Blockchain-driven anomaly detection framework on edge intelligence

Xueshuo Xie, Yaozheng Fang, Zhaolong Jian, Ye Lu, Tao Li & Guiling Wang

CCF Transactions on Networking

ISSN 2520-8462

CCF Trans. Netw.
DOI 10.1007/s42045-020-00044-9

ONLINE FIRST

Springer

CCF Transactions on Networking
https://doi.org/10.1007/s42045-020-00044-9

REGULAR PAPER

Blockchain-driven anomaly detection framework on edge intelligence

Xueshuo Xie[1] · Yaozheng Fang[2] · Zhaolong Jian[1] · Ye Lu[1] · Tao Li[2] · Guiling Wang[3]

Received: 15 June 2020 / Accepted: 29 October 2020
© China Computer Federation (CCF) 2020

Abstract
There are a large number of end devices in an IoT system, which may malfunction due to various reasons, such as being attacked. Anomaly detection of the devices and the whole IoT system normally rely on the analysis of the huge amount of log records generated by the end devices. How to protect the log records from being tampered with and realize the real-time anomaly detection is a challenging task which is still not addressed. Existing works on anomaly detection by the emerging and effective deep learning algorithms require the transfer of log data to cloud servers which incurs high communication overhead and long detection latency, and is subject to the risk of being tampered. In this paper, we propose a novel and efficient hierarchical framework for online anomaly detection in IoT systems atop Blockchain and smart contracts. At the device layer of the hierarchical framework, an efficient feature extractor is developed to preprocess the raw log data which greatly reduces the size of data to be transferred while keeps sufficient information for the anomaly detection model to use. At the cloud layer of the framework, deep learning models use the processed data from the device layer to build the detection model and output normal workflow patterns. In the edge layer of the framework, a permissioned blockchain is built and a series of smart contracts are developed which can guarantee data integrity and achieve automatic anomaly detection based on the model output from the cloud layer. Extensive experiments demonstrate that our framework can reduce the ledger size by 7.1% without detection accuracy reduction compared with traditional centralized solutions and the detection latency is only 0.47ms in our prototype. Our feature extractor can speed up by 3.6x–7.3x times on the execution time with almost the same CPU usage rate compared with state-of-the-art log parsers and encryption solutions, such as AES and RSA.

Keywords Anomaly detection · Feature extractor · Smart contract · On-chain/off-chain

✉ Ye Lu
luye@nankai.edu.cn

✉ Tao Li
litao@nankai.edu.cn

Xueshuo Xie
xueshuoxie@mail.nankai.edu.cn

Yaozheng Fang
fyz@mail.nankai.edu.cn

Zhaolong Jian
jianzhaolong@mail.nankai.edu.cn

Guiling Wang
gwang@njit.edu

[1] College of Computer Science, Nankai University, Tianjin 300350, China

[2] Tianjin Key Laboratory of Network and Data Security Technology, Tianjin 300350, China

[3] New Jersey Institute of Technology, Newark, NJ 07102, USA

1 Introduction

Log analysis is an important tool for anomaly detection in Internet of Things (IoT) systems. In a hierarchical IoT system, a large number of heterogeneous devices may generate a huge volume of log records of operations and activities chronologically. When any anomaly happens, such as performance degradation or device failure, log records can be analyzed for troubleshooting and tracing source of malfunctions. System administrators rely on log records for quick diagnosis to locate errors and exceptions (Xu et al. 2019). Thus, malicious attackers or dishonest participants are incentivized to tamper with log records by adding, modifying, or deleting them Pourmajidi (2018) to avoid being detected, and there are frequent incidents of log tampering in the real world. In addition to raw log records, the workflows extracted from the log files are also a target of attack. Many existing anomaly detection systems (He et al. 2016) train a classifier from the unstructured logs with specific

Published online: 24 November 2020

labels, such as normal or abnormal, or mine unknown types of anomalies (Lou et al. 2010). In these systems, normal workflow are normally mined from log records and then are used to identify execution anomalies in program logic flows (Liu et al. 2019; Xiao et al. 2016). The summarized workflows are naturally an attack and tampering target of anniversaries. In this paper, we aim to protect the mined workflows in addition to the log records by building a log storage and analysis system with immutability, tamper-proof, and traceability for log analysis.

To leverage the intrinsic immutability of blockchain, multiple existing works build log storage and analysis systems atop blockchain (Huang 2019; Pourmajidi and Miranskyy 2018; Pourmajidi et al. 2019). Existing approaches that leverage blockchain for log storage can be broadly classified into three categories: (1) raw logs are directly stored on chain which incurs high storage overhead; (2) the ciphertext are stored on chain which provides data privacy and security but does not support spontaneous online data analysis (Pourmajidi and Miranskyy 2018); and (3) the hash value of log files are stored on chain for storage efficiency (Huang 2019). Even though the above works can achieve tamper resistance in many scenarios, none of the works support efficient automatic online log analysis, especially the analysis for anomaly detection based on the emerging deep learning methodologies (Du et al. 2017), which are in fact in a greater need in the current anomaly detection systems.

Correspondingly, in this paper, we aim to build a tamper-resistance log-based hierarchical anomaly detection framework, named HADS, for IoT systems atop blockchain and smart contract technologies. HADS is designed to support efficient automatic online log analysis using deep neural network (DNN) technologies as well as other emerging machine learning and data mining technologies. Our design has three objectives: (1) to reduce on-chain storage overhead while support the potential heavy deep neural network training, we need to smartly decide how to process the raw log records, which (processed) data is stored on-chain and which data is off-chain, and the data structure to be used; (2) to provide automatic online anomaly detection and reduce detection latency, we need to delicately design the smart contracts to employ; (3) to accommodate the low processing capability of edge devices while achieve spontaneous detection, we need to architect the system components effectively.

There are multiple challenges to achieve aforementioned objectives. Firstly, an effective feature extractor needs to be designed that can run on low power devices to extract the features for DNN training. Note that end devices generate a huge amount of log files. It is not efficient to either transfer all of them to the cloud server for DNN training nor store and process them on-chain. Motivated by Osia et al. (2018), a deep feature extractor will be delicately designed to remove redundant data while keep sufficient information for DNN traning. The second challenge lies in the construction of an auto-update mechanism implemented by smart contract. Note that in an IoT system, new log records are continuously generated and new workflows are frequently identified. The new workflows always require micro-updates by smart contracts, but the pre-deployed smart contracts with limited flexibility has difficulty in dealing with frequent workflow changes and updates Shao et al. (2020). The last challenge in actual deployment is that anomaly detection has to be timely so that administrators can intervene in an ongoing attack or a system performance issue Du and Li (2017); otherwise, it is not useful. If we still adopt centralized solutions, e.g., cloud computing, log data will be collected in devices and transferred to the central server in the service layer for DNN training and inference, and finally the detected results are sent back to devices. Such a procedure may have too long a latency for devices to react to errors or anomalies in time.

To address the above challenges, our HADS framework employs a hierarchical anomaly detection model, which can reduce ledger size and detection latency while keep the same detection accuracy. HADS pushes data preprocessing and workflow detection from the centralized cloud to the distributed network edge and devices of the IoT system, which can process logs and detect anomalies in real-time. We first design an efficient feature extractor, named FLE, that can reserve the feature information of log entries for real-time extraction on the devices. FLE extracts features from the unstructured raw log, realizes one-to-one mapping between raw log entries and features, and achieves the separation of on-chain and off-chain storage for features and large volumes of logs. That is, the raw log entries will be stored off-chain, which the extracted features will be stored on-chain. In the edge layer of the IoT system, a blockchain is built and HADS incorporates a smart contract-based dynamic management mechanism for logs management, storage, and analysis. It can transfer, commit and store features into the ledger and dynamically convert workflow with micro-updates into the smart contracts. In the service layer of the IoT system, any DNN or other machine learning and data mining models can be flexibly implemented and generate workflows from the features received. A series of thorough experiments have proved that HADS can reduce the ledger size and detection latency while keep the same detection accuracy compared with the traditional centralized solutions. The CPU usage and execution time of FLE can speed up to 3.6x–7.3x times compared with state-of-the-art log parsers and encryption solutions, such as AES and RSA. In summary, this paper makes the following contributions:

- A hierarchical blockchain-based anomaly detection model on edge intelligence is designed which consists of a feature extractor running on low-power devices, an edge blockchain with on-chain/off-chain storage

and smart contract-based management, and a detection (DNN) model running on the service layer to mine workflow from the features stored in the edge ledger.
- A feature extractor is developed to use heuristic rules to convert the unstructured raw log into structured features and establish a one-way mapping between the raw log entries and features. Only the features are stored on-chain to reduce data size while satisfy the training requirements. Feature and raw data mapping are stored in an off-chain server for tampering detection.
- A series of smart contracts are developed to manage log data storage, automatically update the detection mechanism based on the dynamically generated workflow by the DNN model, and conduct anomaly detection with low latency.

The remainder of this paper is organized as follows: Sect. 2 introduces the background including log analysis, blockchain, and edge intelligence. Section 3 presents the HADS design in detail. The implementation of on-chain/off-chain storage is introduced in Sect. 4 and smart contract implementation in Sect. 5. We report the system prototype and performance evaluation in Sect. 6. A review of related work is given in Sect. 7. Finally, Sect. 8 concludes the paper.

2 Technical background

In this section, we introduce the background of the main technologies HADS employs, including the log-based anomaly detection, blockchain and smart contract, and edge intelligence.

2.1 Anomaly detection based on log analysis

Anomaly detection aims at uncovering abnormal behaviors from large-scale unstructured log data in a timely manner through training classifiers or mining workflow. Existing works mainly follow four steps: log collection, log parsing, feature extraction, and anomaly detection (He et al. 2016). The collected unstructured raw log entry always records a specific system event with a set of information fields: timestamp, alarm level, and raw message content. The logs normally have two parts: the constant part records the event type of the log message and the variable part carries the runtime information of interest (Zhu et al. 2019). Regarding the second step, log parser is essential for anomaly detection and it converts the raw logs into structured log sequences or events. Currently, there are three main methods to design accurate and efficient log parsers: (1) frequent pattern mining, such as SLCT Vaarandi (2003), LFA Nagappan and Vouk (2010), and LogCluster Vaarandi and Pihelgas (2015; 2) clustering, such as LKE Fu et al. (2009), LogSig Tang et al. (2011), LogMine Hamooni et al. (2016), SHISO Mizutani (2013), and LenMa Shima (2016; 3) heuristic rules, such as AEL Jiang et al. (2008), IPLoM Makanju et al. (2012), Spell Min and Li (2017), Drain He et al. (2017), POP He et al. (2018), and MoLFI Messaoudi et al. (2018). All of these log parsers extract the log event by automatically separating the constant part and variable part, and further transform each log entry into a specific event (Zhu et al. 2019). Considering the length and token position of log entries, an efficient log parsing based on heuristic rules is proved to have better performance (Zhang et al. 2019). Inspired from the previous research, in this paper, we design a lightweight and automatic feature extractor to generate the structured log sequences and only upload them to on-chain storage instead of raw logs. We also choose nine state-of-the-art of these log parsers as the baselines to compare the CPU usage and execute time with our feature extractor.

Regarding the fourth step, anomaly detection, multiple models are proposed. To mine the workflow from log data, some data mining methods, such as building an automaton for the workflow of each management task based on normal executions, check log messages against a set of automata for workflow divergences in a streaming manner (Xiao et al. 2016). A data-driven approach on long short-term memory (LSTM) has also been proposed in Du et al. (2017) to mine workflows from a large volume of system logs for anomaly detection. The mined workflows are then used to identify execution anomalies in program logic flows and output the necessary context information for further diagnosis. We can divide the log-based anomaly detection models into supervised learning and unsupervised learning. However, most of the models, such as support vector machine (SVM), Principal component analysis (PCA), and the aforementioned LSTM DNN, can not be trained on or inference directly from the raw unstructured text. The direct input of the models is normally numerical results, such as feature matrix. The matrix can be extracted from the log sequences by different windows technologies to count the occurrence number of each event, based on the results of aforementioned step three and four.

Since logs are used as the digital evidence, the storage of logs should be immutable, traceable, and tamper-proof. A blockchain-based log system can prevent log from being tampered by sealing logs cryptographically in a hierarchical ledger, as presented in Pourmajidi and Miranskyy (2018). Another two similar works provide immutable log storage as a service (Pourmajidi et al. 2019; Rane and Dixit 2019). To avoid the high cost of storing big files in a blockchain, Huang (2019) utilized the InterPlanetary File System (IPFS) to store log files, used Ethereum to store the hash of log files, and deployed a smart contract to create an index for log files.

