Raft-Forensics: High Performance CFT Consensus with Accountability for Byzantine Faults

Weizhao Tang
Carnegie Mellon Univerisity
USA
wtang2@andrew.cmu.edu

Peiyao Sheng
University of Illinois at Urbana-Champaign
USA
psheng2@illinois.edu

Pronoy Roy
Carnegie Mellon Univerisity
USA
pronoyr@andrew.cmu.edu

Xuechao Wang
University of Illinois at Urbana-Champaign
USA
xuechao2@illinois.edu

Giulia Fanti
Carnegie Mellon Univerisity & IC3
USA
gfanti@andrew.cmu.edu

Pramod Viswanath
Princeton University
USA
pramodv@princeton.edu

ABSTRACT

Crash fault tolerant (CFT) consensus algorithms are commonly used in scenarios where system components are trusted, such as enterprise settings. CFT algorithms offer high throughput and low latency, making them an attractive option for centralized operations that require fault tolerance. However, CFT consensus is vulnerable to Byzantine faults, which can be introduced by a single corrupt component. Such faults can break consensus in the system. Byzantine fault tolerant (BFT) consensus algorithms withstand Byzantine faults, but they are not as competitive with CFT algorithms in terms of performance. In this work, we explore a middle ground between BFT and CFT consensus by exploring the role of *accountability* in CFT protocols. That is, if a CFT protocol node breaks protocol and affects consensus safety, we aim to identify *which* node was the culprit. Based on Raft, one of the most popular CFT algorithms, we present Raft-Forensics, which provides accountability over Byzantine faults. We theoretically prove that if two honest components fail to reach consensus, the Raft-Forensics auditing algorithm finds the adversarial component that caused the inconsistency. In an empirical evaluation, we demonstrate that Raft-Forensics performs similarly to Raft and significantly better than state-of-the-art BFT algorithms. With 256 byte messages, Raft-Forensics achieves peak throughput 87.8% of vanilla Raft at 46% higher latency, while state-of-the-art BFT protocol Dumbo-NG only achieves 18.9% peak throughput at nearly 6× higher latency.

CCS CONCEPTS

• **Security and privacy** → **Distributed systems security**; • **Networks** → *Security protocols*;

KEYWORDS

CFT Protocols, forensics, blockchain

1 INTRODUCTION

Fault tolerance is a cornerstone of distributed systems [39]. Two main types of fault tolerant protocols have dominated the literature: crash fault tolerance (CFT) and Byzantine fault tolerance (BFT) [30, 39]. CFT protocols allow a system to come to consensus on a log of events even in the presence of nodes that may crash, but otherwise follow protocol [25, 32, 42, 52]. BFT protocols provide greater robustness by guaranteeing consensus under not only crash faults, but Byzantine faults as well, including tampering responses or delaying messages [2, 4, 8, 22, 37, 55]. Both types of consensus algorithms provide security guarantees in theory under the assumption that at least a certain fraction of nodes are following the protocol, but the remaining nodes may suffer from crash or Byzantine faults, respectively.

Although both models of fault tolerance have benefits, CFT systems have been more widely adopted in enterprise systems [3, 9, 23, 30, 32]. For example, widely-used systems like etcd [19], CockroachDB [50] and Consul [31] are known to use CFT protocols like Raft [42] and Zookeeper [32]. There are two reasons for this: first, the fault model of CFT protocols is more aligned with common enterprise settings: crash faults are extremely common [40], whereas enterprises are oftene incentivized to prioritize performance over security [17]. Second, BFT protocols are known to exhibit worse performance in terms of transaction throughput and latency than CFT protocols, both theoretically and in practice [5]; this is despite significant interest and effort from the research community towards designing more efficient BFT protocols [22, 27, 55].

The comparative dominance of CFT protocols raises an important question: *What happens if the CFT threat model is violated?* For example, what happens if a node in a CFT protocol is maliciously compromised? This question is particularly relevant when CFT protocols are used in security-sensitive contexts, such as critical infrastructure [30, 46]. The answer to this question is well-understood [39]: the consensus guarantees of the protocol do not hold. The intruder can, in some cases, break consensus for the distributed system in arbitrary ways.

Despite the risk of losing consensus, CFT protocols remain widely used, including in critical infrastructure. For example, recent proposed architecture for a U.S. central bank digital currency (CBDC) suggested that various components of the architecture would use the Raft CFT protocol to provide fault tolerance [38]. If implemented, a CBDC would be a form of critical infrastructure, and the threat of cyberattacks is expected to be significant [21]. For example, a compromised CBDC operator could attempt to fork the ledger and double spend central bank money by tampering with

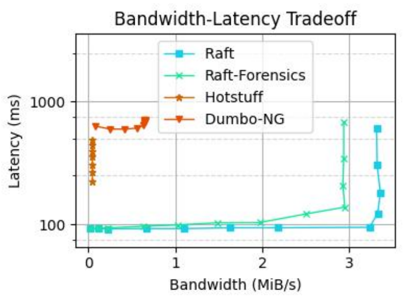


Figure 1: Bandwidth-latency tradeoffs of consensus algorithms at message size of 256 Bytes.

messages. This could lead to financial losses and more importantly, loss of public trust in the system.

1.1 This Paper

We explore a middle ground between the high performance of CFT protocols and the security guarantees of BFT protocols by asking how to modify CFT protocols to offer *accountability*. That is, we do not need the protocol to *withstand* Byzantine faults like a BFT protocol, but if a Byzantine fault occurs that affects the safety of the protocol, we want to be able to identify *which* node was responsible.

An important prior work called PeerReview tackled this problem in the context of general CFT protocols [29]. However, being a general-purpose protocol, it does not always achieve competitive performance with the underlying CFT protocol (details in §C).

Our goal in this work is to design a *practical*, protocol-specific modification of Raft [42]—a widely-used CFT protocol known for its simplicity and low overhead—to endow it with accountability at minimal cost to throughput and latency.

1.1.1 The Raft Consensus Algorithm. The Raft consensus algorithm is a CFT protocol solving the state machine replication (SMR) problem by maintaining the consistency among append-only logs of all nodes in the system. The protocol consists of two major components – log replication and leader election. In log replication, Raft manages to process client requests, maintain consensus among nodes and tolerate crash faults of non-leader (follower) nodes under the coordination of a leader node. Leader election ensures that Raft also tolerates crash faults of the leader node. However, these designs do not suffice to prevent a Byzantine fault from breaking the consensus. For example, a malicious leader can allow two disjoint sets of honest followers to maintain two different logs. §4.1 introduces Raft in detail.

