IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 3 July 2023, accepted 19 July 2023, date of publication 1 August 2023, date of current version 7 August 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3300844

RESEARCH ARTICLE

A Near Real-Time Big Data Provenance Generation Method Based on the Conjoint Analysis of Heterogeneous Logs

YUANZHAO GAO[1], XINGYUAN CHEN[1,2], BINGLONG LI[1], AND XUEHUI DU[1]
[1]Zhengzhou Science and Technology Institute, Zhengzhou 450000, China
[2]State Key Laboratory of Cryptology, Beijing 100878, China

Corresponding author: Xingyuan Chen (chxy302@vip.sina.com)

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB0803603.

ABSTRACT Data provenance is an effective approach for data security supervision. In the distributed, multi-user, and multi-layer big data system, only the provenance generation method, which leverages the information logged at both application and operating system level, has the capacity to completely obtain the provenance information required for data usage supervision. However, the current research on the conjoint analysis of multiple logs is inadequate, and it is difficult for them to effectively integrate the provenance information extracted from different logs, especially in the big data scenario. For the near real-time provenance generation based on the analysis of multiple heterogeneous logs, this paper employs a Hadoop-based big data system as the research object, and proposes a parallel log analysis method based on auxiliary data structures and multi-threading. For the efficient conjoint analysis of multiple logs, 5 auxiliary data structures are constructed as the medium for the correlation and fusion of log information, and a multi-threading method is presented to parallelize the lookup of provenance information. In order to cope with the complex log record generation rules, log analysis methods for nondeterministic records, non-instantaneous operations, and instantaneous batch operations are proposed to generate provenance information correctly. In addition, a provenance generation framework is established to implement the proposed log analysis method. The experimental results show that the log collection time overhead caused by processing files above MB level is less than 0.1%. The proposed method can analyze logs in near real time and generate provenance information correctly.

INDEX TERMS Big data provenance, provenance generation, multi-log conjoint analysis, hadoop.

I. INTRODUCTION

Big data is the innovative engine of economic and social development. While it is developing vigorously, data security issues are becoming increasingly serious. Big data security incidents occur frequently in recent years. Data security supervision and data governance face severe challenges [1], [2].

Data provenance, which describes the origins of data and the operation and processing procedure by which it arrived at the current state, covers the information of arbitrary operations supported by a data management and processing system to access data, and thus is naturally suitable for detecting data security threats (such as data leakage and data abuse) and achieving data security supervision [3], [4].

Obtaining sufficient provenance information is the pre-requisite for the full use of provenance in data security supervision. Big data systems provide diverse data services to numerous users, and generally present multi-layer and distributed characteristics on the data organization [5]. The provenance information required for data security supervision involves multiple users, working nodes [6], and even multiple data management systems at different layers, which brings great challenges to provenance tracking and generation.

The associate editor coordinating the review of this manuscript and approving it for publication was Chong Leong Gan.

Current provenance generation methods include operating system (OS)-level methods, application-level methods, and hybrid methods, which integrate both OS- and application-level methods [7], [8].

OS-level methods generate provenance by intercepting and analyzing system call information [9], [10]. These methods can obtain the information about all the operations occurred in the system, but they suffer from semantic gap problems in that system call information represents only atomic actions. Thus, OS-level provenance information alone is inadequate to effectively explain the behaviors of applications and users [11], [12], especially in the big data system.

Application-level methods include provenance-aware methods and log-based methods. The former captures provenance information by modifying the application's source code to enable provenance generation [13] or leveraging intrusive provenance tracking tools [14]. The latter extracts provenance information from application logs [15], [16] and is more practical. None of them can provide a complete provenance view of data objects in the big data system [7], [17].

Hybrid methods generally adopt both application logs and system call log to generate whole-system provenance information. They take advantage of the rich semantics of application logs and the rich information of system call log [18], and have the capacity to present a data object's entire provenance view in the whole file cycle.

Because the required provenance information is scattered in multiple logs, correlating and fusing the information of heterogeneous logs is a challenge for hybrid methods [15]. Rupprecht et al. [19] proposed URSPRUNG that adopts application-specific sources, which an application naturally produces during its execution (such as logs), and system call information to achieve transparent provenance generation. They did not consider the correlation and integration of related provenance information. Pasquier et al. [20] proposed a hybrid provenance capture mechanism, CamFlow. On the correlation of the OS- and application-level information, they only store related information together in a certain way, but do not integrate them further. Datta et al. [21] proposed a provenance-based auditing framework, which records both OS- and application-level provenance information. They took the system call log as the main provenance source, applied application logs only to supplement some metadata information, and did not propose multi-log conjoint analysis method. To achieve the fusion of the provenance information at two levels, Hassan et al. [7] proposed a provenance tracker, OmegaLog, which leverages the principle that a file operation execution process is also the corresponding application log writing process to link this process and the related application, and then link the OS- and application-level information. However, in the distributed environment, the above-mentioned processes may be not on the same node. Thus, this method is not applicable to the big data system. Yu et al. [18] proposed a heterogeneous log fusion technique, ALchemist, which first normalizes various logs to a canonical representation and then fuses different logs based on their same fields. However, the normalization of diverse logs is very complex. Moreover, they correlate different application logs by taking a system call log as the medium. Because of the multi-layer feature of big data systems in data organization, it is also very difficult to correspond the upper-layer application behavior with the underlying process behavior. In addition, Li et al. [22] proposed an event provenance graph construction method by analyzing the correlations among logs. They calculate the correlations between the logs in a specific event by comparing pairwise log element appearance frequencies in this event, rather than directly analyze the semantic or logical correlations between log elements in different logs.

Existing hybrid provenance generation methods are mostly based on system call information, whose expressiveness is not sufficient to represent the big data operation and processing procedure. Moreover, their research on multi-log conjoint analysis to realize the information fusion of different logs is inadequate and does not consider the provenance generation timeliness. Therefore, it is difficult for them to realize effective data usage monitoring and rapid security threats detection.

To address the above-mentioned problems, this paper employs a Hadoop-based big data system as the research object, and studies the multi-log conjoint analysis method for provenance generation in a distributed, multi-user, and near real-time scenario. Our research faces the following challenges:

(1) Multi-log conjoint analysis. For most provenance types, their attributes have to be extracted from multiple logs. The integration of different log analysis processes and the correlation of the related records in different logs need to be addressed.

(2) Near real-time provenance generation, which requires high efficiency of log processing. When a log is selected to generate a kind of provenance entry, the log analysis efficiency is mainly affected by the lookup of the missing information, which cannot be obtained from this log. The lack of effective lookup method may lead to a backlog of data to be analyzed. In addition, it is necessary to address the near real-time collection and centralized storage of logs distributed in different working nodes.

(3) Log analysis correctness. Log record generation rules indicate how an application or system records the activities that have occurred. For example, when a file copying operation occurs in the Hadoop Distributed File System (HDFS) [23], an ''open'' record that records the file to be copied and a ''create'' record that records the file to be created are generated in the HDFS audit log (HA-Log), but not a ''copy'' record. Hadoop log record generation rules are complex, which brings great challenges to correctly transforming log data into provenance information. Moreover, in the multi-user scenario, because that the operation records of different

users are stored together, an operation may generate multiple non-adjacent log records, which makes log analysis more complex.

HDFS is the data storage foundation of Hadoop. HDFS data are the core of provenance tracking. For efficient and correct HDFS data provenance generation, the main contributions of this paper are summarized as follows:

(1) We propose a parallel multi-log analysis method based on auxiliary data structures and multi-threading, which takes Hadoop application logs as the main provenance source and adopts a system call log as the supplement to the missing information in application logs. Overall, each log is analyzed in parallel and independently. We construct 5 auxiliary structures as the medium for the correlation and fusion of log information, and present a multi-threading method to parallelize the lookup of log missing information, so as to ensure the correct correlation and efficient analysis of heterogeneous logs.

(2) In this paper, log analysis is regarded as deterministic pattern recognition, i.e., the generation of arbitrary provenance information, such as data operation type and object, is regarded as the recognition of a specific pattern. In order to cope with the influence of non-deterministic records on the determination of operation type, we propose a determination method based on an interrupt mechanism. To analyze the operation records of coexisting files generated due to large file reading, we propose a parent-child thread cooperative analysis method. To analyze the batch operation records generated due to the attribute setting/getting of the whole directory, we propose a segmented analysis method.

(3) We present a provenance generation framework to enable the near real-time collection and centralized storage of the required logs stored on distributed nodes, so as to support the implementation of the proposed log analysis method.

(4) The experimental results show that collecting log records generated due to processing files above MB level does not incur obvious time overhead. The log analysis rate is always higher than the log generation rate, and the accuracy reaches 100% when the time thresholds used in the analysis method are all set correctly, which enable near real-time and correct provenance generation to provide a strong data foundation for provenance-based data security supervision.

The rest of the paper is organized as follows. Section II introduces the big data provenance model (BDPM) [24]. Section III specifies the required provenance information based on the BDPM model and the logs to generate it. Section IV describes the auxiliary data structures that are used to assist the multi-log analysis. Section V presents a multi-threading-based missing information seeking method. Section VI presents the log analysis methods based on auxiliary data structures and multi-threading. Section VII introduces a provenance generation framework. Section VIII includes the experimental results and evaluations. Finally, Section IX concludes this paper.

II. BIG DATA PROVENANCE MODEL

Provenance can be represented by a directed acyclic graph. A provenance model formally defines the elements used in provenance description, the dependencies between them, and the rules established on these elements and dependencies to effectively express provenance. It is the foundation of provenance research [25], [26]. Currently, the most widely used provenance model is the PROV-DM model [27] released by the World Wide Web Consortium (W3C) Provenance Working Group. The highly abstract nature of PROV-DM has led to its widespread adoption. For the big data scenario, we extended the PROV-DM model by subtyping and defining new relations, and proposed the BDPM model.