2.2 Blockchain and smart contract

Blockchain is a decentralization platform to maintain an ordered list of blocks, named ledger, which "chained" back to the previous block through containing a hash of the previous block. Every block contains a list of transactions organized as a tree structure, such as Merkle tree in Bitcoin. The transactions store information, such as the sender, receiver, and data, or the (compiled) code of smart contracts, or parameters of function calls of smart contracts. A node in blockchain will send a signed transaction using its private key to the network and validated and propagated by other nodes. Miners are responsible for aggregating valid new transactions into blocks through mining, adding the blocks to the ledger, and propagating the blocks to the network. To govern the addition of new blocks, a consensus mechanism, such as Proof-of-Work (PoW) or Proof-of-Stake (PoS), is used for validating and broadcasting transactions and blocks, resolving conflicts, and achieving the incentive scheme. Due to the security properties of the hash function, computational constraints, and consensus protocols, the data contained in a committed transaction and the historical transactions are seen as immutable in practice. However, every participant can join the network to access all the information on blockchain and validate new transactions because there is no privileged user. Besides, the size of the data on blockchain and the transaction processing rate has limited the scalability Xu et al. (2018).

Smart contract is a program that can be deployed and run on a blockchain to enable wider use of blockchain in scenarios. We can use it to express triggers, conditions, and business logic that will enable more complex programmable transactions, such as the signature of the transaction initiator authorizes the data payload of a transaction or the creation or execution of a smart contract. Obviously, a smart contract should always be deterministic as it is the enforcement force as a judge for the whole network. Therefore, a smart contract should keep integrity and prevent introducing ambiguity, bugs, or vulnerabilities. Even though some cryptography assures its strong integrity and determinacy throughout its life cycle, we still need to delicately design when it is generated and thoroughly tested for security before it is deployed in a blockchain and its integrity should be maintained during execution.

Ethereum is currently widely used as a public and private blockchain that supports Turing-complete language, such as Solidity, compile smart contracts to specify transaction processing. To deploy a smart contract that possesses code pieces, an account balance, and a key-value memory in Ethereum is similarly as a transaction confirmation. Any deployed smart contracts are identified by an address and can be triggered by transactions or contract messages to run autonomously in the blockchain. Ethereum uses gas to limit the number and complexity of transactions that can fit into a block, or prevent infinite contract execution as DoS attacks. An account must enclose sufficient amounts of gas to successfully commit transactions or call a contract function. The latency between a transaction submitting and acknowledgment in Ethereum is around 3 min since there is a 14-s block interval with 11 confirmation blocks, compared with around 1 h on Bitcoin. In this paper, we implement a prototype of HADS on the private Ethereum, so the detailed gas consumption is not our main consideration. However, we still optimize our implementation to minimize costly operations and avoid vulnerabilities using security analyzers before deployed it on the actual network Tsankov et al. (2018).

2.3 Edge intelligence

The integration of artificial intelligence (AI) and edge computing results in new interdisciplinary edge intelligence. To fully unleash the potential of the large volumes of data at the edge, there is an urgent need to push the computing tasks and services of AI from the network core to edge (Ren et al. 2019; Zhou et al. 2019). Currently, the cloud-only approach requires the transfer of significant amounts of data from devices to the cloud, which is inefficient and dispensable. After data transfer, all the processing steps, such as preprocessing, feature extraction, training, and inference, are conducted in a centralized cloud-based layer. This leads to high communication and computation cost, considering the limitation of device energy, communication bandwidth, and server storage (Tang et al. 2019; Thomas et al. 2019). As the computational resources in devices become more powerful and energy-efficient, we can in fact move many computations out of the cloud and onto a hierarchy of IoT devices, such as features extractor and local inferences. Consequently, we need only transmit some processed digest of data and inference outcomes, instead of all the raw data, from end devices to cloud servers. This can significantly reduce data transfer, energy consumption and network traffic. However, the conventional training models do not account for the fact that the smart devices in the system have already performed a local inference. Also, another primary concern is that the prediction accuracy of models will be influenced or not. Inspired by the development of edge intelligence Zhou et al. (2019), there are interesting solutions on edge computing that can offload some computational and storage capability from the cloud to the network edge or devices to reduce the communication and computation latency and improve the quality of services (Lyu et al. 2020; Yin et al. 2018). One possible solution is a fine-grained, layer-level computation partitioning strategy, based on the data and computation variations of each layer within a DNN. Because the layer characteristics of DNN algorithms and hierarchical structure of IoT are similar so that we can integrate them together. The DNN partitioning has significant latency reduction and energy advantages over the status quo approach Kang et al. (2017). Another solution is to design a deep feature extractor that can run on low-power devices to extract approved information using for the model, and achieve high accuracy for primary tasks while removing the redundant data and preserving the sensitive information Osia et al. (2018, 2020).

3 The system architecture of HADS

HADS employs a hierarchical architecture. In this section, we present the details of HADS. Specifically, in HADS, a feature extractor first preprocesses raw logs on devices and generate features of the logs. A feature matrix is built accordingly and dynamically adjusted, which is used to mine legal workflows in real time. Using legal workflows as templates, smart contracts are auto-updated to conduct anomaly detection.

3.1 Hierarchical architecture

As shown in Fig. 1, HADS employs a multi-layer structure for hierarchical IoT systems which consists of device layer, edge layer, service layer, and application layer.

- **Device Layer:** End devices generate a large amount of raw logs, whose online preprocessing can save storage and reduce communication overhead. At the device layer, we deploy a feature extractor based on regular expression, heuristic rules, and one-way function to convert the unstructured raw log entries into structure features, such as the hash value of each log entry. This feature extractor's log preprocessing can reserve the features of each log entry for DNN training in the service layer. Only the extracted features with reduced size will be uploaded to the edge blockchain for immutable storage. , and will be used as input for DNN training at the service layer. Finally, these preprocessed features which remains important log information are used to train the DNN model at the service layer.
- **Edge Layer:** At the edge layer, HADS runs a permissioned blockchain on the decentralized edge server, such as private Ethereum and Hyperledger, where legal users are granted identity credentials for user authorization upon blockchain deployment. The edge blockchain guarantees that all the data assets, such as features, workflows, and smart contracts will not be tampered with. After being processed on the edge devices, the features will be stored into the edge ledger for tamper-proof storage and follow-up training. A mapping between raw logs and structured features is stored into an off-chain file system, such as IPFS. At this layer, HADS employs a series of smart contracts, such as register contract, device snap-

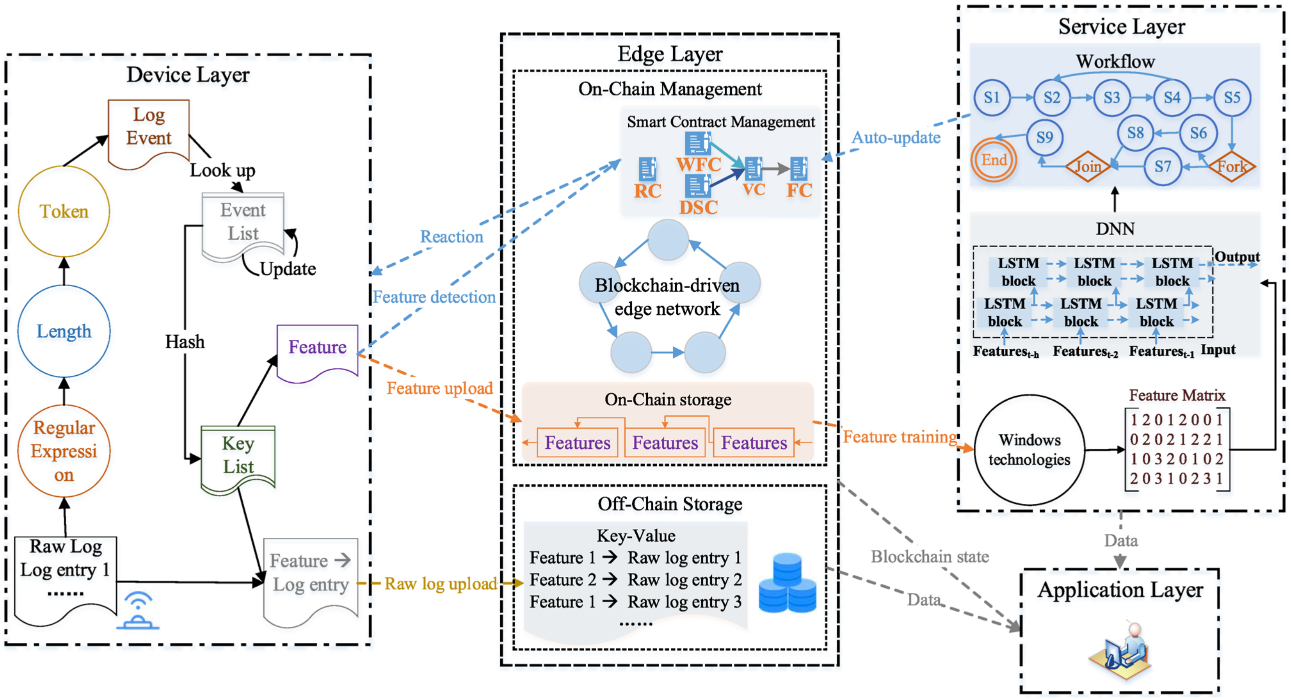


Fig. 1 The HADS overview

shot contract, workflow contract, feedback contract, and verifying contract to complete workflow-based anomaly detection and achieve collaborative defense in the entire network.

- **Service Layer:** A data-driven DNN model is deployed to mine legal workflows as follow-up detection template or train a classifier for anomaly detection. The training data are the preprocessed features stored in the ledger. Different window sizes are used to count the occurrence of each feature and further form a feature matrix, which is used by the DNN model as the input. (Note that in practice, we can replace the DNN model with any other log-based anomaly detection models.) Once the DNN produces new output workflows, a series of smart contracts will dynamically convert the workflows into smart contracts which are used to detect the anomalies in real-time and report to the security policies.
- **Application Layer:** A variety of smart applications are designed to visualize the state of the entire network and devices.

3.2 Feature extractor

As shown in Fig. 2, our online feature extractor, named FLE, consists of two components: (1) a parser that first converts the raw log entries into structured log events and (2) a Hash-map that generates a hash value for each structured log event. Note that the hash values are the extracted features of the raw log entries.

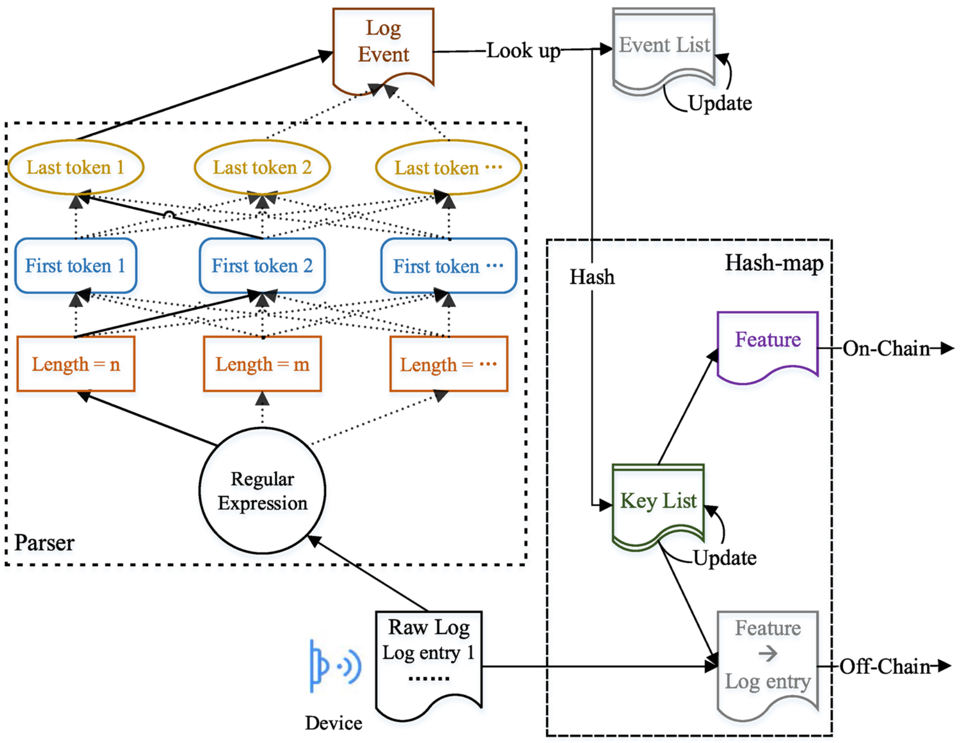
Parser. The raw log entries are usually unstructured and can not be directly used for deep learning model training. Here is an example of raw log sample: “L_i *(Invalid user webmaster from 10.251.42.*, check pass; user unknown, Connection closed by 10.251.42.*)*. Given a raw log sequence $L_1, L_2, L_3, \dots, L_m$, we use the parser to map it into a log event sequence $E_1, E_2, E_3, \dots, E_m$, where m is the total number of log events. The relationship can be formulated as follows:

$$E_i = Parser(L_i), \text{ for } i \in (1, 2, \dots, m). \qquad (1)$$

The previous works on log parsers Zhang et al. (2019) shows that the natural characteristics of logs follow several lightweight heuristic rules: (1) The parameters, such as IP and port number, should be eliminated as the variable parts; (2) The log entries with the same length are likely to associate with the same event; (3) The same tokens in the same positions of different log entries are likely to associate with the same events. In practice, the parsers under these heuristic rules are usually lightweight so that they can run on low-power devices.