1.1.2 Our Contributions. Our contributions are threefold:

- **Accountably-Safe Consensus:** We propose Raft-Forensics, a lightweight modification of the Raft protocol that provably guarantees to expose at least one node that committed Byzantine faults when consensus is violated. Note that we cannot guarantee to detect more than one Byzantine node, as only one malicious node is needed to break CFT consensus; however, for certain classes of attacks involving multiple Byzantine nodes, we are able to detect multiple misbehaving nodes (§B.3). The key intellectual contribution is in identifying the simplest possible algorithmic changes to the consensus protocol, while acquiring accountability for Byzantine faults and retaining crash fault tolerance. Our key finding is such an identification: adding only new signatures during log replication and leader election. Theoretically proving that these changes are sufficient requires a delicate reasoning about what cases can (and cannot) occur in an execution, an analysis of independent interest in the literature on proofs of consensus protocols. As demonstrated in Table 1, Raft-Forensics achieves both Byzantine fault accountability and high performance, whereas other leading approaches typically achieve at most one of the two.
- **Performance Evaluation:** We implement Raft-Forensics as a fork of nuRaft, a popular C++ implementation of Raft. We evaluate its performance compared to Raft and two leading BFT protocols, Hotstuff [55] and Dumbo-NG [22]. We observe in Fig. 1 that Raft-Forensics achieves performance close to vanilla Raft, and has a significant advantage over leading BFT consensus algorithms (experimental details in §7). For instance, it achieves a maximum throughput that is 87.8% the maximum throughput of vanilla Raft, at 1.46× the confirmation latency. In contrast, leading BFT protocols achieve only 18.9% the maximum throughput at 6.78× the transaction confirmation latency.
- **Fault Tolerance in Liveness:** It is impossible to provably expose certain Byzantine faults that only affect liveness. In light of this, we present Live-Raft-Forensics that *tolerates* such faults committed by one node. Algorithmically, Live-Raft-Forensics introduces a new two-shot audit procedure by nodes that suspect liveness faults; it also changes the leader election process to be round-robin rather than the randomized leader election specified by vanilla Raft. We theoretically prove that Live-Raft-Forensics guarantees the completion of every client transaction within finite period of time in a partially-synchronous network.

2 RELATED WORK

The relevant related work can be split into two categories. The first is fault-tolerant protocols (CFT and BFT). The second is accountability for CFT and BFT protcols.

CFT and BFT protocols. CFT protocols are designed to handle crash faults, where nodes may fail but do not exhibit malicious behavior. Paxos [36] is a foundational CFT protocol, whose variants have been widely adopted in distributed systems [3, 6, 12, 49]. However, Paxos is known for its complexity and challenges in implementation [52]. Raft [42] is a CFT protocol that aims to provide a more understandable and easier-to-implement alternative to Paxos. Raft has been adopted in various database systems [1, 44, 50].

BFT protocols are instead designed to withstand Byzantine faults [16]. Because they must withstand a stronger threat model, BFT protocols typically exhibit more complex communication patterns (e.g., increased rounds of communication, larger message sizes, or more all-to-all broadcasts), which reduce performance. Prior work has mainly considered two network settings: partially synchronous and asynchronous (details in §3). In the partially synchronous setting, PBFT [7, 8] is the first practical BFT protocol, featuring quadratic communication complexity in the number of

		Crash Fault Tolerance	Byzantine Fault Tolerance	Byzantine Fault Accountability	Performance
BFT	Hotstuff	$0 \sim n/2$	$n/3$	Yes	Low
	Dumbo-NG	$0 \sim n/2$	$n/3$	Not Studied	Low
CFT	Raft	$n/2$	0	No	High
	Raft-Forensics (ours)	$n/2$	0	**Yes**	**High**

Table 1: Summary of fault tolerance properties. Accountability of BFT algorithms

nodes. HotStuff [55] subsequently provided both optimistic responsiveness and linear communication complexity, at the cost of an extra round of communication. Fast HotStuff [33], DiemBFT-v4 [51], and Jolteon-and-Ditto [24] further optimize the good-case latency over HotStuff at the cost of quadratic complexity during leader rotation. Significant progress has also been made among asynchronous BFT protocols, including HoneyBadger BFT [41], BEAT [15], Dumbo [27], Speeding-Dumbo [26], and Narwhal and Tusk [14]; yet, achieving strong security, low latency, and high throughput simultaneously remains a challenge [22]. Dumbo-NG [22], a recently proposed asynchronous BFT protocol, uses concurrent transaction dissemination and asynchronous agreement to improve latency-throughput tradeoffs. Despite these significant improvements, there is still a gap between the performance of BFT and CFT protocols.

To address these security and performance challenges, some work has incorporated trusted hardware, such as trusted execution environments (TEEs) [45] or *enclaves* (e.g., SGX [13]). The hardware provides non-equivocation guarantees, thus transforming Byzantine failures into crash failures and improving both the fault tolerance threshold and performance [10, 34, 54]. For instance, EngRaft [53] is a secure enclave-guarded Raft implementation that combines the best properties of a performant CFT protocol and a secure SGX enclave. However, these TEE-based designs have fundamental limitations (e.g., safety violations under rollback attacks) that hinder practical deployments [28].

In this work, we compare the performance of Raft-Forensics with leading CFT and BFT protocols (without using trusted hardware), including Raft, Hotstuff, and Dumbo-NG. Our experiments indicate that Raft-Forensics performs close to Raft while holding a significant advantage over both HotStuff and Dumbo-NG.

Accountability. Accountability allows protocols to identify and hold misbehaving participants responsible when security goals are compromised [35]. In the context of CFT and BFT protocols, accountability allows a protocol to identify culpable participants when security assumptions are violated and demonstrate their misconduct. Recent work [47] has examined several widely used BFT protocols and assessed their inherent accountability levels without altering the core protocols. Since CFT protocols are explicitly designed to handle only crash faults, integrating accountability offers a lightweight enhancement to detect Byzantine actors without the performance costs of tolerating them. A prior work [25] explored the accountability of the Hyperledger Fabric blockchain, which features a pluggable consensus mechanism. This study conducted a case analysis of incorporating accountability into a Hyperledger Fabric system underpinned by a CFT protocol, Apache Kafka [23] (called Fabric*). However, this work treats the consensus module as a cluster, offering accountability only at the level of the entire consensus group (not individual nodes within the group).

