DELIA: Distributed Efficient Log Integrity Audit Based on Hierarchal Multi-Party State Channel

Jing Chen, Xin Chen, Kun He, Ruiying Du, Weihang Chen, and Yang Xiang, *Fellow, IEEE*

Abstract—Audit log contains the trace of different activities in computing systems, which makes it critical for security management, censorship, and forensics. However, experienced attackers may delete or modify the audit log after their attacks, which makes the audit log unavailable in attack investigation. In this article, we focus on the log integrity audit in the same domain, in which a number of servers update audit logs for a single or several organizations as an alliance. We propose a distributed efficient log integrity audit framework, called DELIA, which employs the distributed ledger technique to protect audit information, and utilizes the idea of state channel to improve the throughput of distributed ledger. To generate stable state from the rapidly-updated logs in the domain, we propose a log state generation scheme, which not only generates state suitable for audit logs, but also enables mutual supervision within the domain. To overcome the high latency in existing state channel schemes, we propose a hierarchal multi-party state channel scheme, which makes the latency in our framework independent of the number of servers in the domain. We implement DELIA on Ethereum and evaluate its performance. The results show that our framework is efficient and secure in practice.

Index Terms—Audit log, integrity, blockchain, state channel

◆

1 INTRODUCTION

Audit log is a set of security-relevant chronological records, that can be used to reconstruct the events of a computing system for intrusion detection and digital forensics. Especially, organizations usually take advantage of audit logs which are generated from a number of servers to detect attacks. Those servers (e.g., web server, firewall, and intrusion detection system) form a *domain* in an organization. Researchers have shown that by employing attack investigation technique, such as causality analysis, administrators can trace back many attacks by audit logs, even Advanced Persistent Threat (APT) [1], [2].

However, experienced attackers may delete or modify audit log to hide their tracks and hinder the attack investigation [3], [4] by launching penetration testing tools like Metasploit [5], or downloading a simple script [6], [7], [8], [9]. Attackers regularly engage in anti-forensic activities to cover up their attacks. Log tampering is reported as the top evasion tactic by 87 percent of incident response specialists [10]. In 2017, criminals exploited Amazon's multiple vulnerabilities to defraud users, and then they tampered with logs, which makes reverse trace very difficult for Amazon [11]. In 2020, since audit logs were erased, Nintendo realized that it was attacked by hackers until many users complained on the forum about the loss of their account funds [12]. Obviously, the integrity of audit log is a key factor which needs to be guaranteed. Most of traditional audit methods store logs locally in a centralized mode. From the perspective of attackers, once they can compromise a target machine, it is easy for them to delete or modify logs which are the last line of defense to record actual operational behaviors. As a result, attackers leave no suspicious traces and the victim is hard to realize it has been attacked. The main reason is that the trust of the system depends on a single and local server which cannot prove the innocence for itself. To solve this problem, trust dispersion is a good choice.

One option is to outsource the audit log to the cloud, and employ Provable Data Possession (PDP) or Proof of Retrievability (PoR) to examine the log integrity [13], [14]. Unfortunately, since audit log are rapidly-updated, those approaches may introduce unaffordable time for verifying the log integrity, and outsourcing audit data to a third party poses a certain threat to sensitive logs [15]. Another option is to employ the distributed ledger technique, whose security depends on the majority of nodes but not a single one [16]. In [17], [18], and [19], audit logs or the checksums are stored in a distributed ledger. When an auditor needs to examine the log integrity, it compares the ledger data with the audit log to estimate whether the audit log has been modified. Since existing distributed ledger techniques require that every record (or the checksum of the record) is spread to the whole network and stored to the ledger by all participants' consensus, those solutions are inefficient in handling rapid log update events in practice.

A prominent approach to handle the rapidly-updated events in distributed ledger is *state channel* [20], which is a contract instantiated between two parties. It allows the two

---

- *Jing Chen is with the Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China. E-mail: chenjing@whu.edu.cn.*
- *Xin Chen, Kun He, and Weihang Chen are with the School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China. E-mail: cvb543709193@sina.com, {hekun, 2019202210085}@whu.edu.cn.*
- *Ruiying Du is with the Collaborative Innovation Center of Geospatial Technology, Wuhan 430072, China. E-mail: duraying@126.com.*
- *Yang Xiang is with the School of Software and Electrical Engineering, Swinburne University of Technology, Hawthorn VIC 3122, Australia. E-mail: yxiang@swin.edu.au.*

Manuscript received 24 Sept. 2020; revised 2 May 2021; accepted 21 June 2021. Date of publication 25 June 2021; date of current version 2 Sept. 2022.
(Corresponding author: Kun He.)
Digital Object Identifier no. 10.1109/TDSC.2021.3092365

parties at a time to collaboratively maintain a state (e.g., transactions) off-chain. When this approach is applied to the log integrity audit, we can view all the interactive behaviors of the two servers of the same domain within a period of time as a state, which includes the log update, message signature verification, and etc. Since the state is maintained off-chain, the on-chain efficiency is improved. Intuitively, a multiple-party state channel is needed for flexible and rapid interaction combination with different parties in practice. Unfortunately, existing multi-party state channel schemes are exposed to the following two challenges.

The first challenge concerns the state generation, called *state instability*. We observe that audit logs are rapidly-updated in a domain, which makes existing multi-party state channel schemes difficult to determine the current state of the domain. The second challenge concerns the delay of consistency in the domain, called *linear latency*. Even if the state of the domain is determined, existing multi-party state channel schemes require each participant to generate a signature on the state to reach a consensus. This mechanism induces that the system latency grows linearly with the number of participants in multi-party state channel.

In this paper, we aim to propose a Distributed Efficient Log Integrity Audit (DELIA) framework, which guarantees the integrity of audit logs in a domain. Our framework benefits from the distributed ledger technique to protect the verification information of audit logs. First, we design the state generation and verification method of audit log based on state channel. Then, we make state channel suitable for large-scale data operations through our improvement of state channel technology. In summary, we make the following contributions.

1) To the best of our knowledge, it is the first work to propose a Distributed Efficient Log Integrity Audit (DELIA) framework, which provides mutual supervision within a domain. Through our design, log deletion and modification can be detected efficiently by either the servers in the domain or an external auditor.
2) To solve the state instability challenge, we propose a Log State Generation scheme, called LSG. In LSG, we design a number of data structures to represent the current state of the audit logs in a domain. We also propose a verification method over these structures to enable quick integrity verification within the domain.
3) To solve the linear latency challenge, we propose a Hierarchal Multi-party State Channel scheme, called HMSC. In HMSC, all operations for states only induce tiny latency, which is close to constant level.
4) We implement a prototype on Ethereum to evaluate the performance of our DELIA framework in practice. The experimental results show that DELIA is efficient and effective.

2 PROBLEM STATEMENT

2.1 System Model

In our system, there are three kinds of entities: a number of servers which consist of LSG and HMSC modules, an auditor, and a distributed ledger which is maintained by a number of ledger nodes, as shown in Fig. 1. Considering the sensitivity of logs, all participants in different entities should be authenticated and authorized and to form an alliance in the same domain.

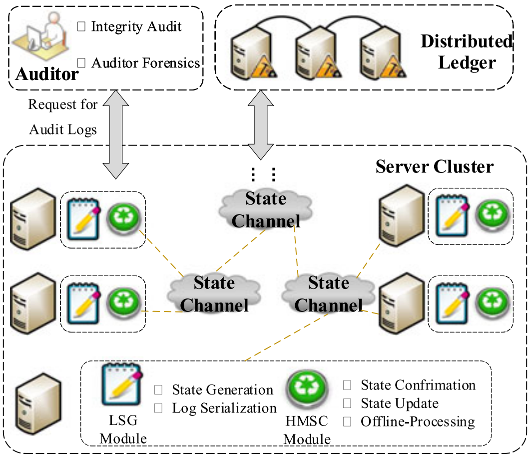


Fig. 1. System model.

The server updates audit logs and interacts with other servers in the domain. The auditor can censor all audit logs, local state information, and the HMSC instance in distributed ledger automatically in cycles. The distributed ledger pre-deploys the HMSC contract and maintains the submitted state in a HMSC instance by all the ledger nodes.

2.2 Threat Model

- Network monitoring attack: Attackers can eavesdrop, delete or modify data transmitted between the log servers and distributed ledger nodes.
- Compromise attack: Attackers can compromise minority servers in the domain. In another words, most of participants in different entities of the domain are honest.
- Sybil attack: Attackers may leverage multiple forged identities to induce calculation errors and information inconsistency in the network.
- Denial of Service (DoS) attack: Attacks can be launched at anytime after initialization, and attackers may cause block access to distributed ledger.

However, we assume that cryptographic algorithms are secure in finite polynomial time, which means that attackers cannot forge/tamper signatures of encrypted messages without the corresponding keys.

2.3 Design Goals

- *Mutual supervision.*The integrity of audit logs is supervised mutually among the servers and distributed ledger nodes in a domain. It allows collaborative attack investigation in that domain.
- *State stability.*The state is generated stably from the rapidly-updated audit logs on various servers. Moreover, the state integrity can be efficiently verified within the domain.
- *Low latency.*The latency in reaching a consensus on a state is independent of the number of participants in

TABLE 1
Notations

Notation	Meaning
n	the number of log servers
p_i	the ith log server
$\delta_{p_i,\xi}$	the ξth record of the ith log server
ψ_{p_i,l_i}	`LogCacheQueue` generated by ith server in l_ith batch
θ	the number of log records in a batch
ω_{p_i,l_i}	hash chain of logs in the l_ith batch of ith server
G_ν	νth global state recording all `LocalState` in a domain
$G_\nu^{\Gamma_x}$	νth global state with signatures from all participants in Γ_x
sid	the unique identity of a state channel instance
Γ	a state channel instance stored in distributed ledger
β	the number of state in a round of HMSC
Δ	the number of layer in HMSC

state channel if there is no dispute. Moreover, the dispute can be resolved efficiently.

- *Offline process.*The disconnection of any server can be detected, and the security of its audit logs can be guaranteed during offline period.

2.4 Notation

To facilitate the understanding, we summarize the main notations in this paper in Table 1.

3 DELIA FRAMEWORK

3.1 Overview

The Distributed Efficient Log Integrity Audit (DELIA) framework is a three-layer architecture including *data layer*, *network layer*, and *audit layer*, as shown in Fig. 3.