The BDPM model consists of three primary types of nodes: entity, activity, and agent, and fifteen types of edges representing the provenance dependencies: derivation, coexistence1, coexistence2, inclusion1, inclusion2, usage, start, end, generation, invalidation, communication, attribution1, attribution2, association, and delegation, as shown in Fig. 1. An entity is something we want to describe the provenance of. It refers to the data object in this paper. An activity is something that acts upon or with entities and changes their state. An agent is something that bears some form of responsibility for an activity occurring, for the existence of an entity, or for the activities of other agents [28]. Coexistence, inclusion, and attribution dependencies may be changed by activities. ''dependency1'' and ''dependency2'' represent the past and current dependency, respectively.

We specify the provenance types and information to be obtained according to the provenance ontology [29] built on the basis of the BDPM model and HDFS data supervision requirements.

III. ADOPTED LOGS

A. REQUIRED PROVENANCE INFORMATION

1) REQUIRED ENTITIES

HDFS data are the core of provenance tracking. In the BDPM ontology, the entity types corresponding to HDFS data is HDFSFile, which includes subtype HFile and HDirectory. An HDFS file is stored in the hosts' local file systems in the form of block files, which are named after HFile data block IDs. They coexist with each other. The provenance information of block files is required to promote the supervision of HFile access behaviors. In this study, the fourth extended file system (Ext4) is used as the local file system, and the entity type corresponding to the block file is E4File. All the required entity types are as follows:

$$\text{HFile} : < \text{ID}, \text{fileType}, \text{name}, \text{path}, \text{replication}, t_c, t_a, t_m, t_d, \text{permission}, \text{size} > . \quad (1)$$

$$\text{HDirectory} : < \text{ID}, \text{fileType}, \text{name}, \text{path}, t_c, t_a, t_m, t_d, \text{permission} > . \quad (2)$$

$$\text{E4File} : < \text{ID}, \text{size} > . \quad (3)$$

where ID represents the identifier of a provenance node. The identifier of an HDFS file is the inode ID assigned to it by

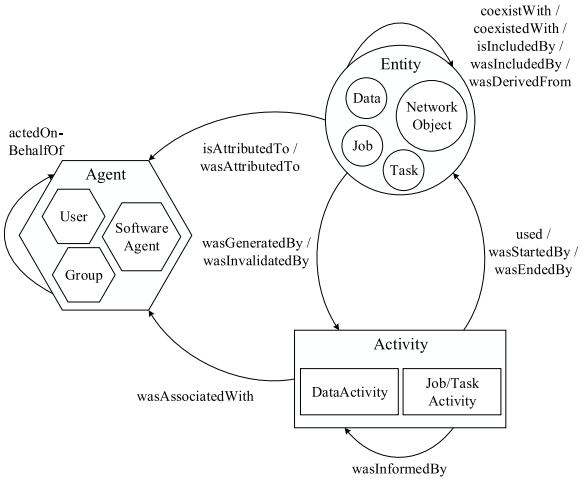


FIGURE 1. Structure of BDPM model.

HDFS. The identifier of a block file is its corresponding block ID. replication denotes a replication factor. t_c, t_a, t_m, and t_d denote the creation time, last access time, last modification time, and deletion time, respectively.

2) REQUIRED ACTIVITIES

In this paper, we focus on HDFS operations and the block file operations occur due to HDFS operations. The required activity type is FileDataOperation:

$$\text{FileDataOperation} : < \text{ID}, [\text{IP}], [t_s, t_e, t_o], [\{\text{args}\}], [\text{finalStatus}], [\text{mode}] > . \quad (4)$$

Activities lack distinct identification information. Their identifiers need to be self-generated. The elements in the square bracket are optional. t_s, t_e, and t_o denote the starting time, ending time, and occurrence time, respectively. {args} denotes activity arguments. finalStatus denotes the operation result, such as success and failure. mode denotes the operation execution mode, which represents the way users access HDFS, including command line interface (CLI), HttpFS, WebHDFS, and network file system (NFS). Considering that the function of NFS is to mount HDFS as part of the local file system [30], we focus on the first three modes.

3) REQUIRED AGENTS

According to the required entity and activity types, the required agent type is User:

$$\text{User} :< \text{ID} > . \quad (5)$$

We assume that the name of each user is unique and immutable, and take the name as the identifier of a user.

4) REQUIRED PROVENANCE DEPENDENCIES

According to the required node types, the required dependency types include all the dependencies except for "start" and "end."

TABLE 1. HA-Log fields.

Field name	Usage
date	Date when an operation was executed
time	Time when an operation was executed
level	Logging level
allowed	Indicates whether an operation was allowed to be executed
ugi (auth)	User who executed an operation (auth is used to identify a user according to the mode of operation)
ip	IP address where an operation occurred
cmd	Record type, which denotes the command that was executed, such as open, create
src	Full file path of the operation input
dst	Full file path of the operation output, may be null
perm	Owner and permission of a file
protocol	Communication protocol, such as RPC and WebHDFS

We select the following five logs to generate the required provenance information.

HDFS logs: HA-Log, EditLog (HDFS transaction log) in the XML format, HttpFS audit log (HTA-Log), and Jetty NameNode log (JNN-Log), and Progger log (P-Log).

HA-Log and EditLog record the operations performed via any execution mode. They are the most important data sources for provenance generation. The information that can be extracted from HA-Log is summarized in Table 1. It has the strongest correlation with the specified provenance information. The HDFSFile operation-related EditLog record (or transaction) types are summarized in Table 2. The operation object is represented by the file's absolute path. The "OP_RENAME" record is generated when a file is moved to the trash, and the "OP_RENAME_OLD" record is generated in other instances of renaming or moving.

HTA-Log and JNN-Log record the timestamp, user name, IP address, type, object, and parameters of operations performed via the HttpFS and WebHDFS mode, respectively.

Progger is a configurable OS-level provenance tracking tool [31], [32]. The information of some local operations occur in the hosts' file systems that Hadoop logs do not record can be obtained from P-Log [33], such as block file operation information. The system call types required in this paper and the information to be logged are summarized in Table 3 (the number in brackets indicates the system call type in P-Log). We briefly introduce the elements in these records:

(1) PID, PPID, SID, and PSID denote the process ID, parent process ID, process ID of a session leader, and parent process ID of a session leader, respectively. Program denotes the process name.

(2) File, Flags, Mode, and FD denote the file absolute path, file access mode, file permission, and file descriptor, respectively. File descriptor is an abstract indicator used to access a file or some other input/output resource, such as a network socket [34].

(3) SocketFD denotes the socket descriptor. sIP, sPort, dIP, and dPort denote the source IP, source port, destination IP, and destination port, respectively.

TABLE 2. File operation-related EditLog record types.

Record type	Meaning	Object	IP	Timestamp	HDFSFile attributes
OP_ADD	File creation	√	√	-	Inode ID, replication factor, t_c, t_a, owner, permission
OP_ADD_BLOCK	*	√	-	-	Block ID
OP_ALLOCATE_BLOCK_ID	*	-	-	-	Block ID
OP_APPEND	*	√	√	-	-
OP_CLOSE	*	√	-	-	Replication factor, t_a, t_m, block set, size, owner, permission
OP_CONCAT_DELETE	Concatenation	√	-	√	-
OP_DELETE	*	√	-	√	-
OP_MKDIR	Directory creation	√	-	√	Inode ID, owner, permission
OP_REMOVE_XATTR	Extended attribute deletion	√	-	-	Extended attributes
OP_RENAME	*	√	-	√	Name
OP_RENAME_OLD	*	√	-	√	Name
OP_SET_ACL	*	√	-	-	ACL
OP_SET_OWNER	*	√	-	-	Owner
OP_SET_PERMISSIONS	*	√	-	-	Permission
OP_SET_REPLICATION	*	√	-	-	Replication factor
OP_SET_XATTR	Extended attribute setting	√	-	-	Extended attributes
OP_SYMLINK	Symbolic link creation	√	-	-	Inode ID, t_c, t_a, owner, permission
OP_TIMES	Timestamp setting	√	-	-	t_a, t_m
OP_TRUNCATE	*	√	√	√	File size after truncation, block set before truncation
OP_UPDATE_BLOCKS	Block information update	√	-	-	Block set

* indicates that the record meaning can be obtained directly from the record type. √ indicates that a record type contains the information listed in the table. - is opposite to √.

TABLE 3. System call types to be logged.

System call type	Record format
SYS_OPEN (0)	Date, Time, HostName, Type, User, PID, PPID, SID, PSID, Program, File, Flags, Mode, FD
SYS_CLOSE (5)	Date, Time, HostName, Type, User, PID, PPID, SID, PSID, Program, File, FD
SYS_READ (6)	Date, Time, HostName, Type, User, PID, PPID, SID, PSID, SocketFD, sIP, sPort, dIP, dPort
SYS_CONNECT (7)	Date, Time, HostName, Type, User, PID, SID, SocketFD, sIP, sPort, dIP, dPort
SYS_ACCEPT (23)	Date, Time, HostName, Type, User, PID, SID, SocketFD, sIP, sPort, dIP, dPort

"SYS_OPEN" and "SYS_CLOSE" are only recorded when the operation object is a block file. "SYS_READ" is recorded when the file reading is executed via a socket descriptor.