In the parser step, we apply designed regular expressions (RE) to the log entries. Each log entry can be described with length, first token and last token according to the above heuristic rules. Specifically, based on the domain knowledge, administrators setup the regular expressions (RE) for some commonly-used variables, such as IP address, port number, etc. When a token from the raw log entry is matched to regular expressions, we replace

Fig. 2 The main steps of feature extractor that can reserve the feature of raw logs



the token with an asterisk. Then, the processed log entry is classified to different groups with the same length, the same first token and the same last token. For example, "*Invalid user webmaster from* *" and "*Connection closed by* *" are two log entries after applying regular expressions. The length of the first log entry and the second log entry are 5 and 4. The first tokens of them are "*Invalid*" and "*Connection*" respectively. Obviously, they belong to different groups because of different lengths and different first tokens. Finally, the new log entry will match an existing log event in the event list. If not match, it will generate a new event and update the event list. This helps the administrators to dynamically maintain the event list.

Furthermore, each raw log entry is classified into the corresponding log event based on the structure. For example, "*Invalid user webmaster from 10.251.42.**" and "*Invalid user webmaster from 17.25.44.**" are two different log entries. However, they belong to the same log event "*Invalid user webmaster from* *". To prepare for the input data for anomaly detection model training, our parser processes the raw log data by removing noises and extracting the structured event information.

Hash-map. From the above step, the raw log will convert to a log event. Such as, "*Invalid user webmaster from 10.251.42.**" convert to "*Invalid user webmaster from* *". In the subsequent use of windows technology to generate the feature matrix, the log event is still so textual that difficult to count the occurrence number of log events. Furthermore, the textual log event contains much useful information and leakage to the malicious attackers that can not store in the public ledger. In the hash-map step, the main purpose is to prevent malicious attackers and semi-honest participants from obtaining sensitive information as well as mining secondary usage information from the event content. One challenge is to minimize the data size without affecting the final performance of the anomaly detection model. Therefore, we establish a one-way mapping (or hash-mapping) from log entries to hash values so that no one can discover sensitive information from the hash value. We can formulate the mapping as follows:

$$F_i = Hash(E_i), \quad \text{for } i \in (1, 2, \ldots, m), \tag{2}$$

where F_i is the corresponding hash value of log event E_i, m is the total number of log events and *Hash* is one of the one-way functions called Hash function. Hash value representation has some advantages. First, it is content independent, which means the malicious attackers or semi-honest participants are not able to infer the content of log event from it because of the security mechanism. Second, it largely reduces the data size because raw log data will not be stored to the edge blockchain ledger.

In Hash-map step, given a new log event, we first check if it already exists in the event list which is a set of log events. If the log event already exists, we retrieve the corresponding hash value from the key list which is a set of hash values mapping from the log events. Otherwise, we insert this new log event to the event list, calculate the hash value with hash function and add the hash value to the key list. Finally, each raw log entry is represented to a hash value as the feature and is uploaded to the on-chain storage. Meanwhile, we also reserve the hash mapping relationships in the off-chain storage so that these fine-grained mappings can be utilized to identify the tampering behavior or be applied to the smart applications.

3.3 Feature matrix generation and detection model training

From raw logs, FLE extracts features in the format of strings. However, contemporary log-based anomaly detection models normally use numeric features as inputs. Thus, we create a numerical feature matrix based on the following procedure: We use windows technologies, such as fixed windows, sliding windows or session windows, to count the occurrence of specific log events. The window size ranges from 1 to n. There are m features in total. The feature numerical matrix M can be computed as:

$$M_{ij} = Count(F_i), \quad \text{for } i \in (1, 2, \ldots, n) \text{ and } j \in (1, 2, \ldots, m), \tag{3}$$

where n is the total number of windows and m is the total number of log events. Here F_i represents log events with feature i and $Count(F_i)$ counts the occurrence of those events under the specific window size.

Taking the feature matrix M or the workflow G as inputs, we are able to train an anomaly detection model (He et al. 2016; Lou et al. 2010). Motivated by Du et al. (2017), we build a Long Short-Term Memory (LSTM) neural network as the anomaly detection model at the service layer. In order to translate the workflow G into a smart contract, we use a directed graph $G = (V, E)$ to represent the workflow, where $V = \{v_i | i \in [1, 2, \ldots, N]\}$ is the set of F_i, $E = \{(v_i, v_j) | v_i, v_j \in V, i \neq j\}$ is the set of edges. The adjacency matrix S of the graph G is calculated as:

$$S_{i,j} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E, \text{ or } (v_j, v_i) \in E, \text{ or } i = j \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

3.4 Auto-update mechanism of smart contract

In this part, we focus on how to translate the workflow G or the adjacency matrix S into a smart contract. The translation processes are dynamic as the logs are being generated on the devices all the time. As shown in Fig. 3, any changes made in the workflow will call for a transaction. Then, the updated workflow is uploaded to the account address of smart contract in the edge blockchain. Our smart contract automatic update mechanism mainly consists of two parts: workflow micro-updates and global parameter updates.

Workflow micro-updates. Intuitively, each log entry, log event, or log feature is the execution step of a log printing statement in the source code. Every task of end devices or applications will generate a sequence of log entries. The order of log entries produced by a task represents the execution order of program events. In general, there are three types of program flows: sequences, branches, and loops. In Fig. 3, each node in the workflow denotes a log event. Here are the examples of a loop "*S2-S3-S4-S2*" and a branch "*S5-S7-S9*". Given a large volume of log records, we can train one workflow in a simple network or a series of workflows in complex networks. A workflows reflect the normal execution flow of a program. Malicious behaviors usually attack the normal execution by disrupting the order of log entries, which also causes the nodes changing in the workflows. By capturing the normal execution patterns of a series of workflows, we can detect the anomaly behaviors in the workflows.

When we use workflow for anomaly detection, we first check whether the context of log event conforms with the rules of normal program flow. For example, if the current log event is "*S3*", the previous log event must be "*S2*", otherwise the program flow is abnormal. Therefore, workflow-based anomaly detection is important for diagnosing the anomalies and understanding how and why they happen. Workflow based anomaly detection provides users the contextual information of last events instead of only providing the diagnose results e.g. either normal or abnormal. In addition, workflow based anomaly detection models are more robust than the log-based anomaly detection models because workflow in a normal environment does not change dramatically. The normal running process follows the logic of the code, especially IoT devices usually only perform some specific tasks.

In practice, micro-updates on existing workflows are frequently needed during system upgrades, code optimizations, task changes and etc. For example, the original branch of the former workflow is "*S5-S7-S9*" or "*S5-S6-S8-S9*" in Fig. 3. When the administrator optimizes a module which produces a different log print statement, the previous log event "*S8*" is changed to "*S10*". In this case, we should adjust the updated workflow to ensure that our anomaly detection algorithm works correctly. In the new workflow, the original branch is replaced with "*S5-S7-S9*" or "*S5-S6-S10-S9*". In fact, once we have deployed the trained workflow, we can simply adjust it with micro-updates. Such as the feature "*S8*" in a previous workflow is replaced with "*S10*" after a period of time during system upgrades. This feature replacement reflects the program execution flow changes. Any drastic changes certainly cause an alarm and attract the administrator's

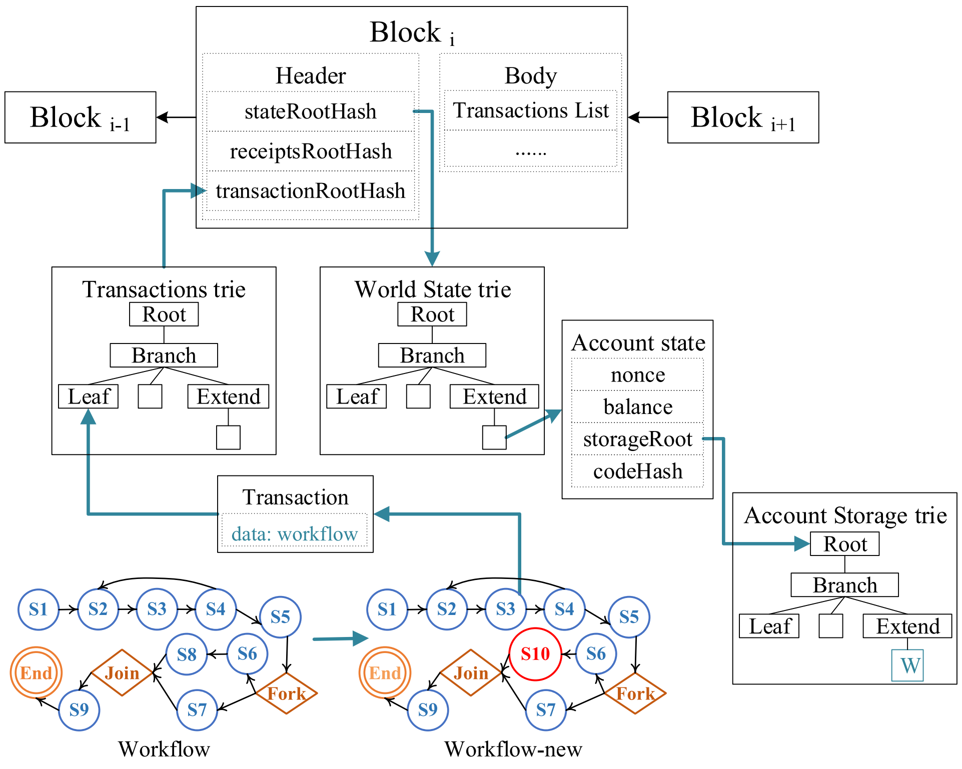


Fig. 3 Smart contract auto-update following micro-updates of workflow

attention. Thus, micro-updates in workflow will reduce operating costs and keep the system stable in the running time of our HADS model.

Global Parameter Updates. As described in Sect. 2.2, smart contracts are programs that run on a blockchain, and cannot be upgraded once it is deployed. This is because the smart contract data stored on the blockchain is immutable. However, anomaly detection is a dynamic process because the attack was unexpected. The workflow for anomaly detection is relatively stable compared with other classifiers, but the workflow is still variable in a period. Thus, we need to upgrade the smart contracts when some micro-updates occur. Due to the inborn immutability of smart contracts, lacking flexibility makes them cumbersome in cases when frequent changes and updates are required. Thus, we need to balance the immutability and upgradability in using smart contracts in security mechanisms otherwise extra overheads and costs are needed.

One possible solution is using an on-chain registry contract to maintain the mapping between user-defined names and the blockchain addresses of the registered contracts. These addresses are advertised off-chain. Every new contract registers its name and the address to the registry contract after being deployed, and it is retrieved for the latest version. Smart contracts can be upgraded by replacing the address of the old version in the registry contract with the address of the new version. This replacement will not break the dependency between the upgraded version and other smart contracts that depend on its functions. This upgrade can be implemented by updating the variable parameters that store the contract address in the registry contract because the registry contract have a permission control module to maintain the writing permission. Note that all the previous values of the variable parameters are still stored on the ledger.