In the data layer, we design data structures used in following layers. In the network layer, we show *Log State Generation* (LSG) scheme, which can handle rapidly-updated audit logs in the domain, and integrate the LSG scheme with the distributed ledger by our *Hierarchal Efficient Multi-party State Channel* (HMSC) scheme. In the audit layer, we integrate the distributed ledger with audit process by HMSC scheme, and present a verification scheme to check the integrity of audit logs. The interaction processes are shown in Fig. 4.

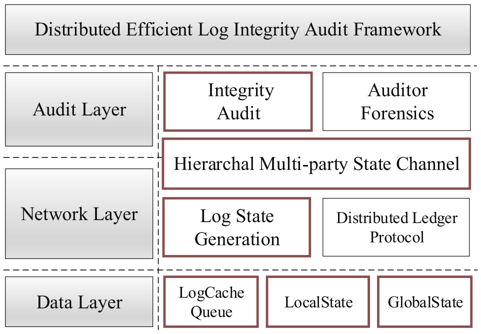


Fig. 3. Distributed efficient log integrity audit framework.

3.2 Data Layer

To handle rapidly updated audit logs in the domain, we introduce three data structures in data layer: `LogCacheQueue`, `LocalState`, and `GlobalState`. To illustrate these data structures, we provide a four-sever example in Fig. 2. The detailed design is as follows.

1) `LogCacheQueue`: Since the audit logs are rapidly-updated, we can neither store all of them in the distributed ledger, nor process them in the state channel directly. To tackle this issue, we collect the audit logs over in the form of a batch which has a fix number of records and is treated as a whole. `LogCacheQueue` is a local hash set, which stores different hash values of each server's log records in a batch. Let p_i ($i \in [1,n]$, where n is the number of servers) be the ith server in a domain and $\delta_{p_i,\xi}$ be the ξth log record in server p_i. LogCacheQueue, ψ_{p_i,l_i} denotes the hash set of the log records of server p_i in a batch. More specifically

$$\psi_{p_i,l_i} = \{H(\delta_{p_i,l_i\theta+1}),\dots,H(\delta_{p_i,(l_i+1)\theta})\}, \tag{1}$$

where l_i is a non-negative integer that denotes the number of batch on the ith server, θ is a global parameter that indicates the number of log records in a batch, and $H(\cdot)$ is a cryptographic hash function.

2) `LocalState`: `LocalState` is a hash chain which links the data of `LogCacheQueue` in the order of batches. To support state channel technique, `LogCacheQueue` on a server needs to be transformed into `LocalState`, which not only contains the information of current batch in `LogCacheQueue`, but also binds all previous batches of audit logs. Let $root_{p_i,l_i}$ be the root of the Merkle hash tree constructed from a `LogCacheQueue` ψ_{p_i,l_i}. We define `LocalState` as follows:

$$\omega_{p_i,l_i} = \begin{cases} H(root_{p_i,0}) & if \quad l_i = 0, \\ H(root_{p_i,l_i} \| \omega_{p_i,l_i-1}) & if \quad l_i > 0. \end{cases} \tag{2}$$

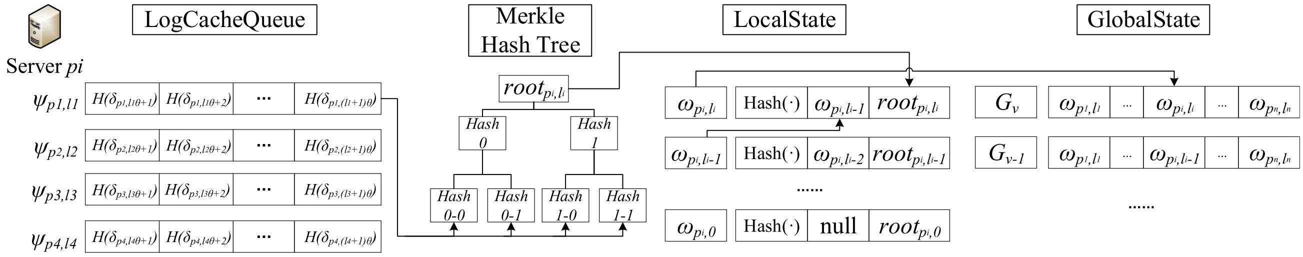


Fig. 2. Transformation among three data structures.

3) `GlobalState`: `GlobalState` is an array recording the `LocalState` of all servers in a domain. To describe the current state of the domain, we denote `GlobalState` as

$$G_\nu = (\omega_{p_1,l_1}, \ldots, \omega_{p_n,l_n}), \tag{3}$$

where ν is its serial number which update a snapshot. That means, `GlobalState` records is a snapshot of current `LocalState` data of all the servers periodically.

3.3 Network Layer

The network layer is mainly responsible for secure data interaction under the supervision of the distributed ledger to ensure the integrity of audit logs. Note that if log records are only generated by a server, attackers may delete or modify these data in a batch. To tackle this issue, we utilize the collaborative monitoring and a real-time broadcasting method among all servers in LSG, which prevent attackers from disturbing the generation of audit logs.

Network layer refers to two phases: LSG phase and HMSC phase. LSG phase consists of two stages: log serialization and state generation. HMSC consists of three stages: off-chain state confirmation, on-chain state update, and off-line processing. LSG phase is only running at the network layer and HMSC phase is realized by the cooperation of network layer and audit layer.

LSG phase (stage ① - ②)

① Log Serialization

To prevent attackers from deleting or modifying audit logs, every server in the domain should maintain the hash values of other servers' log records in a batch to form the data in `LogCacheQueue`. Specifically, the server p_j maintains $\{\psi_{p_1,l_1}, \ldots, \psi_{p_n,l_n}\}$. The details of the log serialization stage is as follows.

- The server p_i broadcasts $H(\delta_{p_i,\xi})(\xi \in [l_i\theta + 1, (l_i + 1)\theta])$ to other servers in the domain once the ξth log record is generated.
- When the server p_j receives $H(\delta_{p_i,\xi})$, it stores this item in its local `LogCacheQueue`.
- The server p_i also stores this item in its local `LogCacheQueue`.

Algorithm 1. Global State Generation

```
procedure GlobalGen$root_{p_i,l_i}, G_{\nu-1}$
    Parse $G_{\nu-1}$ as $(\omega_{p_1,l_1}, \ldots, \omega_{p_i,l_i-1}, \ldots, \omega_{p_n,l_n})$;
    $\omega_{p_i,l_i} \leftarrow H(root_{p_i,l_i} \| \omega_{p_i,l_i-1})$;                ▷ Eq. (2)
    $G_\nu \leftarrow (\omega_{p_1,l_1}, \ldots, \omega_{p_i,l_i}, \ldots, \omega_{p_n,l_n})$;        ▷ Eq. (3)
    return $G_\nu$;
```

Algorithm 2. State Verification

```
procedure StateVerify$\psi_{p_i}, root_{p_i,l_i}, G_{\nu-1}, G_\nu$
    $root'_{p_i,l_i} \leftarrow$ MHTGen$(\psi_{p_i})$;
    if $root'_{p_i,l_i} \neq root_{p_i,l_i}$ then
        return 0;                                   ▷ invalid root
    $G'_\nu \leftarrow$ GlobalGen$(root_{p_i,l_i}, G_{\nu-1})$
    if $G'_\nu \neq G_\nu$ then
        return 0;                           ▷ invalid global state
    return 1;                                 ▷ valid global state
```

② State Generation

Once the server p_i has generated θ records in its l_ith batch which means that the batch is full, it generates a new GlobalState G_ν. MHTGen$(\cdot)$ denotes a function that takes ψ_{p_i,l_i} as input, and outputs the root value $root_{p_i,l_i}$ of the Merkle hash tree as `LocalState`.

- The server p_i invokes MHTGen(ψ_{p_i,l_i}) and obtains $root_{p_i,l_i}$. Then, it invokes Algorithm 1 to obtain `GlobalState` G_ν, where $G_{\nu-1}$ is the latest `GlobalState`. Finally, p_i broadcasts $root_{p_i,l_i}$ and G_ν to other servers in the domain.
- When the server p_j receives $root_{p_i,l_i}$ and G_ν, it calls Algorithm 2 to verify $root_{p_i,l_i}$ and G_ν, where ψ_{p_i,l_i} and $G_{\nu-1}$ are maintained locally on p_j. If they are invalid, p_j broadcasts a failure notification and terminate this stage. Otherwise, its local $G_{\nu-1}$ is replaced by G_ν. Note that this stage provides mutual supervision within the domain.

HMSC phase (stage 3-5)

③ Off-Chain State Confirmation. If Algorithm 2 passes, all servers try to reach a consensus on G_ν in the domain. This process is to confirm the state G_ν generated by LSG phase, and it is implemented by the state confirmation protocol of HMSC described in Section 4.3.2.

Algorithm 3. Log Verification

```
procedure LogVerify$\{\delta_{p_i,1}, \ldots, \delta_{p_i,\xi}\}, \{G_1, \ldots, G_\nu\}$
    $ErrList \leftarrow \emptyset$;        ▷ batches in which audit log is deleted/forged/tampered
    $k \leftarrow 1, l \leftarrow 0$;
    for $k \leq \nu$ do
        Parse $G_k$ as $(\omega_{p_1,l_1}, \ldots, \omega_{p_i,l_i}, \ldots, \omega_{p_n,l_n})$;
        for $l \leq l_i$ do
            $\psi \leftarrow \{H(\delta_{p_i,l\theta+1}), \ldots, H(\delta_{p_i,(l+1)\theta})\}$;        ▷ Eq. (1)
            $root \leftarrow$ MHTGen$(\psi)$;
            if $l == 0$ then
                $\omega \leftarrow H(root)$;
            else
                $\omega \leftarrow H(root \| \omega)$;                              ▷ Eq. (2)
        if $\omega \neq \omega_{p_i,l_i}$ then
            $ErrList \leftarrow ErrList \cup \{l_i\}$;
        $\omega \leftarrow \omega_{p_i,l_i}$
    return $ErrList$;
```

④ On-chain State Update. When the off-chain state channel has been created, the servers as participants in the state channel can record their own operations in audit logs autonomously. To maintain the synchronization between the distributed ledger and the state channel, we could periodically update the state information by state update protocol of HMSC described in Section 4.3.3.

⑤ Offline-processing. When a server is disconnected from networks or the server is down, we need to ensure that our framework still works properly and the data of offline nodes can be protected. It consists of two parts: *Offline Detection* and *Online Notification*.