To improve the log analysis efficiency as a whole, we analyze these logs in parallel. Meanwhile, because that the attributes of some provenance types have to be extracted from multiple logs and the recording of provenance is generally caused by activities. For a specific HDFSFile operation, we select a particular log, which we call starting log, to generate its provenance entry, and generate or change other related provenance entries when analyzing this log. Missing information can be obtained from auxiliary structures whose data are extracted from logs, or sought from other logs by creating a child thread. For HDFSFile creation, the starting log is EditLog. For other operations, the starting log is HA-Log. The relationship between the adopted logs is shown in Fig. 2.

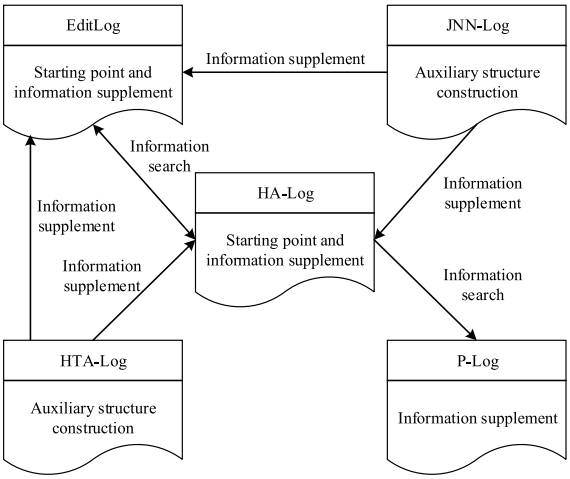


FIGURE 2. Relationship between adopted logs.

IV. AUXILIARY DATA STRUCTURE

Under the condition of multi-user, multi-log, and multiple operation execution modes, we extract part of the log information during the log analysis process, and save it into 5 auxiliary structures to accelerate the lookup of the missing information. These structures are applied to the construction of the proposed log analysis methods. Their functions are as follows:

(1) Auxiliary structures are the medium for the correlation of different logs. A log parser can get the missing information from an auxiliary structure via taking the common elements between this structure and the log as link points, so as to improve the efficiency of seeking missing information.

Compared with ALchemist [18], we do not normalize all the logs, but only extract the required information from logs and organize it in a way that is easier to find, which is more practical.

(2) Auxiliary structures store some state information of files and their operations, which can be used to ensure the correctness of log analysis under the condition of parallelization and multi-threading, and to assist the analysis of multi-record operations that generate other subsequent log records in addition to the first record or record combination, which is used to determine the operation type.

A. HDFSFILE HASH TABLE

HDFSFile hash table (H-Hash) is the most important auxiliary structure. The entry of H-Hash is added when analyzing EditLog, and modified or deleted when analyzing other logs.

Because that the operation object recorded in logs is generally represented by the file's absolute path, we set the key of H-Hash to HDFSFile absolute path. The value of H-Hash stores an HDFSFile state information array in the following format:

$$< \{\text{inodeID}, t_c, \text{fileType}, \text{size}, \text{blockNum}, \{\text{blockID}, \text{blockSize}\}_m, \{\text{userName}, \text{IP}, \text{opType}, \text{opMode}, \text{opID}, \text{inodeID}_{\text{OP}}, t/num_{\text{O}}\}_n\}_k > . \quad (6)$$

k denotes the number of HDFSFiles with the same absolute path supported by H-Hash. Suppose that f_1 and f_2 are two files with the same path that have been deleted and created successively. Because that the analysis of EditLog and HA-Log is asynchronous, if f_2 creation record is analyzed before f_1 deletion record, the setting of k can ensure that f_1 information will not be overwritten by f_2 information and the object of the deletion operation is identified correctly. The identification basis is t_c. Suppose that the timestamp and file path of an operation record, r_1, are t_1 and fp_1, and there are multiple files whose file path is fp_1 in H-Hash. The operation object recorded by r_1 is the file with the largest t_c in the files whose t_c is less than t_1. inodeID is used to convert the path of a file into its provenance ID, as well as to quickly locate the corresponding data entry in another auxiliary structure, the HDFSFile block and hierarchy file (BH-File). fileType is used to help identify whether the operation object is a single file or the whole directory. The latter refers to the operations act upon a directory and its descendant files or directories, which are only supported by the CLI mode. blockNum, blockID, and blockSize are used to analyze operations involving block files. m is the maximum number of blocks supported by H-Hash.

$\{\text{userName}, \text{IP}, \text{opType}, \text{opMode}, \text{opID}, \text{inodeID}_{\text{OP}}, t / num_{\text{O}}\}_n$ is used to store the information of non-instantaneous file data modifying operations and multi-record operations. n is the number of operations supported by H-Hash. IP is only extracted from HA-Log or EditLog. opMode denotes the operation execution mode. opID denotes the provenance identifier of an operation, which is used to supplement the provenance information of multi-record operations when analyzing their subsequent records. $\text{inodeID}_{\text{OP}}$ denotes the inode ID of the first or last object of a whole-directory operation. t denotes the timestamp of an operation record. num_{O} denotes the number of ''open'' records generated due to file reading in HA-Log.

HDFS non-instantaneous data modifying operations include file creation, appending, and truncation. When analyzing these operations, the functions of the above-mentioned information are as follows:

(1) Assisting the multi-threading-based parallel log analysis. These operations change a file's size, data block, and t_m, which need to be extracted from EditLog. Once a file appending or truncation record is met in HA-Log, a child thread is created to seek the corresponding operation record in EditLog based on the operation information stored in H-Hash.

(2) Avoiding recording incorrect file reading range. Suppose that when analyzing a reading record of a file, f_1, in HA-Log, H-Hash shows that f_1 is being appended. Then a data block read by this operation maybe not recorded in H-Hash or a block recorded in H-Hash maybe not read by this operation. It is necessary to get all the blocks that f_1 currently has from EditLog and find the reading records of the corresponding block files from P-Log to determine the blocks that were actually read.

Multi-record operations include single-file operations and whole-directory operations. The subsequent records of these operations can be identified via the operation information stored in H-Hash. In multi-user and near real-time scenarios, storing the information of these operations into H-Hash rather than finding all the records at once avoids skipping too many intermediate records and can effectively deal with the situation that not all the operation records have been generated.

B. HDFSFILE BLOCK AND HIERARCHY FILE

BH-File is created when analyzing EditLog and updated during the analysis of other logs. Each HDFSFile has a BH-File data item.

The data item of HFile records inode ID, file path, and block set. When the number of a file's blocks exceeds the limit of H-Hash, BH-File can be used to obtain its block information. The data format is as follows:

$$< \text{inodeID}, \text{path}, \{\text{blockID}, \text{blockSize}\} > . \quad (7)$$

The data item of HDirectory records inode ID, path, and a collection of subfiles/subdirectories, which are arranged by their names. This collection is used for the analysis of whole-directory operations. The data format is as follows:

$$< \text{inodeID}, \text{path}, \{\text{inodeID}, \text{fileType}, \text{path}\} > . \quad (8)$$

Each BH-File stores the information of HDFSFiles with inode IDs within a certain range to facilitate the quick location of a specific BH-File.

C. LINKED LIST FOR RECORDS NOT ANALYZED IN SEQUENCE

In general, we analyze log records by their storage order. However, for some non-deterministic records, such as r_1, we need to seek and analyze its subsequent record r_2 that is generated due to the same user with r_1 to determine the operation type represented by r_1. If r_1 and r_2 are not adjacent, and r_2 has been analyzed before some of its previous records, we put the location information of r_2 into this linked list (R-List) to avoid repeated analysis.

When analyzing a record, we first determine whether it may appear in R-List according to its type. If possible, we check whether this record is in R-List. If so, we just delete the node corresponding to it in R-List and analyze the next record.

D. LINKED LIST FOR HTA-LOG AND JNN-LOG

When analyzing some operations performed via the HttpFS or WebHDFS mode, it is necessary to leverage HTA-Log or JNN-Log to determine the operation type or get the operation information. Because that no operation generates multiple records in HTA-log or JNN-Log, we store required log records into a linked list by their storage order, i.e., HTA-List and JNN-List. We add nodes from the tail, seek data from the head, and delete a node after obtaining the required information from it. The operation sequence recorded by HA-Log/EditLog is generally consistent with that recorded by HTA-Log/JNN-Log. In most cases, the required information can be obtained from the head of HTA-List/JNN-List, which ensures the efficiency of seeking missing information. The data format is as follows:

$$< t, \text{user}, \text{IP}, \text{opType}, \text{path}, \{\text{args}\} > . \tag{9}$$

V. MULTI-THREADING-BASED MISSING INFORMATION SEEKING METHOD

Because of the complexity and diversity of log information and the analysis asynchronism of different logs, auxiliary structures do not cover all the missing information required for log analysis. Therefore, we propose a multi-threading-based missing information seeking method as a supplement to the auxiliary structures. When neither the record being analyzed nor the related auxiliary structure contains the required information, the log analysis main thread creates a child thread to seek the missing information from other logs or wait for other log parsers to put the required information into the auxiliary structure, and then continues to analyze the subsequent records. Leveraging multi-threading to parallelize the lookup/wait of missing information and log analysis reduces the impact of lack of information on the near real-time generation of provenance information. Child thread creation scenarios are as follows:

(1) The auxiliary structures do not contain the required information type. Such information can only be sought from logs.

(2) The auxiliary structures contain the required information type, but the required information has not yet been generated. If the missing information does not affect the subsequent log analysis, the main thread creates a child thread to wait for the information to be generated (For example, Getting the operation IP from HTA-List for an attribute setting operation performed via the HttpFS mode). Otherwise, the main thread pauses to wait (For example, H-Hash does not contain the entry of the operation object being analyzed).

