Transactions on Parallel and Distributed Systems Auditing Cache Data Integrity in the Edge Computing Environment

Article *in* IEEE Transactions on Parallel and Distributed Systems · December 2020
DOI: 10.1109/TPDS.2020.3043755

Bo Li
Swinburne University of Technology

Feifei Chen
Swinburne University of Technology

Yun Yang
Fudan University

Qiang He
Swinburne University of Technology

Auditing Cache Data Integrity in the Edge Computing Environment

Bo Li, Qiang He, *Senior Member, IEEE,* Feifei Chen, *Member, IEEE,* Hai Jin, *Fellow, IEEE,* Yang Xiang, *Fellow, IEEE,* and Yun Yang, *Senior Member, IEEE*

Abstract—Edge computing allows app vendors to deploy their applications and relevant data on distributed edge servers to serve nearby users. Caching data on edge servers can minimize users' data retrieval latency. However, such cache data are subject to both intentional and accidental corruption in the highly distributed, dynamic, and volatile edge computing environment. Given a large number of edge servers and their limited computing resources, how to effectively and efficiently audit the integrity of app vendors' cache data is a critical and challenging problem. This paper makes the first attempt to tackle this Edge Data Integrity (EDI) problem. We first analyze the threat model and the audit objectives, then propose a lightweight sampling-based probabilistic approach, namely EDI-V, to help app vendors audit the integrity of their data cached on a large scale of edge servers. We propose a new data structure named variable Merkle hash tree (VMHT) for generating the integrity proofs of those data replicas during the audit. VMHT can ensure the audit accuracy of EDI-V by maintaining sampling uniformity. EDI-V allows app vendors to inspect their cache data and locate the corrupted ones efficiently and effectively. Both theoretical analysis and comprehensively experimental evaluation demonstrate the efficiency and effectiveness of EDI-V.

Index Terms—Edge computing, data integrity, data cache, data replica, integrity audit, Merkle hash tree.

1 Introduction

In recent years, the world has witnessed an explosive growth of mobile and Internet-of-Things (IoT) devices. This significantly fuels the variety and sophistication of online applications, such as interactive networked gaming, virtual reality (VR), video analysis and natural language processing [1]. A lot of applications are becoming more and more latency-sensitive and resource-intensive. The high and often unpredictable latency between the cloud and end-users is rendering the conventional cloud computing paradigm unsuitable [2]. App vendors are in urgent need of the ability to deploy their applications in close proximity to end-users to fulfill the increasingly stringent latency requirements [3].

Since it was proposed in 2012 [2], the edge computing (EC) paradigm has gained increasing attention and become a key technique that facilitates the 5G network [4]. In the EC environment, edge servers are equipped with cloud-like computing resources and are attached to base stations or access points located near mobile or Internet-of-Things (IoT) users [5]. App vendors can deploy applications and cache data on those edge servers in an area to serve users with low latency [6]. We take Facebook's new VR application - Horizon [1] as an example. It is a typical application that can significantly benefit from the low latency offered by edge computing because VR users are highly sensitive to latency. Fig. 1 presents a geographical area with five edge servers, denoted as $S_1, ..., S_5$, and twelve Facebook Horizon users. Facebook caches five replicas of a popular VR video D, denoted as $D_1, ..., D_5$, on these edge servers to serve the users within this area. In this way, the Facebook Horizon users in this area can retrieve VR video D from their nearby edge servers with low latency rather than from the remote cloud.

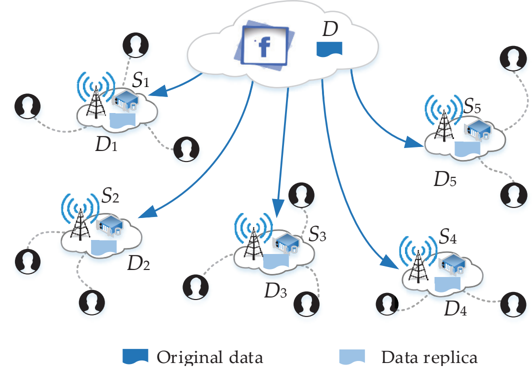


Fig. 1. Facebook Horizon in the EC Environment

However, operating in the highly distributed, dynamic, and volatile edge computing environment, edge servers must not be assumed reliable or trustworthy [7], [8], as they cannot be maintained in-house as those cloud servers. Given limited computing resources and protection mechanisms, data cached on edge servers (referred to as *edge*

- B. Li, Q. He, Y. Xiang and Y. Yang are with School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, Australia. Email: boli@swin.edu.au; qhe@swin.edu.au; yxiang@swin.edu.au; yyang@swin.edu.au.
- F. Chen is with the School of Information Technology, Deakin University, Geelong, Australia. E-mail: feifei.chen@deakin.edu.au.
- H. Jin is with Services Computing Technology and System Lab, Big Data Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China. Email: hjin@hust.edu.cn.

Manuscript received January XX, 2020; revised August XX, 2020.

1. https://www.oculus.com/facebookhorizon/

data hereafter) are subject to intentional and accidental corruptions caused by various events [9], [10]. For example, during a cyber attack, a hacker may delete or tamper with cached edge data replicas. Besides, edge servers may suffer from sudden hardware or software exceptions that can also cause data corruption. From the app vendor's perspective, unauthorized data modifications or data corruption must be detected in a timely manner to ensure the integrity of its edge data. We refer to this problem as the *edge data integrity* (EDI) problem. It is a challenging problem as those edge data are cached on distributed edge servers rather than app vendors' local servers [11].

The data integrity problem has been extensively studied in the cloud computing environment. The provable data possession (PDP) scheme and its variants [12], [13], [14], are the most popular approaches for solving the *cloud data integrity* (CDI) problem, i.e., helping data owners verify the integrity of their data stored in the remote cloud. However, **the EDI problem is fundamentally different from the CDI problem. Its unique characteristics render all the CDI approaches unsuitable in the EC environment.**

First, when verifying the integrity of multiple edge data replicas, a CDI approach only produces a result of Yes or No, with Yes indicating the integrity of all the edge data replicas and No otherwise. Such a result provides app vendors with limited (yet almost useless) information. In addition to whether there is data corruption, an EDI approach must be able to find out where the data corruption is, i.e., which individual edge data replicas are corrupted.

Second, in the EC environment, app vendors raise requests to verify the integrity of their data cached on edge servers. Edge servers respond to these requests by generating and returning data integrity proofs. This is a computationally expensive task in CDI schemes. However, edge servers' computing resources are constrained due to their limited size [15]. Besides, their computing resources are usually shared by multiple app vendors. Thus, an EDI approach must be lightweight on edge servers.

Third, in the EC environment, an app vendor usually needs to cache many data on a large number of edge servers distributed all around the world. Challenging the integrity of massive edge data individually with an CDI approach incurs significant computation costs to the app vendor based on the pay-as-you-go pricing model. Thus, an EDI approach must also be lightweight on app vendors.

Hence, a new cache data integrity audit approach that can accommodate these unique characteristics of the EC environment is in urgent need of app vendors. To tackle this challenge, this paper proposes a novel approach, namely EDI-V, to help app vendors verify the integrity of their edge data and locate corrupted ones based on a new data structure named variable Merkle hash tree (VMHT). EDI-V offers a lightweight and probabilistic solution to the EDI problem by employing the random sampling technique to select part of the VMHT for audit. **To our best knowledge, EDI-V is the first attempt at the EDI problem.** The main contributions of this paper are:

- We study the EDI problem from the app vendors' perspective, who are the owner of those edge data. We point out the threats to the edge data integrity and the objectives of EDI approaches.
- We propose EDI-V, a novel approach to help app vendors efficiently audit the integrity of their edge data and locate the corrupted ones across multiple edge servers. EDI-V offers a probabilistic integrity guarantee.
- We design a new data structure named variable Merkle hash tree. It can not only help generate data integrity proofs efficiently but also ensure the sampling uniformity to ensure the audit accuracy of EDI-V.
- We theoretically analyze and experimentally evaluate the performance of EDI-V. The results demonstrate that EDI-V is capable of auditing the edge data integrity and locating corrupted ones across massive edge servers effectively and efficiently.

The rest of this paper is organized as follows. Section 2 gives an overview for EDI. Section 3 introduces the new variable Merkle hash tree. Section 4 presents the general process of EDI-V, and then discusses the algorithms employed by EDI-V in detail. Section 5 theoretically analyzes the performance of EDI-V. Section 6 experimentally evaluates EDI-V against two representative approaches. Section 7 reviews the related work. Finally, Section 8 summarizes this paper and points out future work.

2 EDI Overview

In this section, we introduce the potential threats to the integrity of edge data in the edge computing environment. Next, we define the objectives of EDI. Then, we introduce the main methodologies used by our EDI-V.

2.1 EDI Threats

In the EC environment, there are two main threats to the edge data integrity.

Threat 1 Unexpected Failures. This can be caused by various factors, such as hardware faults, software exceptions, and cyber attacks. Such failures can lead to data corruption on edge servers.

Threat 2 Cheating Attacks. Cheating attacks may happen during the audit process and prevent app vendors from finding corrupted edge data replicas. First, the edge infrastructure provider (EIP) may not be honest. They may pretend that all the cache data are integral for self-benefit when some of the data are actually corrupted or even lost. In addition, hackers also want to deceit the audit process to hide traces of their attacks. Specifically, EIP and hackers may save a correct data integrity proof previously generated and use it in future audits. This is referred to as the *replay attack* [16], [17], [18]. They may also forge the data integrity proof during the audit to deceive the app vendor. This is referred to as the *forge attack* [19], [20].

Similar to the previous work [16], [17], [18], [19], [20], we assume in this work that the edge servers can correctly respond to the app vendor's requests.

2.2 EDI Objectives

An EDI approach must achieve the three objectives below to correctly, efficiently and securely audit the integrity of app vendors' edge data replicas:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.3043755, IEEE Transactions on Parallel and Distributed Systems

Objective 1 Correctness. Given an original data, an EDI approach should be able to correctly audit the integrity of every cached data replica and point out the corrupted ones.