In HADS, we deploy a private Ethereum at the edge layer, and we also implement a workflow contract to maintain the workflow with micro-updates. In the workflow contract, we definite a variable to store workflows. At first, the workflow contract needs to register its address in a registered contract and maintain write permission for the geth client of DNN models. Only the geth client at the service layer can write the variable parts of the workflow contract. In Ethereum, the variable as a global parameter is stored in the leaf nodes of the World State trie corresponding to the smart contract account, at an account storage trie in account state of world state trie of the smart contract account. As shown in Fig. 3, the workflow is ultimately stored in the leaf node of the Account Storage trie. And then, when the micro-updates of workflow occur, a geth client will submit a transaction that stores the new workflow, to the address of workflow contract. After commit, consensus, mine, etc., this transaction will be verified and committed to the transaction trie of a new block in the ledger. Finally, the world state of blockchain will update the world state trie from the new transactions in the transaction trie. So, the new workflow will be updated to the global parameters of the corresponding smart contract. Obviously, the previous workflows are still stored on the ledger. From now, smart contracts will be updated through consensus on the global parameters that will change with some micro-updates in a workflow. Furthermore, the smart contract ensures the trustable of workflow and achieves the collaborative defense by the consensus of the entire network.

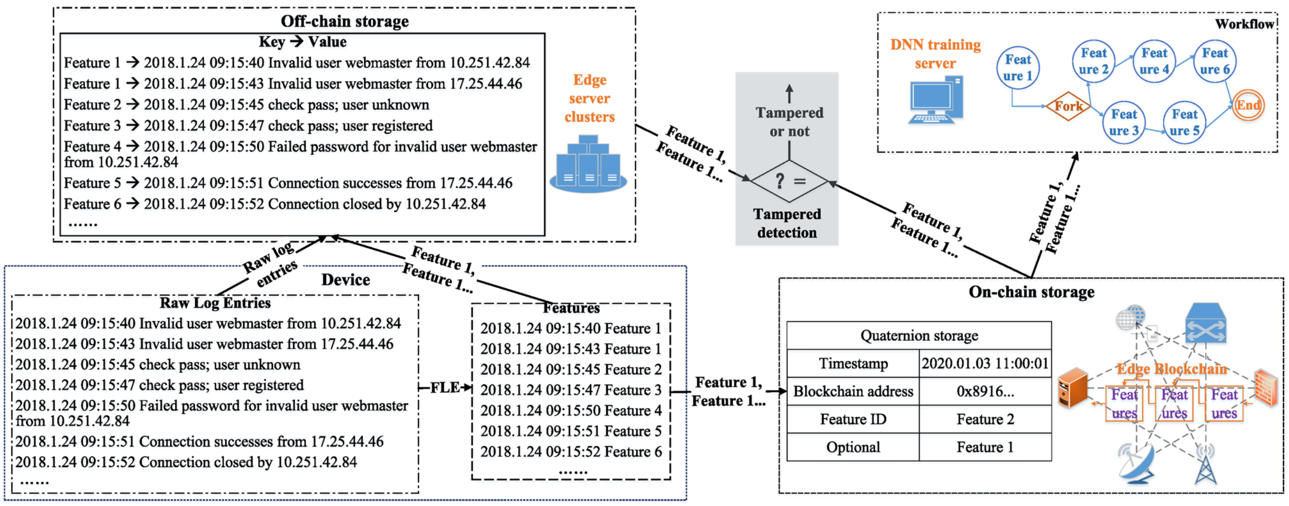
4 On-chain/off-chain storage scheme

In this section, we implement an on-chain/off-chain storage scheme to achieve the tamper-proofing storage for large volumes of logs demonstrated in Fig. 4. To reduce the ledger size, we only store the features to the on-chain storage for DNN training. In addition, we store the mapping from features to raw log entries to the off-chain storage, which is able to help us detect tampered logs.

4.1 Storage scheme implementation

Due to the ledger's full replication across all participants of the blockchain, where it is kept permanently, the capability of storing these data has been greatly restricted. Generally, storing large volumes of data within a single transaction is impossible due to the limited size of the blocks. (For example, Ethereum has a gas limit on the number, computational complexity, and the data size of the transactions that can be included in a block.) To solve this problem, one possible solution is to store the hash value of the raw data on-chain and the raw data off-chain. The hash value is generated by a hash function, such as SHA-256, which is a one-way function. It is easy to compute, but hard to reverse compute as a random input. We use the transaction on a blockchain that has the same hash value to guarantee the integrity of the hash value as well as the raw data. The integrity of the raw data can be validated against the on-chain hash value, and any changes of the raw data will also be detected by the hash value stored on the blockchain.

As mentioned in Sect. 3.2, we obtain a feature and a mapping between the feature and the raw log entry when a new raw log entry is being preprocessed by FLE on the device. The feature is constructed with a unique symbol that does not contain any knowledge on the log entry, so it can be stored in a trustless blockchain. The size of raw log data is always bigger than the size of the hash value, and even through one bit of the data changes, the hash value would be different. When we use the blockchain for large amounts of tamper-proof logs storage, we can use the feature for on-chain storage, and the mapping for off-chain storage.



```
Feature 1 → 2018.1.24 09:15:40 Invalid user webmaster from 10.251.42.84
Feature 1 → 2018.1.24 09:15:43 Invalid user webmaster from 17.25.44.46
Feature 2 → 2018.1.24 09:15:45 check pass; user unknown
Feature 3 → 2018.1.24 09:15:47 check pass; user registered
Feature 4 → 2018.1.24 09:15:50 Failed password for invalid user webmaster
from 10.251.42.84
Feature 5 → 2018.1.24 09:15:51 Connection successes from 17.25.44.46
Feature 6 → 2018.1.24 09:15:52 Connection closed by 10.251.42.84
......
```

```
2018.1.24 09:15:40 Invalid user webmaster from 10.251.42.84
2018.1.24 09:15:43 Invalid user webmaster from 17.25.44.46
2018.1.24 09:15:45 check pass; user unknown
2018.1.24 09:15:47 check pass; user registered
2018.1.24 09:15:50 Failed password for invalid user webmaster
from 10.251.42.84
2018.1.24 09:15:51 Connection successes from 17.25.44.46
2018.1.24 09:15:52 Connection closed by 10.251.42.84
......
```

```
2018.1.24 09:15:40 Feature 1
2018.1.24 09:15:43 Feature 1
2018.1.24 09:15:45 Feature 2
2018.1.24 09:15:47 Feature 3
2018.1.24 09:15:50 Feature 4
2018.1.24 09:15:51 Feature 5
2018.1.24 09:15:52 Feature 6
......
```

Quaternion storage	
Timestamp	2020.01.03 11:00:01
Blockchain address	0x8916...
Feature ID	Feature 2
Optional	Feature 1

Fig. 4 The on-chain/off-chain separate storage scheme

On-chain Storage. From FLE, we can obtain the features for the service training and transmit it to an edge node with JavaScript. The edge node generates a transaction to commit the features and is stored into ledger after achieving consensus. Now, we consider how to store them better in the edge layer because we will read them from the ledger to train a DNN model at the service layer. In the transaction, we design a storage structure that only stores a quaternion (or 4-tuple) to the edge nodes. The quaternion (or 4-tuple) mainly consists of a timestamp, blockchain address, features, and optional field. The timestamp field mainly is used to ensure the feature sequences in order; the blockchain address field can record a feature coming from which device, and also can be used to separate the feature sequences for fine-grained services; the features field mainly record the data upload from the devices; the optional field will record some additional information for the normal requirements in service.

Off-chain storage. From FLE, we can obtain the hash mapping between features and raw log entries and transmit them to an edge or cloud file system, such as IPFS, for off-chain storage. We use a key-value storage structure, where the key represents features and the value points to the raw log entries. In HADS, we compute a hash value for each log entry instead of a big log file that consists of many log entries. This will help us to easily find which log entry has been tampered with. Furthermore, we will recover most of the content that has been tampered with from the event represented by hash value and also provides further information for other smart applications.

Implementation. The core of the storage scheme is that the FLE can run on resource-constrained devices. As mentioned in Sect. 3.2, FLE can extract the structured features to form sequences for the DNN training from the unstructured raw log. This requires it to run in a fine-grained service. In HADS, the hash value is computed by corresponding events instead of raw log entries. So different log entries with the same event will have the same hash value. Therefore, we use the hash value to form the sequences will not influence the feature matrix, and also the training accuracy of DNN. The pseudocode of FLE is shown in Algorithm 1.

Algorithm 1 FLE

```
Input: $log\_entries$
Output: $on\_chain$,  $off\_chain$
1: $event\_list = [e_1, e_2, ...]$;
2: $hash\_list = [h_1, h_2, ...]$;
3: if ($new \quad log\_entries$) then
4:     $reg\_log \leftarrow Regular\_Expression(log\_entries)$;
5:     $length \leftarrow Length(reg\_log)$;
6:     $first\_token \leftarrow Get\_First\_Token(reg\_log)$;
7:     $last\_token \leftarrow Get\_Last\_Token(reg\_log)$;
8: else
9:     $Wait(time)$;
10: end if
11: for $event \in event\_list$ do
12:     $length\_event \leftarrow Length(event)$;
13:     $first\_token\_event \leftarrow Get\_First\_Token(event)$;
14:     $last\_token\_event \leftarrow Get\_Last\_Token(event)$;
15:     if ($length = length\_event$) then
16:         if ($first\_token = first\_token\_event$) then
17:             if ($last\_token = last\_token\_event$) then
18:                 $new\_event = event$;
19:                 for $hash \in hash\_list$ do
20:                     $hash = look\_up(event)$;
21:                     $new\_hash = hash$;
22:                 end for
23:             end if
24:         end if
25:     end if
26: end for
27: $event\_list[].append(new\_event)$
28: $new\_hash \leftarrow Hash(new\_event)$
29: $hash\_list[].append(new\_hash)$
30: $on\_chain = new\_hash$;
31: $off\_chain = new\_hash \quad \& \quad new\_event$;
32: return $on\_chain$,  $off\_chain$.
```

4.2 Tampered detection

In HADS, because log files record the system behavior as audit evidence, the model needs to meet the tamper-proof requirements when storing log data. When tampering with logs occurs, HADS can detect tampering behavior in time, locate and recover some of the tampered content. Naturally, we assume that the data stored on the blockchain is secure and not easily tampered with, because the data stored in the blockchain is immutable. Thus, the on-chain/off-chain storage scheme can achieve the tamper-proof storage since the features stored on-chain serve as a basis for detection.

In tampering detection, the main steps consist of acquisition, comparison and recovery. At first, we obtain the features stored on-chain and the keys stored off-chain. The features and the keys are all calculated from the corresponding log events after processed by FLE. The on-chain features can not be tampered by malicious attackers or dis-honest participants. Then, we compare the features with keys in order. If all of them keep the same, there is no tampering behavior occurring; Otherwise, the corresponding off-chain entries have been tampered. Finally, we can recover some content (log event, except for the variable part of raw logs) of the tampered entries stored off-chain. Because the features stored on-chain still trustful, we can find the log event from the event list and check the hash value. To reduce the system overhead, we can make periodic detection or on-demand detection.

4.3 Efficiency and security analysis

In this part, we focus on the effectiveness and security of on-chain/ off-chain storage schemes. Effectiveness refers to that the features stored on-chain can satisfy the training steps of subsequent DNN. Security refers to the features stored on-chain that will not reveal more information and prevent tampering.

Efficiency. In HADS, we use the features (a hash value of log event) to replace the raw logs as the input of DNN training and inference. This replacement can not influence the performance of log-based anomaly detection models. Because the direct input values of models are feature matrix, which can use windows technology to count the occurrence number of log events. In other words, the models do not care about the content of raw logs. For example, there is a raw log sequence in Fig. 4. If we calculate the hash values of raw log entries, we can not distinguish the log events from different hash values, such as "*Feature 1, Feature 2, Feature 3, Feature 4...*", so we can not get the feature matrix from this method. But, we calculate the hash value of log events in HADS. The log entries with same events can reserve the same hash values, such as "*Invalid user webmaster from 10.251.42.**" and "*Invalid user webmaster from 17.25.44.**" have the same event "*Invalid user webmaster from **", so they keep the same feature "*Feature 1*". And then, we can count the "*Feature 1*" occurrence number is "2" in the feature matrix.

Security. In HADS, the features on-chain is a hash value of log event. The security properties of the one-way function or hash function can guarantee the security of HADS.

