_n\}$ are the whole outsourced original blocks. This algorithm outputs an ordered leaves set $\mathcal{L}$, the proof table $T_{k\times l}$, and the updated root hash value x_{root^*} from the final state bv23Tree Ψ^*.

(1) extract from $\mathcal{SO}$ the largest ordered targeted indices set $\Pi = \{a_1, a_2, \ldots, a_k\}$, by removing the duplicate indices;
(2) read leaves set $\mathcal{L} = \{v_{a_1}, v_{a_2}, \ldots, v_{a_k}\}$ from Ψ;
(3) obtain $T_{k\times l} \leftarrow$ *Proof_Table*(Ψ, Π);
(4) update the file blocks set F according to the sequential executions of all update operations $\{\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_c\}$;
(5) perform each of the update operations in sequence on Ψ and then obtain the final state Ψ^*; more specifically, transform Ψ in terms of each operation, and update the status s_v, rank r_v, and hash value x_v of the affected tree nodes during each transformation; (for the modification operation without transformation, only need to update the hash values of the nodes on the path from the targeted leaf to the root)
(6) **return** $\{\mathcal{L}, T_{k\times l}, x_{\text{root}^*}\}$, where x_{root^*} is the authentication hash value of the root of final state Ψ^*;

Algorithm 3: Algorithm for CSS to perform the batch updates.

Verify_Updates $(x_{\text{root}}, \mathcal{SO}, \mathcal{L}, T_{k\times l}, x_{\text{root}^*}) \rightarrow \{\text{true}, \text{false}\}$. Input parameters x_{root} is client local metadata, update operations set $\mathcal{SO} = \{\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_c\}$ are generated by client herself, $\mathcal{L}$, $T_{k\times l}$, x_{root^*} are provided by CSS as computed in Algorithm 3. This algorithm outputs true if the batch updates are successful, or false otherwise.

(1) extract $\Pi = \{a_1, a_2, \ldots, a_k\}$ from $\mathcal{SO}$ as in Algorithm 3;
(2) **if** *Batch_Verify* $(x_{\text{root}}, \Pi, \mathcal{L}, T_{k\times l}) =$ true **then**
(3) construct partial bv23Tree with x_{root} as the root hash;
(4) **else** {Batch_Verify$(x_{\text{root}}, \Pi, \mathcal{L}, T_{k\times l}) =$ false}
(5) **return** false;
(6) **end if**
(7) perform each update operation $\mathcal{O}_i$ of $\mathcal{SO}$ upon above partial bv23Tree by the same transformations as in Algorithm 3, and then compute the final state root hash $x_{\text{root}'}$;
(8) **if** $x_{\text{root}'} = x_{\text{root}^*}$ **then**
(9) replace local x_{root} with x_{root^*}, **return** true;
(10) **else** $\{x_{\text{root}'} \neq x_{\text{root}^*}\}$
(11) **return** false;
(12) **end if**

Algorithm 4: Algorithm for client to verify the result of batch updates.

cause the structure transformation of bv23Tree, and the maintenance of this transformation is essentially identical to the maintenance of a standard 2-3 tree [20], except that the updating of each affected tree node v should be considered in terms of the 3-element tuple (s_v, r_v, x_v). As in Figure 3, we give an example for the structure transformation of bv23Tree after repetitively inserting (or deleting) the appointed data blocks at the same index position.

Now, suppose that client caches a batch of ordered update operations $\mathcal{SO} \coloneqq \{\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_c\}$, which refer to an ordered indices set $\Pi = \{a_1, a_2, \ldots, a_k\}$, $k \leq c$, by the removal of duplicate indices. To batch-update the remote data, client first issues $\mathcal{SO}$ to CSS and obtains the returned ordered leaves set $\mathcal{L} = \{v_{a_1}, v_{a_2}, \ldots, v_{a_k}\}$ along with the proof table $T_{k\times l}$. As shown in Section 3.3, during the process of executing Algorithm 2 to batch-verify $\mathcal{L}$, client can construct the corresponding partial bv23Tree. Then, after sequentially performing each operation of $\mathcal{SO}$ upon the partial bv23Tree, client can compute by herself what would be the authentication hash value of the final state tree root. Finally, if CSS outputs the same final state root hash value as the one computed by client herself, client outputs true, meaning that CSS correctly performs the batch updates according to $\mathcal{SO}$. Otherwise, client outputs false. As shown in Algorithms 3 and 4, we outline the *Batch_Updates* algorithm performed by CSS and the *Verify_Updates* algorithm performed by client, respectively.

4. DOPOR Solution

4.1. Cloud Server Storage Configuration. As shown in Figure 1, there are two different buffers U and E at CSS side for outsourced data storage. Client's original file F, consisting of n raw blocks $F = \{m_1, m_2, \ldots, m_n\}$, will be separately stored into U and E with different formats, detailed as follows.

U (Unencoded Buffer). To support the efficient reads, buffer U always stores the up-to-date copy of the raw data blocks, and thus the update operations issued from client must be immediately performed upon the appointed blocks of U. All the blocks in U are organized by the bv23Tree as proposed in Section 3, the batch-verifications property of which enables client to batch-read a group of raw blocks from U for improved performance.

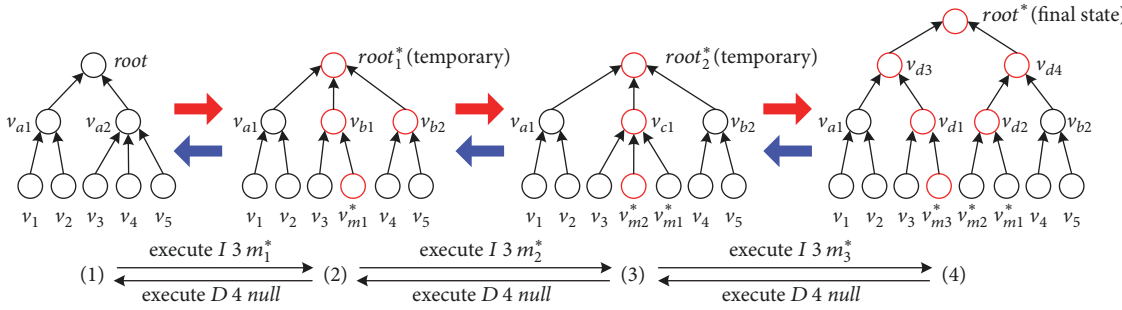

FIGURE 3: From (1) to (4) is the structure transformation of bv23Tree after performing three insertion operations $\{I\ 3\ m_1^*, I\ 3\ m_2^*, I\ 3\ m_3^*\}$ in order. And the reverse transformation from (4) to (1) is based on the sequential executions of three identical deletion operations.

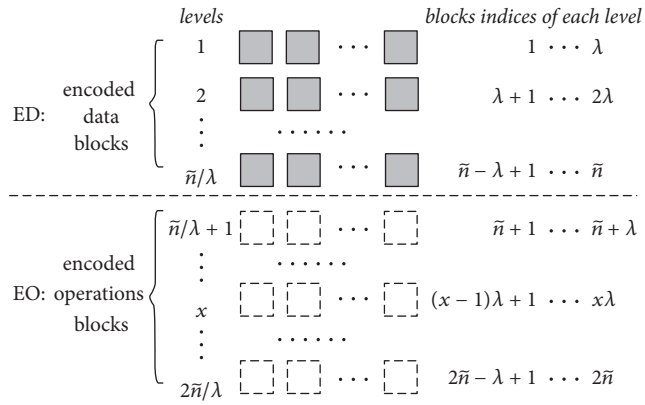

FIGURE 4: CSS-side hierarchical storage structure for erasure-coded buffer E, where each level stores λ encoded blocks along with tags.