Objective 2 Lightweight. There are two aspects in terms of lightweight. First, to accommodate the resource limitation of edge servers, the audit process must be performed with low computational overheads on edge servers. Second, to reduce the costs of app vendors, the computational overheads on the app vendors must also be minimal.

Objective 3 Security. An EDI approach has to provide effective strategies to prevent the EIP and hackers from cheating during the audit process, in particular, from implementing the replay attacks and forge attacks discussed in Section 2.1.

2.3 Main Methodologies

EDI-V employs the following methodologies to achieve the above objectives. In general, to audit the integrity of those data replicas, EDI-V generates a signature for each data replica, and then periodically verifies those signatures to audit the integrity.

Method 1 Generating signature based on Merkle hash tree. A signature is an integrity proof of a data replica. In recent years, Merkle hash tree (MHT) has been widely used to generate data signatures in various distributed systems, such as Tor, Bitcoin, Git [21] and in particular, cloud systems [22], [23], [24]. Fig. 2 shows an example MHT generated from data D, which is a binary tree. Each of its leaf nodes has a hash value of one data block of D. Each of its non-leaf nodes has a hash value of its two child nodes. The root node of the MHT has a hash value of the entire data D as its signature. EDI-V employs a new variable Merkle hash tree (VMHT) to facilitate efficient signature generation for edge data. It will be discussed in detail in Section 3.

Method 2 Reducing complexity via sampling technique. Sampling technique can help reduce the computational overheads incurred by data integrity audit [25], [26], [19]. It provides a probabilistic data integrity guarantee by sampling only a small portion of data blocks. EDI-V implements the sampling technique based on VMHT to provide a lightweight solution to the EDI problem. Briefly, it uses only a specific subtree of the entire VMHT for audit on the edge server side. Given k data replicas to be verified, EDI-V first generates an entire VMHT T on the app vendor side. Then, it randomly samples a total of k subtrees from T. Each subtree T_i is used to audit data replica D_i on edge server S_i. Edge server S_i only needs to generate a corresponding subtree T_i' based on its data replica D_i for the audit. This method can significantly reduce the computational overheads on edge servers. Its application in EDI-V will be discussed in Section 4.2.

Method 3 Ensuring audit accuracy via data block shuffling. To ensure audit accuracy, data blocks must be sampled evenly from D. However, the structure of the VMHT T is fixed when the number of data blocks is given. Then, each individual subtree of T is generated based on a specific set of data blocks. Thus, the audit based on a substree of T is always performed based on the corresponding set of data blocks. We take Fig. 2 as an example. The audit based on the subtree with node n_2^3 as the root node will always be performed on data blocks d_3 and d_4. The audit based on subtree with n_3^3 as its root node will always be performed on data blocks d_5 and d_6. In this way, data block combinations like d_4 and d_5 will never be sampled for the audit. To address this issue, EDI-V shuffles all the blocks of a data replica before sampling. For example, as shown in Fig. 3, given a data replica with 8 blocks, i.e., $d_1, d_2, ..., d_8$, EDI-V generates a random number sh ($sh < m$) and selects a pseudorandom permutation function $Pr(x, y)$ where x is the seed and y is the total number of elements to be permuted. Then, it shuffles those data blocks according to $Pr(sh, 8)$. In this way, data blocks are sampled evenly for the audit.

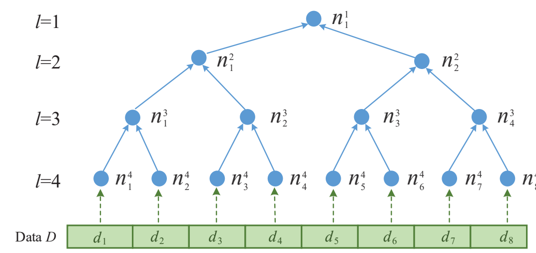


Fig. 2. An Example of an Ideal Merkle Hash Tree

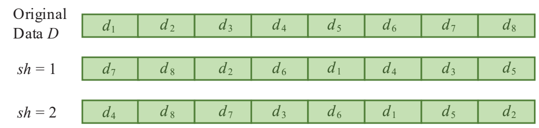


Fig. 3. An Illustration of Shuffling Data Blocks

Method 4 Performing secure audit with random security code. To prevent EIP and hackers from replaying an outdated but correct signature to fool the audit, EDI-V employs the one-time encryption method when generating the signatures. For each audit, EDI-V randomly selects a security code g and delegates it to all edge servers. On the edge server side, EDI-V combines g with each sampled data block to generate the corresponding hash tag of each data block when constructing the VMHT. As each g is used only once, the data signature can be used for only one audit. It helps EDI-V defend against the cheating attacks discussed in Section 2.1.

By integrating the above techniques, EDI-V can solve the EDI problem effectively and efficiently.

3 Variable Merkle Hash Tree

Although the MHT is an effective tool for generating data signatures, it can not be directly used to tackle the EDI problem. In this section, we first introduce the limitation of the traditional MHT, and then discuss our new variable Merkle hash tree that overcomes this limitation. The notations used in this paper are summarized in Table 1.

3.1 Limitation of Traditional MHT

As introduced in Section 2.3, the traditional MHT is a binary tree constructed based on a number of data blocks. For example, as shown in Fig. 2, given a data with 8 blocks, the corresponding MHT T has 8 leaf nodes, i.e., each leaf node corresponds to one block and stores its hash value.

TABLE 1
Key Notations

Notations	Meanings
D	the original data
S_i	the ith edge server
SI_i	identity of the ith edge server
D_i	the ith data replica
DI_i	identity of the ith replica
d_j^i	the jth data block of D_i
m	the number of blocks of a data
k	the number of replicas
$Hash()$	a specific hash function
$Pr()$	a pseudorandom permutation function
H	the height of a tree
l	the level of a tree node
r	the position of a tree node on the same level
n_r^l	tree node on level l with the position of r
T_r^l	a subtree with node n_r^l as its root node
v_r^l	the value of the tree node n_r^l
ln_r^l	the number of leaf node in T_r^l
τ_r^l	the hash value stored in node n_r^l
cn_r^l	the number of tree node n_r^l's children
R	a set of audit requests
R_i	the ith audit request in R
P_i	integrity proof of data replica D_i
sh	a random key used for shuffling data blocks
g	a security code used for generating hash tags
ns	the number of sampled data blocks
nc	the number of corrupted or tempered data blocks
rs	the size of the original data and replica
bs	the size of each data block
ss	sampling scale in each audit request

Every two nodes construct one node as their parent node. For example, nodes n_1^4 and n_2^4 construct node n_1^3 as their parent node. This construction process iterates several times until the root node of the MHT is constructed, such as n_1^1 in Fig. 2.

However, most of the time, the number of blocks of a cache data is not exactly a power of 2. To tackle this problem, when there are not enough nodes to construct their parent node, the last node as well as the corresponding subtree will be duplicated. For example, there are 9 blocks in Fig. 4, thus the 9th leaf node n_9^5 will be duplicated to construct its parent node n_5^4. Then, node n_5^4 and its subtree will be duplicated to construct n_3^3, and so forth. This mechanism makes MHT unsuitable for EDI-V. When EDI-V randomly selects a subtree from the MHT, it may cover vastly different numbers of data blocks, which significantly undermines the sampling uniformity. For example, when sampling a subtree from Fig. 4 with n_1^3 as the root node, it covers 4 data blocks, i.e., d_1 to d_4. However, when sampling a subtree with n_3^3 as the root node, it covers only 1 data block, i.e., d_9. In extreme cases, the right half of the entire MHT covers only one real data block. This will damage the sampling uniformity and reduce the effectiveness of EDI-V.

3.2 VMHT

To overcome the limitation of the traditional MHT, we propose a novel Merkle hash tree named variable Merkle hash tree (VMHT).

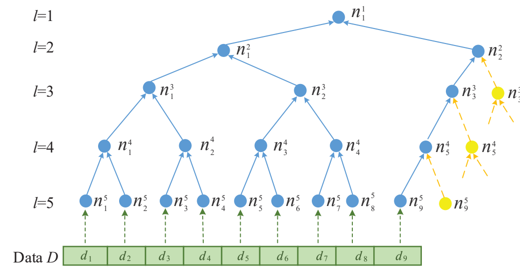


Fig. 4. An Example of Traditional Merkle Hash Tree

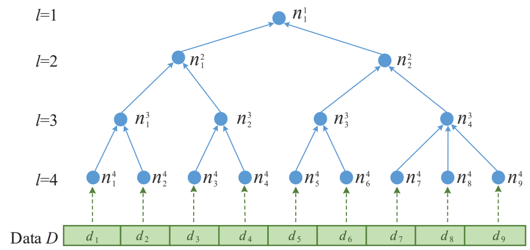


Fig. 5. An Example VMHT Generated from 9 Data Blocks

Definition 1 (Variable Merkle Hash Tree)**.** *A VMHT is an ordered Merkle hash tree, where each non-leaf node has exactly two child nodes but the rightmost one on each level can have either two or three child nodes. VMHT possesses three unique characteristics:*

1) *A VMHT is an ordered tree. Every node contains two serial numbers indicating its level l and position r on level l. In a VMHT, l starts from 1 and increases in top-down order, i.e., the root node's level is 1 and its children's level is 2, and so forth. Nodes on the same level are ordered from 1 from left to right. The leftmost node on each level has a position of 1, i.e., its $r = 1$, and its right cousin node has a position of 2, and so forth. A node on level l with position r is denoted as n_r^l. The subtree with node n_r^l as its root node is denoted as T_r^l.*
2) *The last non-leaf node on each level can have either two or three child nodes while other non-leaf nodes have exactly two child nodes. In addition, for each non-leaf node n_i^l on level l, when n_i^l is not the rightmost node, the corresponding subtree T_i^l is a full binary tree.*
3) *Each node n_r^l has a tuple with five elements, denoted as $v_r^l = < l, r, ln_r^l, cn_r^l, \tau_r^l >$, where ln_r^l denotes the number of leaf nodes in tree T_r^l and cn_r^l denotes the number of child nodes it has (2 or 3 for non-leaf nodes and 0 for leaf nodes). τ_r^l is the hash tag. When n_r^l is a leaf node, τ_r^l is the hash value of the corresponding data block d_r, i.e., $\tau_r^l = Hash(d_i)$, where $Hash()$ can be any hash function, such as MD5, SHA-1, SHA-2, etc. When n_r^l is a non-leaf node, τ_r^l is the hash value of all its child nodes' hash tags.*