One-way Function: A function $f \ : \{0,1\}^* \rightarrow \{0,1\}^*$ is a one-way function, if and only if:

(1) for $\forall\, x$, $f(x)$ can be easily calculated by a polynomial time algorithm;
(2) for a given $f(x)$, x can not be calculated by a polynomial time algorithm.

Even though the malicious attackers or dis-honest participants obtain the features stored on-chain, they are not able to decipher the corresponding raw logs or log events in polynomial time. Since the features on-chain are just hash values of log events, they will not leakage any information. The hash function or one-way function is a compressing algorithm that loses a lot of information and reduces the data size. After processing by hash function or one-way function, no one can recover the raw content easily, and the feature sequence only reserves the order information.

5 Smart contract-based dynamic management mechanism

In this section, we implement a smart contract-based dynamic management mechanism to achieve the secure storage, management, and analysis in Fig. 5. The dynamic management mechanism has three components: (1) data secure storage. The process includes submitting, mining, verifying the transactions that stored in the data field; (2) smart contract auto-update. As mentioned in Sect. 3.3, we update the global parameters for the smart contract's transaction; (3) real-time detection. When the devices upload the feature, it will get the detection result and feedback policies through a device snapshot contract, verifying contract, and feedback contract.

5.1 Mechanism overview

In HADS, the core mechanism is five smart contracts: registry contract (RC), device snapshot contract (DSC), workflow contract (WFC), feedback contract (FC) and verifying contract (VC). The last four contracts need to register in RC before they are called. We give an overview of the smart contract-based dynamic management mechanism in Fig. 5, which mainly consists of the "Edge-Service" loop for workflow micro-updates and the "Device-Edge" loop for feature storage and real-time detection.

"Edge-Service" Loop. The workflow-based DNN anomaly detection service downloads feature sequences from the edge ledger, then generates the workflow for anomaly detection. The detection flow consists of WFC, VC, and FC. WFC stores the workflow where the smart contract is and the workflow will be updated with micro-updates if its content changes. VC uses the workflow to determine whether the feature is abnormal. If an anomaly occurs, FC is called to alert and notify the service for re-training.

"Device-Edge" Loop. The features will be uploaded and stored into the ledger after the detection. The detection flow consists of DSC, VC, and FC. DSC stores the status

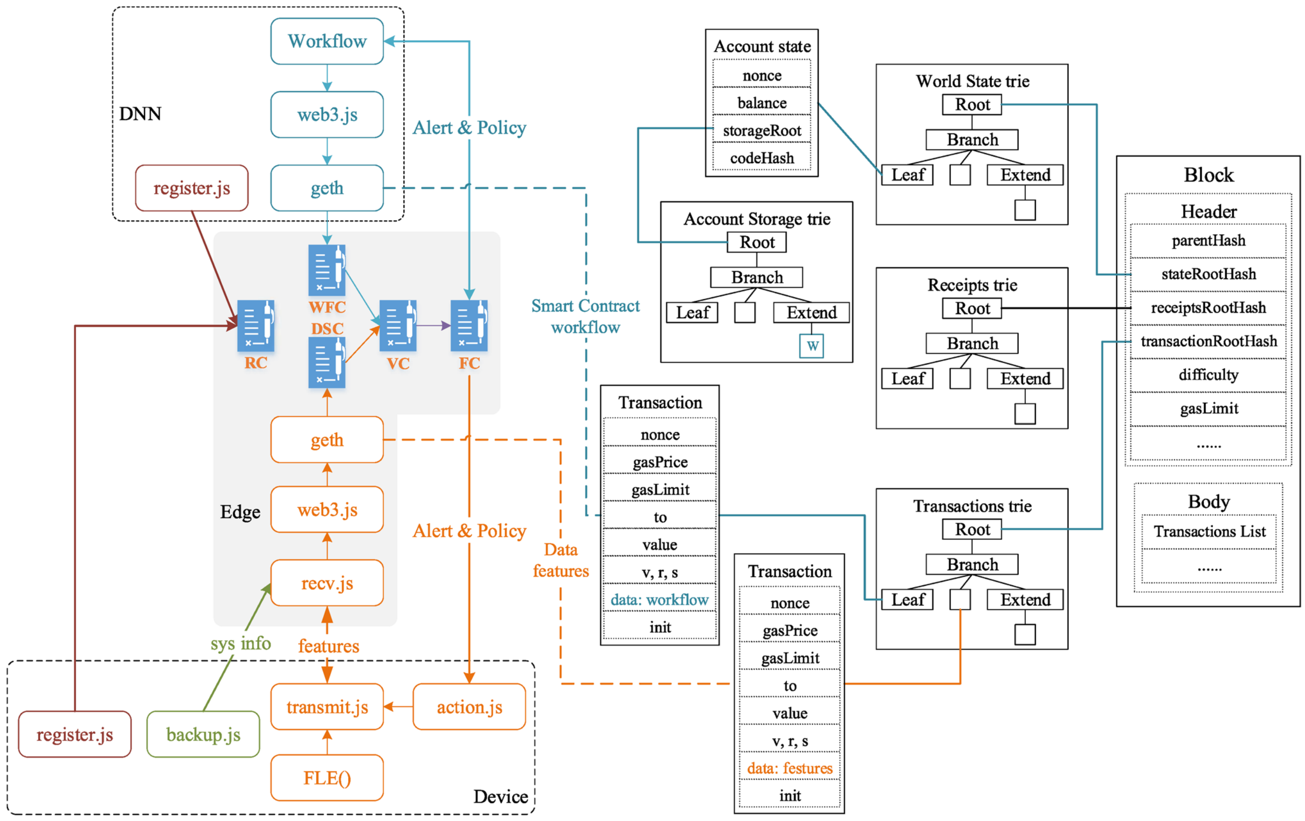


Fig. 5 The smart contract-based dynamic management mechanism

information of the device, in which you can find the connection between previous features and current features. Before submitting a transaction, VC checks whether the transaction is abnormal or not based on its workflow. FC can alert the abnormal feature and generate the corresponding secure policy, then return the alert and policy to the device to respond to the anomalies.

5.2 Smart contract implementation

These smart contracts are the core of HADS, so we generate these contracts under security analysis before their deployment to avoid vulnerabilities. Furthermore, the structural design of the smart contract has a large impact on its execution cost, so we will improve the designed framework to reduce the execution time and deployment cost through off-chain pre-compiled processes. We show a flow chart between five smart contracts in Fig. 6. There are mainly consists of five steps: register, lookup, data transmission, verify, and reaction. And we also demonstrate the pseudocode of core functions in Algorithm 2.

Register. When HADS starts, all the devices, services, and smart contracts need to be registered by calling "*serverReg(), deviceReg(), contractReg()*" in Registry Contract and obtain a blockchain address. RC maintains a trusted addresses list which records the trusted device addresses that have been added to the network, and the smart contract addresses that have been reached a consensus. RC periodically checks and manage the list, and also remove the untrusted devices, services, and contracts. RC provides security read and writes operations for numerous transactions and smart contract public data space in blockchain.

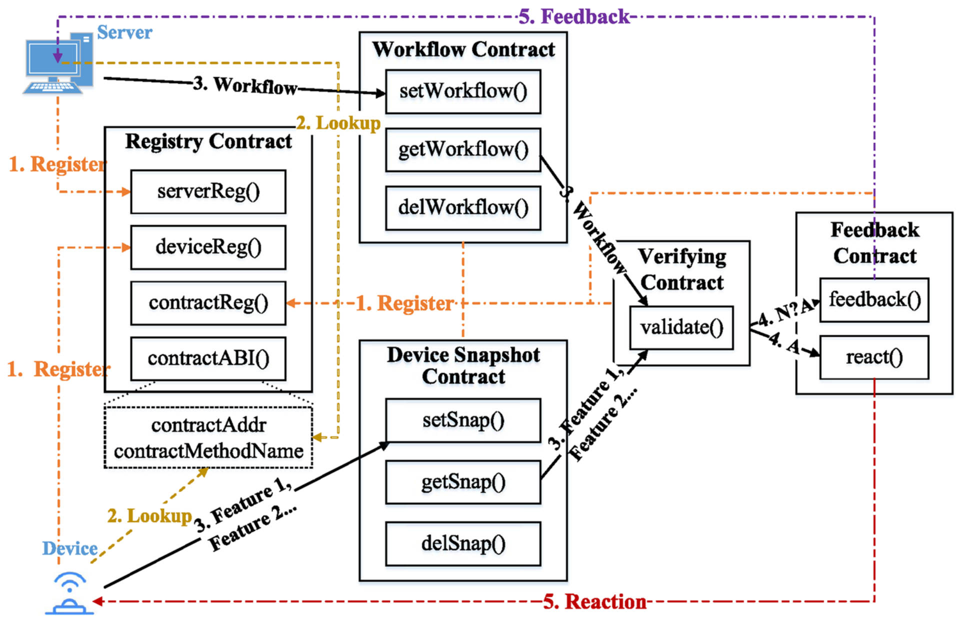
Lookup. When the devices upload the features or the services set and update the workflows, they will first call for "*contractABI()*" in RC to get the "*contractAddr, contractMethodName*" parameters so that they can upload the data to the correct blockchain address. After they get the correct address, the devices can call the"*setSnap()*" in DSC through "*contractABI.setSnap()*" to set a snap-shoot. The snap-shoot helps to quickly find the former log key of the current log key. The devices can call the "*setWorkflow()*" in WC through "*contractABI.setWorkflow()*" to store the current workflow to the blockchain ledger. This method is convenient for administrators to maintain huge amounts of devices and services.

Data Transmission. After the devices and services lookup the corresponding function addresses, the devices will upload the feature sequences to the edge blockchain through "*setSnap()*" and store them into the ledger in order. In addition, the services upload the workflows to the edge blockchain through "*setWorkflow(), Graph()*" and also store them into the ledger as an adjacent matrix.

Verify. The core function of HADS is anomaly detection and implemented in VC by "*validate()*". Before a transaction feature is submitted, VC verifies whether it is abnormal or not. VC utilizes the last feature indexed by the device ID in the DSC and the workflow graph stored in the WFC to detect whether the current upload feature is abnormal or not. If an abnormal occurs, FC is called to alert and generates defense policies.

Reaction. When we find an anomaly occurs in HADS, we can call "*react()*" function to transmit the security policies to devices. These policies defined by administrators will tell

Fig. 6 The flow chart of smart contracts



the devices how to react to the anomalies. And also, we can call "*feedback()*" return the detection result to services. This feedback contains a label of the current features which improves the performance of service training and inference.

Algorithm 2 Smart Contract

```
function CHECK($address$)
    $mapping(address \Rightarrow (mapping(string$
      $\Rightarrow Boolean)));$
    if $address \in address\_list$ then
        $result \leftarrow state(address, type) \leftarrow TRUE;$
    else
        $result \leftarrow FALSE;$
    end if
    return $result.$
end function
function CHECKSTATE($address$)
    if $address \in RC\_state$ then
        $info(address) \leftarrow (sysinfo, address)$
        $result \leftarrow TRUE;$
    else
        $result \leftarrow FALSE;$
    end if
return $result.$
end function
function GRAPH($workflow$)
    if $address \in RC\_state \quad \&\& \quad type ==' server$
then
        $node[] = newArray(count);$
        $line[][] = newArray(count)(count);$
        $vargraph = WFC\_graph(address);$
        $node[] = graph.nodes;$
        $line[][] = graph.line;$
        $result \leftarrow TRUE;$
    else
        $'running.log' \leftarrow error;$
        $result \leftarrow FALSE;$
    end if
    return $err, result, address.$
end function
function VALIDATE($server\_wf, device\_feature$)
    $lastfeatures \leftarrow DSC\_lastfeatures;$
    if $lastfeatures == NULL$ then
        $DSC\_lastfeatures \leftarrow features;$
        $result \leftarrow TRUE;$
    else
        if $line[lastfeatures][features] == TRUE$
then
            $DSC\_lastfeatures \leftarrow features$
            $uploadToBC(features).send$
      $Transactrion();$
            $result \leftarrow TRUE;$
        else
            $DSC\_lastfeatures \leftarrow NULL$
            $uploadToBC(features).send$
      $Transactrion();$
            $result \leftarrow FALSE;$
        end if
        $FC(result, address, features)$;
    end if
    return $result, address, features.$
end function
function FEEDBACK($feature\_result$)
    if $address \in RC\_state$ then
        if $VC\_state == TRUE$ then
            $alert \leftarrow NULL$;
        else
            $alert \leftarrow (address, features)$;
            $policies \leftarrow Policy(features)$;
        end if
    end if
    $TransmitToServer(alert, server\_addr)$;
    $TransmitToDevice(alert, policies, device\_addr)$;
    return
end function
```

6 Evaluation

In this section, we demonstrate that HADS can reduce the ledger size to 7.1% without losing the accuracy of anomaly detection. The FLE framework running on low-power devices outperforms the state-of-the-art encryption solutions and log parsers on both CPU utilization and task execution speed. We implement a system prototype to show that HADS is able to make real-time anomaly detection with an average latency of 0.47 ms. The evaluation part concerns the following questions:

- How accurate can the HADS framework achieve? Is the accuracy affected by reducing data?
- How much ledger size does HADS need compared with the state-of-the-art technologies?
- Why the HADS can work on such small ledge size without losing accuracy?
- How does the FLE framework perform on low-power devices?