E (Erasure-Coded Buffer). Buffer E is organized by the same-sized hierarchical structure, as in Figure 4, where each level has the same size to hold λ encoded blocks with the blocks tags (λ is the security parameter). Moreover, E is equally divided into two parts ED and EO with the same capacity. ED is to store encoded data blocks, and EO is to store encoded operations blocks. At the beginning, client applies erasure code to encode the entire file F into $\tilde{n}$ encoded data blocks, computes the blocks tags, and sequentially stores them into ED in terms of the blocks indices as shown in Figure 4. Subsequently, the content of ED will not be changed until buffer E is rebuilt. As shown in Section 3.4, each update operation $\mathcal{O}$ is of the form "$\mathcal{U}\ a\ m$", which can also be regarded as an operation block. In this case, after a batch of update operations $\mathcal{SO} = \{\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_c\}$ ($c < \lambda$) are performed upon U, client will encode $\mathcal{SO}$ into λ encoded operations blocks and store them along with the blocks tags into a corresponding level of EO. So, EO is empty in the initial state and will be incrementally filled up level by level over time. At last, once EO is full, the rebuilding of E is triggered, as will be described later.

At a high level, by decoding the whole E and sequentially performing the accumulated update operations of EO upon the original data blocks of ED, the latest version of the whole client's outsourced raw data can be recovered. Therefore, the periodic POR audits only need to be deployed against the buffer E, which functions as a backup storage to provide the retrievability guarantee when data loss occurs. To resist the mentioned selective deletion attack as before, every POR audit will sample challenged blocks from each filled level of E. Because each level is entirely encoded, malicious CSS must corrupt a significant portion of the encoded blocks of one level to actually cause the data loss. However, if malicious CSS corrupts that many encoded blocks of one level, it cannot pass the POR audits with overwhelming probability.

4.2. Initialization. We also work in the bilinear setting, where G is a multiplicative cyclic group of prime order p and g is a generator of G. Let $e: G \times G \to G_T$ be the same nondegenerate bilinear map as in [8] that has the following property: for any $\varkappa, \omega \in G$ and $a, b \in \mathbb{Z}_p$, $e(\varkappa^a, \omega^b) = e(\varkappa, \omega)^{ab}$. Let $H: \{0,1\}^* \to G$ be the secure BLS hash function. The initialization of our scheme is described as follows.

(1) **GenKey**($1^{\not\kappa}$): each entity $\mathbb{E} \in \{\text{client, CSS, TPA}\}$ generates a signing key pair ($\text{ssk}_{\mathbb{E}}$, $\text{spk}_{\mathbb{E}}$) for their respective signatures. In addition, client samples a random element $\alpha \xleftarrow{R} \mathbb{Z}_p$ and computes $\omega \leftarrow g^{\alpha}$. α is kept secret by client but ω is public. So, client's private key $\text{sk}_{\text{client}} = (\alpha, \text{ssk}_{\text{client}})$ and her public key $\text{pk}_{\text{client}} = (\omega, \text{spk}_{\text{client}})$.

(2) **GenTags**($\mathrm{sk}_{\mathrm{client}}, F$): client applies erasure code to encode $F = \{m_i\}_{1\le i\le n}$ into $\tilde{n}$ codeword blocks $B = \{b_i\}_{1\le i\le \tilde{n}}$, and each b_i is s sectors long: $b_i := \{b_{ij}\}_{1\le j\le s}$. Client then generates a name fid for F and samples s elements $u_1, \ldots, u_s \xleftarrow{R} G$. For each index i, $1 \le i \le \tilde{n}$, with her secret key α in $\mathrm{sk}_{\mathrm{client}}$, client computes for b_i the corresponding tag $\sigma_i \leftarrow (H(\mathrm{fid} \parallel i) \cdot \prod_{j=1}^{s} u_j^{b_{ij}})^{\alpha}$ and attaches σ_i to b_i.

(3) **OutsourceData**(F, B, Φ): based on $F = \{m_1, m_2, \ldots, m_n\}$, client generates the corresponding bv23Tree Ψ with the root hash x_{root}, as shown in Section 3.1. Then, client outsources $\{F, \Psi\}$ into the buffer U and outsources all encoded data blocks $B = \{b_i\}_{1\le i\le \tilde{n}}$ (along with their tags $\Phi = \{\sigma_i\}_{1\le i\le \tilde{n}}$) into the ED part of buffer E, the layout of which is shown in Figure 4. In addition, let $\mathfrak{sp}$ be a state pointer that denotes the number of the filled levels of buffer E. Clearly, the range of $\mathfrak{sp}$ is $\tilde{n}/\lambda \le \mathfrak{sp} \le 2\tilde{n}/\lambda$, as in Figure 4. Let $c\ell$ denote the number of the challenged blocks from a filled level, and let $\mathbb{P} = \{\mathfrak{sp}, c\ell, g, \omega, \mathrm{fid}, u_1, \ldots, u_s\}$ be the public parameters set for POR audits. Finally, client keeps $\mathbb{P}$ and x_{root} locally, sends $\mathbb{P}$ to TPA, and deletes $\{F, \Psi, B, \Phi\}$ from her local storage.

4.3. Data Access Mechanisms. Based on the bv23Tree of buffer U and the hierarchical configuration of same-sized levels of buffer E, DOPOR supports batch updates that enable the client to perform a batch of update operations upon the outsourced storage, which is suitable for the common scenario of [14] where writes are frequent. Now, after completing the initialization of DOPOR, the client can access and update her outsourced data by the following three protocols.

(1) **ReadBlocks**($x_{\mathrm{root}}, \Pi, \mathfrak{st}$). With her local root hash x_{root} of bv23Tree, the client can batch-read any k appointed raw blocks from CSS, by sending the ordered blocks indices set $\Pi = \{a_1, a_2, \ldots, a_k\}$ as the query to CSS. Here, let buffer U be the CSS state $\mathfrak{st}$. In terms of Section 3, upon receiving Π, CSS accesses appointed raw blocks $\mathscr{M} = \{m_{a_1}, m_{a_2}, \ldots, m_{a_k}\}$ and the tree leaves $\mathscr{L} = \{v_{a_1}, v_{a_2}, \ldots, v_{a_k}\}$ from U, generates $T_{k\times l} \leftarrow$*Proof_Table* (Ψ, Π), and returns $\{\mathscr{M}, \mathscr{L}, T_{k\times l}\}$ to the client. Then, the client batch-verifies the authenticity of $\mathscr{L}$ by calling *Batch_Verify* $(x_{\mathrm{root}}, \Pi, \mathscr{L}, T_{k\times l})$ and finally checks the integrity of all raw blocks of $\mathscr{M}$ according to the corresponding hash values stored in $\mathscr{L}$.

(2) **PerformUpdates**($x_{\mathrm{root}}, \mathcal{SO}, \mathrm{sk}_{\mathrm{client}}, \mathfrak{st}, \mathfrak{sp}$). Suppose that the client keeps $O(\lambda)$ local storage to cache c ($c < \lambda$) ordered update operations $\mathcal{SO} := \{\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_c\}$. Then, the client sends $\mathcal{SO}$ to CSS for performing these c operations in a batch. As shown in Section 3.4, on receiving $\mathcal{SO}$, with the raw data F and bv23Tree Ψ stored in buffer U, CSS can execute *Batch_Updates* $(\mathcal{SO}, \Psi, F)$ to return to the client the results $\{\mathscr{L}, T_{k\times l}, x_{\mathrm{root}^*}\}$ for batch updates, and the client can call *Verify_Updates* $(x_{\mathrm{root}}, \mathcal{SO}, \mathscr{L}, T_{k\times l}, x_{\mathrm{root}^*})$ to authenticate these returned results.