(3) The auxiliary structures contain the required information type, but it is uncertain whether the information corresponding to the current record will be generated. For example, the main thread leverages JNN-List to determine whether an operation was performed via the WebHDFS mode. But there exists no information related to this operation in JNN-List. The main thread cannot determine whether the execution mode was not WebHDFS or the required information has not been put into JNN-List. Thus, it needs to create a child thread to find out whether the corresponding record exists in JNN-Log.

In addition, if a child thread has already been created when analyzing an operation, the main thread does not create a child thread when the above-mentioned situations occur again.

VI. LOG ANALYSIS METHODS

A. OVERALL LOG ANALYSIS PROCESS

Based on the Hadoop source code analysis and a large number of experimental tests, we establish some deterministic log analysis and pattern recognition methods. The overall log analysis process is shown in Fig. 3. The steps in the dashed boxes are only performed for the analysis of specific logs or specific operations. Hereinafter, unless otherwise specified, r_1, op_1, and $o_1/f_1/d_1$ denote the record, operation, and operation object currently being analyzed, respectively. The main log analysis steps are as follows:

(1) Analysis scope identification. According to the pre-divided log analysis scope, determining whether r_1 needs to be analyzed based on r_1 type and o_1 path or name.

(2) Operation type identification.

1) Execution mode identification. For HA-Log, differences in execution modes lead to differences in record characteristics and analysis methods. Thus, to analyze an operation, we need to first identify its execution mode by analyzing the record characteristics on specific fields. For example, the "proto" value of records generated due to the operations performed via the WebHDFS mode is typically "webhdfs." Due to space limitations, we do not introduce all the record characteristics in detail here.

2) Identifying the operation that generated r_1 and its type. For some record types, it is necessary to analyze the H-Hash entry of o_1 at first to determine whether r_1 is the subsequent record of a multi-record operation. If so,

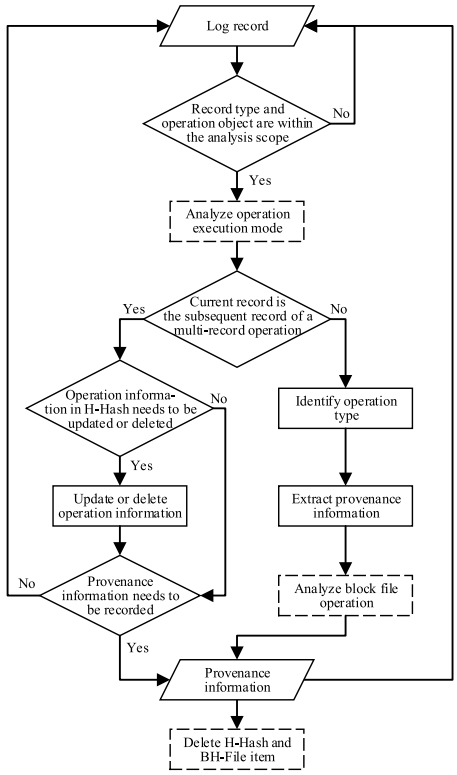


FIGURE 3. Log analysis process.

we further determine whether to update H-Hash and record provenance information according to op_1 type. Otherwise, we analyze op_1 type, where the analysis of non-deterministic records is a challenge.

(3) Operation analysis. Extracting provenance information from logs. The difficulties lie in the analysis of non-instantaneous multi-record operations.

(4) H-Hash and BH-File item deletion, which only involves HA-Log.

B. NON-DETERMINISTIC RECORD ANALYSIS METHOD

In EditLog and HA-Log, some record types can uniquely represent a certain operation type (such as "append" of HA-Log). We call them deterministic records. Some record types may occur in different operations (such as "open" of HA-Log). We call them non-deterministic records. A non-deterministic record may need to be combined with other records to represent a certain file operation. For example, the combination of "open+create" in HA-Log that satisfies some conditions represents file copying.

In order to determine the operation type represented by a non-deterministic record (such as r_1), it is first necessary to identify whether r_1 is the starting record of a record combination, i.e., to find whether there exists a subsequent record of a record combination after r_1. Because that the operation records of the same user may not be adjacent, we draw on the OS interrupt principle, and propose a non-deterministic record analysis algorithm based on an interrupt mechanism, as shown in Algorithm 1.

Algorithm 1 Non-deterministic Record Analysis Algorithm

Input: log_1 that is being analyzed and R-List
Output: Provenance information

1: $breakpoint \leftarrow 0$
2: $r_2 \leftarrow$ readFileRecord(log_1, position(r_1) + 1)
3: **while** $r_2 \neq$ NULL && ((exists(r_2.time) && r_2.time − r_1.time $< t_{\max}$) || !(exists(r_2.time))) **do**
4: **if** $breakpoint = 0$ && r_2.user $\neq r_1$.user && r_2.type $\in T_R$ **then**
5: $breakpoint \leftarrow$ position(r_2)
6: **else if** r_2.user $= r_1$.user && r_2.type $\in T_R$ **then**
7: **break**
8: **end if**
9: $r_2 \leftarrow$ readFileRecord(log_1, position(r_2) + 1)
10: **end while**
11: **if** r_2.time − r_1.time $< t_{\max}$ **then**
12: **if** r_1 and r_2 form a record combination **then**
13: Determine op_1 type and generate op_1 related provenance information
14: **if** $breakpoint \neq 0$ **then**
15: put position(r_2) into R-List
16: **end if**
17: **end if**
18: **end if**

When r_1 has been determined as a non-deterministic record, the log parser seeks forward for the next record r_2, which was generated due to the same user with r_1, within a given time interval threshold, and determines op_1 type according to the find results. During the lookup process, the thread records the breakpoint location when it meets a record that is within the analysis scope (T_R denotes the record types in the analysis scope in line 4) and was generated due to other users for the first time. If r_2 exists, forms a record combination with r_1, and is not adjacent to r_1, after r_2 has been analyzed, its location is stored into R-List. Then, the thread continues to analyze log records from the breakpoint if it is not 0.

For HA-Log, a line of data is a record. For EditLog, the record unit is called transaction, which contains multiple lines, and the unit of measurement for position in Algorithm 1 is transaction. Besides, not all the EditLog records contain time information.

C. NON-INSTANTANEOUS MULTI-RECORD OPERATION ANALYSIS METHODS

The number of records generated due to the same operation in different logs is different. For EditLog, file creation is a multi-record operation. For HA-Log, single-file multi-record operations include reading, copying, and replication factor setting with waiting mode, which is executed via the CLI mode with parameter "-w" to wait for the replication to complete. Whole-directory operations include reading, copying, attribute setting/getting, and search.

Accurate identification of all the records generated due to an operation and the normal or abnormal end of an operation are the difficulties for analyzing the above-mentioned operations. For file creation, file reading executed via non-CLI modes, and file replication factor setting with waiting mode, the identification can be done based on the operation information recorded in H-Hash. For other operations, we propose a parent-child thread cooperative analysis method and a segmented analysis method based on auxiliary structures. We introduce them in detail below.

1) PARENT-CHILD THREAD COOPERATIVE ANALYSIS METHOD

This method is applied to analyze large file reading and whole-directory reading executed via the CLI mode. The size of an HDFS block is 128 MB by default [23]. In this paper, we call a file large file if it is larger than 10 blocks.

HFile reading causes block file reading. Unlike the WebHDFS and HttpFS mode, Hadoop logs do not record the read range of HFile reading performed via the CLI mode. Thus, to accurately obtain the HFile read range at the block level, we make the HA-Log analysis thread create a child thread to seek the corresponding block file reading records in P-Log. Meanwhile, to ensure the correctness of HA-Log analysis, we propose a parent-child thread cooperative analysis method. We first briefly describe the characteristics of the ''open'' record in HA-Log to illustrate the reason for applying this method.

(1) ''open'' record characteristics

An HDFS cluster consists of a single NameNode, which manages the file system namespace, and a number of DataNodes, which stores file data, i.e., block files [35]. When HDFS performs file reading, it typically takes 10 data blocks as a batch to locate the DataNodes where they are located, generates an ''open'' record in HA-Log, and then reads block files. For a batch of blocks, the ''open'' record and the ''SYS_OPEN'' record of the first block file are in one-to-one correspondence, and their time interval is extremely short.

Meanwhile, ''open'' is a non-deterministic record. There exist other operations that generate ''open'' records in addition to file reading. In the CLI mode, the operations with ''open'' as the starting record are shown in Table 4, where ''getBlockLocations'' and ''read'' are not shell commands. They are used to refer to the file operations performed via the Java API, which belongs to the CLI mode.

For an ''open'' record, r_1, which was generated due to an operation performed via the CLI mode, if op_1 type cannot be determined via Algorithm 1, the main thread creates a child thread to identify op_1 type by analyzing the corresponding record of r_1 in P-Log. If f_1 is a large file, it first stores op_1 information into f_1 H-Hash entry and sets ''opType'' to ''UD (undetermined).'' When the main thread meets a f_1 ''open'' record, r_2, generated due to the same user with r_1, if f_1 H-Hash entry shows op_1 type is ''UD,'' it will not create a child thread again.

TABLE 4. Non-deterministic record ''open.''.

CLI command	Potential record combination	Operation type
cp	open + create	Copying
cat, copyToLocal, get, getmerge, read, tail, text	-	Reading
checksum		Checksum acquisition
getBlockLocations		Block location acqusition
read	open + open	Reading
getBlockLocations		Block location acqusition

- means that the file operation does not generate a record combination.

Suppose that op_1 is file reading and f_1 is a large file, if the user (assuming u_1) who executed op_1 got f_1 checksum as well during f_1 reading, or u_1 immediately read f_1 again after op_1 ended abnormally because of network outage, when the main thread meets a f_1 ''open'' record generated due to u_1 again, it cannot determine whether this record was generated due to op_1 correctly based on HA-Log alone, but needs to rely on the corresponding operation records in P-Log.