6.1 Experimental setup

Configuration. As shown in Fig. 7, the experiments are running on a network with ten boards of NVIDIA-JETSON-TK1, and two servers, one is used for DNN training, the other is used for the edge server. The specifications of these devices are listed in Table 1. TK1 devices can simulate IoT devices with resource limitations. We run a three nodes Ethereum network on the edge server to simulate the blockchain-driven edge network. The training server is mainly used for DNN training taking the data stored in the ledger as inputs. Finally, we also design an application to show the state of devices and Ethereum.

Table 1 Specifications of devices

	Device	Edge blockchain	Training server
Hardware	Jetson-TK1	DELL Tower	DELL Tower
CPU	ARMv7	Core i7-2600	Xeon E5-2630
Memory	1.9 GB	12 GB	96,566 MB
OS	Ubuntu 14.04	Windows 10	Ubuntu 16.04

Table 2 Summary of datasets

System	Size	#Log messages	#Log sequences
Benchmark dataset	13.4 MB	104,815	7,940
HDFS dataset	1.49 GB	11,197,691	575,061
Oil industry dataset	11.9 MB	132,602	–

Blockchain Configurations. We use Ethereum to achieve secure storage and flexible architecture in the edge layer. With the geth clients, we create an Ethereum account for each server and device, and configure these nodes to form a private blockchain network, where the edge server plays the roles of miners due to their large computing and storage capability, and the end device and DNN training server serve as light Ethereum nodes that only send transactions using web3.js. We utilize some other JavaScript codes to exchange the information between servers and devices in Fig. 5.

Datasets. We choose the HDFS dataset, HDFS benchmark dataset Zhu et al. (2019), and oil industry dataset to evaluate our system prototype. In Table 2, we illustrate a detailed summary of the datasets. HDFS benchmark dataset contains 104,815 raw log messages, with a size of 13.4 MB, which is often used to evaluate the lightweight metrics, compression ratio, and latency. The size of HDFS dataset is 1.49 GB and it contains 11,197,691 raw log messages. There are 575,061 log sequences separated by block_id in total as HDFS raw log has a unique block_id for each block operation. In the experiment, we use 102,697 (almost 20%) normal sequences for training. The oil industry dataset is collected from the real industry production environment and contains 132,602 raw log messages with 788 anomaly log messages, which is valuable for evaluating the real performance of anomaly detection models.

Measured metrics. Common metrics, such as precision, recall, and F1-Score are used to show that the HADS framework is able to make real-time anomaly detection without

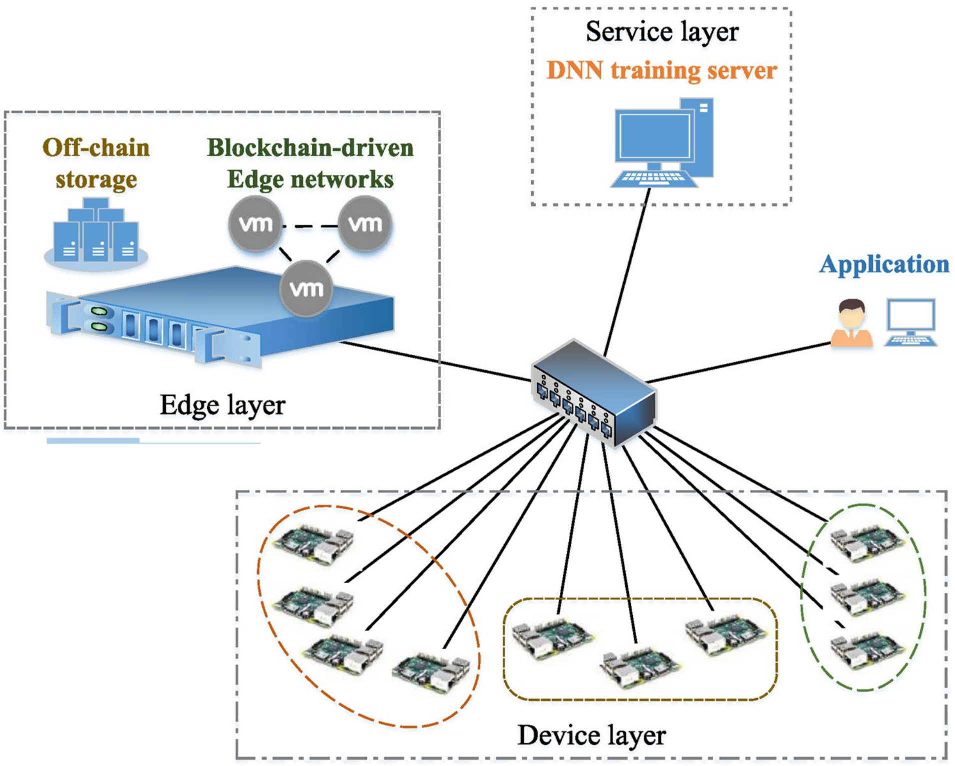


Fig. 7 The experiment network topology

losing detection accuracy. In addition, we present a compression ratio to study the influence of reducing ledger size under different transmission rates. We also analyze the CPU usage and execution time of the proposed FLE framework running on both resource-constrained and non-resource-constrained devices comparing with other widely used log parsers and state-of-the-art encryption solutions.

6.2 Model performance

In this part, we evaluate the performance of HADS on the accuracy, latency, and compression ratio. The experimental results show that our hierarchical model is as accurate as of the traditional centralized models. In the meanwhile, HADS has a lower latency and a higher compression ratio.

Accuracy. We design two comparison experiments on training DNN models for anomaly detection. One is trained with the raw dataset while the other is trained on the processed dataset by using FLE. Fig. 8 shows the comparison results of experiments on the HDFS dataset and the oil industry dataset. In the bar chart, the performance metrics without data size reduction is the same as the performance of the processed data by FLE. The reason is that most of the input data of DNN are usually very redundant for training or inference in our cases. As we discussed in Sects. 3 and 4, we use the hash value of events to replace each log entries in HADS so that the sequences after being pre-processed still keep the same order and the same occurrence of events. Furthermore, a 4-tuple storage structure is utilized to store the ordered sequences in the edge ledger. When we train the DNN model at the service layer, DNN will use the windows technologies to extract the feature matrix by counting the occurrence of events in the ordered sequences.

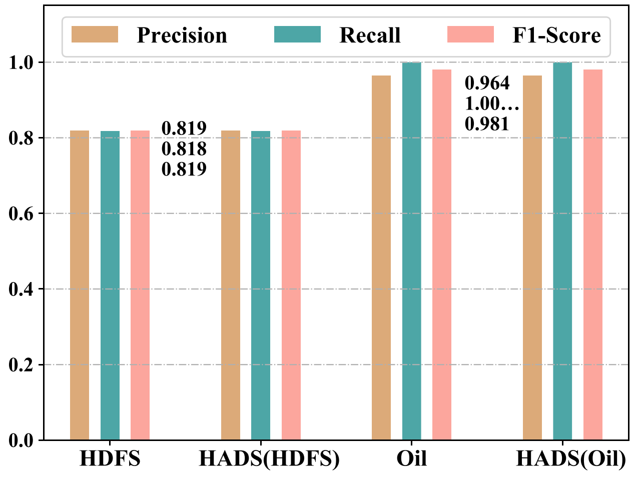


Fig. 8 The accuracy comparison of workflow-based anomaly detection in different datasets. The FLE operation do not influence the accuracy

Latency. We compare the uploading latency and detection latency in the experimental network to prove that HADS can be running in real-time. In Fig. 9, we test the detect latency of uploading features from devices to the edge blockchain, send the results or policies from the edge blockchain to devices, and also record the time cost when we submit $1k$, $2k$, $3kldots20k$ transactions. In addition, we test the uploading latency when we submit $1k, 2k, 3k, \ldots, 20k$ transactions. The average uploading latency is 0.25 ms, and the average detection latency is 0.47 ms, which satisfies the demand for real-time detection. In theory, the latency will be reduced as offloading the detection computational from the services layer to the edge layer, but we still need to further evaluate it at a real network in the future. In this part, we focus on testing the latency of our algorithm without considering the latency of Blockchain, such as the latency of block generation. We also can further optimize the Blockchain ledger structure, consensus algorithm, smart contract, and instruction set extension, so that HADS can adapt to real-time analysis of massive log data.

Compression Ratio. Each node in the blockchain keeps a complete copy of the data, which leads to the increasing scale of the ledger, and seriously affects the scalability of the blockchain. In this paper, we focus on reducing the ledger size and propose a measurement metric called compression ratio which is mainly to measure the ratio of the size of pre-processed data to the size of raw data. The ratio can be expressed by the following equation:

$$Compression\ ratio = \frac{\#\{the\ size\ of\ Pre(data)\}}{\#\{the\ size\ of\ data\}}, \tag{5}$$

where $Pre()$ is the data processing operation. Due to the high cost of data storage on public chain of Ethereum and the privacy leakage risk caused by storing all the raw data, the

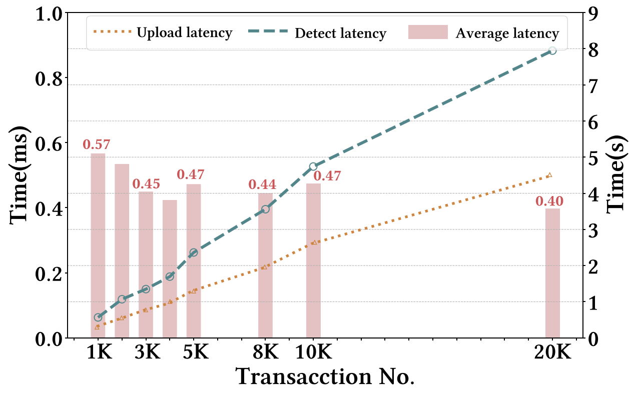


Fig. 9 The network upload latency and detect latency in an experimental environment

compression ratio is an important metric to evaluate the data processing performance and privacy protection.

In this experiment, we take the raw log storage scheme as our baseline, and compare our FLE method with two typical encryption solutions: symmetrical encryption (SE) and asymmetric encryption (AE) which are frequently used for protecting secure storage of logs Pourmajidi and Miranskyy (2018). For the symmetrical encryption approach, we choose advanced encryption standards (AES-256) to encrypt the plaintext and RSA-1024 or RSA-2048 to encrypt the symmetrical key; For the asymmetric encryption approach, we choose RSA-1024 to encrypt the plaintext, rather than RSA-2048 because the encrypting speed of RSA-2048 is so slow that it is rarely used to encrypt the plaintext directly.

Figure 10 shows the comparison results on the HDFS benchmark dataset in two scenarios: (1) the change of the data size before being uploaded the data to the blockchain; (2) the change of the ledger size after storing the data into the ledger. The bar chart in Fig. 10a shows the change of the data size and the dash lines reflect the compression ratio after the different processes. In the bar chart, the results show that only FLE is able to effectively reduce the data size, while the data size increases with SE and AE processing. The reason is that the encrypt schemes need to contain all the information for data decryption. The encryption solutions need to add additional data to fuse, and this will cause data inflation. But we can also get the raw data after the decryption. However, after the FLE is being processed, the additional data will be deleted from the raw data to prevent information leakage. The lines in the figure reflect the compression ratio of FLE is reaching 7.1%, indicating that much data is now required to be stored to the ledger, and this will efficiently reduce the storage cost of gas and ether in Ethereum.

Figure 10b shows the change of ledger size after being stored into the blockchain. Different dash lines in the graph represent the ledger size change and the cost of time when processing the same raw data using SE, AE, and FLE. With the FLE process, the ledger size is reduced from 29.95 MB to 2.10 MB. Moreover, the consensus process speeds up to 2.34x times compared to storing the raw data directly into the ledger. However, with either the SE or the AE process, the ledger size is even larger than the ledger of the raw data.