Fig. 5 demonstrates a 4-level VMHT with 16 nodes generated from an edge data with 9 data blocks. Each leaf node on level 4 ($l = 4$) holds the hash value of the corresponding data block. For example, node n_2^4 holds the hash value of data block d_2, i.e., $n_2^4.\tau = Hash(d_2)$. Each non-leaf node holds the hash value of all its child nodes. For example, node n_1^2 holds the hash value of nodes n_1^3 and n_2^3, i.e., $n_1^2.\tau = Hash(n_1^3.\tau \| n_2^3.\tau)$, where $\|$ means concatenation. The value of node n_1^2 is $v_1^2 = < 2, 1, 4, 2, n_1^2.\tau >$, which means n_1^2's level is 2, position is 1, there are 4 leaf nodes in T_1^2, and n_1^2 has two child nodes. The hash tag can be deduced from $Hash(Hash(n_1^4.\tau \| n_2^4.\tau) \| Hash(n_3^4.\tau \| n_4^4.\tau))$, where $n_i^4.\tau = Hash(d_i)$ $(i \in [1, 4])$. In addition, T_1^3, T_2^3, T_3^3 and T_1^2 are full binary trees.

The advantages of VMHT are twofold. First, each subtree on the same level covers nearly the same number of data blocks. This can help ensure EDI-V's sampling uniformity. Second, it is easy to obtain a subtree without constructing the entire VMHT, i.e., edge servers can directly construct the sampled subtrees based on the corresponding data blocks.

Theorem 1. *Given a VMHT T of height H and its subtree T_r^l, the leftmost leaf node in T_r^l has an position of $2^{H-l}(r-1)+1$ in T on level H.*

Proof. *In VMHT T, node n_r^l has $r-1$ left cousin nodes, and each left cousin node has a corresponding full binary subtree whose height is $H-l+1$. Thus, each subtree has 2^{H-l} leaf nodes. There is a total of $r-1$ such subtrees. Hence, the position of the leftmost leaf node of n_r^l is $2^{H-l}(r-1)+1$.*

With Theorem 1, given a VMHT of height H, it is easy to find out the number of leaf nodes that each subtree has and where those leaf nodes are. Let us take Fig. 5 as an example. Given a subtree T_2^3 we have $r=2$ and $l=3$. Then, we can easily deduce by Theorem 1 that the position of the leftmost leaf node of T_2^3 is 3, i.e., the position of leaf node n_3^4 is 3. Through the tuple of node n_2^3, i.e., $v_2^3 =< 3, 2, 2, 2, n_2^3.\tau >$, we can find that there is a total of 2 leaf nodes in subtree T_2^3. Thus, the two leaf nodes are n_3^4 and n_4^4. In this way, we can directly construct the corresponding subtree T_2^3 without constructing the entire tree displayed in Fig. 5.

4 EDI-V Approach

This section presents and discusses EDI-V in detail.

4.1 Overview

With EDI-V, the app vendor employs a request-response process to audit the integrity of its edge data replicas. This process can be performed either regularly or on-demand. The general framework is shown in Fig. 6, which consists of four main phases:

Phase 1 Setup: In this phase, the app vendor first sets up some essential parameters for the audit. It chooses four random functions, one pseudorandom permutation function $Pr()$, and initializes their random seeds according to its security needs. Second, the app vendor generates the security code g and the entire VMHT based on the original data D.

Phase 2 Request: In this phase, the app vendor generates a set of audit requests and sends them to edge servers. Each edge server receives one unique request. A request contains the edge server's identity, data replica's identity, security code g, and relevant sampling parameters. Please note that different edge servers usually have different sampling parameters.

Phase 3 Response: In this phase, each edge server responds to the app vendor's audit request with a data integrity proof. In order to generate the proof, the edge server must construct a corresponding sub-VMHT.

Phase 4 Verification: After receiving all the responses, the app vendor inspects the received data integrity proofs

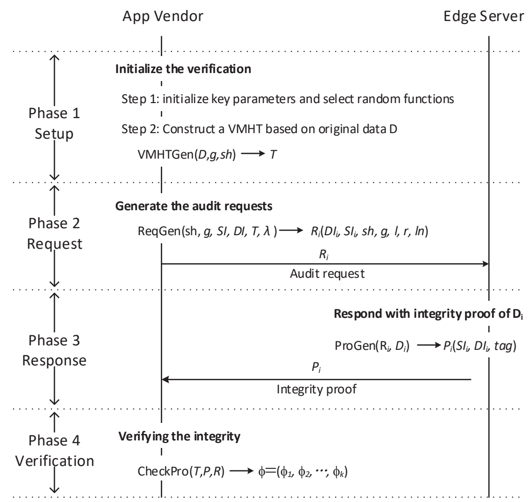


Fig. 6. Framework of the EDI-V Approach

against its own VMHT constructed in Phase 1. By inspecting them one by one, the integrity of each data replica can be audited, and the corrupted ones can be found.

4.2 Process and Algorithms

Now, we discuss the audit process and relevant algorithms in detail according to the general framework introduced in Section 4.1.

4.2.1 *Initialize the Audit*

In Phase 1 (Setup), the app vendor initializes essential parameters and constructs the VMHT T based on the original data D, which consists of two steps:

Step 1. The app vendor selects four random functions and initializes their random seeds. The first function generates the security code g. As introduced in Section 2.3, to protect the audit from replay attacks, EDI-V concatenates a security code g with each original data block when calculating the corresponding hash tags. Generally, to ensure a high level of security, the bit size of g must be adequately large. The second function randomly generates the shuffle key sh. The third and fourth functions generate the sampling parameters, such as the level l and the position r. It also selects the pseudorandom permutation function $Pr()$.

Step 2 Given an original data D with m data blocks, i.e., $D = \{d_1, d_2, ..., d_m\}$, the app vendor employs function **VMHTGen** (Algorithm 1) to construct the corresponding VMHT T.

Function VMHTGen takes the original data D, security code g and shuffle key sh as input and outputs a VMHT T. According to Definition 1, the last non-leaf node on each level in VMHT T can have either two or three child nodes, while others have exactly two child nodes. Thus, given m data blocks in D, the height of the corresponding VMHT, i.e., H, is equal to the height of a full binary tree with no more than m leaf nodes. Therefore, H satisfies $2^{H-1} \leq m < 2^H$, then $H = \lfloor log_2^m \rfloor + 1$.

First, it initializes the height of the T (line 1). Then, it initializes an empty queue Q for saving the current tree nodes (line 2). Next, it shuffles the data blocks of D according to

Algorithm 1 VMHTGen

Input: D, g, sh
Output: a variable Merkle hash tree T
1: initial $H \longleftarrow \lfloor log_2^m \rfloor + 1$
2: initial Queue $Q \longleftarrow null$
3: $D' \longleftarrow D$ s.t. $D' = \{d_{Pr(sh,m)[1]}, ..., d_{Pr(sh,m)[m]}\}$
4: **for** i in $[1, m]$ **do**
5: create leaf node n_i with tuple $(H, i, 1, 0, Hash(g \| d'_i))$
6: Q.add(n_i)
7: **end for**
8: $l \longleftarrow H$;
9: $r \longleftarrow 1$;
10: **while** Q.size > 1 **do**
11: $x \longleftarrow Q$.remove() // the first node in current Q
12: $y \longleftarrow Q$.remove() // the second node in current Q
13: $z \longleftarrow Q$.get(1) // the third node in current Q
14: $w \longleftarrow Q$.get(2) // the fourth node in current Q
15: **if** $z! = null$ and $w == null$ or $z.l \geq w.l$ **then**
16: // new node t has three children
17: create a new node t with tuple $(l-1, r, x.ln+y.ln+z.ln, 3, Hash(x.\tau \| y.\tau \| z.\tau))$
18: t.firstChild $\longleftarrow x$
19: t.secondChild $\longleftarrow y$
20: t.thirdChild $\longleftarrow Q$.remove() //node z
21: **else**
22: // new node t has two children
23: create a new node t with tuple $(l-1, r, x.ln + y.ln, 2, Hash(x.\tau \| y.\tau))$
24: t.firstChild $\longleftarrow x$
25: t.secondChild $\longleftarrow y$
26: **end if**
27: $r \longleftarrow r + 1$;
28: Q.add(t)
29: **if** Q.get(1).$l < l$ **then**
30: $l \longleftarrow l - 1$
31: $r \longleftarrow 1$
32: **end if**
33: **end while**
34: $t \longleftarrow Q$.remove()
35: **return** a VMHT T with root of t

$Pr(sh, m)$ and forms a new data D' (line 3). It then iterates (lines 5-6) to create a total of m leaf nodes based on all blocks of D' and adds them to Q. Each leaf node has a tuple, where $n_i.\tau$ is the hash value of the corresponding block d'_i combined with security code g (line 5). Variable l stores the current level during the construction. For example, when processing the leaf nodes, there is $l = H$. When processing the root node, there is $l = 1$. Variable r stores the position of the parent node to be constructed. Now the algorithm begins to construct the VMHT based on the nodes in Q, starting with the leaf nodes (line 8). The algorithm sets the position of the first node to be constructed to 1 (line 9). Then, it iterates lines 11-32 to generate all the non-leaf nodes in T. It fetches the first two nodes (lines 11-12) and gets the third and fourth nodes from Q (lines 13-14). Please note that the $remove()$ operation will delete the node from Q but the $get(i)$ operation only reads the value of the ith node in Q. It uses lines 15-26 to construct the corresponding parent node. If nodes x, y, z are on the same level but the fourth node w is empty or is on a different level, x, y, z are the only nodes left on the current level l , then nodes x, y, z have the same parent node t (lines 17-20). The parent node t's level is $l-1$. The number of its leaf nodes is equal to the number of its three child nodes' leaf nodes, i.e., $t.ln = x.ln + y.ln + z.ln$. The number of its child nodes is set to 3 (line 17). Otherwise, only nodes x and y have the same parent node t (lines 23-25). The position of the new parent node, i.e., $t.r$, increases by one after each construction (line 27). Then, the algorithm adds node t to Q (line 28). After all the nodes on the same level have been used to construct their parent nodes, it decreases variable l by 1 and resets the position r to 1 (lines 29-32). The above steps are iterated until there is only one node in Q. It is the root node of the final VMHT T.