- Offline Detection: Once a server loses contact with other servers in this domain, this process is triggered. Other servers confirm this server offline through the inquiry mechanism, and ensure the information of its log

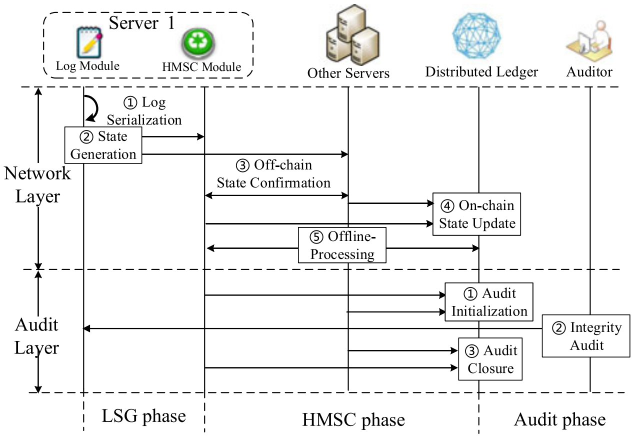


Fig. 4. Entity interactions in network layer and audit layer.

integrity previously stored in the state channel cannot be tampered. It is implemented by the offline state confirmation protocol of HMSC in Section 4.3.5.

- **Online Notification:** Once a server comes back, this server sends notifications to other servers in the domain. Online state confirmation in Section 4.3.5 is called to make the server rejoin the state channel. After the offline server becomes online, this server needs to synchronize all the `GlobalState` data and `LogCacheQueue` data from other servers in the domain.

3.4 Audit Layer

The audit layer is designed to provide audit function, which consists of three stages: *audit initialization*, *integrity audit*, and *audit closure*. Specially, as shown in Fig. 4, HMSC phase refers to the first and third stages.

① **Audit Initialization.** In this stage, the servers in a domain enable the audit function including deployment of log collector, log preprocessor and integrity verification modules. Then, the servers in a domain can initialize different state channel instances according to the network division,which is implemented in instance initialization protocol of HMSC described in Section 4.3.1. This protocol is to create a state channel instance or form a higher layer state channel instance for state interactions between servers and distributed ledger.

② **Integrity Audit.** At any time, an external auditor could access the audit phase to check the integrity of audit logs on a server. To examine the log integrity on p_i, the auditor first obtains $\{\delta_{p_i,1},\ldots,\delta_{p_i,\xi}\}$ from p_i and the states $\{G_1,\ldots,G_v\}$ from the distributed ledger, and then calls Algorithm 3. The output of Algorithm 3 is the batches in which audit logs are deleted/forged/tampered. In other words, the log integrity audit is passed if and only if the output of Algorithm 3 is null.

③ **Audit Closure.** When the servers do not need log integrity protection or the participants of the alliance do not need mutual supervision, log collector, log preprocessor and integrity verification module in each server are closed, and the instance closure protocol of HMSC is invoked to disable the audit function in the state channel, which is implemented by Section 4.3.6.

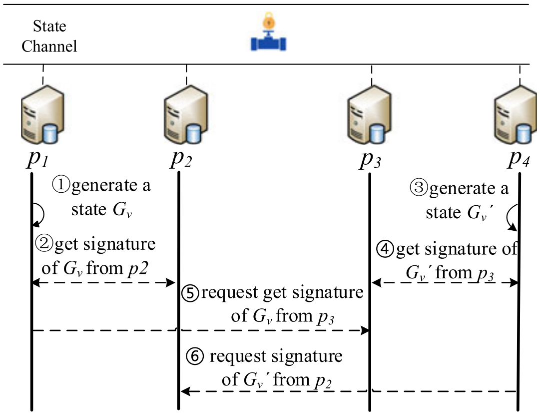


Fig. 5. Consistency problem in the multi-party state channel.

4 HIERARCHAL MULTI-PARTY STATE CHANNEL

4.1 Motivation

The multi-party state channel brings a new perspective to the expansion of distributed ledger. To confirm a state G_v, every participant p_i should make a signature S_{p_i} for the state. The current multi-party state channel is composed by multiple two-party state channels, which means the communication and computation costs linearly increase according to the number of involved participants. Thus, when the scale of participants expands, the verification time and system latency will become unacceptable.

Moreover, since the initilization and closure of each state channel need to consume certain resources, we need to avoid execute these operations frequently. Obviously, if we can find a non-serial model to create multi-party state channels and combine them dynamicly, the system efficiency will be improved significantly.

However, the non-serial model also induce some problems. In the state channel, the state confirmation need the consensus of all members. Due to the network delay, it easily leads to inconsistencies, called $state\ conflict$.

To describe this problem, we illustrate an example that a state channel has four participants p_1, p_2, p_3, and p_4 as in Fig. 5. Suppose p_1 generates a new state G_v and requests a signature S_{p_2} from p_2. Simultaneously, p_4 generates a new state G_v' and requests a signature S_{p_3} from p_3. When p_3 receives (G_v, S_{p_1}, S_{p_2}), p_3 has already owned another state (G_v', S_{p_4}, S_{p_3}). These two states have the same serial number and the same signature quantity. For p_2, it has the same problem. As a result, p_2 and p_3 confuse to decide the correct state, which leads to $state\ conflict$.

Therefore, how to design an efficient, robust, and secure multi-party state channel is still a open problem which is worth studying. Following the principle of cooperative autonomous, we need to design a non-serial multi-party state channel. To avoid the state conflict, a rotating leader is elected to organize the state channel. For the requirement of dynamic combination, we form a hierarchical channel through the leader to improve the joint efficiency and enhance the scalability.

4.2 Overview of HMSC

To solve the above mentioned problems, we design an efficient hierarchal multi-party state channel scheme, called

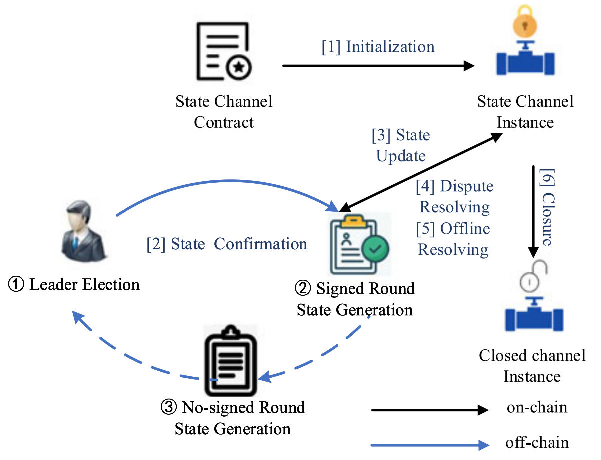


Fig. 6. An overview of HMSC.

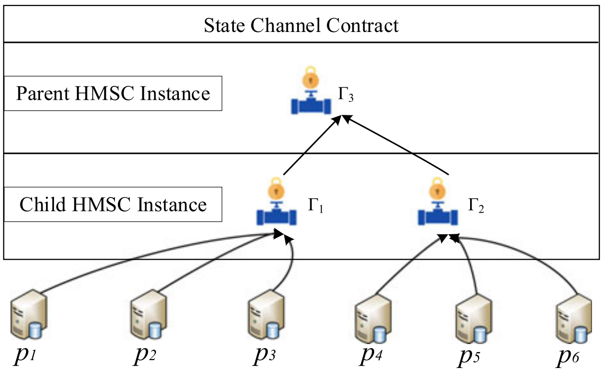


Fig. 7. A two-layer example of HMSC.

HMSC. Specially, it is implemented by a smart contract which consists of executable codes pre-deployed on a distributed ledger. In HMSC, we exploit a state channel contract (SCC) includes six protocols: instance initialization (Init), state confirmation (Confirm), state update (Update), dispute resolving (DisputeResolving), instance closure (Close), and offline resolving (OfflineResolving), as shown in Fig. 6.

Since state channels are temporary unions, which are composed by parts of participants, to finish special tasks, we can launch multiple state channels simultaneously. When different participants need to cooperate with each other, i.e., joint audit, the efficient way is to form federal state channels but not close the old ones and create a new one each time. As shown in Fig. 7, it is an example with two state channels Γ_1 (p_1, p_2, p_3) and Γ_2 (p_4, p_5, p_6). Because each state channel is independent and autonomous, the leader selected by each state channel is used to build a hierarchical state channel and coordinate internal interactions. When Γ_1 and Γ_2 need to be federated, as a hierarchical structure, their leaders negotiate a temporary federation Γ_3. As a result, Γ_3 can manage six participants $p_1, p_2, \ldots, p_6$ uniformly which not only promotes efficiency, but also improves the consensus strength. In Γ_3, each state should be approved in the whole federation which means that more monitors are involved.

4.3 Protocol Design

4.3.1 Instance Initialization Protocol

This protocol aims to create a state channel instance, which ensures that pre-negotiated participants could join the same instance. Suppose p_1, ..., p_n intend to join the same instance. Let $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a digital signature scheme (e.g., ECDSA), in which Gen is the key generation algorithm, Sign is the signing algorithm, and Vrfy is the signature verification algorithm. Each participant p_i has its own public key PK_{p_i} and private key SK_{p_i} that are generated by Gen. Let (G_v, S_{p_i}) be the state and signature of the state using SK_{p_i}. The participants could send a message InitMsg composed of $\mathcal{P} = \{PK_{p_1}, \ldots, PK_{p_n}\}$ to SCC. Then, a state channel instance $\Gamma := \{sid, \mathcal{P}, G, \mathsf{OffList}, \mathsf{NewLeader}, \mathsf{Parent}, \mathsf{ChildList}\}$ is created in SCC, where sid represents the unique identity of the instance, $\mathcal{P}$ is the set of all participants' public keys in this instance, G represents the latest state in the instance, OffList is a list of offline servers, NewLeader is a reserved field for dispute resolving protocol, Parent reprensents the identity of parent instance, and ChildList reprensents the identity list of all child instances. Specially, OffList, NewLeader, Parent, and ChildList are null in initialization process.

To federate multiple state channels, a parent state channel needs to be generated. This process could be executed in initialization or subsequent processes. Suppose Γ_1, ..., Γ_m intend to form a hierarchal state channel, then their leaders should send a message FederateMsg composed of $(sid_1, \ldots, sid_m, \mathcal{P}_1, \ldots, \mathcal{P}_m)$ to SCC. After that, a parent state channel instance $\Gamma_a := \{sid_a, \mathcal{P}_a, G, \mathsf{OffList}_a, \mathsf{NewLeader}_a, \mathsf{Parent}_a, \mathsf{ChildList}_a\}$ is stored in SCC, where sid_a is randomly generated, $\mathcal{P}_a$ are the set of participants' public keys in child state channels, Parent_a is null, and $\mathsf{ChildList}_a$ includes $sid_1, \ldots, sid_m$ which belong to its child state channel instances. In Fig. 7, we show a two-layer example of HMSC. p_1, p_2, p_3 form Γ_1, p_4, p_5, p_6 form Γ_2, and Γ_3 is federated by Γ_1 and Γ_2. The state channel instance at top of HMSC is called root state channel instance.