In contrast, we aim to identify and attribute Byzantine faults to individual misbehaving consensus replicas participants. Fabric* introduces two primary modifications. First, parties must sign every message they send. Second, it enforces a deterministic block formation algorithm to eliminate ambiguity. However, these changes are neither necessary nor sufficient for ensuring accountability in Raft. Additionally, Fabric* ensures liveness only when all parties are honest while our Live-Raft-Forensics approach (§8.2) guarantees liveness even in the presence of a single Byzantine corruption. Finally, [25] does not empirically evaluate their system, whereas we evaluate performance empirically.

PeerReview [29] builds a framework for accountability that applies to general distributed systems. Although it accounts for Byzantine faults in Raft as Raft-Forensics does, it has substantially higher overhead communications and space requirements than Raft-Forensics, which we discuss in §C in detail. In particular, under high workloads, PeerReview has 5× higher communication overheads than Raft-Forensics and a prohibitive spatial overhead compared to a constant spatial overhead in Raft-Forensics. PeerReview requires nodes to audit each other, instead of assuming a central auditor as we do (§3). To address this difference, we also analyze the theoretical complexity of a variant of PeerReview, which we term HubReview, in §C. However, even HubReview incurs substantially higher communication and memory overhead than Raft-Forensics. PeerReview also fails to account for Byzantine behavior by the client, while such Byzantine behavior is included in our model (§3). Therefore, Raft-Forensics is not replaceable by PeerReview.

3 MODEL

We assume there exist $n = 2f + 1$ nodes in the system, where f is the maximum number of nodes that may encounter crash faults. We also assume the system is partially synchronous, i.e., there exists a global stabilization time (GST) and a constant time length Δ, such that a message sent at time t is guaranteed to arrive at time $\max\{\text{GST}, t\} + \Delta$. The GST is unknown to the system designer and not measurable by any component of the system. Table 2 lists the relevant notation for Raft-Forensics.

Threat Model. In addition to the above assumptions by vanilla Raft, we further assume the existence of *one* node that encounters Byzantine faults, which is controlled by an adversarial party. This node may or may not overlap with the f crash-faulty nodes. Consequently, by manipulating the Byzantine faults, it can threaten both liveness and safety of the system. When liveness is compromised, it is difficult to find evidence to accuse the adversarial node, which

Symbol	Description	Requirement
f	Max number of crash-faulty nodes	
n	Number of nodes	$= 2f + 1$
Δ	Max message delay after GST	
$\mathcal{V}$	Set of all nodes	

Table 2: Parameters in system design

motivates our fault-tolerant design in §8. Therefore, we focus on accounting for Byzantine faults that attack safety.

Under the framework of Byzantine failures, we make the following assumptions about the adversary's capabilities. The adversary is capable of determining whether, when, and what to send to every honest node. It is also capable of reading and copying the persistent states of every honest node without any delays. In addition, it has full information about the network traffic, for example, end-to-end latency between every pair of nodes. However, it is unable to influence the honest nodes or the communication among them. Based on these capabilities, we list some adversarial strategies in §4.2.

We assume the existence of a Public Key Infrastructure (PKI). Each node u maintains a pair of secret and public keys $s[u]$ and $p[u]$ respectively, and $p[u]$ is known by every party in the system, including other nodes, clients, and auditors. A node u can use its secret key $s[u]$ to create an unforgeable signature over (the hash of) an arbitrary message m, denoted by $\sigma_{s[u]}(\texttt{Hash}(\mathrm{m}))$, and the signature can be verified with u's public key, $p[u]$. A preimage-resistant cryptographic hash function `Hash` is known to every node. Both signing a message and verifying the signature can be executed in time that is polynomial in message size.

Client. Clients are the source of requests in the distributed system. As in vanilla Raft, clients can submit requests to an arbitrary server node who processes the requests. If the node is not the active leader, it will forward the requests to the leader. After the request is committed, a client receives a tamper-evident receipt from the leader, which includes the index and the term of the log entry handling the request. Clients are unable to influence server machines or inter-node communications.

In addition, we assume that each client request is sequenced and tamper-evident; i.e., an adversarial server node cannot modify a client request without being detected. We allow clients to query any node about whether a given log entry (with index and term, §4.1) conflicts with the node's log list, and these queries will be answered if the node is honest. We also allow clients to submit receipts to the auditor (described below), where a receipt is effectively a notification from a leader to the client that the client's transaction has been executed. The precise form of the receipt is part of the design problem described in §4.3.

Auditor. In order to identify the adversary, we introduce an *auditor* into the system. It is permitted to query the full state of any node (details in §5.1.1) and accept receipts from the client, which come from the server node who processed the client's requests. In response, an honest node always provides the required information to the auditor; a Byzantine node can respond arbitrarily. The auditor determines the safety of the system by checking data legitimacy and consistency among the nodes, as a function of the received state information. However, auditors are unable to influence the system directly. Our main research goal is to define modifications to the consensus protocol and an auditing algorithm that jointly enable an auditor to uncover the identity of the adversarial node if the State Machine Safety property is violated or a legitimate client receipt conflicts with the state machine logs (described next).

4 BACKGROUND AND PROBLEM STATEMENT

4.1 Background on Raft

The Raft consensus algorithm [42] manages a replicated log among multiple distributed nodes in a distributed system. In normal cases, Raft maintains the system consensus by letting a *leader* node coordinate the other nodes, namely the *follower* nodes. Once the leader receives a client request, it wraps the request into a log entry and marks it *uncommitted*. Then, the leader broadcasts the entry to all the followers, who replicate the entry and acknowledge the success in replication. After the leader successfully replicates the entry to more than half of the nodes, it marks the entry *committed*, applies the entry to the state machine and responds to the client. It also notifies the followers to commit the entry in the same way. In Raft, this entire procedure is called *log replication*.

During log replication, the nodes may differ in their log lists. We use a metric called *freshness*[1] to evaluate and compare the log lists between nodes. Generally, a node is fresher than another means the node is more progressive in replicating log entries. We formally define freshness as below.