4.2.2 Generate Requests

In Phase 2 (Request), the app vendor generates a set of audit requests and sends them individually to each corresponding edge server. An audit request is defined as follows:

Definition 2 (Audit Request)**.** *An audit request R_i is a tuple that consists of 7 elements, i.e., $R_i = < DI_i, SI_i, sh, g, l, r, ln >$, where DI_i is data replica D_i's identity, SI_i is edge server S_i's identity, sh is the shuffle key, g is the security code, l and r are two randomly selected positive integers used for defining the sampled subtree, ln is the number of the leaf nodes in the relevant subtree T_r^l.*

Take Fig. 5 as an example, assume that EDI-V samples the subtree T_3^5 in audit request R_i, the random integer $R_i.l$ is the level of n_3^5, i.e., $R_i.l = 5$ and the random integer r is n_r^l's position, i.e., $R_i.r = 3$.

Given a total of k data replicas to be verified, the app vendor employs function **ReqGen** (Algorithm 2) to generate k corresponding audit requests. For ease of presentation, we assume that data replica D_i is deployed on edge server S_i and denote the corresponding audit request as R_i. ReqGen takes six parameters as input, including the security code g, shuffle key sh, a set of identities of hired edge servers $SI = \{SI_1, SI_2, ..., SI_k\}$, a set of identities of cached data replicas $DI = \{DI_1, DI_2, ..., DI_k\}$, the VMHT T generated by function VMHTGen, and a security parameter λ. It outputs a set of audit requests $R = \{R_1, R_2, ..., R_k\}$.

ReqGen first obtains the height of T (line 1), and then randomly selects a positive integer l such that $1 \leq l < H$ based on the security parameter λ (line 2). Next, it executes a breadth-first search to obtain all the nodes on level l in T (lines 3-10). It stores the number of nodes on level l as N (line 11). It then iterates lines 13-17 for k times to generate all the audit requests. Specifically, it randomly selects an integer r such that $r \leq N$ (line 15). This ensures that each audit request has a randomly selected non-leaf node n_r^l on level l. It obtains the number of leaf nodes in the corresponding subtree T_r^l from $n_r^l.ln$ (line 16). Then, it constructs an audit request (line 17). Finally, all the audit requests are returned (line 19).

4.2.3 Respond with Integrity Proof

In Phase 3 (Response), each edge server SI_i generates an integrity proof of the data replica cached on it. Then it sends the proof back to the app vendor. The integrity proof is defined as follows:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.3043755, IEEE Transactions on Parallel and Distributed Systems

Algorithm 2 ReqGen

Input: sh, g, SI, DI, T and λ
Output: a set of audit requests $R = \{R_1, R_2, ..., R_k\}$
1: $H \longleftarrow T.height$
2: select a random integer l according to λ, s.t. $1 \leq l < H$
3: initialize queue $Q \longleftarrow null$
4: Q.add(n_1^1)
5: **while** Q.size > 0 and Q.get(1).$l < l$ **do**
6:     $node \longleftarrow Q$.remove()
7:     **for all** $node$.child **do**
8:         $Q.add(node.child)$
9:     **end for**
10: **end while**
11: $N \longleftarrow Q$.size
12: **for all** $i \in [1, k]$ **do**
13:     get edge ith server's identity SI_i from SI
14:     get data ith replica's identity DI_i from DI
15:     select a random positive integer r s.t. $1 \leq r \leq N$
16:     $ln \longleftarrow Q.get(r).ln$
17:     generate $R_i \longleftarrow < DI_i, SI_i, sh, g, l, r, ln >$
18: **end for**
19: **return** R

Definition 3 (Integrity Proof)**.** *An integrity proof is a tuple that consists of 3 elements, i.e.,* $P_i =< DI_i, SI_i, tag >$*, where* DI_i *is the data replica* D_i*'s identity,* SI_i *is edge server* S_i*'s identity and* tag *is hash value of the sampled VMHT node specified in the corresponding request* R_i*.*

Edge server S_i employs function **ProGen** (Algorithm 3) to produce its integrity proof. ProGen takes two parameters as input: the received audit request R_i and data replica D_i in its cache. First, it obtains the number of blocks of D_i (line 1), and calculates the height of the entire VMHT generated based on all the data blocks in D_i (line 2). Please note that edge server S_i does not need to construct the entire VMHT. Next, it obtains three parameters ln, l, r from the received request (lines 3-5). Then, it shuffles D_i to obtain a new piece of data D'_i according to the results of $Pr(sh, m)$ (line 6). Next, it samples a set of blocks $Sample$ from D'_i according to parameters ln, l, r, H (line 7). Please note that the indexes of these data blocks are $2^{H-l}(r-1)+1, 2^{H-l}(r-1)+2, ..., 2^{H-l}(r-1)+ln$, which can be easily deduced via Theorem 1. It employs function VMHTGen to generate a VMHT T' based on $Sample$ (line 8), then sets tag as the hash value of T''s root node, i.e., $tag = n_1^1.\tau$ (line 9). Finally, it constructs the integrity proof P_i and returns (lines 10-11).

4.2.4 Verify the Integrity

When initializing the audit in Phase 1 (Setup), the app vendor has generated a VMHT T based on its original data D. This tree holds all the *correct* values of those integrity proofs. Given an integrity proof P_i generated based on parameters r and l, $P_i.tag$ should be equal to $n_r^l.\tau$ if the corresponding data replica D_i is correct. In Phase 4 (Verification), given a set of integrity proofs $P = \{P_1, P_2, ..., P_k\}$ received from edge servers, the app vendor employs function **CheckPro** (Algorithm 4) to inspect each data replica and locate the corrupted ones sequentially.

Function CheckPro initializes the audit result of each data replica by setting $\phi_i = 0$ ($i = 1, ..., k$) (lines 1-3),

Algorithm 3 ProGen

Input: an audit request R_i and data replica D_i
Output: integrity proof P_i of data replica D_i
1: $m \longleftarrow$ number of blocks of D_i
2: $H \longleftarrow \lfloor log_2^m \rfloor + 1$
3: $ln \longleftarrow R_i.ln$
4: $l \longleftarrow R_i.l$
5: $r \longleftarrow R_i.r$
6: shuffle D_i according to $R_i.sh$, i.e., $D'_i \xleftarrow{Pr(sh,m)} D_i$
7: $Sample = \{d^{i'}_{2^{H-l}(r-1)+1}, ..., d^{i'}_{2^{H-l}(r-1)+ln}\}$ from D'_i
8: construct a VMHT T' based on $Sample$ via VMHTGen
9: $tag \longleftarrow n_1^1.\tau$, where n_1^1 is the root node of T'
10: $P_i \longleftarrow < SI_i, DI_i, tag >$
11: **return** P_i

Algorithm 4 CheckPro

Input: a VMHT T, a set of proofs of integrity P, a set of audit requests R
Output: the audit result $\Phi = \{\phi_1, ..., \phi_k\}$ for all edge data replicas
1: **for** i in [1,k] **do**
2:     Initialize the result $\phi_i \longleftarrow 0$
3: **end for**
4: obtain l from R
5: initialize queue $Q \longleftarrow null$
6: Q.add($n_1^1 \in T$)
7: **while** Q.size > 0 and Q.get(1).$l < l$ **do**
8:     $node \longleftarrow Q$.remove()
9:     **for all** $node$.child **do**
10:         $Q.add(node.child)$
11:     **end for**
12: **end while**
13: **for** i in [1,k] **do**
14:     $r \longleftarrow R_i.r$
15:     $tag' \longleftarrow P_i.tag$
16:     $tag \longleftarrow Q$.get(r).τ
17:     $\phi_i \longleftarrow tag == tag'$? 1 : 0
18: **end for**
19: **return** Φ

where $\phi_i = 1$ means that data replica D_i is correct, and 0 otherwise. Then, it obtains the level l from the audit request R (line 4). In Algorithm 2 all the sampled nodes are on the same level. Thus, the values of l in all the audit requests are the same. Similar to function ReqGen, function CheckPro also employs the breadth-first search to find all the non-leaf nodes on level l in T (lines 4-11). Next, it iterates lines 13-18 to compare each integrity proof P_i received from edge server S_i against the corresponding node in Q. To verify integrity proof P_i, it obtains the position r from request R_i (lines 14), the hash value $tag' = P_i.tag$ from P_i (line 15), and the $correct$ hash value tag from Q according to r (line 16). By comparing tag' and tag, the integrity of data replica D_i on edge server S_i can be verified (lines 17). After all the iterations, this algorithm returns (line 19).

5 Performance Analysis

This section theoretically analyzes the performance of EDI-V, including its correctness, efficiency, and security guarantee.

TABLE 2
Efficiency Analysis

	Phase 1	Phase 2	Phase 3	Phase 4
Time Complexity	O(m)	O(m+k)	O(m)	O(m+k)
Space Occupation	384(2m-1)	320k	0	32k
Communication Overheads	/	320k	320k	/

5.1 Correctness

To prove the correctness of EDI-V, we first prove that EDI-V can precisely verify the integrity of the sampled data blocks, and then prove that it can detect data corruption with a probability guarantee.