4.3.2 State Confirmation Protocol

In this protocol, participants aim to reach a consensus on a given state G_v. In HMSC, every instance runs the state confirmation protocol by rounds. To solve the state conflict problem, there would be a leader to manage the state channel in a round. Specially, the leader responds to coordinates other participants to generate a signed round state and β no-signed round states (β is not fixed in different state channel instances). The difference of these two types of round states is that the former needs the signatures of all participants and the latter only needs the signature of the leader. We use $G_v^{\Gamma_x}$ to represent signed round state, which includes a state generated in Γ_x with signatures from its participants or leaders of its child state channel instances. Obviously, the signed round state is more secure but the no-signed round state is more efficient. To obtain better synthetic ability, we use the former as secure anchor and the latter as temporary memory point. Once any participant finds abnormal situation, it can call the dispute resolving protocol described in Section 4.3.4 to roll back to the latest secure anchor and exclude the inconsistent participant. The state confirmation protocol in a round consists of three processes:

① Leader Election: At the beginning of a round in every state channel instance, a leader should be elected. To achieve election fairness, every participant should have the same opportunity as the leader. The system chooses the participant

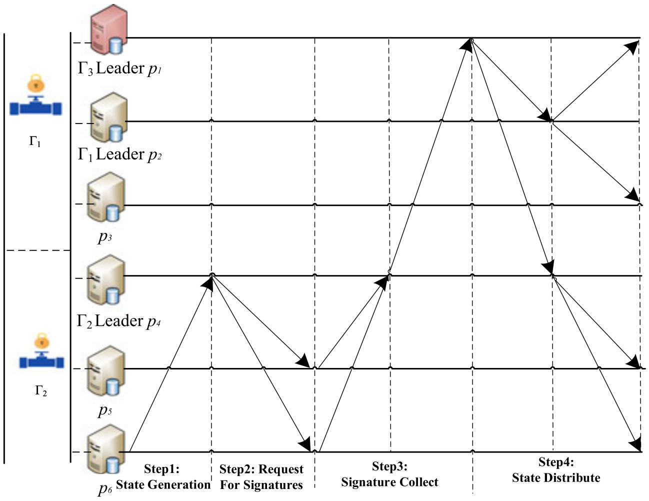


Fig. 8. Signed round state generation.

with the minimum value of PK_{p_i} in $\mathcal{P}$ as the leader. This leader will not join the election of next round, and the next leader is the participant with the next smallest PK_{p_i} in $\mathcal{P}$. If no such participant exists, the system starts over from the minimum value in $\mathcal{P}$.

② Signed Round State Generation: In this process, there are four steps. To illustrate clearly, we take an example of a state channel instance formed by six servers as shown in Fig. 8, while p_1 is elected as the leader in Γ_3 which is root state channel instance, p_2 is leader of Γ_1, and p_4 is leader of Γ_2.

Step 1 : The participant p_6 in Γ_2 generates G_ν and sends it to its leader p_4. Step 2: This leader p_4 signs $(G_\nu|p_4)$ and broadcasts (G_ν, S_{p_4}) to all its participants or leaders of state channel instances in its ChildList. All participants or leaders of child state channel instances verify the leader's signature and generate their own signatures. Step 3: p_4 collects signed states which include signatures from all participants or leaders of child state channel instance. Then p_4 sends $G_\nu^{\Gamma_2}$ to its parent leader, and this step stops until the $G_\nu^{\Gamma_2}$ reaches root state channel instance Γ_3. Step 4: The leader of root state channel instance p_1 distributes this signed round state $G_\nu^{\Gamma_2}$ to the participants in all its subordinative state channels by this step recursively.

③ No-signed Round State Generation: Once the signed round state generation is completed, the leader p_4 in this state channel instance can still conduct β no-signed round states in the rest of this round. Every state channel instance could have different value of β. Every leader in HMSC receiving a no-signed round state sends it to the leader of root state channel instance recursively. When the leader of root state channel instance approves a no-signed round state, it broadcasts the state to all participants through its descendant leaders.

Let G_{ν^*} be the latest state conducted by p_1, and $G_{\nu'}$ be the new state from p_1. If $\nu^* < \nu + \beta$ and ν' is equal to $\nu^* + 1$, p_1 signs $G_{\nu'}$ with its private key SK_{p_1} and broadcasts $(G_{\nu'}, S_{p_1})$ to all participants.

When other participants receive $(G_{\nu'}, S_{p_1})$, they can check whether the serial number of their local latest state is equal to $\nu' - 1$. If the equation does not hold, it means the participants are not synchronized and they need to interact with their leaders to synchronize all the states to their local storage.

4.3.3 State Update Protocol

In this protocol, the state changed by participants can be recorded to the distributed ledger. Only signed round states can be accepted by the distributed ledger, and the no-signed round states are stored off-chain as temporary cache. A participant could submit a signed round state to its state channel instance. Let Γ_x be a state channel instance in HMSC, num_{off} is the number of addresses stored in OffList, and $G_\nu^{\Gamma_x}$ is the signed state submitted to the distributed ledger. If ChildList of Γ_x is null, num is the number of its participants. Otherwise, num is the number of entries in ChildList. The distributed ledger stores this state $G_\nu^{\Gamma_x}$, if the following $State\,Update\,Rules$ are satisfied.

1) $\nu > \nu^*$, where ν^* is the serial number of the latest state conducted by the root channel's leader;
2) $\forall$ PK_{p_i} in OffList, p_i's `LocalState` data ω_{p_i,l_i} in G_ν must be the same with ω_{p_i,l_i^*} in G_{ν^*};
3) There should be $num - num_{off}$ signatures in $G_\nu^{\Gamma_x}$. For every signature S_{p_i} in $G_\nu^{\Gamma_x}$, the output of Vrfy(PK_{p_i}, S_{p_i}) should be accepted, and there must be one signature which contains the leader's public key;
4) If its ChildList is not null, the minimum serial number of state stored in its child state channel instances should be less than ν or it is null.

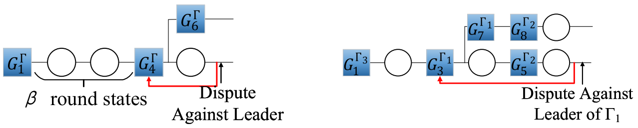
The first rule ensures that an old state cannot be updated to the instance. The second rule guarantees that an offline server's `LocalState` in a signed round state could not be changed in instance. The third rule requires that each online participant in the state channel Γ_x must sign on the consensus state G_ν, which ensures that G_ν has reached in consensus off-chain. The fourth rule checks if the serial number ν of the updating state is bigger than the minimum serial number of Γ_x's child state channel instances. If not, the state stored in Γ_x's child channels who has a bigger serial number should be set null. The reason is that if any serial number of state stored in Γ_x's child state channel instances is bigger than that of itself, it means that it is an old state, which is meaningless.

4.3.4 Dispute Resolving Protocol

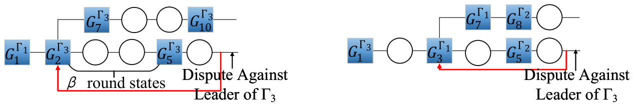
This protocol aims to prevent malicious participants from updating or generating old and invalid states. There are two malicious situations. The first situation is that some participants except the leader are malicious. The second situation is that the leader is malicious, even colludes with a small number of other participants. The first situation is easy to be excluded because the leader could detect the malicious behaviors. Then, we focus on analyzing the second situation which is more pervasive.

Since the no-signed round state is only signed by the leader, it seems that a malicious leader can violate the protocol unconstrainedly. For example, the malicious leader may collude with another participant, and only process the state generated by that participant while ignoring others' states. To resist malicious leaders, we design the dispute resolving protocol to ensure that the states updated in state channel are in line with all participants' consensus and cannot be tampered by the leader alone.

For malicious leader, there are mainly two kinds of disputes, one is against malicious leader of state channel instance



(a) Dispute in HMSC whose Parent and ChildList are null (b) Dispute in HMSC whose ChildList is null



(c) Dispute against other state channel leader using own state (d) Dispute against other state channel leader using child channels' states

Fig. 9. Different dispute situations.

whose ChildList is null, the other is against malicious leader of state channel instance whose ChildList is not null. We choose checkpoints from anchor points which are those signed round states to roll back the state channel instance to a safe state. The specific checkpoint selection algorithm is shown in Algorithm 4 which takes all states as input and the serial number of state in checkpoint as output. To execute the dispute resolving protocol, a participant submits a message DisputeMsg composed of signed round state in the checkpoint as the request to change the leader. This request is considered valid, if *State Update Rules* are satisfied.

Algorithm 4 is able to handle the two kinds of disputes mentioned above (ChildList is null or not null). We present four cases of these two kinds in Fig. 9, in which the white circle means the no-signed round state which does not need the signatures of all participants and the blue square means the signed round state which needs the signatures of all participants. Suppose there is a dispute against the leader of state channel Γ_x:

① Γ_x has no child channels: The checkpoint should be the latest signed round state of Γ_x, corresponding to lines 3 to 8 in Algorithm 4. According to our state confirmation protocol in Section 4.3.2, the latest signed round state generated by Γ_x whose ChildList is null should be signed by all participants in Γ_x. The malicious leader only colludes with a small number of participants in our assumption, so this state contains signatures from the honest non-leader participants and is correct. In Fig. 9a, there is only one state channel Γ and G_4^{Γ} is the latest signed round state. In Fig. 9b, Γ_1 and Γ_2 are child channels of Γ_3, while the dispute is against Γ_1. State $G_3^{\Gamma_1}$ is Γ_1's latest signed round state and should be selected as a checkpoint.

② Γ_x has child channels: There are two ways to select checkpoints, and we use the better one.