If the above results pass the client's authentication, the client then applies an erasure code to encode $\mathcal{SO}$ into λ encoded operations blocks $\{b_{i_1}, b_{i_2}, \ldots, b_{i_\lambda}\}$. Based on local state pointer $\mathfrak{sp}$, client first computes the indices $\{i_1, i_2, \ldots, i_\lambda\}$ of these λ encoded blocks in order $i_\tau = \mathfrak{sp}\cdot\lambda + \tau$, $1 \le \tau \le \lambda$, and computes the tag σ_{i_τ} for each b_{i_τ} by the same tag formula as in **GenTags**$(\cdot)$ of initialization. Secondly, client outsources these λ blocks $\{b_{i_1}, b_{i_2}, \ldots, b_{i_\lambda}\}$ along with their tags into the $(\mathfrak{sp}+1)$th level of buffer E (i.e., the corresponding empty level of EO). Finally, client sends to TPA the updated state pointer $\mathfrak{sp}^* = \mathfrak{sp} + 1$ with her signature and empties her $O(\lambda)$ local storage for caching the next c ordered update operations. Overall, through this protocol, the CSS state $\mathfrak{st}$ consists of U and EO as above.

(3) **Rebuild**($x_{\mathrm{root}}, \mathrm{sk}_{\mathrm{client}}, \mathfrak{st}, \mathfrak{sp}$). Once in every n update operations, when EO of buffer E gets filled (i.e., $\mathfrak{sp}$ is equal to $2\tilde{n}/\lambda$ as in Figure 4), the periodic rebuilding for E is triggered. Since buffer U stores the up-to-date copy of all raw data blocks, client can carry out this rebuilding based on U for the performance improvement, instead of decoding the whole E and applying all operations of EO on the original data of ED. (ED and EO will not be decoded and combined, unless client detects data corruption within CSS and wants to retrieve the whole data.)

Benefitting from the ability of batch-reading multiple blocks from CSS and our hierarchical configuration, client can rebuild E with only $O(\lambda)$ local memory that is the same size as a level of the hierarchical structure, which is the significant improvement when compared to existing schemes [14, 18, 19] that require such rebuilding with $O(n)$ client local memory. In this protocol, the CSS state $\mathfrak{st}$ consists of U and E. At the beginning, client runs **ReadBlocks**$(\cdot)$ to batch-read the first c ($c < \lambda$) raw blocks from U. After encoding these c blocks into λ codeword blocks, client computes the corresponding λ tags by the same way as in **GenTags**$(\cdot)$, where the indices of λ codeword blocks of each level are shown in Figure 4. Then, client outsources to CSS these λ codeword blocks and tags that will be stored in the first level of ED. Subsequently, client can empty her $O(\lambda)$ local memory and batch-read from U the next batch of c blocks, which are processed and outsourced to CSS by the same procedures as above, except that each batch of λ processed codeword blocks and tags will be stored in the corresponding different level of ED (e.g., the second batch will be stored in the second level of ED, etc.). After the last batch of blocks from U are processed and outsourced, client authorizes CSS to update the whole ED with all above outsourced codeword blocks and tags, and CSS simultaneously empties the whole EO. Finally, client publishes the updated state pointer $\mathfrak{sp}^*$ to TPA, meaning that the rebuilding is over. Overall, the amortized bandwidth of rebuilding E is $O(1)$ per update operation, since this rebuilding is executed only once every n update operations.

4.4. Outsourced Proof of Retrievability (OPOR)

(1) **RandomChal**($\mathbb{B}, t, \mathbb{P}$). The periodic random challenges are generated based on the time-based bitcoin pseudorandom

source $\mathbb{B}$. More specifically, given the current time t, the tool getblockhash(t) [12] from the bitcoin source $\mathbb{B}$ can output the hash of the latest block that has arisen since time t in the bitcoin block chain. As shown in [12], for a future time t, no adversary can predict the hash of a bitcoin block that will arise in the future. In addition, for a past time t, the hash of previous bitcoin block, returned by getblockhash(t), is objective and irrefutable against any adversary. Thus, let $\text{coin}^{(t)}$ denote the output of getblockhash(t); then $\text{coin}^{(t)}$ can be considered as a secure pseudorandom coin for time t.

With the public parameter $\mathfrak{z}, \mathfrak{c} \in \mathbb{P}$, TPA can generate the random POR challenge of length $\mathfrak{z} \cdot \mathfrak{c}$, by calling the same probabilistic algorithm Sample($\text{coin}^{(t)}, \mathfrak{z} \cdot \mathfrak{c}$) as in [12] that $\text{coin}^{(t)}$ is obtained from getblockhash(t) for the different time t. Specifically, to ensure that a POR challenge can sample the same amount of $\mathfrak{c}$ blocks from each filled level of buffer E (i.e., ED and EO) at CSS, TPA first calls Sample($\text{coin}^{(t)}, \mathfrak{c}$) to choose a random $\mathfrak{c}$-elements subset $I^* = \{s_{t,1}, s_{t,2}, \ldots, s_{t,\mathfrak{c}}\}$ of $[1, \lambda]$, and then it computes the $\mathfrak{z} \cdot \mathfrak{c}$-elements set I as follows:

$$I = \left\{\zeta \cdot \lambda + \tau \mid 0 \le \zeta \le \mathfrak{z} - 1,\ \tau \in I^*\right\}, \tag{5}$$

Moreover, for each $i \in I$, as in [12], TPA also depends on Sample($\text{coin}^{(t)}, \mathfrak{z} \cdot \mathfrak{c}$) to choose a corresponding random element $\alpha_i \xleftarrow{R} \mathbb{Z}_p$. Finally, let $Q^{(t)} \coloneqq \{(i, \alpha_i)\}_{i \in I}$ denote an $\mathfrak{z} \cdot \mathfrak{c}$-elements set, which is regarded as the POR challenge at time t and sent to CSS by TPA.

(2) **GenProof**($Q^{(t)}, \mathfrak{z}t$). Here, let buffer E be the CSS state $\mathfrak{z}t$. Upon receiving challenge $Q^{(t)}$, for each index i specified by $Q^{(t)}$, CSS reads the corresponding block $b_i \coloneqq \{b_{ij}\}_{1 \le j \le s}$ and tag σ_i from E. Now CSS performs exactly as in the public verification scheme of [8] by computing the values $\mu_j^{(t)}$, $1 \le j \le s$, along with $\sigma^{(t)}$ as follows:

$$\begin{aligned} \mu_j^{(t)} &\longleftarrow \sum_{(i,\alpha_i) \in Q^{(t)}} \left(\alpha_i \cdot b_{ij}\right) \in \mathbb{Z}_p, \\ \sigma^{(t)} &\longleftarrow \prod_{(i,\alpha_i) \in Q^{(t)}} {\sigma_i}^{\alpha_i} \in G. \end{aligned} \tag{6}$$

At last, CSS sends to TPA its response $\rho^{(t)} \parallel \text{sig}_{\text{CSS}}$, where $\rho^{(t)} \coloneqq (\mu_1^{(t)}, \mu_2^{(t)}, \ldots, \mu_s^{(t)}, \sigma^{(t)})$, and sig_{CSS} is CSS's signature to provide nonrepudiation.

(3) **PORAudit**($\mathbb{P}, Q^{(t)}, \rho^{(t)}$). Based on the public parameters set $\mathbb{P}$ and the challenge $Q^{(t)}$, TPA can compute his own auditing parameter $\xi^{(t)}$ as follows:

$$\xi^{(t)} \longleftarrow \prod_{(i,\alpha_i) \in Q^{(t)}} H\left(\text{fid} \parallel i\right)^{\alpha_i} \in G. \tag{7}$$

Then, after TPA parses the CSS's response to obtain $\rho^{(t)}$, TPA will audit $\rho^{(t)}$ by checking the following equation:

$$e\left(\sigma^{(t)}, g\right) \stackrel{?}{=} e\left(\xi^{(t)} \cdot \prod_{j=1}^{s} {u_j}^{\mu_j^{(t)}}, \omega\right). \tag{8}$$

If this verification does not pass, TPA informs the client of this abnormal situation, meaning that the data loss occurs. Finally, TPA must generate and store the following log $\Lambda^{(t)}$ that corresponds to the challenge at time t:

$$\Lambda^{(t)} \coloneqq \left(t, \text{coin}^{(t)}, \xi^{(t)}, \rho^{(t)} \parallel \text{sig}_{\text{CSS}}\right). \tag{9}$$

(4) **CheckLogs**($\mathbb{B}, T, \Lambda^{(T)}, \mathbb{P}, \text{sk}_{\text{client}}$). To protect against malicious TPA who might violate the above auditing process or even collude with CSS, the client can verify TPA's work by checking TPA's logs. However, instead of checking the accumulated TPA's logs one by one, client is able to batch-check multiple TPA's logs all together, so such a client's batch-checking against TPA is only seldom performed in practice, as shown in [12].