Lemma 1. *EDI-V can precisely verify the integrity of the sampled blocks of a data replica if the hash function $Hash()$ is collision-resistant.*

The proof of Lemma 1 can be found in Appendix A.1.

Theorem 2. *Given a data replica D' that consists of m blocks, $D' = \{d_1, d_2, ..., d_m\}$, EDI-V can successfully detect the corruption with the probability of $1 - (\frac{m-nc}{m})^{ns}$, where ns is the number of sampled data blocks, and nc is the number of corrupted blocks of D'.*

The proof of Theorem 2 can be found in Appendix A.1. **It demonstrates that EDI-V achieves the first objective introduced in Section 2.2, i.e., the correctness objective.**

5.2 Efficiency

The time complexity, space occupation and communication overheads of EDI-V are summarized in Table 2, where k is the number of edge servers, m is the number of blocks of each data replica, $Hash()$ is SHA-256, and security code g is 128 bits, a normal integer is 32 bits. A detailed analysis can be found in Appendix A.2. **This theoretically proves that EDI-V achieves the second objective introduced in Section 2.2, i.e., the lightweight objective.**

5.3 Security Guarantee

Now we prove that EDI-V can tackle the cheating attacks discussed in Section 2.1, i.e., the replay attack and the forge attack.

Theorem 3. *With EDI-V, an edge server cannot pass the audit by either a replay attack or a forge attack if the hash function $Hash()$ is collision-resistant and the random functions employed by EDI-V are secure.*

The proof of Theorem 3 can be found in Appendix A.3. **It ensures that EDI-V achieves the third objective introduced in Section 2.2, i.e., the security objective.**

6 Experimental Evaluation

In this section, we experimentally evaluate EDI-V against the traditional PDP approaches and an EDI approach that is implemented based on the traditional Merkle hash tree.

6.1 Baseline Approaches

To find the baseline approaches for comparison, we thoroughly and extensively investigated numerous variants of the PDP approaches used in the cloud computing environment. However, only a few are designed for auditing multiple data replicas [23], [19], [27], [28], [29]. Two representative schemes are MR-PDP [29] and MuR-DPA [23]. Given an original data, MR-PDP [29] employs classical symmetric cipher to mask data replicas into different ones and then employs traditional PDP schemes to audit them. Users have to obtain the corresponding key to unmask the data before using it. Thus, MR-PDP incurs extremely high computational overheads to both data owners and users. MuR-DPA [23] employs MR-MHT (multi-replica Merkle hash tree) to audit multiple data replicas. However, MuR-DPA employs only one MR-MHT to reserve all the blocks of all the data replicas, which makes it unsuitable for edge computing scenarios where the number of data replicas to be audited is huge. Furthermore, all of them audit the data replicas in batch. As a result, they can not locate corrupted data during the audit. Hence, neither MR-PDP nor MuR-DPA can be directly employed as the baseline approaches in our evaluation.

Almost all the PDP-based approaches employ the *Homomorphic Verifiable Tag* (HVT) [25] to verify data integrity. Thus, we derive a PDP-based approach, denoted as GPDP, as one of the baselines in our evaluation. GPDP is implemented by extending the MR-PDP approach [29]. To perform a fair comparison against EDI-V, we remove the data replica masking and unmasking mechanisms from GPDP to accelerate it. We also reverse the entities in GPDP so that it can be applied to EDI scenarios. In this way, GPDP's audit process consists of four phases. In Phase 1 (Setup), the app vendor generates necessary parameters, including finding the large primes and creating the cyclic group, etc. In Phase 2 (Request), the app vendor creates audit requests and sends to edge servers. In Phase 3 (Response), each edge server generates an HVT as the integrity proof and sends it back to the app vendor. In Phase 4 (Verification), the app vendor verifies all the integrity proofs in a single batch.

As EDI-V employs VMHT to facilitate the sampling, to study the impact of VMHT, we also implement EDI-V based on the traditional MHT, denoted as EDI-T, as the second baseline approach for comparison.

6.2 Performance Metrics

Efficiency. The efficiency is measured by the total computation time taken to audit all the edge data replicas: the lower the better. Both EDI-V and EDI-T use the same hash function to generate the MHT, the former is based on VMHT whilst the latter is based on traditional MHT. The difference between their time consumption is negligible. Here, we focus on the comparison in the efficiency between EDI-V and GPDP in the experiments.

Effectiveness. The effectiveness is measured by the ratio of the amount of located corrupted edge data replicas to the total amount of corrupted edge data replicas. GPDP's audit result only indicates whether all the data replicas are correct or there are corruption(s), but cannot locate the corrupted data replicas. Thus, it is infeasible to compare the data corruption localization performance between EDI-V and GPDP. Both EDI-V and GPDP are probabilistic approaches. Their performance in data corruption detection is solely dependent on the number of sampled data blocks. Thus, their corruption detection abilities are theoretically the same when sampling the same number of data blocks. Here, we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2020.3043755, IEEE Transactions on Parallel and Distributed Systems

TABLE 3
Parameter Settings

Parameter	Varied value	Fixed value
Replica Scale (k)	$2^6, 2^7, ..., 2^{10}$	2^8
Replica Size (rs)	$2^{28}, 2^{29}, ..., 2^{31}, 2^{32}$	2^{30}
Block Size (bs)	$2^{17}, 2^{18}, ..., 2^{21}$	2^{19}
Sampling Scale (ss)	$7, 8, 9, 10, 11$	9

focus on the comparison in the effectiveness between EDI-V and EDI-T. With the same sampling parameters, higher ratio indicates higher effectiveness.

6.3 Experiment Setup

In the experiments, we randomly generate both the original data and the corresponding data replicas to be audited. Each data is divided into different numbers of blocks according to the block size and the file size. To simulate data corruption, we randomly alter a portion of the data blocks in the replica according to the corruption rate cr. For example, when the corruption rate is set to 3%, given a data replica with 1,000 blocks, we randomly alter 30 data blocks to corrupt the data replica. To evaluate EDI-V comprehensively, we vary four experiment parameters listed as follows to simulate a variety of EDI scenarios:

1) **Replica Scale (k)**, measured by the number of data replicas cached on edge servers. It varies from 64 to 1,024. In this way, we can evaluate EDI-V's scalability with k.
2) **Replica Size (rs)**, measured by the size of an individual data replica. The unit of rs is byte, i.e., $rs = 2^{30}$ means the replica size is 1 GB. By varying rs from 2^{28} to 2^{32}, we can validate EDI-V's performance in handling data replicas of different sizes.
3) **Block Size (bs)**, measured by the size of each block of a data replica. The unit of bs is also byte. Given a fixed replica size, a bigger block size will result in fewer data blocks. In this way, the corresponding VMHT will have different numbers of nodes to be generated. By varying bs from 2^{17} to 2^{21}, we can evaluate EDI-V's performance in handling different numbers of data blocks.
4) **Sampling Scale (ss)**, measured by the height of the sampled subtree for each data replica. Please note that in function VMHTGen (Algorithm 1), H is the height of the entire VMHT T, l is the level of the selected node, thus there is $ss = H - l + 1$. By varying ss, we can evaluate how the sampling scale impacts the performance of EDI-V.

Table 3 summarizes the parameter settings used in the experiments. We vary each parameter while fixing the other parameters to simulate various EDI scenarios. Each time we vary the value of one parameter, the experiment is repeated 100 times and the average results are reported.

Please note that the above parameters are not necessarily to be used simultaneously in different scenarios. For example, in terms of efficiency, in Phase 1 (Setup), EDI-V needs to generate the original VMHT. Its computational overheads is affected by both the replica size rs and the block size bs. In Phase 2 (Request), both EDI-V and GPDP generate requests and send them to each edge server, their computational overheads are affected by the replica scale k. In Phase 3 (Response), each edge server generates its integrity proof individually. The corresponding computational overheads are affected by the block size bs and the sampling scale ss. In Phase 4 (Verification), the efficiency of EDI-V and GPDP are impacted by the replica scale k, the block size bs, and the sampling scale ss. Because GPDP can not locate corrupted data replicas, the corruption rate cr is not used when comparing EDI-V and GPDP.

All the approaches are implemented in Java with Java encryption library Chilkat [2]. For GPDP, we set the large prime as 512 bits and the certainty of primes generation as 2^{-64}. For EDI-V and EDI-T, we employ SHA-256 as the hash function. All the experiments are conducted on a machine equipped with Intel Core i5-7400T processor and 8GB RAM, running Windows 10 Professional 64bit.

6.4 Experimental Results

This section first compares the overall efficiency of EDI-V and GPDP, then reports the efficiency of each phase. In Phase 2 (Request), both EDI-V and GPDP only generate a few parameters for those requests and then send them to each edge server. It takes almost no time. Thus, we do not evaluate their computational overheads in this phase. At last, it compares the effectiveness of EDI-V and EDI-T.

6.4.1 Efficiency Comparison of the Whole Process

Table 4 compares the overall efficiency of EDI-V and GPDP during the entire audit process. **On average, EDI-V is three orders of magnitude more efficient than GPDP across different parameter settings**.

We can see that the impact of replica scale k on EDI-V is negligible, indicated by its stable time consumption when k varies from 64 to 1,024. The reason is that the increase in k only impacts Phase 4 (Verification), where EDI-V takes almost no time to verify the received k proofs. This impact is negligible compared to the overall time consumption during the entire audit. In contrast, GPDP's time consumption increases rapidly as k increases. On average across the five cases, EDI-V is 6,098.40 times more efficient than GPDP. This observation indicates that EDI-V is suitable for auditing massive edge data replicas.