Definition 4.1 (Freshness). A log entry x's freshness is denoted by the tuple (t_x, j_x), which represents the `term` and `index` of x. Entry x is *as fresh as* entry y if $t_x = t_y \wedge j_x = j_y$. x is *fresher* than y if $(t_x > t_y) \vee (t_x = t_y \wedge j_x > j_y)$. In contrast, x is *staler* than y if $(t_x < t_y) \vee (t_x = t_y \wedge j_x < j_y)$.

Under this definition, any finite set of entries can be *totally ordered* by node freshness, implying that each entry has a freshness rank inside the set. For convenience, we extend the concept of freshness to nodes. The freshness of a node is equal to the freshness of its last local log entry, which could still be uncommitted.

It is trivial to tolerate the followers' crash faults during log replication, but Raft requires *leader election* to tolerate crash faults from the leader. Leader election is initiated by a follower who detects that the current leader stops sending heartbeat messages. The follower turns into a *candidate* and asks every other node for vote. A voter votes for a candidate if and only if the candidate's log entries are not *staler* than its own. When a candidate obtains votes from more than a half nodes, it becomes the new leader. Leader election ensures that 1) empirically, leader election eventually ends with a single candidate turning into a leader, and 2) the new leader's logs are fresher than or same as more than a half nodes.

Notice that Raft maintains consensus only via communication between the leader and the followers. This gives Raft optimal communication complexity, but it also introduces vulnerabilities against Byzantine faults committed by the leader.

[1]The original Raft paper uses the concepts, but has no such abstraction. We formalize this for convenience.

4.2 Safety properties

Raft [42] proves five safety properties, which together guarantee that consensus is reached between any pair of non-faulty nodes.

- **Election Safety**: at most one leader can be elected in a given term.
- **Leader Append-Only**: a leader never overwrites or deletes entries in its log; it only appends new entries.
- **Log Matching**: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.
- **Leader Completeness**: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
- **State Machine Safety**: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

In their setting, there are no Byzantine faults, so these conditions must hold for all nodes. However, if there exists one Byzantine node, the conditions can be easily violated. We list two examples of attacks violating State Machine Safety, where Examples 4.2 and 4.3 can be launched by an adversarial leader and follower, respectively.

Example 4.2 (Forking). Let a be a Byzantine leader. It divides the $2f = n - 1$ followers into two disjoint subsets P, P' of equal size. From two distinct client requests, it creates two log entries x_i and x_i', both having the same index i and the same predecessor x_{i-1}. Then, it broadcasts x_i to P and x_i' to P', respectively and exclusively. Being replicated to $f + 1$ nodes (including itself), both entries x_i and x_i' are committed by the leader a.

Example 4.3 (Bad vote). Suppose a is a Byzantine follower and v is an honest candidate for leader who lacks the latest committed log entry B. Suppose v can only acquire exactly f votes since the other $f + 1$ nodes have already replicated B (including a). However, the adversarial follower a casts an illegal vote for v, allowing v to win the election undeservingly. This results in v committing a different entry with the same index as log entry B, thereby violating the safety property.

4.3 The Accountability Properties

We emphasize that both Raft and Raft-Forensics solve the state machine replication (SMR) problem. In other words, the consensus of Raft variants is equivalent to the consistency of *state machines*, which is precisely stated by the State Machine Safety property. Although the other four properties are important, their violations sometimes do not lead to consensus failures over committed log entries between two honest nodes. Therefore, we regard State Machine Safety as our core safety property. Note that it is **not** our goal to guarantee that the property always holds in the existence of a Byzantine node. Instead, we answer the question of how to find the culprit after an attack takes place and damages security.

To summarize, our goal is to design a protocol and an audit algorithm that satisfies the following definition of accountability.

Definition 4.4 (Accountability). Let C denote a consensus algorithm that solves the SMR problem. C has *accountability* if there exists a polynomial-time auditing algorithm $\mathcal{A}$, such that

(a) $\mathcal{A}$ takes the states of C as input.
(b) If the State Machine Safety property is violated, $\mathcal{A}$ outputs a non-empty set of nodes and irrefutable proof that each member of the set violated protocol. Otherwise, $\mathcal{A}$ outputs $\perp$.

Furthermore, we extend our goal to designing a protocol that satisfies client accountability as defined below. Client accountability ensures that a client can always trust the receipts it receives from a leader.

Definition 4.5 (Client Accountability). Let C denote a consensus algorithm that has accountability. C has *client accountability* if there exists a polynomial-time auditing algorithm $\mathcal{A}'$, such that if an honest node has a different committed log entry at the same height as the entry in a client's legitimate receipt, $\mathcal{A}'$ outputs a non-empty set of (server) nodes and irrefutable proof that shows the culpability of each node in the set.

5 DESIGN

In order to participate in Raft consensus, there are three basic activities: voting for candidate leaders, creating log entries and acknowledging log entry replications. We modify Raft in such a way that nodes must leave irrefutable cryptographic evidence of each basic activity.

Determining the minimal added states and procedures that are necessary for accountability is delicate. While it is straightforward to propose a set of modifications that guarantee accountability, minimizing the associated overhead required careful refinement. Each added state in Raft-Forensics is critical for Thm. 6.1 of accountability and the auditing algorithm in Alg. 1.

5.1 Raft-Forensics

We describe the design of Raft-Forensics. §5.1.1 first lists our modifications to the states a node must maintain. Based on the states, §5.1.2 specifies how the nodes interact with each other during log replication, and §5.1.3 specifies interactions during leader election. For ease of understanding, we describe the modifications in text here. We provide precise pseudocode in Appendix §B.1.

5.1.1 Maintained States. A node in vanilla Raft maintains three persistent states: (1) its current term `currentTerm`, (2) the latest node that it voted for, and (3) the log list. A persistent state must be updated on stable storage before responding to RPCs. We add a few other persistent states: (4) the hash pointer cache, (5) the leader signatures, (6) the commitment certificate (CC), and (7) the election list. We describe each in detail below, but briefly: the *hash pointers* are tamper-evident proofs of the elements and their order in a log list. The *leader signatures* ensure that every entry is authenticated by its creator. The *commitment certificate* and the *election list* preserve irrefutable traces of votes for an entry or a leader.

Log List. A log list is an ordered list of *log entries* maintained by both Raft-Forensics and vanilla Raft. The basic format of a log entry includes three fields: 1) `term`, a non-decreasing number marking a continuous leadership during when the entry is created; 2) `index`, the sequence number starting at 0 for a universal initial entry; and 3) `payload`: the content of the log entry.