Client can check the latest TPA's log for a minimal check, since this log reflects the latest status of retrievability for the outsourced data [12]. More generally, to perform a batch-checking against TPA, client selects a point-in-time set $T = \{t_1, t_2, \ldots, t_k\}$ and sends T to TPA, where each $t \in T$ marks the time of a past challenge.

Upon receiving T, for each $t \in T$, in terms of $\rho^{(t)}$ and $\xi^{(t)}$ stored in log $\Lambda^{(t)}$, TPA computes

$$\begin{aligned} \mu_j^{(T)} &\longleftarrow \sum_{t \in T} \mu_j^{(t)} \in \mathbb{Z}_p \quad \text{for } 1 \le j \le s, \\ \sigma^{(T)} &\longleftarrow \prod_{t \in T} \sigma^{(t)} \in G, \\ \xi^{(T)} &\longleftarrow \prod_{t \in T} \xi^{(t)}. \end{aligned} \tag{10}$$

Then, TPA responds to client with his proof $\varrho^{(T)} \coloneqq (\mu_1^{(T)}, \mu_2^{(T)}, \ldots, \mu_s^{(T)}, \sigma^{(T)}, \xi^{(T)})$, which is also signed by TPA.

Based on the public parameters $\mathfrak{z}$ and $\mathfrak{c}$, for each $t \in T$, the client is able to reconstruct alone each past challenge $Q^{(t)}$ as described in the protocol **RandomChal**($\cdot$), by using Sample($\text{coin}^{(t)}, \mathfrak{z} \cdot \mathfrak{c}$) with the pseudorandom $\text{coin}^{(t)}$ obtained from getblockhash(t). So, client can compute her own checking parameter $\eta^{(T)}$ as follows:

$$\eta^{(T)} \longleftarrow \prod_{t \in T} \prod_{(i,\alpha_i) \in Q^{(t)}} H\left(\text{fid} \parallel i\right)^{\alpha_i} \in G. \tag{11}$$

After verifying TPA's signature on the proof $\varrho^{(T)}$, client first checks that whether $\eta^{(T)}$ is equal to $\xi^{(T)}$ of $\varrho^{(T)}$. If $\eta^{(T)} \neq \xi^{(T)}$, client outputs false, confirming that TPA was irresponsible for the past POR audits. Finally, client checks the following equation with her secret key $\alpha \in \text{sk}_{\text{client}}$:

$$\left(\eta^{(T)} \cdot \prod_{j=1}^{s} {u_j}^{\mu_j^{(T)}}\right)^{\alpha} \stackrel{?}{=} \sigma^{(T)}. \tag{12}$$

If the above client's check fails, client outputs false, which means that there exists collusion among TPA and CSS

and that the data corruption has occurred within CSS. The correctness of (12) is demonstrated as follows:

$$\begin{aligned}\sigma^{(T)} &= \prod_{t\in T}\sigma^{(t)} = \prod_{t\in T}\prod_{(i,\alpha_i)\in Q^{(t)}}\sigma_i^{\alpha_i}\\ &= \prod_{t\in T}\prod_{(i,\alpha_i)\in Q^{(t)}}\left(H(\text{fid}\parallel i)^{\alpha_i}\cdot\prod_{j=1}^{s}u_j^{\alpha_i\cdot b_{ij}}\right)^{\alpha}\\ &= \left(\prod_{t\in T}\prod_{(i,\alpha_i)\in Q^{(t)}}H(\text{fid}\parallel i)^{\alpha_i}\right.\\ &\quad\left.\cdot\prod_{j=1}^{s}u_j^{\sum_{t\in T}\sum_{(i,\alpha_i)\in Q^{(t)}}(\alpha_i\cdot b_{ij})}\right)^{\alpha} = \left(\eta^{(T)}\cdot\prod_{j=1}^{s}u_j^{\mu_j^{(T)}}\right)^{\alpha}.\end{aligned} \tag{13}$$

5. Security Analysis

Similar to the analysis of [8, 12], we evaluate the soundness of our DOPOR scheme according to three parts: unforgeability, liability, and extractability.

Theorem 1 (unforgeability). *It is computationally infeasible for any adversary $\mathcal{A}$ to forge a proof that can pass verifier's check, if the Computational Diffie-Hellman (CDH) problem and the Discrete Logarithm (DL) problem are hard.*

Proof. Since CSS does not check any proof throughout the whole process of executing DOPOR, there are only two cases to be discussed.

Case 1. TPA plays the role of verifier to check the proof returned from CSS during executing the protocols **GenProof**(·) and **PORAudit**(·) as shown in Section 4.4. In this case, observe that both CSS and TPA perform exactly the same as the BLS-based public verification scheme of [8], so the unforgeability guarantee immediately follows from the work of [8]. As shown in [21], the BLS scheme is secure when the CDH problem is hard in bilinear groups, based on which the unforgeability of BLS-based public scheme has been proven in [8] and thus omitted here.

Case 2. The client acts as the verifier to check TPA's logs as in the protocol **CheckLogs**(·) of Section 4.4. To pass the client's check with (12), TPA should return the correct proof $(\mu_1^{(T)}, \mu_2^{(T)}, \ldots, \mu_s^{(T)}, \sigma^{(T)})$. Now, assume that TPA is able to forge the proof. As shown in [8], due to the security of BLS scheme, the BLS-based homomorphic verifiable tag σ_i is unforgeable, and thus the aggregated tag $\sigma^{(T)}$ is also unforgeable. So, the only choice for TPA is to generate the forged aggregated block, denoted with $\tilde{\mu}_1^{(T)}, \tilde{\mu}_2^{(T)}, \ldots, \tilde{\mu}_s^{(T)}$, as the response to client's check. Then, for (12) to be satisfied, we have

$$\left(\eta^{(T)}\cdot\prod_{j=1}^{s}u_j^{\tilde{\mu}_j^{(T)}}\right)^{\alpha} = \sigma^{(T)}. \tag{14}$$

In addition, according to the correct proof, we have

$$\left(\eta^{(T)}\cdot\prod_{j=1}^{s}u_j^{\mu_j^{(T)}}\right)^{\alpha} = \sigma^{(T)}. \tag{15}$$

Note that $\eta^{(T)}$ is the parameter computed by client herself, and the security of $\eta^{(T)}$ is ensured by the security of bitcoin pseudorandom source of [12]. Based on the security of BLS scheme, we can learn that

$$\begin{aligned}&\prod_{j=1}^{s}u_j^{\mu_j^{(T)}} = \prod_{j=1}^{s}u_j^{\tilde{\mu}_j^{(T)}} \Longrightarrow\\ &\prod_{j=1}^{s}u_j^{\Delta\mu_j^{(T)}} = 1,\end{aligned} \tag{16}$$

where $\Delta\mu_j^{(T)} = \tilde{\mu}_j^{(T)} - \mu_j^{(T)}$ and $\tilde{\mu}_j^{(T)} \neq \mu_j^{(T)}$.