GPDP samples only a fixed number of data blocks for the audit. Thus, its time consumption is not significantly impacted by the replica size rs. Similarly, on the edge server side, EDI-V samples a fixed number of data blocks to construct the VMHT. However, on the app vendor side, it constructs the entire VMHT based on all the data blocks of the original data D. This increases the time consumed by EDI-V. However, EDI-V is still 5,375.95 times more efficient than GPDP on average across five cases.

When the block size bs varies from 128 KB to 2 MB, both EDI-V and GPDP need more time to complete the audit. However, the time taken by EDI-V increases much slower than that of GPDP. For example, EDI-V's time consumption increases from 8.24 s to 12.01 s by 1.46 times. In contrast, GPDP's time consumption increases from 2,336.13 s to 537,237.18 s by 299.97 times. On average across five

2. http://www.chilkatsoft.com/java-encryption.asp

TABLE 4
Time Consumption of Entire Process (s)

Parameters	Replica Scale (k)					Replica Size (rs)				
	2^6	2^7	2^8	2^9	2^{10}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}
EDI-V	9.01	9.02	9.02	9.02	9.03	2.94	4.92	9.02	16.70	32.53
GPDP	8,971.02	17,791.05	36,091.03	70,827.82	141,510.91	36,091.03	36,091.03	36,091.03	36,091.03	36,091.03
Parameters	**Block Size (bs)**					**Sampling Scale (ss)**				
	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	7	8	9	10	11
EDI-V	8.24	8.52	9.02	10.10	12.01	8.29	8.54	9.02	9.99	11.96
GPDP	2,366.13	9,098.82	36,091.03	139,700.77	537,237.18	9,617.62	18,683.40	36,091.03	71,513.88	140,664.82

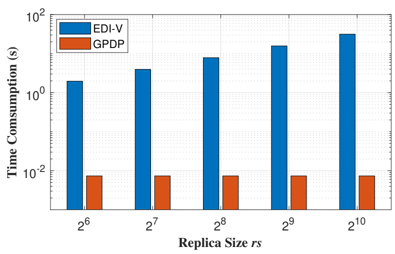


Fig. 7. Efficiency vs. rs in Phase 1

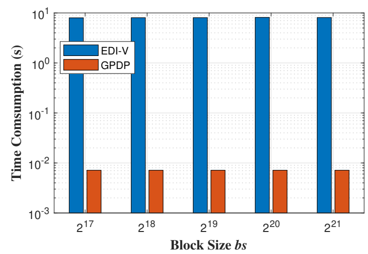


Fig. 8. Efficiency vs. bs in Phase 1

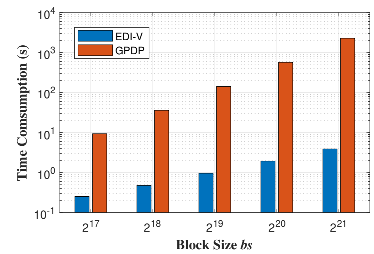


Fig. 9. Efficiency vs. bs in Phase 3

cases, EDI-V is 12,781.79 times more efficient than GPDP. The reason is that in Phase 3 (Response), GPDP's VHT generation does not scale with the block size as well as the Hash operations performed by EDI-V to generate VMHTs. Besides, in Phase 4 (Verification) on the app vendor side, GPDP needs to generate an HVT based on the data blocks sampled by all the edge servers in Phase 3 (Response). This further increases its time consumption. Similar phenomena are observed when the sampling scale ss increases from 7 to 11. When more data blocks are sampled, EDI-V takes more time to create the VMHT. Similarly, GPDP needs more time to create HVTs. On average across five cases, EDI-V is 5,252.47 times faster than GPDP.

6.4.2 Efficiency Comparison in Phase 1 (Setup)

Fig. 7 compares the efficiency of EDI-V and GPDP in Phase 1 (Setup) where the replica size rs varies from 256 MB to 4 GB and the block size bs is 512 KB. When rs increases, the number of blocks in the original data D increases. Thus, EDI-V needs to spend more time to construct a bigger VMHT. On average, its time consumption is 12.21 s across five cases, and the maximum time consumption is 31.55 s on average when the replica size is 4 GB. This is acceptable for two reasons: 1) this process needs to be performed only once for each audit; 2) the app vendor usually has access to (virtual) machines far more powerful than our desktop, which can significantly reduce the time consumption in this phase.

Fig. 8 compares the efficiency where the block size bs varies from 128 KB to 2 MB and the replica size rs is 1 GB. The time consumption of EDI-V is 8.06 s on average across all the cases. Interestingly, the time consumption of EDI-V is not affected significantly by bs. The reason is that, no matter what the block size is, SHA-256 iterates on each 512-bit chunk to obtain the final hash result. Thus, when rs is fixed, the number of the chunks in the corresponding data is fixed too. The overall time consumption for generating all the leaf nodes of the corresponding VMHT is also fixed. The generation of all the non-leaf nodes is very fast, i.e., 14.89 ms on average when the block size is 256 KB and 1.03 ms when the block size is 2 MB.

As mentioned before, GPDP only initializes its security parameters in this phase. It is irrelevant to either rs or bs. Thus, its time consumption in this phase is stable, i.e., 7.25 ms on average.

6.4.3 Efficiency Comparison in Phase 3 (Response)

Next, we compare the efficiency of EDI-V and GPDP in Phase 3 (Response). Here, we analyze the average time consumption on a single edge server. Fig. 9 shows the experimental results where the block size bs varies from 128 KB to 2 MB, and the replica size rs is 1 GB. The sampling scale ss is 9, i.e., at least a total of $2^{9-1} = 256$ blocks are sampled by EDI-V and exactly 256 blocks are sampled by GPDP. We can find that bs can affect the performance of both EDI-V and GPDP, indicated by the increases in their time consumption when bs increases. However, **EDI-V is two orders of magnitude more efficient than GPDP in this phase**. EDI-V consumes an average of 1.52 s across all cases while GPDP consumes 613.85 s, i.e., 403.85 times as EDI-V. The reason is GPDP spends too much time on processing large integers and performing exponential operations when generating an HVT.

Similar phenomena are observed in Fig. 10 where the sampling scale ss increases from 7 to 11, i.e., the number of sampled blocks increases from 64 to 1,024. When more data blocks are sampled, function ProGen (Algorithm 3) takes more iterations to generate the data integrity proof. This increases EDI-V's time consumption. On average, GPDP consumes 220.73 s across all five cases while EDI-V consumes only 1.51 s, 145.19 times faster than GPDP.

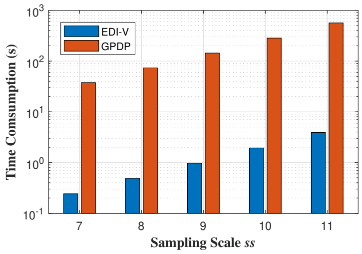


Fig. 10. Efficiency vs. ss in Phase 3

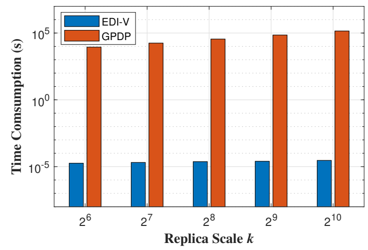


Fig. 11. Efficiency vs. k in Phase 4

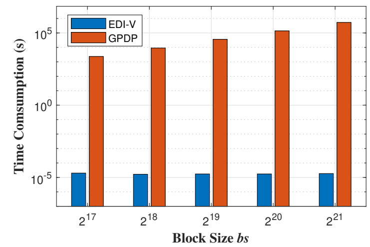


Fig. 12. Efficiency vs. bs in Phase 4

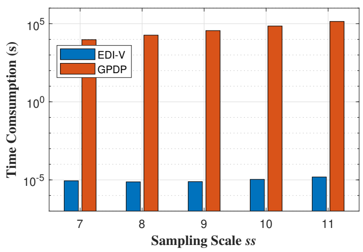


Fig. 13. Efficiency vs. ss in Phase 4

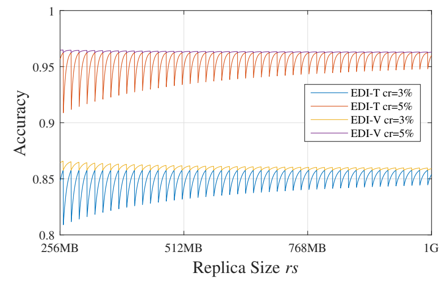


Fig. 14. Comparison of Effectiveness ($ss = 7$)

The above observations confirm that: 1) PDP-based approaches tend to incur high computational overheads to edge servers, which conflicts with the EDI objective 2 discussed in Section 2.2; 2) EDI-V is lightweight on edge servers, making it more suitable than PDP-based approaches in EDI scenarios.

6.4.4 Efficiency Comparison in Phase 4 (Verification)

In this phase, app vendor audits all the received data integrity proofs. Fig. 11 illustrates the time consumption when the replica scale k varies from 64 to 1,024. As k increases, the time consumptions of both EDI-V and GPDP increase, but in very different manners. For example, GPDP's time consumption increases from 8,827 s to 141,367 s by more than 16 times. In contrast, EDI-V's time consumption increases from 0.018 ms to 0.029 ms only by 1.6 times. The reasons are twofold. First, GPDP has to aggregate a total number of k received HVTs by multiplying them one by one before comparison. Second, it converts each data block to a large integer, adds up all the integers, and obtains the overall HVT via an exponential computation. Both operations are computationally expensive. In Phase 1 (Setup), EDI-V has generated the VMHT and all the hash values. Thus, in Phase 4 (Verification), it produces the verification results by simply comparing the hash values in the received data integrity proofs against the corresponding hash values in its own VMHT, which consumes only 0.023 ms on average across all cases in the experiments.