Field Name	Description
Commitment Certificate (CC) Fields	
`term`	term number of the entry
`index`	index of the entry
`pointer`	hash pointer $h_{\texttt{index}}$
`voters`	A list of at least $f + 1$ voters
`signatures`	A list of the signatures by corresponding voters
Vote Request Fields	
`leader`	the ID of elected leader
`term`	the term of proposed election
`freshness`	(`term`, `index`) of the latest log entry owned by `leader` before election
`pointer`	hash pointer $h_{\texttt{index}}$
Leader Certificate (LC) Fields	
`request`	vote request that voters have signed over
`voters`	A list of at least $f + 1$ voters
`signatures`	A list of the signatures by corresponding voters
Client Receipt Fields	
`entries`	log sub-list starting with the one with the client transaction and ending with the one with CC
`pointer`	hash Pointer of the starting entry's predecessor
`certificate`	CC of the last entry in `entries`

Table 3: Elements of a commitment certificate, a vote request, a leader certificate, and a client receipt. All elements in this table are specific to Raft-Forensics.

Hash Pointers. In Raft-Forensics, each node computes the hash pointer of entry indexed by i as follows. (|| denotes concatenation.)

$$h_0 = \texttt{zero}, \quad h_i = \texttt{Hash}(h_{i-1} \| \texttt{term}_i \| \texttt{index}_i \| \texttt{payload}_i), \ \forall i \in \mathbb{N}.$$

It suffices for each node to maintain the hash pointer of only the latest entry and the latest *committed* entry. The pointer ensures that the log list cannot be tampered with.

Leader Signatures. For each unique term in the log list with last entry indexed by i, each node maintains the signature $\sigma_{s[\ell]}(h_i)$ over the hash pointer h_i signed by the leader ℓ creating the entry. This signature irrefutably shows that the leader has locked itself to the entire branch ending with the signed entry. Should a different signature exist on a conflicting branch, an auditor can directly expose the leader. All signatures are stored in a dictionary keyed by the term number.

Commitment Certificate. Raft-Forensics adds the requirement to store commitment certificates (CC's) of only the latest committed log entry. The CC irrefutably shows that the corresponding log entry has been replicated to a quorum of nodes. A CC consists of four fields: `term`, `index`, `voters` and `signatures`, as shown in Table 3. The `signatures` may include the leader's own signature on the log entry or the followers' signatures which is mandatory for an acknowledgement to the leader of entry replication. Having produced such a signature, a node is considered a voter over the entry in question. Consequently, `voters` consists of the identities of the voters and `signatures` consists of their digital signatures on the entry.

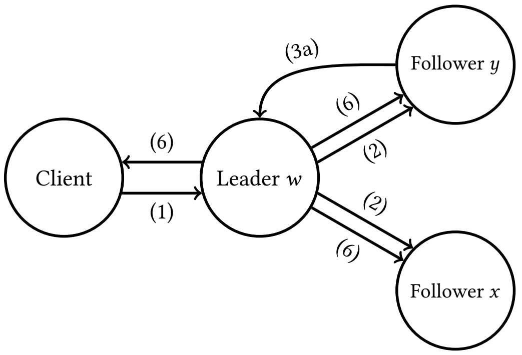
Election List. Like leader signatures, the election list is also a dictionary keyed by term, where each value is the corresponding leader certificate for the term. A *leader certificate* (LC) consists of three fields: `request`, `voters` and `signatures`, where `request` further consists of three fields, as listed in Table 3. `request` includes information about the leader's freshness at the beginning of the election. `voters` and `signatures` are analogous to those for a CC, except that the signatures are created on the hash of `request` instead of a hash pointer. The certificate irrefutably shows that by the election, the leader is at least as fresh as (Def. 4.1) a quorum of nodes in the system. Like leader signatures, the number of LC's also equals the number of different terms in the log list.

5.1.2 *Log Replication.*

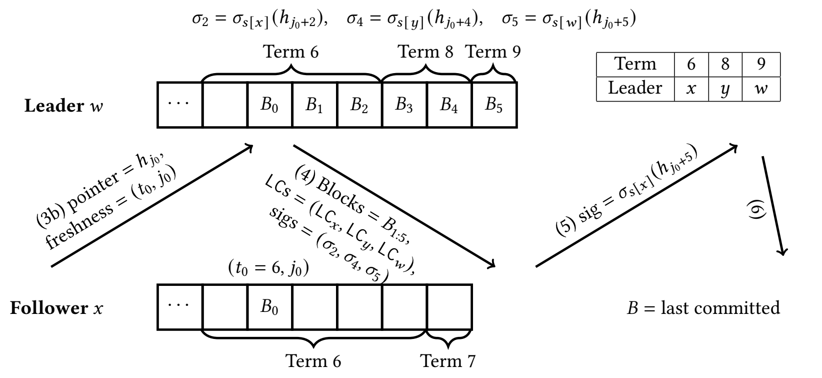
Raft-Forensics also modifies the log replication process. We demonstrate the process by a toy example, illustrated in Figure 2. In this example, the system has a leader w and only two followers x and y (Figure 2a). The steps are:

(1) A client submits a request to the current leader w.
(2) w creates a new log entry (namely B_5 with index i), appends it to its local log list, and broadcasts it to all the follower nodes x and y. In Raft-Forensics, along with B_5, the leader must attach its signature $\sigma_{s[w]}(h_i)$ such that the recipients can verify the integrity of the chain. (Recall that h_i is the hash pointer respective to the entry's index i.) Then, as in vanilla Raft, the leader waits for acknowledgements from at least f followers, signaling success in replicating the entry system-wide.
(3) When a follower receives B_5, it checks whether the entry can be directly appended to its log list. This depends on whether the follower has the entry's predecessor B_4, and whether the leader's signature is correct; the latter condition is specific to Raft-Forensics. This check yields three possible cases:
    (3a) If B_5's predecessor B_4 is exactly the follower's last log entry, the follower appends B_5 to its log list and acknowledges the leader with its signature on h_i. In this example, follower y acknowledges w with signature $\sigma_{s[y]}(h_i)$.
    (3b) if B_5 is too fresh to append, i.e., the follower does not have its predecessor, the follower asks the leader for **log synchronization**.
    (3c) If B_5 is not fresher than the follower's last log entry, the follower rejects it.
(6) After the leader w receives f (out of $2f$) acknowledgements, it assembles a CC (Table 3) by the f signatures within the acknowledgements plus its own signature on the entry. Then, it broadcasts the CC to all the followers, who will accept once they verify the signatures and ensure the corresponding entry does not conflict with their local committed entries. This may trigger log synchronizations (see below) if the new entry is too fresh. Once accepting a CC, both the leader and the followers apply it to their state machines and replace their stored CC's with the latest one. The leader also notifies the client with a receipt formatted as in Table 3, which convinces the client that their request has been committed if the CC can be verified.