For any two given elements $g_1, g_2 \in G$, we have $u_j = g_1^{p_j} g_2^{q_j} \in G$, where $p_j, q_j \in \mathbb{Z}_p$. Hence, (16) is transformed as follows:

$$\begin{aligned}&\prod_{j=1}^{s}\left(g_1^{p_j}g_2^{q_j}\right)^{\Delta\mu_j^{(T)}} = 1 \Longrightarrow\\ &g_2 = g_1^{-(\sum_{j=1}^{s}p_j\cdot\Delta\mu_j^{(T)})/(\sum_{j=1}^{s}q_j\cdot\Delta\mu_j^{(T)})}.\end{aligned} \tag{17}$$

Obviously, (17) means that malicious TPA can solve the DL problem, which is in conflict with the assumption that DL problem is hard. Therefore, it is infeasible for TPA to forge a proof to pass the client's check. This completes our proof. □

Theorem 2 (liability). *If any adversary $\mathcal{A}$ attempts to cheat or frame the honest entity who has been well behaving, the honest entity can output incontestable evidence to confirm the misbehavior of adversary $\mathcal{A}$ in case of lawsuit.*

Proof. It is clear that if the honest entity can protect against the collusion of the other two malicious entities, then this honest entity can certainly protect against any single malicious entity. Hence, to prove Theorem 2, it suffices to consider the following three cases where only one entity is honest.

Case 1. Honest client defends against the collusion of CSS and TPA. Obviously, CSS has incentive to collude with TPA only when the outsourced data corruption has occurred at cloud side. In this case, once the corrupted data blocks are challenged, according to Theorem 1, both CSS and TPA cannot forge an effective proof to pass client's check against TPA's log, unless they can solve the DL problem (but this probability is negligible). Therefore, the false output by client when executing the protocol **CheckLogs**(·) is the incontestable evidence to identify the collusion of CSS and TPA.

Case 2. Honest TPA defends against the collusion of client and CSS. As shown in the protocol **PORAudit**(·) of Section 4.4, TPA completes his auditing work by computing the auditing parameter $\xi^{(t)}$ and verifying (8), where all these processes are reproducible and undeniable for the malicious entities.

More specifically, on the one hand, the nonrepudiation of parameter $\xi^{(t)}$ is derived from the objectivity of challenge $Q^{(t)}$, which is computed based on the secure bitcoin pseudorandom source and the public parameters $\{\mathcal{sp}, \mathcal{cl}\}$. On the other hand, all other inputs involved in verifying (8) are also undeniable for both client and CSS; for example, $(\mu_1^{(t)}, \mu_2^{(t)}, \ldots, \mu_s^{(t)}, \sigma^{(t)})$ are signed by CSS's signature, and $\{g, \omega, u_1, \ldots, u_s\}$ are the public parameters confirmed by all entities. Hence, the honest TPA can provide his logs $\Lambda^{(t)}$ in case of lawsuit, which includes all above incontestable evidence enabling the playback of all past TPA's auditing work to prove the innocence of TPA.

Case 3. Honest CSS defends against the collusion of client and TPA. When malicious client colludes with TPA to falsely accuse CSS of corrupting the ith block b_i, CSS can output the intact $b_i = \{b_{ij}\}_{1 \le j \le s}$ and its tag σ_i as the incontestable evidence. As shown in [8], each tag σ_i constructed and outsourced by client herself is unforgeable. Based on the security of BLS signature scheme, as long as the above b_i and σ_i output by CSS satisfy (18), then CSS is innocent.

$$e(\sigma_i, g) \stackrel{?}{=} e\left(H(\text{fid} \parallel i) \cdot \prod_{j=1}^{s} u_j^{b_{ij}}, \omega\right). \tag{18}$$

This completes the proof of Theorem 2. □

According to Theorem 2, all the three entities have to behave properly in DOPOR. In this case, the extractability during the TPA' audits against CSS can immediately follow from the work of [8], since the procedure of TPA's audits of DOPOR corresponds to the public verification scheme of [8]. As for the extractability of performing **CheckLogs**(·) protocol, we have the following theorem.

Theorem 3 (extractability). *During the client's checking against TPA, if client does not output false after checking TPA's logs, then there exists a deterministic extraction algorithm, based on which client can extract the challenged file blocks by the repetitive interactions with TPA.*

Proof. According to Theorem 1, to pass client's check during the execution of protocol **CheckLogs**(·), TPA has to respond to client with the correct proof $\varrho^{(T)}$ that includes the aggregated block $(\mu_1^{(T)}, \mu_2^{(T)}, \ldots, \mu_s^{(T)})$, and each $\mu_j^{(T)}$ is a linear equation of the following form:

$$\mu_j^{(T)} = \sum_{t \in T} \mu_j^{(t)} = \sum_{t \in T} \sum_{(i, \alpha_i) \in Q^{(t)}} \left(\alpha_i \cdot b_{ij}\right), \tag{19}$$

where T is a set of point-in-times chosen by client herself, and all the coefficients α_i are dominated by these point-in-times as shown in the protocol **RandomChal**(·) of Section 4.4.

Now, suppose client chooses the appropriate point-in-times in the past to generate different set T and checks TPA's logs for a polynomial number of times by sending different T to TPA; then, client can get a total of s systems of linear equations that are built upon the challenged target blocks. Finally, by solving these s systems, client can extract all target blocks. □

When referring to dynamic updates, as shown in Section 4.3, the procedure of performing updates is divided into two parts: (1) performing the batch updates upon buffer U according to the algorithms *Batch_Updates* (·) and *Verify_Updates* (·), as shown in Section 3.4, the essence of which is based on the property of batch verifications of bv23Tree, while the security of this property is ensured by Theorem 4; (2) outsourcing a batch of encoded operations blocks and their tags to buffer E, the security of which is directly ensured by the security of the unforgeable tags, the periodic executions of POR audits against buffer E, and the erasure code scheme.

Theorem 4. *Assuming the existence of a collision-resistant hash function $h(\cdot)$, for any k ordered indices set $\Pi = \{a_1, a_2, \ldots, a_k\}$ appointed by the client, the corresponding proof table $T_{k \times l}$ generated using the bv23Tree ensures the integrity of all the k appointed leaves $\mathscr{L} = \{v_{a_1}, v_{a_2}, \ldots, v_{a_k}\}$ returned from CSS with overwhelming probability.*

Proof. As shown in Section 3.2, upon receiving the appointed ordered indices set $\Pi = \{a_1, a_2, \ldots, a_k\}$, CSS should respond to client with the corresponding leaves $\mathscr{L} = \{v_{a_1}, v_{a_2}, \ldots, v_{a_k}\}$ and proof table $T_{k \times l}$. Now, suppose that CSS tries to act dishonestly; then, the possible ways for CSS to misbehave can be covered by the following two cases.

Case 1. Malicious CSS either forges some leaves within $\mathscr{L}$ or forges some items within the proof table $T_{k \times l}$. As shown in Section 3.3, client possesses the public tree root hash x_{root} and will verify both $\mathscr{L}$ and $T_{k \times l}$ by calling the algorithm *Batch_Verify* (·), the procedure of which is to recalculate the public x_{root} by iteratively hashing the values of all tree nodes included in $\mathscr{L}$ and $T_{k \times l}$ in terms of the specified order. Apparently, the above forged $\mathscr{L}$ or $T_{k \times l}$ can enable *Batch_Verify*(·) to output the same root hash value as x_{root}, meaning that CSS is able to find the collisions against the hash function $h(\cdot)$, which contradicts the assumption that $h(\cdot)$ is collision-resistant. Therefore, it is a negligible probability for malicious CSS to forge $\mathscr{L}$ or $T_{k \times l}$.