Fig. 12 and Fig. 13 show similar phenomena to Fig. 11. In Fig. 12, the block size bs varies from 128 KB to 2 MB while both rs and ss are fixed. To produce the verification results, GPDP spends 144,330 s on average across all cases while EDI-V takes only 0.018 ms on average. In Fig. 13 where the sampling scale ss increases from 7 to 11, GPDP spends 55,172 s to produce the final result while EDI-V spends only 0.01 ms. Thus, **EDI-V is six orders of magnitude more efficient than GPDP in this phase.**

Through the comparison, we can find that: 1) traditional PDP-based approaches are too time-consuming to be employed in EDI scenarios; and 2) EDI-V is lightweight on the app vendor. Its lightweight characteristic accommodates app vendors' need, i.e., to ensure high edge data integrity by verifying its massive edge data replicas cached on a large number of edge servers efficiently.

6.4.5 Effectiveness Comparison of EDI-V and EDI-T

As discussed in Section 3.1, VMHT overcomes the limitation of the traditional MHT and achieves a high audit accuracy. We validate this by comparing the accuracy of EDI-V and EDI-T under different experiment settings. Fig. 14 shows the results of EDI-V and EDI-T when the corruption rate cr is 3% and 5%, respectively, and the data block size bs is 256 KB. The replica size rs increases from 256 KB to 1 GB. The sampling scale ss is 7, i.e., each sampling has at least 64 leaf-nodes in the corresponding MHT. Please note that the sampling does not necessarily mean that 64 corresponding data blocks are sampled. This is the core difference between EDI-V and EDI-T. As discussed in Section 3.1, EDI-V can ensure that at least 64 corresponding data blocks are sampled in every case. However, EDI-T can only ensure that at most 64 corresponding data blocks are sampled in the best-case scenarios. As the corruption detection ability is highly related to the number of sampled data blocks, sampling fewer data blocks can damage the detection accuracy.

From Fig. 14 we can easily find that, in both scenarios where 3% and 5% of the data blocks are corrupted, EDI-V's performance is much more stable than EDI-T. When the number of data blocks is slightly larger than the power of 2, EDI-T's audit accuracy drops significantly. The reason is that the rightmost subtree on each level only covers a few data blocks, making it difficult for EDI-T to detect data corruption. For example, when there are 1,025 blocks in the data replica, the rightmost subtree of the corresponding traditional MHT, i.e., the right subtree of its root node, covers only 1 data block. During the sampling, if EDI-T chooses a subtree from the right half part, its accuracy would be very low. This confirms the conclusion drawn in Section 3.1 that the traditional MHT is not suitable for solving the EDI problem.

An interesting finding is that, with different numbers of data blocks, EDI-V's audit accuracy also fluctuates slightly. The reason is that, in contrast to EDI-T, when the number of data blocks is not the power of 2, the rightmost node on each level may have three child nodes. Thus, given the same sampling parameters, more data blocks will be sampled, which leads to higher audit accuracy. In fact, we can easily find in Fig. 14 that the lower bound of EDI-V's performance is the upper bound of EDI-T's performance.

7 Related work

As a prospective distributed computing paradigm, edge computing has gained wide attention from researchers and raised lots of research problems, such as edge user allocation [30], [31], [32], [33], computation offloading [4], edge data caching/distribution [1], [34], [35], and service placement [7], [36], [37], etc.

In the EC environment, app vendors, such as Facebook and YouTube, can deploy their applications/services on edge servers to serve their users with low service latency. Wang et al. formulated the multiple service placement problem in the EC environment as a combinatorial optimization problem, and proposed an approach named ITEM to solve it while satisfying the economic and service quality constraints [36]. Pasteris et al. studied the placement of multiple services in a heterogeneous EC environment with the aim to maximize the total reward [37]. Deng et al. explored the microservice-based application deployment problem and proposed an approach to minimize the overall deployment cost while fulfilling the resources and performance constraints [38]. Taking a step further, Zhao et al. considered the service composite property and proposed an approach named GASS to deploy microservice-based applications with sequential combinatorial structures [39]. Li et al. studied service reliability in the EC environment. They proposed an approach named READ to deploy multiple instances of an edge application while fulfilling the budget constraint and maximizing the overall application robustness [7]. Deng et al. investigated service level agreement compliance in the EC environment. They proposed a reinforcement learning-based approach to maximize service trustworthiness gain by dynamically allocating appropriate resources according to the system states [40].

As edge servers become many app users' entry point to various online applications and services, caching popular data on edge servers can significantly reduce their data retrieval latency and has attracted many researchers' attention in recent years. To name a few, Halalai et al. proposed an approach named Agar that selects data blocks to be cached on edge servers to minimize users' access latency [41]. Similarly, Xia et al. proposed a data caching approach to minimize users' average data retrieve latency with a given data caching budget [1]. Cao et al. proposed an auction-based approach that allocates edge servers' storage resources to maximize app vendors' revenue [42]. Liu et al. proposed an approach to maximize app vendor's data caching revenue at the edge, taking data caching cost, data transmission cost, and users' access latency into account [43]. Xia et al. investigated the edge data caching problem in dynamic scenarios where new data need to be cached and outdated data need to be decached on the fly. They proposed a Lyapunov optimization-based approach to solve this problem, with the aim to minimize the overall cost, including data caching cost, data migration cost, and quality-of-service (QoS) penalty [34]. The popularity of edge data caching raises the edge data integrity (EDI) problem because data cached on edge servers are subject to corruption caused by various events in the highly distributed and dynamic EC environment.

In the cloud computing environment, tremendous data are collected and stored in the cloud, which raises the need to remotely verify the data integrity in the cloud [44]. This problem has been intensively studied in the past decade. Provable Data Possession (PDP) [25] and Proof of Retrievability (POR) [45] are the most popular approaches. Various variants of PDP and POR have been proposed to offer new features, such as data dynamics [14], [46], public verifiability [14], [47], and privacy preservation [14], [47], [48]. A few approaches were proposed to verify the integrity of multiple data replicas [19], [23], [27], [28]. Unfortunately, none of these approaches can be employed to tackle the EDI problem as discussed in Section 1.

Compared with the cloud computing environment facilitated by large-scale and centralized data centers, the EC environment is much more distributed, dynamic, and volatile. Many security issues have been identified in the edge computing environment [9], [10], [49], [50]. Although the data integrity problem has been extensively studied in the cloud computing environment, it is a new and open problem in the EC environment. Very recently, Tong et al. employed the conventional PDP scheme to verify the integrity of users' data stored on edge servers [11]. However, they focused on the protection of users' privacy and did not tackle the EDI problem from the app vendor's perspective - how to verify the integrity of massive data replicas efficiently. Its theoretical efficiency is the same as the GPDP approach implemented in our experiments. Due to its low efficiency as demonstrated in Section 6, it cannot deal with massive edge data.

To ensure the effectiveness of EDI-V, we also proposed a new type of Merkle hash tree, i.e., VMHT. There are a few variants of PDP that employ hash algorithms or MHT or its variants to verify the integrity of remote data in the cloud [17], [19], [22], [24]. Zhu et al. proposed a cooperative PDP scheme to verify the integrity of data stored over multi-cloud. They employed a hierarchical hash structure with three layers to manage all the replicas of one file [19]. Tian et al. employed the dynamic hash table (DHT) to audit dynamic data stored in the cloud. They employed the DHT to manage all the files stored in the cloud. Each file has a corresponding linked list that stores all blocks [17]. Liu et al. proposed a cloud data audit scheme to support dynamic and fine-grained data updates. In their approach, a rank-based Merkle hash tree was created to manage each file. In the same tree, different leaf nodes contain different numbers of blocks [22]. He et al. proposed a dynamic group-oriented PDP approach, which employs traditional MHT to generate data integrity proofs. Each leaf node in the tree is generated based on the homomorphic tags of multiple blocks [24].

However, none of the above data structures is suitable for auditing massive edge data replicas. To address this issue and enable fast audit of massive edge data replicas, we proposed a novel VMHT. It allows EDI-V to efficiently and effectively audit the integrity of massive data replicas cached on a large number of edge servers.

8 Conclusion and Future Work

In this paper, we studied the Edge Data Integrity (EDI) problem from the app vendor's perspective. We analyzed the threats to data integrity in the edge computing environment, discussed the EDI objectives and the audit process. To facilitate the efficient and effective audit of massive edge data replicas, we proposed a novel approach named EDI-V. It provides a probabilistic data integrity guarantee. We also proposed a new variable Merkle hash tree (VMHT) to generate the integrity proof of each replica. VMHT can also help EDI-V uniformly sample data blocks during the audit. By both theoretical analysis and experimental evaluation, we demonstrated that EDI-V can audit the integrity of massive edge data replicas and locate corrupted ones with high efficiency and effectiveness.

In our future work, we will enhance EDI-V with new capacities, such as privacy preservation, data dynamics, etc. We will also investigate how to efficiently repair corrupted edge data replicas.

Acknowledgments

This research is funded by Australian Research Council Discovery Projects (DP180100212 and DP200102491). Qiang He is the corresponding author of this paper.

References

[1] X. Xia, F. Chen, Q. He, G. Cui, P. Lai, M. Abdelrazek, J. Grundy, and H. Jin, "Graph-based optimal data caching in edge computing," in *International Conference on Service-Oriented Computing*. Springer, 2019, pp. 477–493.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *workshop on Mobile cloud computing*. Helsinki, Finland: ACM, 2012, pp. 13–16.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[4] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[5] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2333–2345, 2018.

[6] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[7] B. Li, Q. He, G. Cui, X. Xia, F. Chen, H. Jin, and Y. Yang, "READ: Robustness-oriented edge application deployment in edge computing environment," *IEEE Transactions on Services Computing*, 2020.

[8] G. Cui, Q. He, F. Chen, H. Jin, and Y. Yang, "Trading off between user coverage and network robustness for edge server placement," *IEEE Transactions on Cloud Computing*, 2020.

[9] R. Roman, J. Lopez, and M. Mambo, "Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges," *Future Generation Computer Systems*, vol. 78, pp. 680–698, 2018.

[10] C. Esposito, A. Castiglione, F. Pop, and K.-K. R. Choo, "Challenges of connecting edge and cloud computing: A security and forensic rerspective," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 13–17, 2017.