Log Synchronization. A log synchronization is triggered when a follower receives a replication of a new log entry that is too fresh to append, or a commitment notification of an entry that is fresher than its latest block. In Fig. 2b, we demonstrate the process with a



(a) The overall flow chart of a log replication.


$\sigma_2 = \sigma_{s[x]}(h_{j_0+2}), \quad \sigma_4 = \sigma_{s[y]}(h_{j_0+4}), \quad \sigma_5 = \sigma_{s[w]}(h_{j_0+5})$
Term 6
Term 8
Term 9
Leader w
$\cdots$ | | B_0 | B_1 | B_2 | B_3 | B_4 | B_5
Term | 6 | 8 | 9
Leader | x | y | w
(3b) pointer $= h_{j_0}$, freshness $= (t_0, j_0)$
(4) Blocks $= B_{1:5}$, LCs $= (\mathsf{LC}_x, \mathsf{LC}_y, \mathsf{LC}_w)$, sigs $= (\sigma_2, \sigma_4, \sigma_5)$
(5) sig $= \sigma_{s[x]}(h_{j_0+5})$
(6)
$(t_0 = 6, j_0)$
Follower x
$\cdots$ | | B_0 | | | |
Term 6
Term 7
B = last committed

(b) Log synchronization. To generalize over commitment-triggered synchronizations, we let w send its signature σ_5 and LC_w, which are redundant in this case.

Figure 2: Flow charts of messages in log replication. The system has 3 nodes and is currently in Term 9.

simple example, where follower x triggers the process after leader w sends an entry B_5 that is too fresh to append.

(3b) x initiates the synchronization by replying with the freshness (t_0, j_0) and the pointer h_{j_0} of its latest *committed* entry B_0 to the leader. Here we assume i, the index of B_5, equals $j_0 + 5$.

(4) If the leader has a conflict with h_{j_0}, it drops the request. Otherwise, it responds to the follower with the entire sub-chain starting from the entry with index $(j_0 + 1)$ and ending with the leader's latest entry. For each unique term of the entries, the leader must also send the LC and the leader's signature on the term's last entry.

(5) If the sub-chain, the LC's and the signatures are all valid, the follower replaces all current uncommitted entries with the sub-chain, and acknowledges the leader with its signature of the sub-chain's last entry.

In summary, our modifications to log replication guarantee that two consensus activities will inevitably leave evidence. During the creation of a log entry, a leader must attach its signature to authenticate the entry. During acknowledgement of entry replication, a follower must send its signature to the leader, which proves the follower has replicated the entry. These signatures are collected into CC's.

5.1.3 Leader Election. In vanilla Raft, a leader election is typically initiated by a follower who detects that the current leader fails to send periodic heartbeat messages that signal its liveness. The initiating follower turns into a *candidate*, which is an intermediate role between the follower and leader roles. A leader election eventually ends with a candidate turning into a leader. This role switching requires that the candidate collect a quorum of votes, i.e., at least $f + 1$ votes in total, including its own vote.

Recall that we aim to track the activities of voting for candidates. We make the following modifications in Raft-Forensics to ensure that the voters for every leader are traceable. When a node v_C becomes a candidate, it broadcasts a vote request to every other node that is structured as in Table 3. Then, v_C waits for responses from followers. As required in vanilla Raft, a follower v_F votes for v_C if v_F is not fresher than v_C and v_F's term is less than the term in v_C's request. Otherwise, v_F rejects the vote request. When v_F decides to vote, it must send its signature on the vote request back to v_C in Raft-Forensics, instead of "ok" in vanilla Raft.

If the candidate collects $f + 1$ votes (including its own), it assembles an LC with the $f + 1$ signatures included in the votes, which irrefutably shows the candidate is fresh enough to gain $f + 1$ votes. To finally turn into a leader, the candidate must send the LC to every other node. Every node who agrees with the new leader keeps the LC in storage, corresponding to the state of election list in §5.1.1.

5.1.4 Auditing. During an audit, each node provides the auditor with data including the log list containing only committed entries, the leader signatures, the CC of the last committed log entry, and the election list. After collecting this data from all nodes, the auditor checks the data integrity of each node, i.e., whether the data of each node is internally consistent (e.g., the index field is incrementing, all certificates include the correct number of valid signatures from other nodes, etc.). The integrity checking algorithm is specified in Appendix B.2. If this check passes, the node's data can be used for the consistency checks in the next step.

Then, the auditor checks data consistency between the node with the longest (committed) log list and all the other nodes. The consistency checking algorithm should detect whether they disagree in consensus, i.e., whether they have committed different entries at the same index. If so, the algorithm outputs a set of the Byzantine nodes who caused the disagreement with the leader signatures, the CC's and the LC's. Otherwise, the algorithm outputs an empty set. Theorem 6.1 theoretically prove that such algorithm exists, and we give an example in Algorithm 1.

Finally, the auditor finds the Byzantine nodes in the union of all the outputs of the auditing algorithm. Under our assumption that there exists only one Byzantine node, the size of the output set should be exactly one if consensus fails. However, if the assumption is relaxed and multiple adversarial nodes participate in breaking the consensus, it is possible that the algorithm returns a set with multiple nodes. In this case, the set is not guaranteed to include every Byzantine node, but every node in the set must be Byzantine.

5.1.5 Summary. To summarize, the key differences between vanilla Raft and Raft-Forensics are:

A. Maintained States: hash pointers, leader signatures, a CC of the latest committed block, and an election list maintaining all LC's.
B. Log Replication.
  B1. A leader attaches its signature to the latest entry it creates.
  B2. A follower provides a leader the follower's signature over the last entry it replicates from the leader.
  B3. An honest node can only commit an entry if it obtains a CC for the entry or one of its descendants.
  B4. After a transaction is completed, a client receives a receipt with a CC of a descendant (or self) of the log entry containing the transaction.
C. Leader Election.
  C1. To vote for the candidate, a follower must send the its signature over the candidate's vote request.
  C2. A follower can only accept a new leader if it obtains a valid LC from the leader.