(2) Parent-child thread cooperative analysis method for a single file

To address the interference brought by the ''open'' record generation rule to the correct analysis of large file reading, we propose a parent-child thread cooperative analysis method by leveraging the correspondence between the ''open'' record generated due to HFile reading and the ''SYS_OPEN'' record of the first block file in a batch (hereinafter referred to as first block). This method takes message queue as the communication medium of parent and child threads.

Each time the child thread finds a ''SYS_OPEN'' record of f_1 first block, it sends the record's timestamp to the message queue constructed for op_1. When the main thread meets a f_1 ''open'' record r_k, if the H-Hash entry of f_1 shows that it is being read by op_1, it gets a timestamp, t, from the corresponding queue, and compares whether the interval between the timestamp of r_k and t is within a given threshold. If so, it can be determined that r_k was generated due to op_1.

The main and child thread analysis algorithms are Algorithm 2 and Algorithm 3, respectively. Suppose that the main thread cannot determine op_1 type and has created a thread to analyze the corresponding records of r_1 in P-Log.

Algorithm 2 describes the analysis process when the main thread meets a f_1 ''open'' record again. In line 3, if the condition is true, the main thread continues to analyze whether r_2 was generated due to op_1. In line 5, the main thread waits for the child thread to identify op_1 type. ''RD'' refers to file reading. In lines 12-14, if $|t - r_2.\text{time}| > t_{\max}$, it is determined that r_2 was not generated due to op_1, and the main thread continues to seek forward for the f_1 ''open'' record corresponding to t. In line 16, if $op_1.num_O$ is reduced to 1, it indicates that all the ''open'' records generated due to op_1 have been analyzed. The main thread deletes op_1 information in the H-Hash entry of f_1 and releases the related message queue.

Algorithm 2 Main Thread HA-Log Analysis Algorithm

Input: HA-Log, H-Hash, R-List, and MQ, which refers to the message queue
Output: Provenance information

1: r_2 ← readFileRecord(HA-Log, ($breakpoint \neq 0$? $breakpoint$: position(r_1) + 1))
2: **while** $r_2 \neq$ NULL **do**
3: **if** r_2.src = f_1 && r_2.cmd = "open" && r_2.ugi = op_1.userName **then**
4: **if** op_1.opType = "UD" **then**
5: Query f_1.hashEntry periodically until op_1 information is deleted or op_1.opType is changed to "RD"
6: **else if** op_1.opType = "RD" **then**
7: **if** MQ = NULL **then**
8: Query MQ periodically until it is not NULL
9: **else**
10: Get a timestamp, t, from MQ
11: **end if**
12: **if** $|t - r_2.\text{time}| > t_{\max}$ **then**
13: Seek forward for the f_1 "open" record corresponding to t and put it into R-List
14: **end if**
15: $op_1.num_O$ - -
16: **if** $op_1.num_O = 1$ **then**
17: Delete op_1 information in f_1.hashEntry and release MQ
18: **end if**
19: **end if**
20: **end if**
21: r_2 ← readFileRecord(HA-Log, position(r_2) + 1)
22: **end while**

Algorithm 3 describes the analysis process that the child thread seeks f_1 block file "SYS_OPEN" records generated due to op_1, and determines op_1 type as well as whether op_1 ended abnormally. In line 1, according to r_1.time, the child thread seeks f_1 "SYS_OPEN" record within a certain time range after r_1.time. If it is found, op_1 is file reading. In line 3, num_O is calculated according to f_1 length and the first block read by op_1. If op_1 did not actually read all the data blocks after this block, op_1 can still be analyzed correctly by using the file reading exception handling methods in Lines 19-26. In line 6, the PID and SID value of the block file reading process and the corresponding TCP connection establishing process can be used to determine whether pr_2 was generated due to op_1. Similarly, the PID and SID value of the related "SYS_CONNECT" records can be used to determine whether pr_1 and pr_2 were generated due to the operations executed by the same user in line 11. Line 18 indicates two cases where op_1 ended abnormally. The former indicates that the next f_1 block file "SYS_OPEN" record was not found beyond the given time threshold. The latter indicates that the user re-executed f_1 reading from where it was interrupted after op_1 ended abnormally. Lines 19-26 show that how the child thread deals with op_1 exception according to the analysis progress of the main thread. In line 19, $num_{\text{O-A}}$, which refers to the actual number of "open" records generated due to op_1, is calculated according to the last block read by op_1. $num_{\text{O-E}}$ refers to the expected "open" record number in line 20. If $num_{\text{O-A}}$ is equal to $num_{\text{O-E}}$, the child thread just exits. Else, line 21 indicates that all the "open" records generated due to op_1 have been analyzed by the main thread, whereas line 23 is the opposite. And the child thread executes different processing.

Algorithm 3 Child Thread P-Log Analysis Algorithm

Input: P-Log, BH-File, H-Hash, and MQ
Output: Provenance information

1: seek f_1 block file "SYS_OPEN" record pr_1
2: **if** pr_1 is found **then**
3: Calculate num_O, put it into $op_1.num_O$ of f_1.hashEntry, and set op_1.opType to "RD"
4: pr_2 ← readFileRecord(P-Log, position(pr_1) + 1)
5: **while** $pr_2 \neq$ NULL && pr_2.Time − pr_1.Time < $t_{\max}$ **do**
6: **if** pr_2 is a f_1 block file "SYS_OPEN" record generated due to op_1 **then**
7: **if** pr_2.File is a first block of f_1 **then**
8: Send pr_2.Time to MQ
9: **else if** pr_2.File is the last block file of f_1 **then**
10: **break**
11: **else if** pr_2.File = pr_1.File && pr_2 and pr_1 were generated due to the same user **then**
12: **break**
13: **end if**
14: $pr_1 \leftarrow pr_2$
15: **end if**
16: pr_2 ← readFileRecord(P-Log, position(pr_2) + 1)
17: **end while**
18: **if** pr_2.Time − pr_1.Time > $t_{\max}$ || pr_2.File = pr_1.File **then**
19: Calculate $num_{\text{O-A}}$
20: **if** $num_{\text{O-A}} < num_{\text{O-E}}$ **then**
21: **if** $op_1.num_O = num_{\text{O-E}} - num_{\text{O-A}} + 1$ **then**
22: Delete op_1 information in f_1.hashEntry and release MQ
23: **else**
24: $op_1.num_O$ += $num_{\text{O-A}} - num_{\text{O-E}}$
25: **end if**
26: **end if**
27: **end if**
28: **end if**

(3) Parent-child thread cooperative analysis method for the whole directory

Whole-directory operations do not have a definite ending flag in HA-Log. To identify the ending record of the whole directory reading, we get the inode ID of the last operation object (assuming o_L) in d_1 with depth first traversal, i.e. file reading order, according to the file hierarchy stored in BH-Files, and put it into op_1 information of d_1 H-Hash entry. However, due to the dynamic change of files in d_1 or op_1 exception, the actual o_L may be different from the recorded value. In order to correctly identify all the "open" records generated due to op_1, we still adopt the parent-child thread cooperative analysis method, whereas the analysis process of the child thread needs to be adjusted as follows:

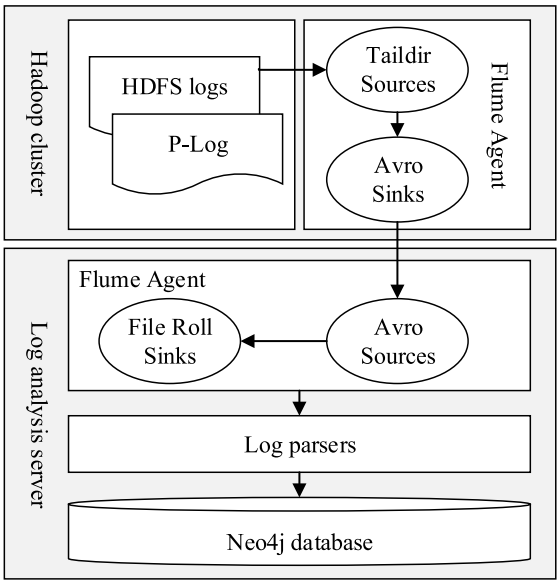


FIGURE 4. Provenance generation framework.

1) The child thread seeks the block file "SYS_OPEN" records of d_1's descendant files in turn with depth first traversal. Each time it finds a first block "SYS_OPEN" record (Not limited to large files), besides the record timestamp, it also sends the file inode ID and an operation ending flag to the message queue. If the found first block is the last one read by op_1, the flag is set to 1. Otherwise, it is set to 0.

2) If the next "SYS_OPEN" record was not found beyond the time threshold, the child thread determines that op_1 ended abnormally. It releases the message queue after all the messages have been read by the main thread, and deletes all the op_1 information in H-Hash.

Besides "open" records, the main thread analyzes the "listStatus" records (Indicating the operation object is a directory) generated due to op_1. Each time it meets such a record, it stores op_1 information into this directory's (assuming d_k, $k \geq 1$) H-Hash entry as one of the bases for identifying whether an "open/listStatus" record of d_k's child file/directory is generated due to op_1.

2) SEGMENTED ANALYSIS METHOD

This method is applied to analyze whole-directory attribute setting/getting operations and file search. These operations do not involve block files. Thus, the parent-child thread cooperative analysis method does not apply to them.