[11] W. Tong, B. Jiang, F. Xu, Q. Li, and S. Zhong, "Privacy-preserving data integrity verification in mobile edge computing," in *2019 IEEE 39th International Conference on Distributed Computing Systems*. IEEE, 2019, pp. 1007–1018.

[12] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, no. 5, pp. 2795–2808, 2016.

[13] F. Guo, H. Zhang, H. Ji, X. Li, and V. C. Leung, "An efficient computation offloading management scheme in the densely deployed small cell networks with mobile edge computing," *IEEE/ACM Transactions on Networking*, 2018.

[14] J. Li, L. Zhang, J. K. Liu, H. Qian, and Z. Dong, "Privacy-preserving public auditing protocol for low-performance end devices in cloud," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 11, pp. 2572–2583, 2016.

[15] M. Chen, Y. Hao, K. Lin, Z. Yuan, and L. Hu, "Label-less learning for traffic control in an edge network," *IEEE Network*, vol. 32, no. 6, pp. 8–14, 2018.

[16] H. Wang, "Identity-based distributed provable data possession in multicloud storage," *IEEE Transactions on Services Computing*, vol. 8, no. 2, pp. 328–340, 2014.

[17] H. Tian, Y. Chen, C.-C. Chang, H. Jiang, Y. Huang, Y. Chen, and J. Liu, "Dynamic-hash-table based public auditing for secure cloud storage," *IEEE Transactions on Services Computing*, vol. 10, no. 5, pp. 701–714, 2015.

[18] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," in *ACM Conference on Computer and Communications Security*. ACM, 2009, pp. 213–222.

[19] 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.

[20] K. Yang and X. Jia, "An efficient and secure dynamic auditing protocol for data storage in cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 9, pp. 1717–1726, 2012.

[21] F. Tschorsch and B. Scheuermann, "Bitcoin and beyond: A technical survey on decentralized digital currencies," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2084–2123, 2016.

[22] C. Liu, J. Chen, L. T. Yang, X. Zhang, C. Yang, R. Ranjan, and R. Kotagiri, "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, 2013.

[23] C. Liu, R. Ranjan, C. Yang, X. Zhang, L. Wang, and J. Chen, "Murdpa: 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, 2014.

[24] K. He, J. Chen, Q. Yuan, S. Ji, D. He, and R. Du, "Dynamic group-oriented provable data possession in the cloud," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[25] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *ACM Conference on Computer and Communications Security*. Alexandria, Virginia, USA: ACM, 2007, pp. 598–609.

[26] L. Li, Y. Yang, and Z. Wu, "Fmr-pdp: Flexible multiple-replica provable data possession in cloud storage," in *2017 IEEE Symposium on Computers and Communications*, 2017, pp. 1115–1121.

[27] J. Li, H. Yan, and Y. Zhang, "Efficient identity-based provable multi-copy data possession in multi-cloud storage," *IEEE Transactions on Cloud Computing*, 2019.

[28] A. F. Barsoum and M. A. Hasan, "Provable multicopy dynamic data possession in cloud computing systems," *IEEE Transactions on Information Forensics and Security*, vol. 10, pp. 485–497, 2015.

[29] R. Curtmola, O. Khan, R. Burns, and G. Ateniese, "Mr-pdp: Multiple-replica provable data possession," in *IEEE International Conference on Distributed Computing Systems*. Beijing, China: IEEE, 2008, pp. 411–420.

[30] P. Lai, Q. He, M. Abdelrazek, F. Chen, J. Hosking, J. Grundy, and Y. Yang, "Optimal edge user allocation in edge computing with variable sized vector bin packing," in *International Conference on Service-Oriented Computing*. Springer, 2018, pp. 230–245.

[31] P. Lai, Q. He, G. Cui, X. Xia, M. Abdelrazek, F. Chen, J. Hosking, J. Grundy, and Y. Yang, "Edge user allocation with dynamic quality of service," in *International Conference on Service-Oriented Computing*. Springer, 2019, pp. 86–101.

[32] Q. He, G. Cui, X. Zhang, F. Chen, S. Deng, H. Jin, Y. Li, and Y. Yang, "A game-theoretical approach for user allocation in edge computing environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 515–529, 2019.

[33] G. Cui, Q. He, X. Xia, P. Lai, F. Chen, T. Gu, and Y. Yang, "Interference-aware SaaS user allocation game for edge computing," *IEEE Transactions on Cloud Computing*, 2020.

[34] X. Xia, F. Chen, Q. He, J. Grundy, M. Abdelrazek, and H. Jin, "Online collaborative data caching in edge computing," *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[35] ——, "Cost-effective app data distribution in edge computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 31–44, 2021.

[36] L. Wang, L. Jiao, T. He, J. Li, and M. Mühlhäuser, "Service entity placement for social virtual reality applications in edge computing," in *IEEE International Conference on Computer Communications*. IEEE, 2018, pp. 468–476.

[37] S. Pasteris, S. Wang, M. Herbster, and T. He, "Service placement with provable guarantees in heterogeneous edge computing systems," in *IEEE International Conference on Computer Communications*. IEEE, 2019, pp. 514–522.

[38] S. Deng, Z. Xiang, J. Taheri, K. A. Mohammad, J. Yin, A. Zomaya, and S. Dustdar, "Optimal application deployment in resource constrained distributed edges," *IEEE Transactions on Mobile Computing*, 2020.

[39] H. Zhao, S. Deng, Z. Liu, J. Yin, and S. Dustdar, "Distributed redundancy scheduling for microservice-based applications at the edge," *IEEE Transactions on Services Computing*, 2020.

[40] S. Deng, Z. Xiang, P. Zhao, J. Taheri, H. Gao, J. Yin, and A. Y. Zomaya, "Dynamical resource allocation in edge for trustable internet-of-things systems: A reinforcement learning method," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 9, pp. 6103–6113, 2020.

[41] R. Halalai, P. Felber, A. M. Kermarrec, and F. Taïani, "Agar: A caching system for erasure-coded data," in *IEEE International Conference on Distributed Computing Systems*, 2017, pp. 23–33.

[42] X. Cao, J. Zhang, and H. V. Poor, "An optimal auction mechanism for mobile edge caching," in *IEEE International Conference on Distributed Computing Systems*, 2018, pp. 388–399.

[43] Y. Liu, Q. He, D. Zheng, X. Xia, F. Chen, and B. Zhang, "Data caching optimization in the edge computing environment," *IEEE Transactions on Services Computing*, 2020.

[44] Y. Deswarte, J. J. Quisquater, and A. Saïdane, "Remote integrity checking," in *Working Conference on Integrity and Internal Control in Information Systems*. Springer, 2003, pp. 1–11.

[45] A. Juels and B. S. Kaliski Jr, "Pors: Proofs of retrievability for large files," in *ACM Conference on Computer and Communications Security*. ACM, 2007, pp. 584–597.

[46] E. Shi, E. Stefanov, and C. Papamanthou, "Practical dynamic proofs of retrievability," in *ACM SIGSAC Conference on Computer & Communications Security*. ACM, 2013, pp. 325–336.

[47] Z. Hao, S. Zhong, and N. Yu, "A privacy-preserving remote data integrity checking protocol with data dynamics and public verifiability," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1432–1437, 2011.

[48] Y. Yu, M. H. Au, G. Ateniese, X. Huang, W. Susilo, Y. Dai, and G. Min, "Identity-based remote data integrity checking with perfect data privacy preserving for cloud storage," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 4, pp. 767–778, 2016.

[49] S. Yi, Z. Qin, and Q. Li, "Security and privacy issues of fog computing: A survey," in *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 2015, pp. 685–695.

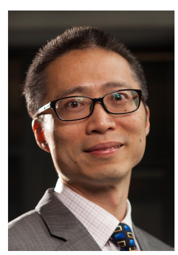
[50] I. Stojmenovic, S. Wen, X. Huang, and H. Luan, "An overview of fog computing and its security issues," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 10, pp. 2991–3005, 2016.

Bo Li received the BS and MS degree from the School of Information Science and Technology from Shandong Normal University in 2003 and 2010, respectively. He worked as an academic visitor at Shandong University from 2014 to 2015 and at Swinburne University of Technology in 2018. He is currently a Ph.D. candidature in Swinburne. His research interests include software engineering, edge computing, and network security issues.

Qiang He received his first PhD degree from Swinburne University of Technology, Australia, in 2009 and his second PhD degree in computer science and engineering from Huazhong University of Science and Technology, China, in 2010. He is a senior lecturer at Swinburne. His research interests include service computing, software engineering, cloud computing and edge computing. More details about his research can be found at https://sites.google.com/site/heqiang/.

Feifei Chen received her PhD degree from Swinburne University of Technology, Australia in 2015. She is a lecturer at Deakin University. Her research interests include software engineering, cloud computing and green computing.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. Jin received his PhD in computer engineering from HUST in 1994. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.



Yang Xiang received his PhD in Computer Science from Deakin University, Australia. He is currently a full professor and the Dean of Digital Research & Innovation Capability Platform, Swinburne University of Technology, Australia. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. He is also leading the Blockchain initiatives at Swinburne. In the past 20 years, he has published more than 300 research papers in many international journals and conferences. He is the Editor-in-Chief of the SpringerBriefs on Cyber Security Systems and Networks. He serves as the Associate Editor of IEEE Transactions on Dependable and Secure Computing, IEEE Internet of Things Journal, and ACM Computing Surveys. He served as the Associate Editor of IEEE Transactions on Computers and IEEE Transactions on Parallel and Distributed Systems. He is the Coordinator, Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP). He is a Fellow of the IEEE.

Yun Yang received his PhD degree from the University of Queensland, Australia, in 1992, in computer science. He is currently a full professor in the School of Software and Electrical Engineering at Swinburne University of Technology, Melbourne, Australia. His research interests include software technologies, cloud and edge computing, workflow systems, and service computing. He is currently serving as an Associate Editor for the IEEE Transactions on Parallel and Distributed Systems.