Using Own State. Select the second last signed round state generated by Γ_x as the checkpoint, the process is corresponding to lines 10 to 16 in Algorithm 4. In Fig. 9c, Γ_x is Γ_3 and it has two child channels Γ_1 and Γ_2. Recall our state confirmation protocol in Section 4.3.2, the signed round states generated by parent channel Γ_3 ($G_2^{\Gamma_3}$ and $G_5^{\Gamma_3}$) do not include signatures of non-leader participants. In our assumption, the leaders are malicious, so we cannot guarantee the correctness of $G_2^{\Gamma_3}$ and $G_5^{\Gamma_3}$. In addition, the leader generated $G_5^{\Gamma_3}$ is having a dispute now, therefore we consider $G_5^{\Gamma_3}$ is corrupted and cannot be selected as a checkpoint. However, the leader between $G_2^{\Gamma_3}$ and $G_5^{\Gamma_3}$ (which is different from the leader who generated $G_5^{\Gamma_3}$, because the leader should be re-elected before generating the signed round state) does not gain any dispute, so we believe the leader generated $G_2^{\Gamma_3}$ is honest and $G_2^{\Gamma_3}$ could be used for resolving dispute.

Using Child Channel State. For every child state channel of Γ_x, we find out its latest signed round state. Among these states, we select the earliest one as a checkpoint, corresponding to lines 17 to 23 in Algorithm 4. In Fig. 9d, Γ_x is Γ_3 and it has two child state channels Γ_1 and Γ_2. The latest signed round state of Γ_1 is $G_3^{\Gamma_1}$, which means that all participants in Γ_1 have reached consensus on $G_3^{\Gamma_1}$ and its previous states. Similarly, Γ_2 also reached consensus on its latest signed round state $G_5^{\Gamma_2}$ and its previous states. Because $G_3^{\Gamma_1}$ is earlier than $G_5^{\Gamma_2}$, all participants of Γ_1 and Γ_2 have reached consensus on $G_3^{\Gamma_1}$ and its previous states. From the instance initialization protocol in Section 4.3.1, we know that all participants of Γ_3 come from its child channels Γ_1 and Γ_2, so all participants of Γ_3 have reached consensus on $G_3^{\Gamma_1}$ and its previous states. Therefore, $G_3^{\Gamma_1}$ can be selected as a checkpoint.

Then, we use the voting mechanism to arbitrate whether the leader or the dispute initiator is honest. Any participant in the channel instance can generate a signed voting message for the leader or the participant that initiates the dispute by sending DisputeMsg. According to the vote, one participant between leader and the dispute initiator will be excluded to ensure that our protocol can continue operating normally.

Algorithm 4. Checkpoint Selection

```
1: procedure CheckpointSelection$G_1, \ldots, G_v, \Gamma_x$
2:   Parse $\Gamma_x$ as $(sid_x, \ldots, \text{ChildList}_x)$;
3:   if ChildList$_x$ $== null$ then
4:     $k \leftarrow v$;
5:     for $k \geq 0$ do
6:       $k--$;
7:       if $G_k^{\Gamma_x}$ exists then
8:         return $k$;
9:   else
10:    $k1 \leftarrow v, k2 \leftarrow v, count \leftarrow 0$;
11:    for $k1 \geq 0$ do
12:      $k1--$;
13:      if $G_{k1}^{\Gamma_x}$ exists then
14:        $count++$;
15:        if $count == 2$ then
16:          break;
17:    for $\Gamma_b$ in ChildList$_x$ do
18:      $k3 \leftarrow v$
19:      for $k3 \geq 0$ then
20:        $k3--$;
21:        if $G_{k3}^{\Gamma_b}$ exists then
22:          if $k3 \leq k2$ then
23:            $k2 \leftarrow k3$
24:    return $max(k1, k2)$;
```

If the leader is malicious, a new leader is selected randomly in this state channel instance, which could be decided by the hash of latest block, and its address is stored into NewLeader$_x$. Then, all state channel instances in HMSC roll back to this checkpoint. Then, the next state received by Γ_x must be G_{v+1} and leader's signature must be Sign(G$_{v+1}$|NewLeader$_x$). In this process, we lose $\nu - \nu^*$ states for resolving dispute, we call it state loss. The corresponding participants can know the new leader by checking $\mathsf{NewLeader}_x$ and continue state confirmation process. After dispute resolved, $\mathsf{NewLeader}_x$ will be set null.

4.3.5 Offline Resolving Protocol

In a real environment, equipment failure or network disconnection are common. To deal with this situation, we design the offline resolving protocol, in which other participants could also work normally when one participant goes offline. In our protocol, we use a OffList to record the offline servers.

The detection of offline server has been mentioned in Sectoin 3.3, this protocol mainly concerns the communications between participants and the state channel instance. The offline resolving protocol consists of two steps: (1) offline state update; (2) online state update.

- Offline State Confirmation: The leader broadcasts the first `GlobalState` data generated after a participant goes offline and the hash of its `LogCacheQueue` data $(G_\nu, H(\psi_{p_i,l_i}))$ to other participants in $\mathcal{P}$. The other participants also inquiry this participant to check if it is offline. If it passes, a participant replies with its signed `GlobalState` data. Then the leader collects the signatures from other participants. At last, the leader calls the state update protocol of HMSC to submit OfflineMsg composed of `GlobalState` data and an offline tag $(G_\nu, S_{p_1}, \dots, S_{p_{i-1}}, S_{p_{i+1}}, \dots, S_{p_n}, OFFLINE : (PK_{p_i}, H(\psi_{p_i,p_1}))$.

  When the state channel instance receives OfflineMsg, it first checks if it satisfies $\mathit{State\ Update\ Rules}$. Then to prevent other participants in this state channel from intentionally excluding this node, it starts a challenge time which is used for cancelling this offline request. Specifically, after the state channel receives and checks OfflineMsg, it waits for one block generation time (block height could be obtained from the smart contract) to receive the request from the offline server. After the waiting time, if the participant is not actually offline, it could send its signed latest state, (G_ν, S_{p_i}) to state channel to request cancelling. In the waiting time, if the state channel instance does not receive any cancelling message, it adds PK_{p_i} and $H(\psi_{p_i,l_i})$ to its own OffList. Otherwise, the offline request does not take effect.
- Online State Confirmation: The leader first broadcasts the first `GlobalState` data generated after an offline participant goes online and collects the signatures from other participants. Then, the leader calls the state update protocol of HMSC to submit OnlineMsg composed of `GlobalState` and an online tag $(G_\nu, S_{p_1}, \dots, \dots, S_{p_n}, ONLINE : PK_{p_i})$. When the state channel instance receives OnlineMsg, it checks if it satisfies $\mathit{State\ Update\ Rules}$. If it passes, this state channel instance removes the address of offline participant to its own OffList. Otherwise, the online request fails.

4.3.6 Instance Closure Protocol

This protocol aims to close a state channel instance, such that no participant can submit state any longer. Traditional approach in the two-party state channel scheme is to utilize an on-chain challenge period. In this period, a state is submitted to the distributed ledger, and is accepted by the distributed ledger if no newer state is submitted to the distributed ledger during that period. However, this approach requires an unaffordable waiting time.

To tackle this issue, we design an off-chain instance closure protocol as follows. When a state channel intends to close the root state channel instance, it generates a state with a special identifier $(G_\nu|Close)$, and requests a signed state as shown in signed round state generation which has the signatures from all participants of root state channel instance. When a participant intends to close the state channel instance which does not have parent state channel, it could also generate $(G_\nu|Close)$, and requests a signed state as shown in signed round state generation. Finally, any participant could send a message CloseMsg composed of $(G_\nu|Close)^{signed}$ to SCC.

If multiple state channel instances intend to be disassociated, which means the closure of their parent state channel instance. The closure process could be done as follows. Let this parent state channel instance be Γ_a. Every state channel instance in ChildList of Γ_a generates a signed round state on a new state with a special identifier $(G_{\nu_{ci}}|Close)^{\Gamma_a}$. The leader of Γ_a collects all these signed round states and sends a DisassociateMsg composed of $(G_\nu^{\Gamma_1}, \dots, G_\nu^{\Gamma_m})$ to SCC. SCC verifies these signatures and closes this state channel instance Γ_a.

5 CAPABILITY AND SECURITY ANALYSIS

Theorem 1. *Time complexity of HMSC depends on β and ration of dispute q. When the β increases and q decreases, the time complexity convergence to $o(1)$.*

Proof. Note that one leader could receive β no-signed round states and one signed round state. The leader needs to interact with other participants three times when generating a signed round state as shown in Fig. 8. While it only needs one time interaction when generating a no-signed round state. The main factor for determining the efficiency of our scheme is the cost of Sign and Vrfy, here we analyse the times of generating and verifying signatures in our scheme. We define the servers in a state channel is n, the states processed in a round is $\beta + 1$ and the round times is r. The average number of signatures per state could be represented by the sum of number of signatures in case of no dispute and number of signatures in case of dispute divided by total number of state. In case of no dispute, the sum of number of signatures is composed of signed round states' signatures which is $rn(1-q)$ and no-signed round states' signatures which is $r\beta(1-q)$. In case of dispute, the sum of number of signatures is only composed of signed round states' signatures which is rnq. In single HMSC, it could be represent as follow:

$$T(n) = \frac{(rn + r\beta)(1-q) + rnq}{r(1+\beta)(1-q) + rq} = \frac{n + (1-q)\beta}{1 + (1-q)\beta}. \tag{4}$$

In multi-layer HMSC, Δ is the number of layer and $n_{i,j}$ is the number of participants in ith layer and jth state channel instance. The average number of signatures per state could be represented by the sum of the average number of signatures per state of all state channel minus their shared no-signed round states' signatures

$$T(n) = \sum_{i=1}^{\Delta} \sum_{j=1}^{\eta_i} \left(\frac{n_{i,j} + (1 - q_{i,j})\beta}{1 + (1 - q_{i,j})\beta} - 1 \right) + 1. \tag{5}$$

It could be seen from above that when the the percentage of honest participants decreases, $T(n)$ is nearly close to n. If there is no malicious participant and β is big enough, $T(n)$ is close to 1. Note that when the β is large and malicious participant exsist, it will repeatedly trigger $DisputeResolve$, and could bring large state loss and greatly reduce efficiency of HMSC. □

Security Goals. We define security goals that guarantee that an adversary described in the threat model (denoted as $\mathcal{A}$) cannot affect the update of global states G_ν (signed and no-signed round states), which ensures the integrity of the audit logs. Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the participants of a DELIA instance and $\mathcal{H} = \{p \,|\, p \in \mathcal{P} \wedge p \text{ is honest}\}$, we should defend:

- (S1) Network monitoring attack: When all participants $p \in \mathcal{P}$ are honest, a monitoring attacker $\mathcal{A}$ cannot affect the update of global states G_ν.
- (S2) Compromise attack: Compromised participants cannot modify G_ν generated by $p_h (p_h \in \mathcal{H})$, nor submitted corrupted states G'_ν.
- (S3) Sybil attack: An attacker $\mathcal{A}$ who could disguise as multiple participants cannot modify G_ν generated by $p_h (p_h \in \mathcal{H})$, nor submitted corrupted states G'_ν.
- (S4) DoS attack: The local state ω_{p_i, l_i} of log server p_i under DoS attack will not be tampered by the attacker.