A single attribute setting/getting operation is an instantaneous operation. Thus, the time interval between adjacent records generated by the whole-directory operation is extremely short. Taking advantage of this feature, we propose a segmented operation analysis method. We take the adjacent records generated by the same operation as a segment and analyze them continuously. When a segment is interrupted, if it is determined that this operation does not end, we store the timestamp of this segment's last record into H-Hash for subsequent segment identification.

This method is divided into an intra-segment record analysis algorithm and a segment identification algorithm, as shown in Algorithm 4 and Algorithm 5. Suppose that op_1 information has been stored into d_1 H-Hash entry, and $\text{inodeID}_{\text{OP}}$ is set to the inode ID of the current o_L.

Algorithm 4 covers the possible situations that may be encountered when analyzing a segment's last record. The HA-Log parser finds out whether o_L still exists in HDFS, analyzes the sequence between r_3.src (assuming o_3) and o_L, and gets o_L^*, which refers to the updated last operation object in d_1, through BH-File. In line 4, if r_2 and r_3 were generated due to the same user, it is determined that r_2 belongs to the current segment. Line 8 indicates that the segment is interrupted. If op_1 does not end, the parser records r_3.time as shown in line 11. Except the situation that o_3 is before o_L, if the last operation object in d_1 has changed, the parser also stores the inode ID of o_L^* into H-Hash in line 17. Line 27 indicates that o_L does not exist.

Algorithm 4 Intra-segment Record Analysis Algorithm

Input: HA-Log, BH-File, and H-Hash
Output: Provenance information

```
1:  r2 ← readFileRecord(HA-Log, position(r1) + 1)
2:  r3 ← r1
3:  while r2 ≠ NULL do
4:    if r2.ugi = r3.ugi then
5:      if r2.cmd = "listStatus" then
6:        Store op1 information into r2.src.hashEntry
7:      end if
8:    else
9:      if oL exists then
10:       if r3.src, o3, is before oL then
11:         Record r3.time into op1 information in the
            H-Hash entries of the directories on o3.path
            between o3.parent and d1
12:       else if o3 is oL then
13:         if oL is still the last operation object in d1 then
14:           Record op1 ending time and delete op1 H-
              Hash information
15:         else
16:           Get the last operation object in d1 for now,
              o*L
17:           Record r3.time and o*L.inodeID into H-Hash
              as line 11 does
18:         end if
19:       else
20:         Get o*L
21:         if o3 is o*L then
22:           Execute the pseudocode in line 14
23:         else
24:           Execute the pseudocode in line 17
25:         end if
26:       end if
```

```
    else
      Execute the pseudocode from line 20 to 25
    end if
    break
  end if
  r3 ← r2
  r2 ← readFileRecord(HA-Log, position(r2) + 1)
end while
```

Algorithm 5 describes when the log parser meets a record r_1 that is suspected to be generated due to op_1, it makes further confirmation by comparing whether the time interval between r_1.time and the timestamp of the previous segment's last record is within a given time threshold. If not, it is determined that op_1 ended abnormally.

Algorithm 5 Segment Identification Algorithm

Input: HA-Log and H-Hash
Output: Provenance information

1: **if** the H-Hash entry of r_1.src's parent directory contains op_1 information && r_1.cmd = op_1.opType && r_1.ugi = op_1.userName **then**
2: **if** r_1.time $- op_1.t < t_{\max}$ **then**
3: Analyze this segment leveraging Algorithm 4
4: **else**
5: Record op_1 ending time as $op_1.t$ and delete op_1 H-Hash information
6: **end if**
7: **end if**

VII. PROVENANCE GENERATION FRAMEWORK

In this section, we establish a provenance generation framework to support the implementation of the proposed log analysis method via the near real-time collection and centralized storage of the adopted logs. The framework is shown in Fig. 4.

A. LOG COLLECTION

The adopted logs are all stored in the local file system of Hadoop hosts. EditLog, HA-Log, and JNN-Log are stored on NameNode. HTA-Log is stored on the node that runs HttpFS server. P-Logs are stored on all the nodes that run Progger.

We adopt the distributed streaming data collection system Apache Flume [36] to collect the adopted logs. Flume supports collecting and aggregating large amounts of log data from many different data sources and sending them to a centralized destination. By using Flume Taildir source, whenever new files are created or new lines are appended to existing files in the specified directories, Flume can collect the fresh data in nearly real time. Collected data are first sent to the log analysis server via Avro sinks and sources. Then, Avro sources send different logs to different directories to ensure their identifiability through File Roll sinks. In particular, the P-Log of each node is collected into s single file. An HDFS block typically has 3 replicas that are stored on different DataNodes for high availability [30]. When an HFile reading operation occurs, it is uncertain where the corresponding block file reading operations occur. With the above collection method, no matter which P-Log the required information is originally in, it can be sought from a single file.

B. LOG ANALYSIS

Log analysis is implemented by a set of log parsers. We divide the log analysis mode into two categories: analysis mode and search mode. The analysis parser analyzes each log record, generates provenance nodes and dependencies defined by BDPM ontology, and stores them into Neo4j database, whereas the search parser seeks required information from logs. Each parser is responsible for one analysis mode of a log, and an analysis parser can call search parsers through specific link points to obtain the required information.

C. PROVENANCE STORAGE

Neo4j is a high-performance, scalable, and distributed graph database [37], [38]. It models the objects and relationships between them as the nodes and edges of a graph, which makes it easy to represent highly connected data and semi-structured data. Neo4j is fully compliant with ACID (atomicity, consistency, isolation, and durability), supports efficient graph traversal, and has rich graph query functions. Thus, we store the generated provenance information into Neo4j.

VIII. EXPERIMENTS

In this section, we evaluate the time overhead incurred by log collection as well as the timeliness and correctness of the proposed log analysis methods.

A. ENVIRONMENT

The experiments were conducted on top of a small Hadoop cluster with 12 nodes and a log analysis server, which were built based on the VMware virtualization platform, where the cluster consists of a NameNode and 11 DataNodes.

Host machine configuration: Windows 10 Pro, Intel Core I5-8250U quad core CPU@1.6 GHz, 500 GB of hard drive, 32 GB of memory.

Virtual machine configuration: CentOS 6.5, 40 GB of hard driver. Each Hadoop node has 2 GB of memory. The log analysis server has 4 GB of memory.

Software releases: Apache Hadoop 2.10.1, Apache Flume 1.7.0, and Neo4j 3.5.15.

B. LOG COLLECTION TIME OVERHEAD EVALUATION

We use the MapReduce job, a typical big data processing mode, to evaluate the time overhead caused by log collection to Hadoop data processing.

Obviously, the log collection time overhead has a positive correlation with the amount of log data generated during Hadoop data processing. When the amount of data to be processed is fixed, the smaller the average file size, the more

TABLE 5. Log collection time overhead test results.

M (MB)	Job runtime without log collection (s)	Job runtime with log collection (s)	Time overhead
4000	119.63	119.56	-0.06%
4	113.89	113.76	-0.13%
2	114.33	114.40	0.06%
0.8	119.05	120.07	0.86%
0.4	125.24	128.58	2.76%
0.2	141.53	148.22	4.73%
0.08	184.29	198.69	7.81%

the number of files, and the larger the amount of log data. We used 7 datasets of 4000 MB, which contain 1, 1000, 2000, 5000, 10000, 20000, and 50000 files, respectively, to evaluate the relationship between the log collection time overhead and the average file size. To facilitate dataset generation and experimental result analysis, the datasets containing multiple files were generated by copying a seed file. The MapReduce jobs used to test the time overhead on large file processing and batch small file processing were Wordcount and MultiFileWordCount, respectively. The test results are shown in Table 5. M denotes the average file size.

Table 5 shows that when M is larger than 1 MB, the time overhead is minimal. The proposed method can be integrated into the big data processing system well. When M drops below 1 MB, log collection incurs obvious time overhead. When M is 0.08 MB, the time overhead is 7.81%.

The decrease in M, i.e., the increase in the number of files, increases the amount of data to be read/write for log collection as well as the amount of CPU and memory resources occupied by log collection, which is one of the reasons for the increase in the job runtime. However, even without log collection, the decrease in M itself still causes the increase in the job runtime because of the dispersion of data to be processed. Compared with when M is 4 MB, when M is 0.08 MB, the job runtime without log collection increases by 61.81%. Therefore, properly merging small files can reduce or even eliminate the time overhead caused by log collection and improve job execution efficiency. This is also applicable to simply running MapReduce jobs without log collection.

C. LOG ANALYSIS METHOD TIMELINESS EVALUATION

EditLog and HA-Log are the most important data sources for provenance generation, and they are both starting logs whose analysis tasks are heavier than other logs'. On the premise of considering actual situations, we set some conditions, which are conducive to the rapid generation of log records, to test the highest generation rate (r_c) of both logs and the average analysis rate (r_a) of the generated log data. In all the tests, if r_a is always higher than r_c, it indicates that the proposed method can achieve near real-time provenance generation.

1) EDITLOG TEST

In all the HDFSFile operations, only the analysis of HDFSFile creation starts with EditLog. If users only execute this operation, the generation rate of the records within the analysis scope is highest, whereas the analysis rate is lowest. If r_a is always higher than r_c under this condition, it indicates that the proposed method can meet the near real-time requirements on EditLog analysis. Thus, this test only executes HDFSFile creation.

For file creation, r_c is related to file size, creation manner, replication factor, and the number of users (n_u). Obviously, r_c is negatively correlated with file size. Because we found that the data files on most popular data sharing platforms (such as Wikipedia, Google, and Amazon) are generally larger than 1 KB, we constructed a dataset S, which contains 1 million files. The sizes of these files are all around 1 KB. The creation manner includes creating all the files one by one by executing 1 million creation commands and creating the whole directory at once by executing only 1 command. We select the latter, which has higher log generation rate. The replication factor is set to the default value of 3. Under the above conditions, we test the changes of r_c and r_a as n_u increases from 1 to 12. All the users perform operations in parallel on different nodes. Each n_u is tested 10 times. r_c takes the highest value of the test results, whereas r_a takes the average.