Case 2. Malicious CSS launches the replacing attack; that is, CSS returns the replaced $\widetilde{\mathscr{L}}$ where some appointed leaves are replaced with other existing leaves of bv23Tree and the corresponding proof table $\widetilde{T}_{k \times l}$ that is correctly generated based on $\widetilde{\mathscr{L}}$. Without loss of generality, suppose that $\widetilde{\mathscr{L}} = \{v_x, v_y, \ldots, v_{a_k}\}$, where v_x and v_y are not the appointed leaves; that is, $x \neq a_1$ and $y \neq a_2$. In this case, although the final hash value output by *Batch_Verify* (·) is equal to the public x_{root}, the final value of the variable $\mathcal{IS}_1$ within *Batch_Verify* (·) will be $\{x, y, \ldots, a_k\}$ instead of the specified $\{a_1, a_2, \ldots, a_k\}$, which contradicts the expected results as shown in Section 3.3. So, client will still output false meaning that above malicious attack is detected by client.

In short, if there exists a collision-resistant hash function, malicious CSS has to return all the appointed leaves along

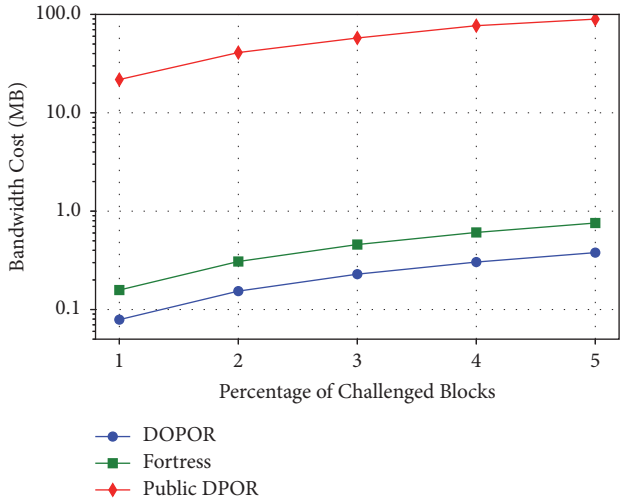

Figure 5: TPA-CSS bandwidth cost for a POR audit, with respect to the fraction of challenged blocks of the total number of encoded blocks.

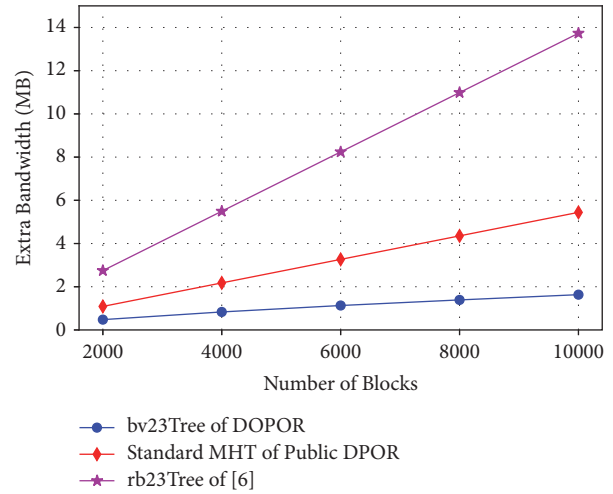

Figure 6: Extra bandwidth cost consumed by client for verifying the leaves of all blocks batch-read from CSS, according to different authenticated data structures.

with the correct proof table to pass client's batch verifications upon these leaves. □

6. Performance Evaluation

Our experiments were deployed using Python language on the Linux system with Intel Xeon E5-2609 CPU running at 2.40 GHz, 16 GB of RAM, and 7200 RPM 600 GB Serial ATA drive with a 32 MB buffer. The cryptographic operations were implemented based on the Python Cryptography Toolkit [22] and Pypbc library [23], and we used the 80-bit security parameter that means the order p of group G is of 160-bit length. We chose 1 GB raw data file F for testing and relied on the (9, 12) erasure code for encoding. For ease of comparison, all block sizes are set to 4 KB as in [6, 14]. Our results are an average of 20 rounds.

6.1. POR Audits Cost. During each POR audit, since the number of challenged blocks $|I|$ is far less than the total number of encoded file blocks $\widetilde{n}$ (e.g., the percent $|I|/\widetilde{n} = 1\%$ as in [12]), the time consumed in proof computation (or proof verification) will not be the bottleneck for CSS (or TPA). The POR audit phase of DOPOR corresponds to the execution of the public verification construction of [8], the efficient computation performance of which has been confirmed as shown in previous studies [5, 6, 10]. Therefore, the computation time of POR audit phase is not the primary concern in our DOPOR scheme, and we will focus on evaluating the bandwidth cost of this phase. Figure 5 depicts the total TPA-CSS bandwidth cost for executing POR audit once, for various percents of challenged blocks. Here, with regard to the given parameters $c\ell$ and λ in DOPOR, the percentage of challenged blocks is equal to $c\ell/\lambda$. It is obvious that the public DPOR of [14] results in a large communication overhead since it must transfer all challenged blocks during each audit, which greatly affects the bandwidth performance. By relying on the technologies of blockless verification and homomorphic authenticators (tags) to compress the proof size, the bandwidth costs of both our DOPOR and the static OPOR (i.e., Fortress) of [12] are only dominated by the sizes of challenges released from TPA and thus gradually increase with the percents of challenged blocks. Note that the bandwidth cost of DOPOR is always less than that of Fortress, since TPA only needs to send a single challenge for each audit in DOPOR, but there are two parallel challenges for TPA to be sent in Fortress. This is due to the fact that, during each audit, Fortress requires CSS to respond with two different responses [12]: one is used by TPA for auditing CSS and the other will be used by client for checking TPA's work. And thus these two responses correspond to two parallel challenges in Fortress. However, as shown in Section 4.4, DOPOR enables CSS to respond with only one response, which is based on the public key cryptosystem to support both TPA's auditing and client's checking. So, within DOPOR, there is only one challenge corresponding to the above sole CSS's response.

6.2. Read Cost. When client reads a batch of raw blocks from CSS, the integrity of these blocks is guaranteed by the authenticated data structure of the up-to-date buffer U. In Figure 6, we evaluate the extra bandwidth cost (i.e., not including the bandwidth of transferring the blocks themselves) incurred on client side for batch-verifying the leaves of all returned raw blocks with the proposed proof table of bv23Tree, when compared to the costs of the rb23Tree method [6] and the standard MHT method [14] that can only verify all appointed leaves by transferring their respective proof paths. As shown, with the increasing number of blocks batch-read from CSS, the extra bandwidth cost caused by rb23Tree is much higher than MHT, since the basic component of proof path of rb23Tree is an 8-element tuple *mark* with a larger size. However, owing to the proof table that avoids transferring all repetitive node values within different proof paths, our bv23Tree incurs the lowest extra bandwidth cost among the three dynamic structures. Likewise, as shown

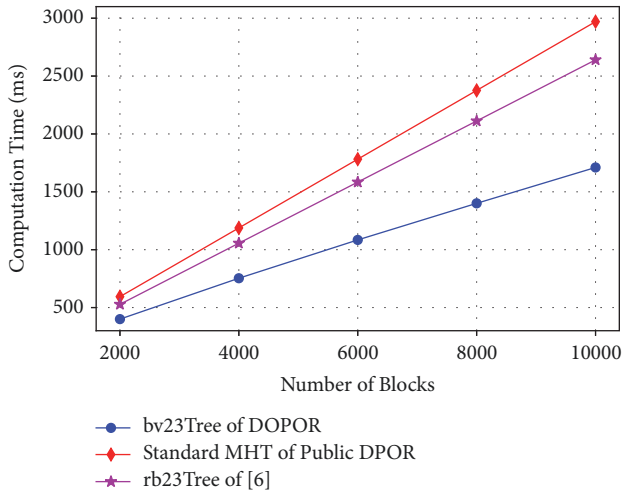

Figure 7: Computation time spent by client for verifying the integrity of all blocks batch-read from CSS, according to different authenticated data structures.