Figure 10c and d show the further experiments when uploading with different amount of items per transaction. When we upload 50,000 items to the blockchain per transaction, the ledger size changes greatly and the compression ratio is further improved to 1/55. If in the same ledger size, the data items contained in the ledger also change obviously when uploading in different items/Transaction, and the compression ratio can also further improve to 1/12. This also indicates that the number of network-accessible devices can be increased. Thus, FLE also improves the performance of

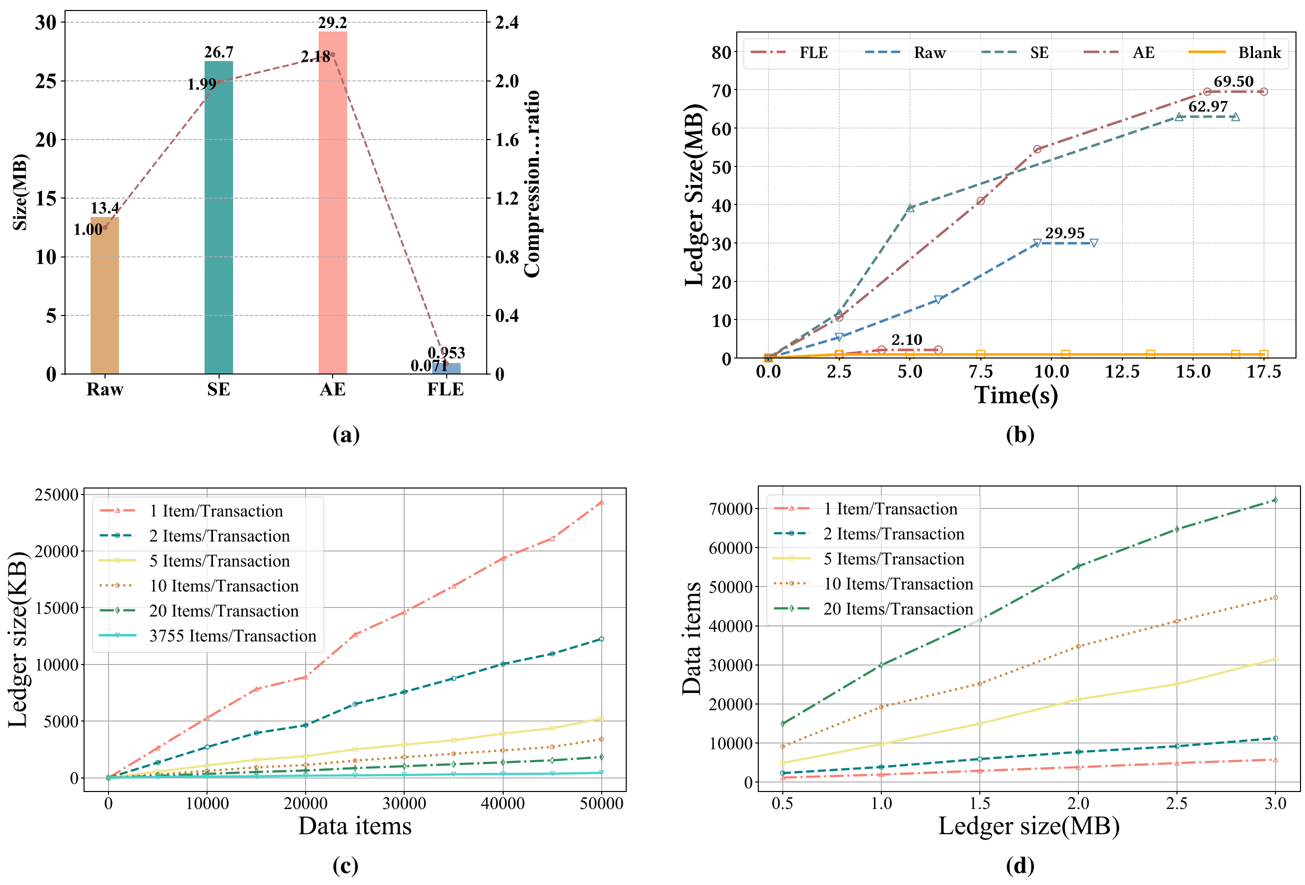


Fig. 10 The compress ratio comparison: **a** the data size between raw data and after the process of SE, AE, and FLE; **b** the ledger size when uploading raw data and after the process of SE, AE, and FLE, and blank data; **c** the ledger size when uploading in different items/Transaction; **d** the data items in the same ledger size when uploading in different items/transaction

blockchain and allows more devices access to the networks by removing irrelevant data and reducing data size as well as preventing data tampering.

6.3 Extractor performance

In this section, we study CPU usage and execution performance of different extractors. We first compare FLE with previous log parsers Zhu et al. (2019) to show the efficiency of the log process, and then compare it with the above encryption solutions. Our experimental results show that FLE can run on resource-constrained devices. This evaluation is measured in two aspects: (1) CPU usage indicates the maximum demand for resources; (2) Execution time indicates the maximum time that an operation takes up resources. The shaded area in Fig. 11 intuitively reflects the need for resources.

Comparing with different log parsers. In this part, we study the CPU usage and execution time of ten different log parsers (Zhang et al. 2019; Zhu et al. 2019) and the experimental results are shown in Fig. 11a. We also make a comparison between simulated cloud servers and simulated resource-constrained devices. In Fig. 11a, we observe that the instantaneous maximum CPU usage reaches 18.3% and the run-time is 0.497 s when we run FLE to process 6000 items on the cloud server. Logcluster, FLE, AEL, Drain, and LFA still keep lightweight on NVidia TK1. When we process 2000 items with FLE on NVidia TK1 to process, the instantaneous maximum CPU usage reaches 25.5% and the run-time is 0.475 s. Thus, we can conclude that FLE is lightweight enough for the resource-constrained end devices because it replaces the heavy loops and transverse steps with some lightweight matches and lookup steps. In addition, the hash computation is called only when a new event is generated.

Comparing with Different Encryption Technologies. In this part, we compare the CPU usage and the execution time of different encryption solutions running on TK1. Figure 11b shows the comparison results of processing 2000 HDFS log messages on NVidia TK1. A blank experiment where

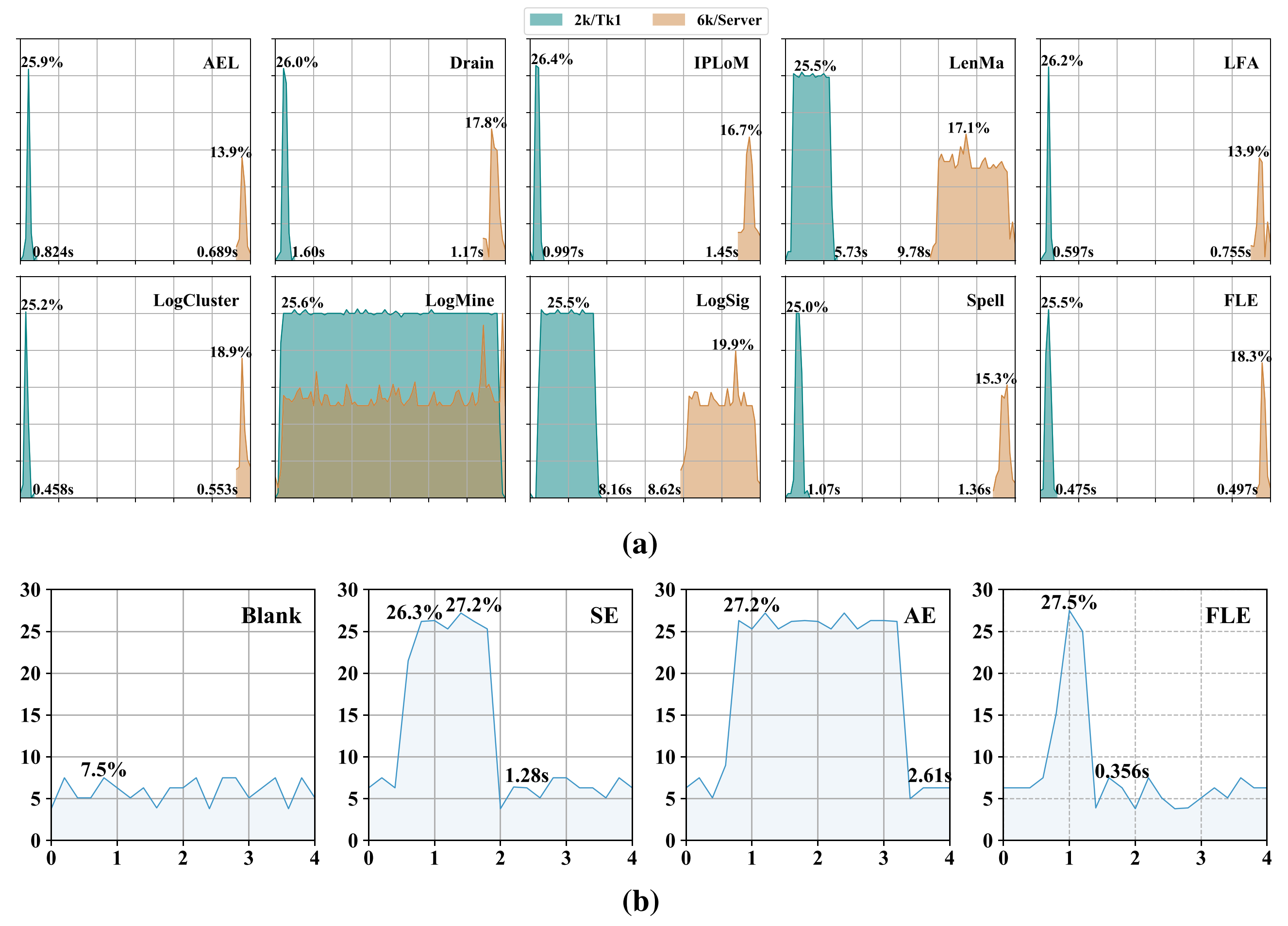


Fig. 11 The comparison of CPU usage and execution time: **a** compared with previous log parsers on serves and NVidia TK1; **b** compared with symmetrical encryption (SE), asymmetric encryption (AE), FLE, and blank experiment on TK1 using 2000 HDFS log messages

Table 3 The execution time on TK1 using HDFS benchmark dataset

Operation	SE	SE advance	AE	FLE
Execution time	2.33 s	32.40 s	57.68 s	19.44 s

runs an upload program is to simulate the normal activities on the edge device. The less shade area in the figure indicates that fewer CPU resources are needed in the resource-constrained end devices. The FLE operation achieves the best performance with the lest execution time and almost the same maximum instantaneous CPU utilization as the other two methods. The task execution speed speeds up to 7.3 x times compared with AE, and 3.6 x times compared with SE. Thus, the FLE operation effectively reduces energy consumption when running on low-power devices. Furthermore, HADS achieves the same level of tamper-proof security without heavy encryption.

Table 3 shows the comparison of the execution time of different compression methods on the HDFS dataset. As the size of the dataset increases, the execution time greatly increases. In the columns of SE, the execution time is only 2.33 s when we choose RSA-1024 to encrypt the symmetrical key, however, this encryption scheme is not secure enough. When we choose RSA-2048 to encrypt the symmetrical key under the premium of security, the execution time of SE Advance reaches 32.4 s which is almost 1.67 x times to the FLE. Furthermore, the execution process of FLE speeds up to 2.97 x times compared with AE. This is because the encrypt operations are always time-consuming. Comparing with other approaches, the FLE method is a rapid extractor that only needs to walk through the log data a few times to obtain the structured log events. The results in two scenarios indicate that FLE is not only a very lightweight encryption tool but also can more efficiently run on low-power devices compared with the state-of-the-art encryption technologies.