Theorem 2 (S1). *Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the participants of a DELIA instance, every $p \in \mathcal{P}$ is honest and $p_i (i \in [1, n])$ attempts to submit a global state G_ν. Then for a monitoring adversary $\mathcal{A}$ as described in the threat model, the state updated to the distributed ledger will be G_ν exactly.*

Proof. In LSG phase, we encrypt data by Transport Layer Security (TLS) protocol, so that a monitoring attacker $\mathcal{A}$ cannot interfere with the normal state generation process in LSG phase.

While in HMSC phase, the communication between distributed ledger node and log servers is transparent, because these data must be published in distributed ledger. This means $\mathcal{A}$ can get all the signed round states.

When updating G_ν^Γ to distributed ledger, even though this message is transparent, $\mathcal{A}$ cannot modify the state G_ν nor the serial number ν directly because he cannot forge the signatures. If $\mathcal{A}$ tries to replace G_ν^Γ with $G_{\nu^\star}^\Gamma$ ($\nu > \nu^\star$, which means $G_{\nu^\star}^\Gamma$ is an earlier state that $\mathcal{A}$ could get from previous update), it will be prevented by our state update protocol's first rule as described in Section 4.3.3. □

Theorem 3 (S2). *Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the participants of a DELIA instance, $\mathcal{H} = \{p \,|\, p \in \mathcal{P} \wedge p \text{ is honest}\}$ and $|\mathcal{H}| > n/2$. Then compromised participants $p_m (p_m \in \overline{\mathcal{H}})$ as described in the threat model cannot modify G_ν generated by $p_h (p_h \in \mathcal{H})$ nor submitted corrupted states G'_ν.*

Proof. In addition to the basic monitoring ability, a compromised participant p_m can forge a state G' and sign it with his private key SK_{p_m}. But the hash value $H(\delta_{p_i,\xi})$ of log records should be broadcasted immediately at LSG phase, so every $p \in \mathcal{P}$ could calculate the faithful state G_ν.

The two kinds of malicious behaviour, where the compromised participants p_m are trying to modify a normal state G_ν or submit a corrupted state G'_ν, are basically the same. The only difference is which participant (p_m or p_h) submits the state to its leader at the beginning of the state confirmation protocol. After that, all the processes are identical, so we treat them as the same situation.

Suppose p_m attempts to replace G_ν with G'_ν or submit G'_ν.

① If G_ν is a no-signed round state, only the leader of root state channel instance p_{rld} will sign it. Every $p \in \mathcal{P}$ will store $(G'_\nu, S'_{p_{rld}})$ or $(G_\nu, S_{p_{rld}})$ as temporary cache. Suppose p_{rld} colludes with some other p_m(s) and store G'_ν instead of G_ν. The following state $G_{\nu+1}$ contains some information about G_ν. If p_m stores $\ldots, G'_\nu, G_{\nu+1}, \ldots$, it will be an error in audit phase. If p_m stores $\ldots, G'_\nu, G'_{\nu+1}, \ldots$, this incosistency will be found in the next signed round state update, and p_h can call the dispute resolving protocol as described in Section 4.3.4. Then we can use the voting mechanism to exclude the malicious leader ($|\mathcal{H}| > n/2$ required) and roll back to a proper checkpoint.

② If G_ν is a signed round state, it should include signatures from all participants in the channel. As long as there are honest participants p_h in the channel, p_m cannot forge (G'_ν, S'_{p_h}) and update G'_ν. If there is no p_h in this channel and all the leaders are compromised, G'_ν would be updated to the distributed ledger, but the leaders will be re-elected in the next signed round state update and p_h in other channels will find this inconsistency, just like the situation in ①. The dispute resolving protocol would roll back the states to an appropriate checkpoint. □

Theorem 4 (S3). *Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the participants of a DELIA instance, $\mathcal{H} = \{p \,|\, p \in \mathcal{P} \wedge p \text{ is honest}\}$, $|\mathcal{H}| > n/2$. Then a Sybil adversary $\mathcal{A}$ who could disguise as multiple participants cannot modify G_ν generated by $p_h (p_h \in \mathcal{H})$ nor submitted corrupted states G'_ν.*

Proof. The basic idea to defeat Sybil attack in DELIA is identity validation. The instance initialization protocol described in Section 4.3.1 ensures that only the pre-negotiated participants could join the same state channel instance, because each p_i should send `InitMsg` containing all other participants' public keys to `SCC` in order to create a channel.

Therefore, the Sybil attacker $\mathcal{A}$ could only interact with the state channel instance with the identities of those participants whose private key has been compromised, which is actually equivalent to the compromise attack in Theorem 3. If $\mathcal{A}$ tries to modify a faithful state G_ν or update a corrupted state G'_ν, the dispute resolving protocol will prevent it and exclude the compromised participants from the state channel. □

Theorem 5 (S4). *Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the participants of a DELIA instance, $\mathcal{A}$ be a DoS adversary with the ability to block* *access to any log server. Then for an honest log server* p_i *under DoS attack,* $\mathcal{A}$ *cannot tamper* p_i*'s local state* ω_{p_i,l_i}*.*

Proof. If log server p_i is attacked by DoS attacks, it cannot submit signatures in the state confirmation stage. The leader would know that p_i is offline and broadcast the global state G_v and the hash of p_i's LogCacheQueue $H(\psi_{p_i,l_i})$, as described in Section 4.3.5. The state channel instance will add PK_{p_i} and $H(\psi_{p_i,l_i})$ to its own OffList. In the subsequent update, the second rule of the state update protocol will ensure that p_i's `LocalState` data ω_{p_i,l_i} in G_v remains unchanged until the DoS attack on the p_i ends and p_i goes online again. □

If DoS attack is against the distributed ledger nodes, due to the decentralized characteristic, DELIA could replace the nodes being attacked by other nodes and log servers could still communicate with the state channel instance. However, DoS attacks cannot be defended absolutely. We only guarantee that the log records will not be tampered during DoS attacks, and DoS attacks on minority servers will not stop the entire system.

6 EVALUATION

6.1 Implementation

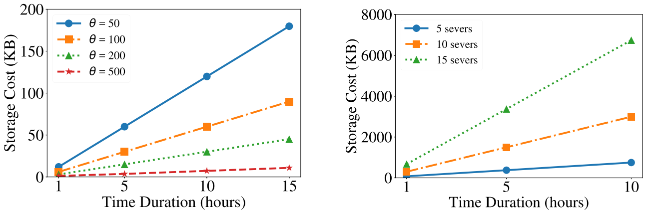
To validate our state channel scheme, we develop a prototype of DELIA, which employs Ethereum 1.8.1 [21] as the distributed ledger platform. We use solidity to realize our HMSC contracts on Ethereum, which is the implementation of state channel on the distributed ledger. Each server in the domain could communicate with HMSC instance on Ethereum by web3js APIs. The algorithms and protocols in DELIA are coded by Java and the communication between two servers is protected by TLS. The hash function in our prototype is SHA256. To simplify the log production process and test the system under different parameters, we use a log reader program and public audit log dataset [22] for simulating log generation. According to this dataset, the total amount of audit log data is 15.6 GB and the average log record generation rate is up to 2420 items per minute.

The domain consists of 15 servers with Intel celeron E4300(2.6GHz) CPU, 8G RAM, and Ubuntu 16.04 64bit operation system. Moreover, the distribute ledger consists of 10 servers as the ledger nodes with Inter(R) Xeon(R) CPU E5-2682 v4 @2.5GHz, 8G RAM, and Ubuntu 16.04 64bit operation system.

6.2 Result Analysis

6.2.1 Performance of LSG

The first is the analysis of storage efficiency in LSG phase. Since every server needs to store all `LocalState` and `GlobalState` data for further verifications, it is necessary to evaluate the storage cost on the server side. While the length of `LogCacheQueue` is always constant, we don't consider it in our evaluation. As shown in Fig. 10a, the storage cost for `LocalState` data decreases as θ increases. Note that θ is the number of records that are involved in `LocalState` data and the total number of records is fixed, which means that bigger θ implies generating less `LocalState` data and less storage cost. Fig. 10b shows the storage cost for `GlobalState` data in different number of participants in a state channel instance. Since the server needs to maintain `GlobalState` data which consists of all `LocalState` data from the servers in a state channel instance, the storage cost grows linearly with the time duration and the number of participants in a state channel instance. However, the storage cost for these two types of data is acceptable in practice. When 15 servers run for 10 hours generating 2,178,000 log records, there only takes 7 MB storage cost.



(a) Storage Cost for `LocalState` data with Different θ

(b) Storage Cost for `GlobalState` data with Different Number of Participants

Fig. 10. Stroage cost in LGS phase.

The second is the analysis of verification delay in LSG phase. We evaluate the verification delay of LSG, which aims to generate `LocalState` and `GlobalState` data from audit logs. Table 2 describes the `LocalState` data generation time, which grows with θ increasing. This is because that `LocalState` data are calculated by the root of the Merkle hash tree which is generated from θ items (see Eq. (2)). The computation cost is quite small in practice, which is only 123ms when $\theta = 500$.

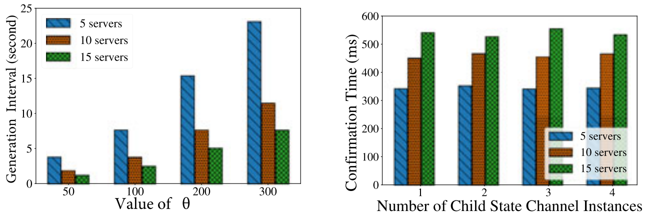
Once `LocalState` data are generated on a server, `GlobalState` should be generated and verified immediately. One goal of LSG is to generate stable `GlobalState` data, which means that the `GlobalState` data generation interval needs to be greater than the state confirmation time. From Fig. 11a, we can find that the interval time decreases as θ decreases and the number of participants in a state channel instance increases. The reason is that `LocalState` data on a server is generated more frequently when θ decreases, and the generation frequency of `LocalState` data increases when the number of participants in a state channel instance increases. Even if $\theta = 50$ and there are 15 servers in a state channel instance, the `GlobalState` data generation interval is 1.28s, which is enough for state confirmation shown in Fig. 11b and means that our framework can generate stable `GlobalState`.