The test results are shown in Fig. 5. r_c rises with the increase in n_u at first, and then stabilizes. The peak r_c is 292.3 KB/s (About creating 108 files per second). r_a rises with the increase in the log data volume at first because that establishing the connection between the log parser and Neo4j incurs time overhead, but the proportion of this overhead in the total log analysis time gradually decreases as the log data volume increases. With the increase in n_u, the interleaving degree of the operation records generated due to different users increases, which increases the EditLog analysis complexity and the amount of data to read/write for BH-File update, and leads to the decline in r_a. But r_a stabilizes finally, and is always higher than r_c.

2) HA-LOG TEST

For HA-Log, operation type, execution mode, and execution manner are the main influencing factors of r_c and r_a. HDFSFile operations can be divided into instantaneous operations and non-instantaneous operations according to their runtime. The former contains various attribute getting/setting operations. Their log generation rate is high, but usage frequency is low. The latter refers to file read/write operations. Their log generation rate is related to the file size and is lower compared with that of instantaneous operations, whereas their usage frequency is higher. Thus, we test two cases where non-instantaneous operations are executed separately and both operations are executed simultaneously, but do not specifically test the case where instantaneous operations are executed separately. Here, the whole-directory reading and permission setting are used for the test.

Using dataset S, we first test r_c and r_a changes of file reading as n_u increases. When r_c peaks, we fix the number of users executing file reading, gradually increase the number

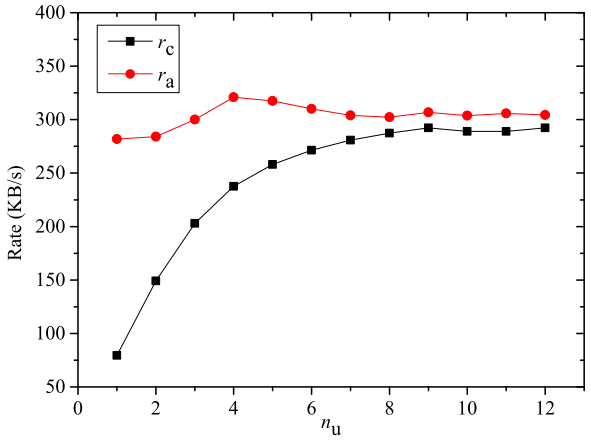


FIGURE 5. EditLog r_c and r_a test results.

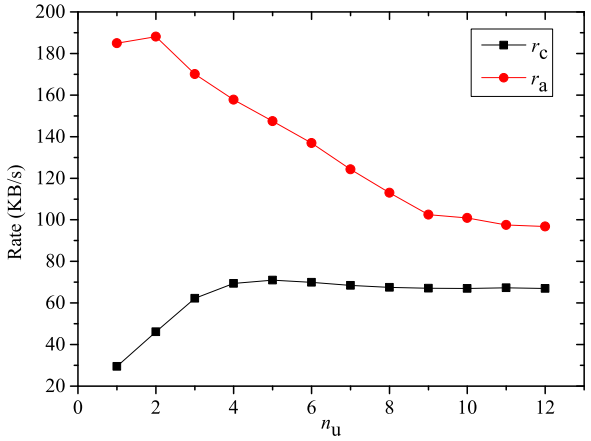


FIGURE 6. HA-Log r_c and r_a test results of file reading.

of users executing permission setting, and further test the changes of r_c and r_a.

The test results of file reading are shown in Fig. 6. When n_u was 5, r_c reached a peak of 71 KB/s (About generating 418 records per second). The r_a trajectory of HA-Log is similar to that of EditLog, but the change range is larger. Whole-directory reading is analyzed by the main and child threads cooperatively. The increase in n_u leads to the increase in the number of block file operation records in P-Log and the number of child threads, which further leads to the increase in the time taken by child threads to seek block file "SYS_OPEN" records and the cases that the main thread waits for child thread messages. Thus, r_a declines significantly. Finally, both r_c and r_a stabilize, and r_a is always higher than r_c.

In Fig. 7, rate 1 refers to the file reading test results when n_u is greater than 5. Rate 2 refers to the test results of two operations executed simultaneously. Compared with the former, the latter improves obviously. When n_u is 12, r_c is 111.53 KB/s (About generating 602 records per second). The growth rate of r_c is larger and it is still on the rise when n_u is 12. This is because that the segmented analysis method, which does not create child threads to analyze P-Log, is more

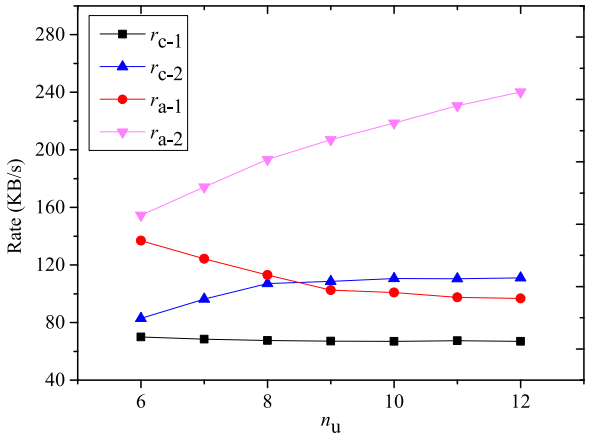


FIGURE 7. HA-Log r_c and r_a test results of two operations.

efficient than the parent-child thread collaborative analysis method, and the proportion of users who perform permission setting gradually increases.

D. LOG ANALYSIS METHOD CORRECTNESS EVALUATION

The proposed log analysis methods are deterministic pattern recognition methods established on the basis of Hadoop source code analysis and a large number of experimental tests. However, in the multi-user scenario, the correctness of these methods still faces uncertainties. Because the records generated due to the same operation may not be adjacent, determining whether some patterns occur depends on the time interval between related records. For some time thresholds in the log analysis methods, whether they are set too large or too small, they may lead to wrong determination, which are the main influencing factors of the method correctness.

Considering the record generation rules of various operations in EditLog and HA-Log, we test whether the following scenarios that use time thresholds as the basis for pattern recognition face the risk of wrong determination. For these scenarios, if the time threshold is set too large, the records generated by different operations may be determined as generated by a single operation. If the threshold is set too small, the records generated by a single operation may be determined as generated by different operations.

(1) EditLog: In the CLI mode, determining whether a "OP_MKDIR" record is the starting record of a whole-directory creation operation according to the time interval between the parent "OP_MKDIR" record and its descendant "OP_ADD" record.

(2) HA-Log: In the CLI mode, determining whether a "open" record is the starting record of a file copying operation according to the time interval between the "open" and "create" records with the same "ugi" value.

(3) HA-Log: In the CLI mode, determining whether a replication factor setting operation adopts the waiting mode according to the time interval between the "setReplication" and "open" records with the same "src" value.

TABLE 6. Time threshold test results.

Scenario	Maximum interval test operation	Maximum interval (ms)	Minimum interval test operations	Minimum interval (ms)
1	Whole-directory creation	1203	Directory creation and file creation	1552
2	File copying	625	File reading and creation	1669
3	File replication factor setting with waiting mode	695	File replication factor setting and reading	1507
4-1	Whole-directory permission setting	940	File listing and permission setting	1609
4-2	Whole-directory permission setting	45	File permission setting and listing	1529

(4) HA-Log: In the CLI mode, determining whether a whole-directory operation occurs according to the time interval between the ''listStatus'' record and its preceding or following record that has the same ''ugi'' value with it. The existence of these two cases relates to the record generation rules of HA-Log. Taking the whole-directory permission setting as an example, if the permission parameter is consistent with the current permission of the directory (assuming d_1), only a ''listStatus'' record with ''src'' value as d_1 is generated in HA-Log. Otherwise, a ''setPermission + listStatus'' record combination is generated.

By executing Hadoop shell commands via a bash script, we first test the maximum time interval between the above-mentioned records when they are generated due to a single operation under the condition that 12 users perform operations in parallel on different nodes (near full-load state). Then, we test the minimum time interval between them when they are successively generated due to different operations under the single-user condition(near no-load state). If the former is smaller than the latter, the corresponding scenario has a safe range for the time threshold setting. Otherwise, it faces the risk of wrong determination.

The test results are shown in Table 6. Scenario 4 is divided into 2 sub-scenarios. Scenario 4-1 and 4-2 indicate that the ''listStatus'' record precedes and follows the record that has the same ''ugi'' value with it. Table 6 shows that the minimum time intervals are between 1.5s and 2s, whereas the maximum time intervals are all below 1.5s. Thus, every scenario has a safe threshold range. By setting the time thresholds in these ranges, the correctness of the log analysis methods reaches 100%, which provides a strong data foundation for provenance-based data security supervision.

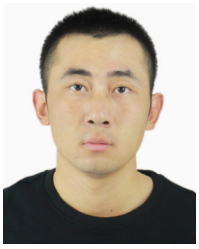
IX. CONCLUSION

The establishment of data provenance can lay a solid foundation for data security supervision. To address the problems of generating big data provenance in near real-time based on multi-log analysis, this paper proposes a parallel multi-log analysis method based on auxiliary data structures and multi-threading, which supports the efficient analysis of non-deterministic records, non-instantaneous operations and instantaneous batch operations. The experimental results show that the proposed method can correctly generate provenance information in near real-time and brings little time overhead to large file processing. In the future, we will further expand provenance tracking beyond HDFS to improve the effectiveness of provenance on data supervision.