While these differences are algorithmically simple, the exact set of changes required to guarantee accountability (without hurting performance) requiring reasoning about several corner cases. We discuss these in §6 and the proof of Theorem 6.1.

6 RAFT-FORENSICS ACCOUNTABILITY

6.1 Main Result

We present our main result, Theorem 6.1, which shows Raft-Forensics is able to provide accountability when consensus fails. We provide a sketch of proof and include the full proof in Appendix A.1. The main challenge in writing this proof is enumerating the correct and minimal set of cases for analysis.

THEOREM 6.1 (RAFT-FORENSICS ACCOUNTABILITY). *Raft-Forensics achieves accountability (Def. 4.4).*

Proof Sketch. In Raft-Forensics, a violation of State Machine Safety is equivalent to the following statement: there exist two honest nodes u and v with **legitimate** log chains and certificates, but neither log chain is a prefix of the other. Data legitimacy is specified in Appendix B.2.

To analyze a conflict, we should first check whether each term is used by only one leader. If not (Case 0), we can accuse the voter who voted for both leaders. Otherwise, we proceed to cross-checking the log lists. We let t_u, t_v denote the terms of u and v's CC's. Without loss of generality, we assume $t_u \geq t_v$. In this step, our tactics rely on whether the logs have conflicting entries at term t_v.

If so (Case 1), we accuse the leader that created the entries. Otherwise (Case 2), we search for t_v's succeeding term t_ℓ in u's log list, where the leader of t_ℓ is ℓ. We show that under this case, there must exist a voter that voted for v's CC and ℓ's leader candidate at the same time. We prove that this is illegal in the full proof in Appendix A.1. By analyzing these cases, we show that if State Machine Safety is violated, we can correctly identify (at least) 1 Byzantine node. ∎

6.2 End-to-End Safety for Clients

We have thus far focused on the consistency *between server nodes.* However, there still exists safety concerns with respect to the client. Although after log replication, a client receives a legitimate receipt, it is still possible that the log entry is **uncommitted** by the leader, and overwritten later by a conflicting entry.

Example 6.2 (Commitment Fraud). Suppose the leader is adversarial. It replicates the log entry ε including the client's request to f honest nodes, assembles a CC of ε, and replies the CC to the client. However, it withholds the CC from all the followers and commits a new log entry ε' at the same position of ε. All nodes overwrite ε with ε', while the client still thinks ε is already committed.

In this example, the State Machine Safety property is not violated. (Same for the other 4 properties in §4.2) However, this attack may cause serious safety problems such as double spending in practice. To account for this type of Byzantine fault, we extend the scope of auditing to the clients. As described in Table 3, when a transaction is complete, a client gets a receipt from the leader that includes 1) `entries` including all the log entries starting from the one with the transaction and ending with the one with CC; 2) `pointer` of the **predecessor** of the starting log entry; 3) `certificate`. Both the client and the auditor can determine whether the receipt is legitimate by verifying the signatures in `certificate` over the hash pointer of the ending log entry derived from `pointer`. If `certificate` is valid, it is critical in revealing Byzantine nodes if the log entry conflicts with a log list maintained by an honest node.

With the above extensions to auditing, we show in Proposition 6.3 that for Raft-Forensics, there exists an algorithm that is guaranteed to find at least one Byzantine node if a client detects a commitment fraud. The proof of Proposition 6.3 is in §A.2.

PROPOSITION 6.3 (CLIENT ACCOUNTABILITY). *Raft-Forensics achieves client accountability (Def. 4.5).*

7 EVALUATION

We implement the log replication component of Raft-Forensics[2] in C++ based on nuRaft v1.3 [18] by eBay. We choose to focus on the log replication component as this is the "average case" behavior, and leader changes are expected to be rare. With roughly 2,500 lines of code, our implementation fully expands nuRaft with our OpenSSL-based designs in log replication, which correctly reflects the throughput and latency performances between leader elections. We choose the SHA-256 hash function and Elliptic Curve Digital Signature Algorithm (ECDSA) over the secp256r1 curve. For commitment certificates, we used concatenated ECDSA signatures by all the signers.

We evaluate Raft-Forensics in two phases – online phase (§7.1) and offline phase (§7.2). In the online phase, we benchmark the performance of Raft-Forensics over a WAN. Generally, Raft-Forensics is significantly better than both BFT protocols in both peak throughput and latency-throughput tradeoff, but slightly worse than vanilla Raft. In the offline phase, we inspect the auditing procedure that scans server logs for adversarial behaviors. The auditing procedure has computational complexity linear in the size of data, where data chunking improves its efficiency in memory requirements.

7.1 Online Evaluation

Setup on AWS. We evaluate Raft-Forensics over a WAN to demonstrate a geo-redundant deployment for increased resilience [11].

[2]URL redacted for double-blind review.

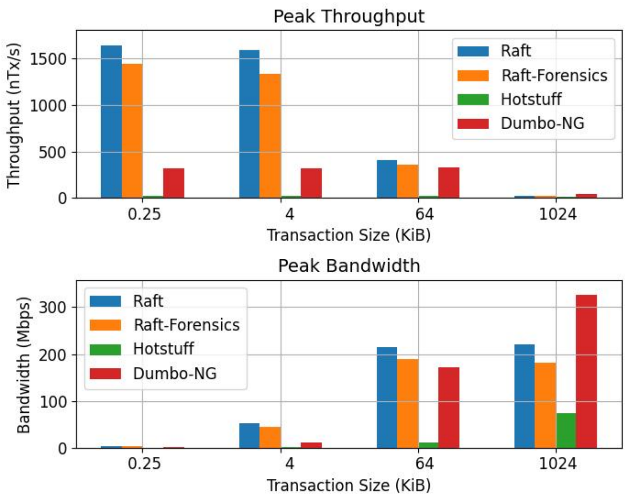


Figure 3: Peak transaction and bandwidth throughputs of consensus algorithms.

We simulated the WAN environment by deploying Raft-Forensics and other baseline protocols on 4 dedicated `c5.large` instances on AWS, where each instance has 2 vCPUs and 4 GB Memory. Because some typical applications of Raft-Forensics require the nodes to be distributed domestically, we deployed the instances in distinct AWS datacenters (Virginia, Ohio, California and Oregon) in the US.