6.2.2 Performance of HMSC

We compare HMSC with [23] called MSC, which is a representative scheme of multi-party state channel. As shown in Table 3, *ledger* means whether the protocol needs to interact with the distributed ledger directly. *Verificiation* means the Sign and Vrfy delays of the signature alogrithm. *Communication* means the network delay between servers. Fig. 12a shows the delays of HMSC and MSC in optimistic state confirmation process, while there is no dispute. β represents the round number of no-signed round state in a round. We conduct 100 round state confirmation processes and find that the delay of HMSC keeps stable and the delay of MSC grows up with the number of participating servers increasing. The reason is that our scheme

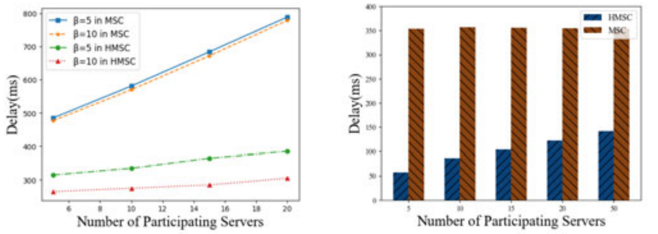
TABLE 2
`LocalState` Generation Time

θ	Generation Time (in Millisecond)
50	22
100	35
200	55
500	123



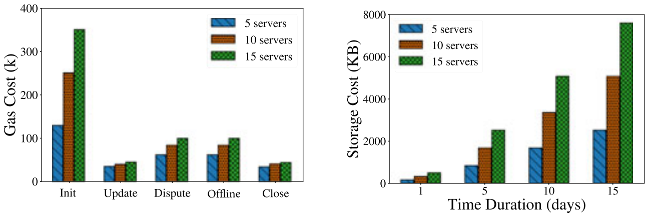
(a) GlobalState Generation Interval with Different Number of Participants (b) Signed Round State Confirmation Time with Different Number of Participants

Fig. 11. `GlobalState` Generation interval and signed round state confirmation time.



(a) Delay comparison in State Confirmation without dispute (b) Delay comparison in State Confirmation with dispute

Fig. 12. Delay comparison in state confirmation process.



(a) Gas Cost in Ethereum in Different Number of Participants (b) Storage Cost in Ethereum in Different Number of Participants

Fig. 13. Performance of HMSC.

leverages leader's unified coordination to instead the linear validation when the state channel confirms states. Thus, HMSC is more efficient in optimistic state confirmation process. Fig. 12b shows the delays in pessimistic state confirmation process. We can find that the delay of HMSC is much lower than that of MSC. The reason is that, when a dispute occurs, HMSC just invokes the voting mechanism among the participants in the state channel while MSC needs to contact with the distributed ledger, which induces more time. As a result, HMSC has obvious advantages in the whole state confirmation process which is the most frequent process in multiparty state channel, thus, HMSC is more suitable for audit scenario where states are generated and confirmed frequently.

Then, we evaluate the gas cost (which represents computing resources in Ethereum) in the five on-chain protocols: Init, Update, DisputeResolving, OfflineResolving, and Close, which need to be executed in Ethereum. From Fig. 13a, we can find that Init protocol requires more gas than that in other protocols, since every server in a state channel instance has to submit its request for joining the HMSC instance. The gas required in Update and Close protocols are almost the same and much smaller than that in Init protocol. This is because only one server needs to send its request to Ethereum in these two protocols. Although there is also one server that sends its request to Ethereum in DisputeResolver protocol, the gas required in DisputeResolver are higher than that in Update. The reason is that Ethereum need to select a new leader in this protocol. At a gas cost of 25 Gwei (which is measurement unit of gas), Init for a state channel instance of 15 servers would cost approximately 0.00085 Ether (unit of Ethereum currency), which equals to $0.178. This implies that our scheme is efficient from the perspective of economics.

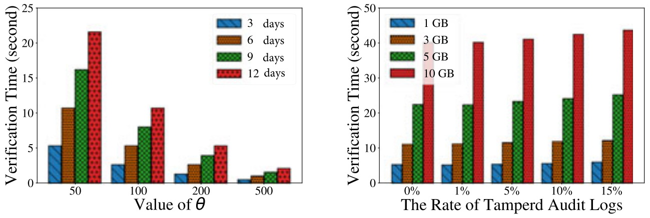
Finally, we evaluate the storage cost on Ethereum as shown in Fig. 13b. When DELIA runs for 15 days in a HMSC composed of three state channel instances including 15 servers, it only takes up 8MB at most. Therefore, HMSC is efficient in terms of storage requirements in the distributed ledger.

6.2.3 Performance of Log Integrity Audit

Figs. 14a and 14b show the verification time in integrity audit stage. As shown in from Fig. 14a, verifiaction time decreases when θ increases. As shown in from Fig. 14b, the ratio of tampered audit logs has limited effect on the verification time. The verification time grows linearly with the size of audit logs, since the auditor has to compute the hash value of every log record and generate `LocalState` data

TABLE 3
Comparison With Previous Multi-Party State Channel

Protocol	Dziembowski(2019)[23]			Present work		
	Communication	Verification	Ledger	Communication	Verification	Ledger
Initialization	$O(n)$	-	×	-	$O(n)$	√
Optimistic confirmation	$O(n)$	$O(n)$	×	$O(n)$	$O(1)$	×
Pessimistic confirmation	$O(n)$	$O(n)$	√	$O(n)$	$O(n)$	×
Update	-	$O(n)$	√	-	$O(n)$	√
DisputeResolving	$O(n)$	$O(n)$	√	$O(n)$	$O(n)$	√
OfflineResolving	-	-	-	$O(n)$	$O(n)$	√
Closure	$O(n)$	-	×	$O(n)$	$O(n)$	√



(a) Verification Time at Different θ

(b) Verification Time at Different Rate of Tamperd Audit Logs

Fig. 14. Verification time in log integrity audit.

from these hash values. The most time-consuming part in integrity audit stage is the generation of `LocalState` data. Since every `LocalState` data is generated based on the previous one, the computation cannot be performed in parallel.

6.2.4 *Performance of Voting Mechanism*

In state confirmation process, we employ a voting mechanism to arbitrate whether the leader or the dispute initiator is honest. In this part, we evaluate the delays of voting mechanism from two aspects: communication and computation.

Fig. 15 indicates that the computation delay raises and the communication delay keeps stable with the increase of the number of participating servers. The reason is that the offline resolving protocol just involves the servers who have the same leader. Each server in the same state channel broadcasts it's DisputeMsgs, collects and responses DisputeMsgs, individually. Even if the number of participating servers reaches 50, the communication delay is less than 120 ms and the computation delay is less than 100 ms. Thus, the voting process is efficient and it can effectively support our offline resolving protocol.

7 RELATED WORK

7.1 Log Integrity Audit

Provable Data Possession(PDP) [13] and Proof of Retrievability (PoR) [14] are designed for examining the integrity of archived data, such as audit logs . These methods require the administrator to outsource audit logs to a third party (e.g., a cloud server) and provide probabilistic proof that the third party stores the files. Armknecht *et al.* [24] extended PoR scheme, which enables an external auditor to execute the PoR protocol with the cloud on behalf of the data owner. Hanling *et al.* [25] pointed out that the size of remote storage can be the most expensive factor, thus they proposed a simple PoR scheme to minimize storage overhead. Liu *et al.* [26] exploited disconnected ORAM operations and designed a two-layer encryption scheme to reduce evict cache size from GB/MB to KB level. Guo *et al.* [27] presented a communication-efficient and fast protocol for verifiable aggregation. However, the sensitive logs may be leaked at the third party in these solutions. Liu *et al.* [28] proposed a novel message-locked integrity auditing scheme for encrypted data, which solves data leaking. Liu *et al.* [29] proposed the hybrid model named EncodeORE, which achieves acceptable security and appropriate ciphertext length to reduce information leakage. Unfortunately, those solutions depend on the security of the third party which may be a potential risk point.

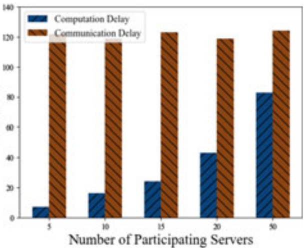


Fig. 15. Delays of voting mechanism.

Some researchers employ the distributed ledger technique for log integrity protection. In [18], Andrew Sutton *et al.* proposed a linked-data-based method, which utilizes the distributed ledger technology to create tamper-proof audit logs. Their solution provides proofs of log manipulation and non-repudiation which are useful in data sharing environments. In [19], Ashar Ahmad *et al.* presented BlockAudit, a scalable and tamper-proof system that leverages audit logs and security property of distributed ledger to enable secure and trustworthy log integrity audit. In [17], Jordi Cucurull *et al.* used an immutable log generation method to ensure the integrity, authenticity, and non-repudiation of updated audit logs, and stored the integrity proofs in Bitcoin's blockchain. In [30], Gaurav Panwar *et al.* presented an auditing framework, which leverages zero knowledge proofs, Pedersen commitments, Merkle trees, and public ledgers to create a scalable mechanism for auditing electronic surveillance processes involving multiple actors. However, these schemes directly store every log or checksum to the distributed ledger which induces unaffordable cost towards massive log data.

7.2 State Channel

Since the blockchain-based systems have the limitation on throughput, which makes it hard to use them directly for microtransactions, the state channel technology attracts widespread attention as a solution. The payment channel, which is a special sub-class of state channel, is first proposed in [31]. Payment channel allows two users exchange their money rapidly without sending every middle transaction to the ledger. When the whole transactions are completed, users send the final result to the ledger, the ledger only needs to process final transaction, which promotes the throughput of blockchain-based systems. The whole transaction process only needs to interact with ledger when constructing channels and settlement. However, every time users conduct microtransactions, they have to construct a new payment channel, which takes a lot of time to initial channel and settle results. Thus, researches mainly concerns about related routing protocols [32], [33], channel rebalancing [34], and channel hubs [35]. The purpose of these studies is to construct new appointed transaction route based on existing state channels instead of construct a new payment channel. These routing schemes need all nodes in each state channel to participant interactions, which induces privacy risk of transactions and high cost of intermediate nodes. Then, researchers focus on the generalization of payment channel and foundations of state channel [36]. Dziembowski *et al.* [20], [37] proposed virtual state channel, which makes the establishment of the state channel no longer need the participation of all nodes in related state channels. Note that, intermediate nodes do not participate execution after the virtual state channel is established,, which can protect the privacy of transaction details and reduce the cost of themselves. This solution permits to build channels over multiple state channels, but it still only supports two users in one channel.