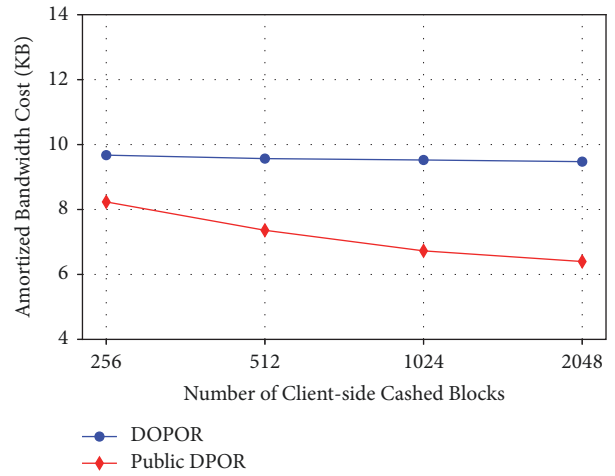

Figure 8: Client-CSS amortized bandwidth cost for each block when writing all cached blocks in a batch to CSS.

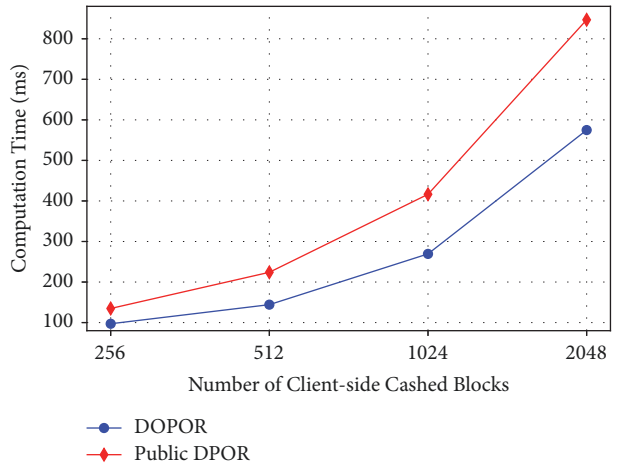

Figure 9: Computation time spent by CSS when writing all cached blocks in a batch to CSS.

in Figure 7, based on bv23Tree, client also further reduces the computation time spent for verifying the integrity of all returned blocks, due to the fact that the proof table enables client to batch-verify all returned leaves together just by computing the tree root hash once, avoiding the straightforward way of rb23Tree and standard MHT that client has to verify different leaves one after another by repeatedly computing the tree root hash with different proof paths.

In conclusion, our results show that the more the blocks batch-read from CSS, the more the repetitive node values omitted for transferring and computing according to proof table, and thus the client will save more costs based on bv23Tree for improving the performance of reads.

6.3. Write Cost. Now, we evaluate the performance of writes for DOPOR and the public DPOR of [14], both of which apply the client-side cache measure for performing writes; that is, client will cache locally a group of raw blocks (contained in the update operations of modification or insertion as in Section 3.4) and write these blocks in a batch to CSS, as shown in the protocol **PerformUpdates**(·) of Section 4.3. For DOPOR of this experiment, the parameter c is the number of client-side cached blocks, which determines the parameter λ according to the erasure-coding rate.

Figure 8 depicts the client-CSS amortized bandwidth cost for writing each 4 KB raw block. With the increasing number of client-side cached blocks, our results show that DOPOR incurs 17%~48% more amortized bandwidth than the public DPOR, due to the fact that DOPOR needs to transfer the additional encoded operations blocks besides the raw blocks. However, recall that the required bandwidth cost for frequent POR audits in the public DPOR is orders of magnitude higher than that in DOPOR (Figure 5), and DOPOR achieves a stronger security level by protecting against malicious TPA and collusion than the public DPOR. Furthermore, as shown in Figure 9, the public DPOR incurs an average of 45% more computation time at CSS side than DOPOR, since during performing writes the public DPOR must rebuild the corresponding levels of an MHT-based hierarchical structure located on CSS's disk, which results in a lot of additional disk I/O time when compared to DOPOR that does not need to do such rebuilding.

6.4. Client-Side Checking Cost. As shown in [12], although client should not be embroiled in the most frequent POR audits, it is necessary to give client the capability of checking TPA's past work to protect against the malicious TPA.

Since both DOPOR and the Fortress scheme of [12] adopt the aggregation technology to compress the proof size, the client-TPA bandwidth costs during checking TPA are alike for these two schemes, so in this experiment we focus on measuring the client's computation time of two investigated schemes when batch-checking TPA's logs, as shown in Figure 10. Here, one log corresponds to a past POR audit performed by TPA with challenging 1% of the total

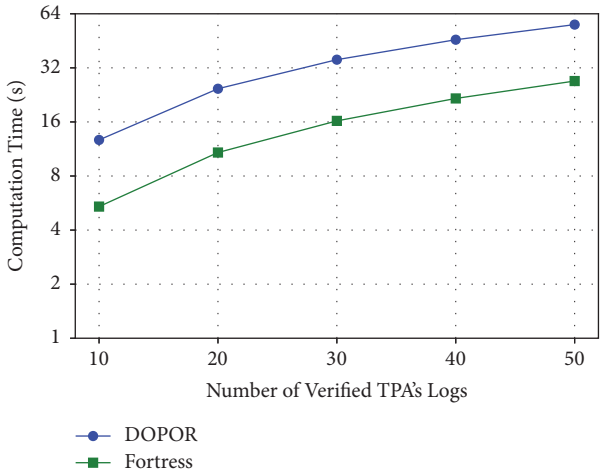

Figure 10: Computation time spent by client for batch-checking TPA's logs (including the time to access the bitcoin source).

encoded blocks. Compared to Fortress, DOPOR requires more time at client side for checking TPA, due to the fact that the exponentiation operation on the elliptic curve of DOPOR incurs more computation cost than the module operation of Fortress. However, recall that DOPOR enables the efficient data dynamics that cannot be supported in Fortress, so the additional client-side time cost incurred by DOPOR, that is, the distance between the line of DOPOR and that of Fortress as in Figure 10, can be regarded as the price for dynamism under the OPOR setting. Indeed, this additional dynamism cost can be tolerated by client to a great extent, since the client's checking against TPA is an optional verification and is only seldom executed in practice [12]; for example, client might just batch-check TPA's logs once in several months or even a year.

7. Related Work

Nowadays, with the rapid development of cloud computing, more and more cloud applications are designed upon the big data stored at CSS side, such as the service quality evaluation [24] and cloud service recommendation [25, 26]. However, how to guarantee the storage security of the big data is a critical challenge for mobile clients in the setting of cloud computing. Proof of Retrievability (POR) is a kind of security measure that builds upon cryptographic proofs to ensure the correctness and retrievability of client's big data outsourced to cloud. Juels and Kaliski Jr. [7] proposed the first POR scheme by utilizing the "sentinels" technique, where client can conceal some sentinel blocks among other original data blocks for remote POR audits before outsourcing her data. But this proposal can only support a limited number of POR audits, since performing the audits will expose the corresponding sentinels, so the frequent audits cannot be sustained once all sentinels are exhausted. Based on the pseudorandom functions (PRFs) and BLS signatures [21], Shacham and Waters [8] proposed two improved POR schemes with private verification and public verification, respectively. Both of these schemes enable an unlimited number of audits against CSS and simultaneously compress the response of CSS into one aggregated block along with a small authenticator value for optimized auditing bandwidth. Subsequently, Dodis et al. [27] generalized the constructions of [7, 8] by combining the concepts of POR with the coding and complexity theory. In view of the importance of data dynamics, Cash et al. [18] provided a Dynamic POR (DPOR) scheme based on the ORAM technique. Because ORAM will incur the heavy bandwidth overhead for client when performing dynamic updates under the POR setting, by replacing ORAM with the FFT-based constructible code and the hierarchical storage structure, Shi et al. [14] designed a more efficient private DPOR scheme than that of [18] and simultaneously applied the MHT structure to turn this private DPOR into a public DPOR scheme. Furthermore, with the observations made upon previous POR studies, Etemad and Küpçü [19] proposed a general framework to construct efficient DPOR and defend against the selective deletion attack described in [14].