REFERENCES

[1] D. Singh, ''Towards data privacy and security framework in big data governance,'' *Int. J. Softw. Eng. Comput. Syst.*, vol. 6, no. 1, pp. 41–51, May 2020.
[2] H. Tao, M. Z. A. Bhuiyan, M. A. Rahman, G. Wang, T. Wang, M. M. Ahmed, and J. Li, ''Economic perspective analysis of protecting big data security and privacy,'' *Future Gener. Comput. Syst.*, vol. 98, pp. 660–671, Sep. 2019.
[3] O. I. Abiodun, M. Alawida, A. E. Omolara, and A. Alabdulatif, ''Data provenance for cloud forensic investigations, security, challenges, solutions and future perspectives: A survey,'' *J. King Saud Univ. Comput. Inf. Sci.*, vol. 34, no. 10, pp. 10217–10245, Nov. 2022.
[4] B. Pan, N. Stakhanova, and S. Ray, ''Data provenance in security and privacy,'' *ACM Comput. Surv.*, vol. 55, no. 14, pp. 1–36, Apr. 2023, doi: 10.1145/3593294.
[5] T. R. Rao, P. Mitra, R. Bhatt, and A. Goswami, ''The big data system, components, tools, and technologies: A survey,'' *Knowl. Inf. Syst.*, vol. 60, no. 3, pp. 1165–1245, Sep. 2019.
[6] I. M. Abbadi and J. Lyle, ''Challenges for provenance in cloud computing,'' in *Proc. TAPP*, Heraklion, Crete, Greece, 2011, pp. 1–6.
[7] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, ''OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis,'' in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2020, pp. 1–16.
[8] B. Pérez, J. Rubio, and C. Sáenz-Adán, ''A systematic review of provenance systems,'' *Knowl. Inf. Syst.*, vol. 57, no. 3, pp. 495–543, Feb. 2018.
[9] Z. Li, Q. A. Chen, R. Yang, Y. Chen, and W. Ruan, ''Threat detection and investigation with system-level provenance graphs: A survey,'' *Comput. Secur.*, vol. 106, pp. 1–16, Apr. 2021.
[10] M. Zipperle, F. Gottwalt, E. Chang, and T. Dillon, ''Provenance-based intrusion detection systems: A survey,'' *ACM Comput. Surv.*, vol. 55, no. 7, pp. 1–36, Dec. 2022.
[11] A. Bates, W. U. Hassan, K. Butler, A. Dobra, B. Reaves, P. Cable, T. Moyer, and N. Schear, ''Transparent web service auditing via network provenance functions,'' in *Proc. 26th Int. Conf. World Wide Web*, Apr. 2017, pp. 887–895.
[12] O. Q. Zhang, R. K. L. Ko, M. Kirchberg, C. H. Suen, P. Jagadpramana, and B. S. Lee, ''How to track your data: Rule-based data provenance tracing algorithms,'' in *Proc. IEEE 11th Int. Conf. Trust, Security Privacy Comput. Commun.*, Liverpool, U.K., Jun. 2012, pp. 1429–1437.
[13] M. Abediniala and B. Roy, ''Facilitating asynchronous collaboration in scientific workflow composition using provenance,'' in *Proc. ACM Human-Comput. Interact.*, vol. 6, pp. 1–26, Jun. 2022.
[14] J. Teon, M. A. Gulzar, and M. Kim, ''Influence-based provenance for dataflow applications with taint propagation,'' in *Proc. SoCC*, New York, NY, USA, 2020, pp. 372–386.
[15] D. Ghoshal and B. Plale, ''Provenance from log files: A BigData problem,'' in *Proc. Joint EDBT/ICDT Workshops*, Genoa, Italy, Mar. 2013, pp. 290–297.
[16] F. Psallidas, A. Agrawal, C. Sugunan, K. Ibrahim, K. Karanasos, J. Camacho-Rodríguez, A. Floratou, C. Curino, and R. Ramakrishnan, ''OneProvenance: Efficient extraction of dynamic coarse-grained provenance from database logs,'' 2023, *arXiv:2210.14047*.
[17] K. K. Muniswamy-Reddy, U. J. Braun, D. A. Holland, P. Macko, D. Maclean, D. W. Margo, M I. Seltzer, and R. Smogor, ''Layering in provenance systems,'' in *Proc. ATC*, San Diego, CA, USA, 2009, pp. 1–14.

[18] L. Yu, S. Ma, Z. Zhang, G. Tao, X. Zhang, D. Xu, V. E. Urias, H. W. Lin, G. Ciocarlie, V. Yegneswaran, and A. Gehani, "ALchemist: Fusing application and audit logs for precise attack provenance without instrumentation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.
[19] L. Rupprecht, J. C. Davis, C. Arnold, Y. Gur, and D. Bhagwat, "Improving reproducibility of data science pipelines through transparent provenance capture," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3354–3368, Aug. 2020.
[20] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proc. Symp. Cloud Comput.*, Santa Clara, CA, USA, Sep. 2017, pp. 405–418.
[21] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "ALASTOR: Reconstructing the provenance of serverless intrusions," in *Proc. USENIX*, Boston, MA, USA, 2022, pp. 2443–2460.
[22] T. Li, X. Liu, W. Qiao, X. Zhu, Y. Shen, and J. Ma, "T-trace: Constructing the APTs provenance graphs through multiple syslogs correlation," *IEEE Trans. Dependable Secure Comput.*, early access, May 8, 2023, doi: 10.1109/TDSC.2023.3273918.
[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. MSST*, Incline Village, NV, USA, 2010, pp. 1–10.
[24] Y. Gao, X. Chen, and X. Du, "A big data provenance model for data security supervision based on PROV-DM model," *IEEE Access*, vol. 8, pp. 38742–38752, 2020.
[25] Y. Simmhan, P. Groth, and L. Moreau, "Special section: The third provenance challenge on using the open provenance model for interoperability," *Future Gener. Comput. Syst.*, vol. 27, no. 6, pp. 737–742, Jun. 2011.
[26] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche, "The open provenance model core specification (v1.1)," *Future Gener. Comput. Syst.*, vol. 27, no. 6, pp. 743–756, Jun. 2011.
[27] L. Moreau, P. Groth, J. Cheney, T. Lebo, and S. Miles, "The rationale of PROV," *J. Web Semantics*, vol. 35, pp. 235–257, Dec. 2015.
[28] W3C. (Apr. 30, 2013) *PROV-DM: The PROV Data Model*. [Online]. Available: https://www.w3.org/TR/2013/REC-prov-dm-20130430/
[29] Gewuzhao. (Feb. 3, 2021) *Big-Data-Provenance-Ontology*. [Online]. Available: https://github.com/gewuzhao/big-data-provenance-ontology
[30] T. White, "The Hadoop distributed filesystem," in *Hadoop: The Definitive Guide*, 4th ed. Sebastopol, CA, USA: O'Reilly, 2015, pp. 53–56.
[31] R. K. L. Ko and M. A. Will, "Progger: An efficient, tamper-evident kernel-space logger for cloud data provenance tracking," in *Proc. IEEE 7th Int. Conf. Cloud Comput.*, Anchorage, AK, USA, Jun. 2014, pp. 881–889.
[32] T. Corrick, "Progger 3: A low-overhead, tamper-proof provenance system," M.S. thesis, Dept. Comput. Sci., Waikaato Univ., Hamilton, New Zealand, 2021.
[33] Y. Gao, X. Fu, B. Luo, X. Du, and M. Guizani, "Haddle: A framework for investigating data leakage attacks in Hadoop," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, San Diego, CA, USA, Dec. 2015, pp. 1–6.
[34] HANDWIKI. (Jun. 27, 2023) *File Descriptor*. [Online]. Available: https://handwiki.org/wiki/File_descriptor
[35] APACHE. (Jun. 18, 2023). *HDFS Architecture*. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#NameNode_and_DataNodes
[36] APACHE. (Oct. 24, 2022). *Apache Flume*. [Online]. Available: https://flume.apache.org/
[37] J. J. Miller, "Graph database applications and concepts with Neo4j," in *Proc. SAIS*, Atlanta, GA, USA, 2013, pp. 141–147.
[38] F. Holzschuher and R. Peinl, "Performance of graph query languages: Comparison of cypher, Gremlin and native access in Neo4j," in *Proc. Joint EDBT/ICDT Workshops*, Genoa, Italy, Mar. 2013, pp. 195–204.



YUANZHAO GAO received the M.S. and Ph.D. degrees in computer science and technology from the Zhengzhou Science and Technology Institute, Zhengzhou, China, in 2017 and 2021, respectively. His research interest includes big data security.

XINGYUAN CHEN received the Ph.D. degree in communication and information systems from the Zhengzhou Science and Technology Institute, Zhengzhou, China, in 2003. He is currently a Professor with the Zhengzhou Science and Technology Institute. He is also a Ph.D. Supervisor with the School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China. His research interests include network and information security, cloud computing security, and OS security.

BINGLONG LI received the Ph.D. degree in computer software and theory from the Zhengzhou Science and Technology Institute, Zhengzhou, China, in 2007. He is currently an Associate Professor with the Zhengzhou Science and Technology Institute. He has been funded under the National Natural Science Foundation of China (NSFC). His research interests include digital forensics and information security.

XUEHUI DU received the Ph.D. degree in computer science and technology from the Zhengzhou Science and Technology Institute, Zhengzhou, China, in 2011. She is currently a Professor with the Zhengzhou Science and Technology Institute. Her research interests include cloud computing, big data security, cyberspace security, and access control.