Multi-party state channel is the extension of traditional two-party state channel. In [23], the researchers proposed a multi-party scheme which is built on top of two-party state channel. Their multi-party state channel scheme is more suitable for scenarios related to virtual coin. However, due to the complexity of off-chain communication protocol, their scheme is difficult to be realized in our scenario. Compared with their solution, our multi-party state channel scheme is built on top of the original distributed ledger and easy to implement. Moreover, our scheme is more efficient in the process of pre-determining participants in the state channel.

8 CONCLUSION

In this paper, we propose a distributed efficient log integrity audit framework, called DELIA. We adopt the distributed ledger technique to protect the verification materials, and utilize the idea of state channel to improve the throughput of the distributed ledger system. To generate stable state and provide mutual supervision in the domain, we propose a log state generation scheme, called LSG. With the help of LSG, rapidly-updated audit logs can be recorded in the state channel. To solve the high latency challenge in existing multi-party state channel schemes, we propose an hierarchal efficient multi-party state channel scheme, called HMSC. Then, the latency is dramatically reduced in amortized analysis. Extensive experiments demonstrate that DELIA is highly efficient in practice.

ACKNOWLEDGMENTS

This research was supported in part by the National Natural Science Foundation of China under grants No. 61772383, U1836202, 62076187; by the Joint fund of Ministry of Education of China for Equipment Pre-research under grant No. 6141A02033341.

REFERENCES

[1] Y. Kwon *et al.*, "MCI : Modeling-based causality inference in audit logging for attack investigation," in *Proc. Netw. Distrib. Sys. Secur. Symp.*, 2018.

[2] Y. Liu *et al.*, "Towards a timely causality analysis for enterprise security," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[3] S. McClure, J. Scambray, G. Kurtz, and Kurtz, *Hacking Exposed: Network Security Secrets and Solutions*. New York, NY, USA: McGraw-Hill, 2009.

[4] T. M. Corporation, "Capec-268: Audit log manipulation." Accessed: Aug. 17, 2020. [Online]. Available: https://capec.mitre.org/data/definitions/268.html

[5] The world's most used penetration testing framework. Accessed: Aug. 17, 2020. [Online]. Available: https://www.metasploit.com/

[6] Dark Laboratory, "A better generation of logcleaners." Accessed: Aug. 17, 2020. [Online]. Available:https://web.archive.org/web/20070218231819/http://darklab.org/jot/logcleanng/logcleaner-ng_1.0_lib.html

[7] S. Hales, "Last door log wiper." Accessed: Aug. 17, 2020. [Online]. Available: https://packetstormsecurity.com/files/118922/LastDoor.tar

[8] K. Haniradi, "mig-logcleaner-resurrected." Accessed: Aug. 17, 2020. [Online]. Available: https://github.com/Kabot/mig-logcleaner-resurrected

[9] maldevel, "Clearlogs." Accessed: Aug. 17, 2020. [Online]. Available: https://sourceforge.net/projects/clearlogs/

[10] Global incident response threat report. Accessed: Aug. 17, 2020. [Online]. Available: https://www.carbonblack.com/resources/tipping-point-election-covid-19-create-perfect-storm-cyberattacks/

[11] Amazon. Accessed: Aug. 17, 2020. [Online]. Available: http://chinaplus.cri.cn/news/china/9/20171112/51048.html

[12] Nintendo. Accessed: Aug. 17, 2020 [Online]. Available: https://edition.cnn.com/2020/06/09/tech/nintendo-300000-accounts-hacked/index.html

[13] G. Ateniese *et al.*, "Provable data possession at untrusted stores," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 598–609.

[14] A. Juels and B. S. Kaliski Jr,"PORS: Proofs of retrievability for large files," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 584–597.

[15] K. He, J. Chen, Q. Yuan, S. Ji, D. He, and R. Du, "Dynamic group-oriented provable data possession in the cloud," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 3, pp. 1394–1408, May/Jun. 2021.

[16] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." Accessed: Aug. 17, 2020. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[17] J. Cucurull and J. Puiggali, "Distributed immutabilization of secure logs," in *Proc. Int. Workshop Secur. Trust Manage.*, 2016, pp. 122–137.

[18] A. Sutton and R. Samavi, "Blockchain enabled privacy audit logs," in *Proc. Int. Semantic Web Conf.*, 2017, pp. 645–660.

[19] A. Ahmad, M. Saad, M. Bassiouni, and A. Mohaisen, "Towards blockchain-driven, secure and transparent audit logs," in *Proc. 15th EAI Int. Conf. Mobile Ubiquitous Syst. Comput. Netw. Serv.*, 2018, pp. 443–448.

[20] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 949–966.

[21] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014. Accessed: Aug. 17, 2020. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[22] J. Zhu *et al.*, "Tools and benchmarks for automated log parsing," in *Proc. 41st Int. Conf. Softw. Eng. Softw. Eng. Pract.*, 2019, pp. 121–130.

[23] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, "Multi-party virtual state channels," in *Proc. Annu. Int. Conf. Theory Appl. Cryptogr. Techn.*, 2019, pp. 625–656.

[24] F. Armknecht, J. Bohli, G. O. Karame, Z. Liu, and C. A. Reuter, "Outsourced proofs of retrievability," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 831–843.

[25] M. Hanling, G. Anthoine, J. Dumas, A. Maignan, C. Pernet, and D. S. Roche, "Poster: Proofs of retrievability with low server storage," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 2601–2603.

[26] Z. Liu, B. Li, Y. Huang, J. Li, Y. Xiang, and W. Pedrycz, "NEWMCOS: Towards a practical multi-cloud oblivious storage scheme," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 4, pp. 714–727, Apr. 2020.

[27] X. Guo *et al.*, "VERIFL: Communication-efficient and fast verifiable aggregation for federated learning," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 1736–1751, 2021.

[28] X. Liu, W. Sun, W. Lou, Q. Pei, and Y. Zhang, "One-tag checker: Message-locked integrity auditing on encrypted cloud deduplication storage," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2017, pp. 1–9.

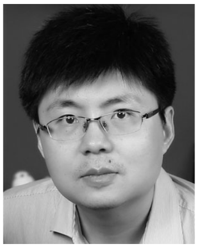
[29] Z. Liu *et al.*, "ENCODEORE: Reducing leakage and preserving practicality in order-revealing encryption," *IEEE Trans. Dependable Secure Comput.*, early access, Oct. 9, 2020, doi: 10.1109/TDSC.2020.3029845.

[30] G. Panwar, R. Vishwanathan, S. Misra, and A. Bos, "SAMPL: Scalable auditability of monitoring processes using public ledgers," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 2249–2266.

[31] R. Pass and A. Shelat, "Micropayments for decentralized currencies," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 207–218.

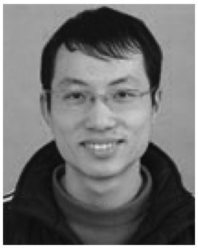
[32] G. Malavolta, P. Moreno-Sanchez , A. Kate, and M. Maffei, "Silentwhispers: Enforcing security and privacy in decentralized credit networks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017.

[33] Anno, "Lightning-onion." Accessed: Aug. 17, 2020. [Online]. Available: https://github.com/lightningnetwork/lightning-onion
[34] R. Khalil and A. Gervais, "Revive: Rebalancing off-blockchain payment networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 439–453.
[35] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017.
[36] Counterfactual. Accessed: Aug. 17, 2020. [Online]. Available: https://www.counterfactual.com
[37] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 106–123.

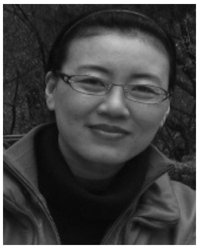


Jing Chen received the PhD degree in computer science from the Huazhong University of Science and Technology, Wuhan. Since 2015, he has been a full professor with Wuhan University. He has authored or coauthored more than 100 research papers in many international journals and conferences, including the *IEEE Transactions on Dependable and Secure Computing*, *IEEE Transactions on Information Forensics and Security*, *IEEE Transactions on Mobile Computing*, INFOCOM, *IEEE Transactions on Computers*, and *IEEE Transactions on Parallel and Distributed Systems*. His research focuses on computer science, especially in network security and cloud security. He is currently a reviewer for many journals and conferences, including the *IEEE Transactions on Information Forensics*, *IEEE Transactions on Computers*, and *IEEE/ACM Transactions on Networking*.

Xin Chen received the master's degree in information security from Wuhan University, Wuhan, China, in 2020. His research interests include blockchain and computer network.



Kun He received the PhD degree in computer science from Computer School, Wuhan University. He is currently an associate professor with Wuhan University. He has authored or coauthored more than 20 research papers in many international journals and conferences, including the *IEEE Transactions on Computers*, *IEEE Transactions on Dependable and Secure Computing*, *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Parallel and Distributed Systems*, USENIX Security, and INFOCOM. His research interests include cryptography, network security, mobile computing, and cloud computing.



Ruiying Du received the BS, MS, and PhD degrees in computer science from Wuhan University, Wuhan, China, in 1987, 1994, and 2008, respectively. She is currently a professor with the School of Cyber Science and Engineering, Wuhan University. She has authored or coauthored more than 80 research papers in many international journals and conferences, including the *IEEE Transactions on Parallel and Distributed Systems*, *International Journal of Parallel and Distributed Systems*, INFOCOM, SECON, TrustCom, and NSS. Her research interests include network security, wireless network, cloud computing, and mobile computing.

Yang Xiang (Fellow, IEEE) received the PhD degree in computer science from Deakin University, Australia. He is currently a full professor and the dean of digital research and innovation capability platform with the Swinburne University of Technology, Australia. He is also leading the Blockchain initiatives at Swinburne. Since the past 20 years, he has authored or coauthored more than 300 research papers in many international journals and conferences. His research interests include cyber security, which covers network and system security, data analytics, distributed systems, and networking. He was an associate editor for the *IEEE Transactions on Computers* and the *IEEE Transactions on Parallel and Distributed Systems*. He is currently the editor-in-chief of *Springer Briefs on Cyber Security Systems and Network*s, an associate editor for *IEEE Transactions on Dependable and Secure Computing*, *IEEE Internet of Things Journal*, and the *ACM Computing Surveys*. He is the coordinator, of Asia IEEE Computer Society Technical Committee on Distributed Processing.