On the other hand, Provable Data Possession (PDP), first proposed by Ateniese et al. [17], is a closely related research direction that focuses on ensuring the integrity of outsourced data. The difference between POR and PDP is that POR applies the erasure code but PDP does not. As shown in [19], the security level of POR is stronger than that of PDP, since POR ensures that *the whole* outsourced data can be retrieved by client, when compared to PDP that only guarantees the integrity of *most of* the outsourced data. Given that some existing public auditing schemes [5, 15, 16] are designed without involving erasure code, these schemes can be classed as the variants of PDP. Zhu et al. [4] presented a cooperative PDP (CPDP) scheme for distributed Multicloud Storage setting. Wang et al. [5] designed the random masking technology to protect client's outsourced data from leaking to TPA during the audits. Erway et al. [13] proposed the first Dynamic PDP (DPDP) scheme to support efficient data updates using Skip List. And then variant authenticated structures were proposed for data dynamics, such as the standard MHT method [9], rank-based MHT [15], multireplica MHT [16], and rb23Tree [6]. However, all these authenticated structures can only verify different leaves one by one, which is an inefficient way for client when there are many leaves that need to be verified.

In addition, as shown in [12], when referring to public verification (auditing), the potential security risk is that TPA might also be malicious. But this risk has not been considered by all the above public schemes. Outsourced Proof of Retrievability (OPOR), proposed by Armknecht et al. [12], is the first scheme to protect against malicious TPA under the public POR setting. However, the OPOR construction of [12] only supports the static data, which is the limitation that should be further solved.

8. Conclusions

As a stronger security model in the context of remote data auditing, Outsourced Proof of Retrievability (OPOR) focuses on dealing with the dilemma that client hopes to resort to TPA for assessing the storage security of her outsourced

data, while TPA might be malicious and collude with CSS to cheat client. In this paper, we propose a concrete DOPOR scheme to support data dynamics under the environment of OPOR. Our DOPOR scheme is constructed based on a newly designed authenticated data structure, called bv23Tree, which not only relies on the property of balanced tree to guarantee the expected logarithmic complexity in any case of dynamic updates, but also enables client to batch-verify multiple appointed leaves all together for improved performance. Under the setting of employing erasure code, by separating the updated data from the original data and adopting the hierarchical structure of same-sized levels to uniformly store all encoded data, DOPOR can efficiently support batch reads and updates upon outsourced storage according to the feature of batch verifications of bv23Tree. When compared to the state of the art, our experiments show that DOPOR incurs a lower bandwidth cost for frequent TPA's audits than the original static OPOR scheme, and the overall performance of DOPOR for reads and writes is comparable to that of existing public Dynamic POR scheme.

Conflicts of Interest

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

Acknowledgments

This work is supported by NSFC (Grant no. 61502044).

References

[1] S. Guarino, E. S. Canlar, M. Conti, R. Di Pietro, and A. Solanas, "Provable Storage Medium for Data Storage Outsourcing," *IEEE Transactions on Services Computing*, vol. 8, no. 6, pp. 985–997, 2015.

[2] J. Yu, K. Ren, and C. Wang, "Enabling cloud storage auditing with verifiable outsourcing of key updates," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1362–1375, 2016.

[3] K. He, J. Chen, R. Du, Q. Wu, G. Xue, and X. Zhang, "DeyPoS: Deduplicatable Dynamic Proof of Storage for Multi-User Environments," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3631–3645, 2016.

[4] Y. Zhu, H. Hu, G.-J. Ahn, and M. Yu, "Cooperative provable data possession for integrity verification in multicloud storage," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2231–2244, 2012.

[5] C. Wang, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for data storage security in cloud computing," in *Proceedings of the IEEE INFO-COM*, pp. 525–533, March 2010.

[6] Z. Ren, L. Wang, Q. Wang, and M. Xu, "Dynamic proofs of retrievability for coded cloud storage systems," *IEEE Transactions on Services Computing*, vol. PP, no. 99, 2015.

[7] A. Juels and B. S. Kaliski Jr., "Pors: proofs of retrievability for large files," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, pp. 584–597, ACM, Alexandria, VA, USA, November 2007.

[8] H. Shacham and B. Waters, "Compact proofs of retrievability," in *Advances in Cryptology—ASIACRYPT 2008: Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 2008*, vol. 5350 of *Lecture Notes in Computer Science*, pp. 90–107, Springer, Berlin, Germany, 2008.

[9] Q.-A. Wang, C. Wang, K. Ren, W.-J. Lou, and J. Li, "Enabling public auditability and data dynamics for storage security in cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 5, pp. 847–859, 2011.

[10] C. Wang, S. S. M. Chow, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for secure cloud storage," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 362–375, 2013.

[11] H. Tian, Y. Chen, C. C. Chang, H. Jiang, Y. Huang, and J. Liu, "Dynamic-hash-table based public auditing for secure cloud storage," *IEEE Trans. Services Comput*, 2015.

[12] F. Armknecht, J.-M. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter, "Outsourced proofs of retrievability," in *Proceedings of the 21st ACM Conference on Computer and Communications Security, CCS 2014*, pp. 831–843, USA, November 2014.

[13] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *Proceedings of the Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, pp. 213–222, ACM, Chicago, Ill, USA, November 2009.

[14] E. Shi, E. Stefanov, and C. Papamanthou, "Practical dynamic proofs of retrievability," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013*, pp. 325–336, Germany, November 2013.

[15] C. Liu, J. Chen, L. T. Yang et al., "Authorized public auditing of dynamic big data storage on cloud with efficient verifiable fine-grained updates," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 9, pp. 2234–2244, 2014.

[16] C. Liu, R. Ranjan, C. Yang, X. Zhang, L. Wang, and J. Chen, "MuR-DPA: Top-Down Levelled Multi-Replica Merkle Hash Tree Based Secure Public Auditing for Dynamic Big Data Storage on Cloud," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2609–2622, 2015.

[17] G. Ateniese, R. Burns, R. Curtmola et al., "Provable data possession at untrusted stores," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, pp. 598–609, Virginia, Va, USA, November 2007.

[18] D. Cash, A. Küpçü, and D. Wichs, "Dynamic Proofs of Retrievability via Oblivious RAM," in *Advances in Cryptology – EUROCRYPT 2013*, vol. 7881 of *Lecture Notes in Computer Science*, pp. 279–295, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[19] M. Etemad and A. Küpçü, "Generic efficient dynamic proofs of retrievability," in *Proceedings of the 8th ACM Cloud Computing Security Workshop, CCSW 2016*, pp. 85–96, Austria.

[20] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley Publishing Co., Reading, Mass.-London-Amsterdam, 1975.

[21] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," in *Advances in cryptology—ASIACRYPT 2001 (Gold Coast)*, vol. 2248 of *Lecture Notes in Comput. Sci.*, pp. 514–532, Springer, Berlin, 2001.

[22] "Python Cryptography Toolkit (PyCrypto)," 2014, https://pypi.python.org/pypi/pycrypto.

[23] "Python 3 bindings for libpbc (Pypbc)," 2017, https://github.com/debatem1/pypbc.

[24] L. Qi, W. Dou, Y. Zhou, J. Yu, and C. Hu, "A context-aware service evaluation approach over big data for cloud applications," *IEEE Transactions on Cloud Computing*, 1 page, 2015.

[25] L. Qi, X. Xu, X. Zhang et al., "Structural balance theory-based e-commerce recommendation over big rating data," *IEEE Transactions on Big Data*, 2016.

[26] L. Qi, X. Zhang, W. Dou, and Q. Ni, "A Distributed Locality-Sensitive Hashing-Based Approach for Cloud Service Recommendation From Multi-Source Data," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2616–2624, 2017.

[27] Y. Dodis, S. Vadhan, and D. Wichs, "Proofs of Retrievability via Hardness Amplification," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 5444, pp. 109–127, 2009.