7 Related work

Blockchain-assisted Log Storage. Blockchain is being used as a decentralization platform to maintain the data storage consistently in a ledger. And the data contained in a committed transaction along with the historical transactions are seen as immutable in practice Xu et al. (2018). Therefore, the blockchain-assisted log storage scheme can keep the natural character of logs in terms of immutability, traceability, and tamper-proof. The previous solutions can be divided into two categories: (1) store the encrypted logs. In Pourmajidi and Miranskyy (2018), the author proposes a blockchain-based log system, which collects the logs from different providers and avoids log tampering by sealing the logs cryptographically first and then adding them into a hierarchical ledger. This system provides an immutable platform for log storage. Similarly, using a immutable log storage as a service was also proposed in Pourmajidi et al. (2019) Rane and Dixit (2019). Another approach illustrating the design of a weblog storage system based on Hyperledger resulting in higher throughput and lower latency is mentioned in Wang et al. (2018). Blockchain is also used to protect justice Logs in Belchior et al. (2019; 2) store the hash value. In Huang (2019), the authors utilize the InterPlanetary File System(IPFS) to store log files and use Ethereum blockchain to store the hash and the index of the log files. Different from them, HADS uses a fine-grained hash-map for each log entries, which makes a trade-off between the log secure storage and efficient usage.

Log-based anomaly detection. Log-based anomaly detection aims to mine abnormal behaviors on time through training classifiers or mining workflow Lou et al. (2010). These detections help administrators quickly locate and resolve accident issues. The previous models mainly involve four steps: log collection, log parsing, feature extraction, and anomaly detection. The anomaly detection can be further divided into supervised learning models and unsupervised learning models based on the training dataset with or without labels He et al. (2016). The supervised models all require a clear label on normal and abnormal events in the training dataset. In contrast, the unsupervised models that work based on the abnormal event which is always as an outlier point from the normal event, do not need labeled training data. Recently, a deep neural network model utilizing Long Short-Term Memory (LSTM) has been proposed to model a system log as a natural language sequence in Du et al. (2017). To solve the model aging or concept issue, the authors proposed confidence-guided multiple algorithms to jointly detect the anomalies in Xie et al. (2019) Xie et al. (2020). Another important branch is workflow-based anomaly detection methods, which learn workflows from the normal execution path, and detect anomalies when it deviates from the model trained from logs under normal execution Xiao et al. (2016). These previous approaches only consider how to improve the accuracy, but not focus on the log secure storage and actual deployment. To convert the unstructured raw log into a structured event, log parsing is an essential task for anomaly detection. Frequent pattern mining, clustering, and heuristic rules will help to extract the log event by automatically separating the constant part and variable part, and further transform each log entry into a specific event Zhu et al. (2019). Considering the length, token position of log entries, an efficient log parsing based on heuristic rules will achieve better performance. And also, the author gives a confidence-guided evaluation for log parsing inner quality in Xie et al. (2020). Inspired by the previous research, we design an online and automatic feature extractor to generate and upload the structured log sequences instead of raw logs.

8 Conclusion and future work

In this paper, we propose a hierarchical blockchain-driven anomaly detection framework that uses an on-chain/off-chain scheme for immutable and tamper-proof logs storage. We also design a series of smart contracts to manage logs storage and build an auto-update mechanism of smart contracts with the dynamic workflow. To achieve this goal, we first design a feature extractor that runs on the resource-constrained end devices to extract features based on heuristic rules. The content-independent features can be stored on a trustless blockchain to reduce ledger size and satisfy the DNN training requirements. And then, we present a smart contract auto-update method to handle the dynamic changing workflow at the edge layer. The smart contract can be updated through consensus on the global parameters for some micro-updates in a workflow. Our approach provides sound results on multiple datasets in an experimental network and system prototype. HADS can reduce ledger size without losing detection accuracy, and the FLE outperforms state-of-the-art encryption technologies on resource usage.

Acknowledgements This work is partially supported by the National Key Research and Development Program of China (2018YFB2100300), the National Natural Science Foundation (61872200), the People's Republic of China ministry of education science and technology development center (2019J02019), the CERNET Innovation Project (NGII20180306, NGII20190402) and the Natural Science Foundation of Tianjin (19JCZDJC31600, 19JCQNJC00600).

Compliance with ethical standards

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

Belchior, R., Correia, M., Vasconcelos, A.: Justicechain: Using blockchain to protect justice logs. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", pp. 318–325. Springer (2019)

Du, M., Li, F.: Atom: efficient tracking, monitoring, and orchestration of cloud resources. IEEE Trans. Parallel Distrib. Syst. **28**(8), 2172–2189 (2017)

Du, M., Li, F., Zheng, G., Srikumar, V.: Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1285–1298. ACM (2017)

Fu, Q., Lou, J.G., Wang, Y., Li, J.: Execution anomaly detection in distributed systems through unstructured log analysis. In: Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on, pp. 149–158. IEEE (2009)

Hamooni, H., Debnath, B., Xu, J., Zhang, H., Jiang, G., Mueen, A.: Logmine: fast pattern recognition for log analytics. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, pp. 1573–1582. ACM (2016)

He, P., Zhu, J., He, S., Li, J., Lyu, M.R.: Towards automated log parsing for large-scale log data analysis. IEEE Trans. Dependable Secure Comput. **15**(6), 931–944 (2018)

He, P., Zhu, J., Zheng, Z., Lyu, M.R.: Drain: An online log parsing approach with fixed depth tree. In: Web Services (ICWS), 2017 IEEE International Conference on, pp. 33–40. IEEE (2017)

He, S., Zhu, J., He, P., Lyu, M.R.: Experience report: system log analysis for anomaly detection. In: Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on, pp. 207–218. IEEE (2016)

Huang, W.: A blockchain-based framework for secure log storage. In: 2019 IEEE 2nd International Conference on Computer and Communication Engineering Technology (CCET), pp. 96–100. IEEE (2019)

Jiang, Z.M., Hassan, A.E., Hamann, G., Flora, P.: An automated approach for abstracting execution logs to execution events. J. Softw. Mainten. Evolut. Res. Pract. **20**(4), 249–267 (2008)

Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J., Tang, L.: Neurosurgeon: collaborative intelligence between the cloud and mobile edge. ACM SIGARCH Comput. Architect. News **45**(1), 615–629 (2017)

Liu, J., Ren, J., Dai, W., Zhang, D., Zhou, P., Zhang, Y., Min, G., Najjari, N.: Online multi-workflow scheduling under uncertain task execution time in iaas clouds. IEEE Transactions on Cloud Computing (2019)

Lou, J.G., Qiang, F., Yang, S., Jiang, L., Wu, B.: Mining program workflow from interleaved traces. In: ACM Sigkdd International Conference on Knowledge Discovery & Data Mining (2010)

Lyu, F., Ren, J., Cheng, N., Yang, P., Li, M., Zhang, Y., Shen, X.: Lead: large-scale edge cache deployment based on spatio-temporal wifi traffic statistics. IEEE Trans. Mob. Comput. (2020)

Makanju, A., Zincir-Heywood, A.N., Milios, E.E.: A lightweight algorithm for message type extraction in system application logs. IEEE Trans. Knowl. Data Eng. **24**(11), 1921–1936 (2012)

Messaoudi, S., Panichella, A., Bianculli, D., Briand, L., Sasnauskas, R.: A search-based approach for accurate identification of log message formats. In: Proceedings of the 26th IEEE/ACM International Conference on Program Comprehension (ICPC18). ACM (2018)

Min, D., Li, F.: Spell: Streaming parsing of system event logs. In: IEEE International Conference on Data Mining (2017)

Mizutani, M.: Incremental mining of system log format. In: Services Computing (SCC), 2013 IEEE International Conference on, pp. 595–602. IEEE (2013)

Nagappan, M., Vouk, M.A.: Abstracting log lines to log event types for mining software system logs. In: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, pp. 114–117. IEEE (2010)

Osia, S.A., Shamsabadi, A.S., Sajadmanesh, S., Taheri, A., Katevas, K., Rabiee, H.R., Lane, N.D., Haddadi, H.: A hybrid deep learning architecture for privacy-preserving mobile analytics. IEEE Internet of Things Journal (2020)

Osia, S.A., Taheri, A., Shamsabadi, A.S., Katevas, K., Haddadi, H., Rabiee, H.R.: Deep private-feature extraction. IEEE Trans. Knowl. Data Eng. **32**(1), 54–66 (2018)

Pourmajidi, W.: Scalable blockchain-assisted log storage system for cloud-generated logs (2018)

Pourmajidi, W., Miranskyy, A.: Logchain: Blockchain-assisted log storage. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 978–982. IEEE (2018)

Pourmajidi, W., Zhang, L., Steinbacher, J., Erwin, T., Miranskyy, A.: Immutable log storage as a service. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 280–281. IEEE (2019)

Rane, S., Dixit, A.: Blockslaas: Blockchain assisted secure logging-as-a-service for cloud forensics. In: International Conference on Security&amp; Privacy, pp. 77–88. Springer (2019)

Ren, J., Zhang, D., He, S., Zhang, Y., Li, T.: A survey on end-edge-cloud orchestrated network computing paradigms: transparent computing, mobile edge computing, fog computing, and cloudlet. ACM Comput. Surv. (CSUR) **52**(6), 1–36 (2019)

Shao, W., Wang, Z., Wang, X., Qiu, K., Jia, C., Jiang, C.: Lsc: online auto-update smart contracts for fortifying blockchain-based log systems. Inf. Sci. **512**, 506–517 (2020)

Shima, K.: Length matters: clustering system log messages using length of words. arXiv:1611.03213 (2016)

Tang, L., Li, T., Perng, C.S.: Logsig: Generating system events from raw textual logs. In: Proceedings of the 20th ACM international conference on Information and knowledge management, pp. 785–794. ACM (2011)

Tang, W., Ren, J., Zhang, K., Zhang, D., Zhang, Y., Shen, X.: Efficient and privacy-preserving fog-assisted health data sharing scheme. ACM TIST **10**(6), 1–23 (2019)

Thomas, A., Guo, Y., Kim, Y., Aksanli, B., Kumar, A., Rosing, T.S.: Hierarchical and distributed machine learning inference beyond the edge. In: 2019 IEEE 16th International Conference on Networking, Sensing and Control (ICNSC), pp. 18–23. IEEE (2019)

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82 (2018)

Vaarandi, R.: A data clustering algorithm for mining patterns from event logs. In: IP Operations & Management, 2003.(IPOM 2003). 3rd IEEE Workshop on, pp. 119–126. IEEE (2003)

Vaarandi, R., Pihelgas, M.: Logcluster-a data clustering and pattern mining algorithm for event logs. In: Network and Service Management (CNSM), 2015 11th International Conference on, pp. 1–7. IEEE (2015)

Wang, H., Yang, D., Duan, N., Guo, Y., Zhang, L.: Medusa: Blockchain powered log storage system. In: 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS), pp. 518–521. IEEE (2018)

Xiao, Y., Joshi, P., Xu, J., Jin, G., Hui, Z., Jiang, G.: Cloudseer: workflow monitoring of cloud infrastructures via interleaved logs. ACM Sigarch Comput. Architect. News **44**(2), 489–502 (2016)

Xie, X., Jin, Z., Han, Q., Huang, S., Li, T.: A confidence-guided anomaly detection approach jointly using multiple machine learning algorithms. In: International symposium on cyberspace safety and security, pp. 93–100. Springer (2019)

Xie, X., Jin, Z., Wang, J., Yang, L., Lu, Y., Li, T.: Confidence guided anomaly detection model for anti-concept drift in dynamic logs. Journal of Network and Computer Applications, pp. 102659 (2020)

Xie, X., Wang, Z., Xiao, X., Lu, Y., Huang, S., Li, T.: A confidence-guided evaluation for log parsers inner quality. Mobile Networks and Applications, pp. 1–12 (2020)

Xu, S., Qian, Y., Hu, R.Q.: Data-driven network intelligence for anomaly detection. IEEE Netw. **33**(3), 88–95 (2019)

Xu, X., Pautasso, C., Zhu, L., Lu, Q., Weber, I.: A pattern collection for blockchain-based applications. In: Proceedings of the 23rd European Conference on Pattern Languages of Programs, pp. 1–20 (2018)

Yin, H., Wang, Z., Jha, N.K.: A hierarchical inference model for internet-of-things. IEEE Trans. Multi-Scale Comput. Syst. **4**(3), 260–271 (2018)

Zhang, L., Xie, X., Xie, K., Wang, Z., Lu, Y., Zhang, Y.: An efficient log parsing algorithm based on heuristic rules. In: International Symposium on Advanced Parallel Processing Technologies, pp. 123–134. Springer (2019)

Zhou, Z., Chen, X., Li, E., Zeng, L., Luo, K., Zhang, J.: Edge intelligence: paving the last mile of artificial intelligence with edge computing. Proc. IEEE **107**(8), 1738–1762 (2019)

Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., Lyu, M.R.: Tools and benchmarks for automated log parsing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 121–130. IEEE (2019)