Baseline Protocols. We compare the performance of Raft-Forensics against Raft [18], Hotstuff [43] and Dumbo-NG [20]. We used eBay's NuRaft [18] for Raft. For both Raft variants, we let the leader machine spawn transactions. We chose the implementation in GoLang on Github [43] for Hotstuff. To avoid consensus failures of Hotstuff caused by resource contention, we used an extra instance in Virginia to submit client transactions. We chose the official implementation [20] for Dumbo-NG, where all servers spawn transactions by default. For a fair comparison between both Raft variants and the BFT protocols, we disabled transaction batching for BFT protocols by setting batch sizes to 1. This may have a negative impact on throughput at small transaction sizes, but the bandwidth throughput suffers less (high transaction sizes simulate batching).

Experimental Settings. We benchmark each protocol by two metrics – transaction latency and throughput. Latency is measured by the average time difference between when a transaction is confirmed and when it is sent to the servers. Throughput is measured by the average number of transactions processed per second during an experiment. The experiments are configured by two key parameters – transaction size and number of concurrent clients. The transaction sizes range from 256 Bytes to 1 MB. For each transaction size, we sweep the number of concurrent clients sending transactions. Under each configuration of transaction size and client concurrency, we run all the nodes and client processes simultaneously for 1 minute for Dumbo-NG which requires up to 40 seconds to warm up, and 20 seconds for the other algorithms that do not require warm-up. We measure transaction latency and throughput by the average of 5 repeated runs to reduce random perturbations. Typically, as the number of clients increases, throughput increases linearly at first, and plateaus when the consensus protocol is saturated. In contrast, latency is insensitive to the number of clients before the saturation, but rapidly increases when the bottleneck throughput is reached.

We compare the performance of Raft-Forensics against other consensus protocols in two major aspects.

- **Peak throughput.** We measure the peak throughput of each consensus protocol by the maximum number of transactions processed per second over all the number of concurrent clients. Fig. 3 presents the performance of all protocols under transaction sizes 256 Bytes, 4 KiB and 64 KiB. In comparison to Raft, Raft-Forensics has an approximately 10% loss in peak throughput under various transaction sizes, which is brought by the cryptographic operations involved. At all transaction sizes, Raft-Forensics is superior to both BFT protocols in terms of peak throughput. Note that Dumbo-NG is less sensitive to transaction sizes, possibly benefiting from the concurrent execution of transaction dissemination and asynchronous agreement. Exploiting this gain can be a direction for future works in CFT protocols.
- **Latency-Throughput tradeoff.** Under each transaction size, we measure the latency-throughput curve parameterized by number of concurrent clients. Fig. 4 shows the latency-throughput tradeoffs of the four protocols under various transaction sizes. In these plots, Hotstuff's tradeoff appears vertical because the lowest implemented level of client concurrency already achieves peak throughput, so increasing the number of client threads linearly increases latency. Generally, the tradeoff of Raft-Forensics is slightly worse than Raft, but significantly better than Hotstuff and Dumbo-NG under small transaction sizes. Although Raft-Forensics is outperformed by Dumbo-NG in throughput at higher transaction sizes, it has a significant advantage in latency.

7.2 Offline Evaluation

We next evaluate the offline performance of log auditing. In more detail, Theorem 6.1 states that we can find at least 1 culprit when State Machine Safety is violated, and we further show *how* the culprit is found by Algorithms 1 and 2. Because the algorithm requires that the node's data is legal, we present Algorithm 4 in Appendix B.2 which checks data integrity before the audit.

Complexity Analysis. Recall that n denotes the number of nodes. Let H denote the length of the longest chain and Λ the number of elections in total. Table 4 summarizes the computational complexity of different parts of the auditing process.

The total time complexity of auditing is asymptotically optimal (linear to the size of data $n(H+\Lambda)$, which is required at minimum to ensure data legitimacy), where the complexity of global consistency checks does not depend on the chain length H. On the other hand, the spatial complexity $\Theta(n(H+\Lambda))$ may be acceptable for disk storage, but unacceptable for memory. In this case, chunked storage of log chain is necessary. For instance, for a chunk size $\Theta(\log H)$, the spatial complexity decreases to $\Theta(n(\log H+\Lambda))$, while the time complexity remains the same. Notably, the time complexity of global consistency check slightly increases to $O(n(\Lambda+\log^2 H))$, but is still much less than that of legitimacy checks.

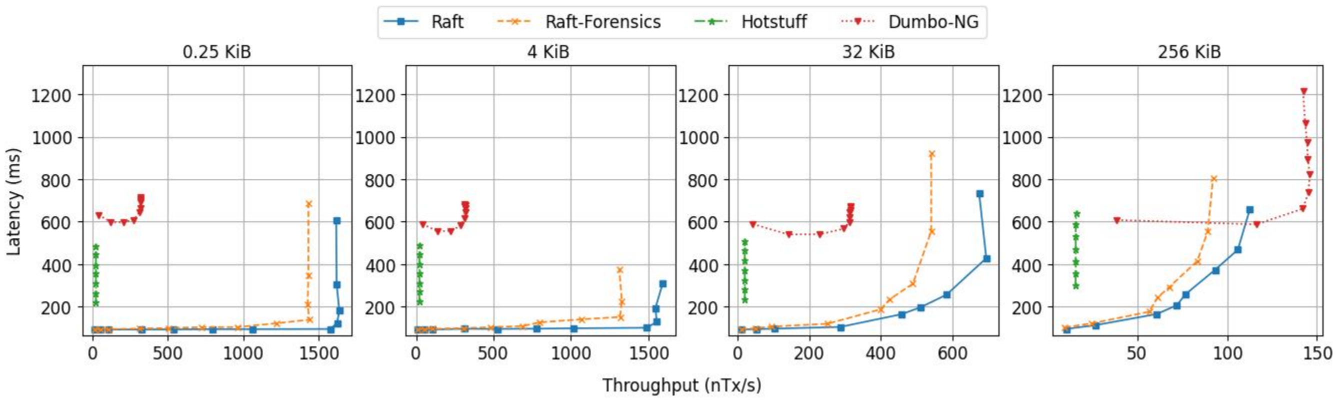


Figure 4: Latency-throughput tradeoff

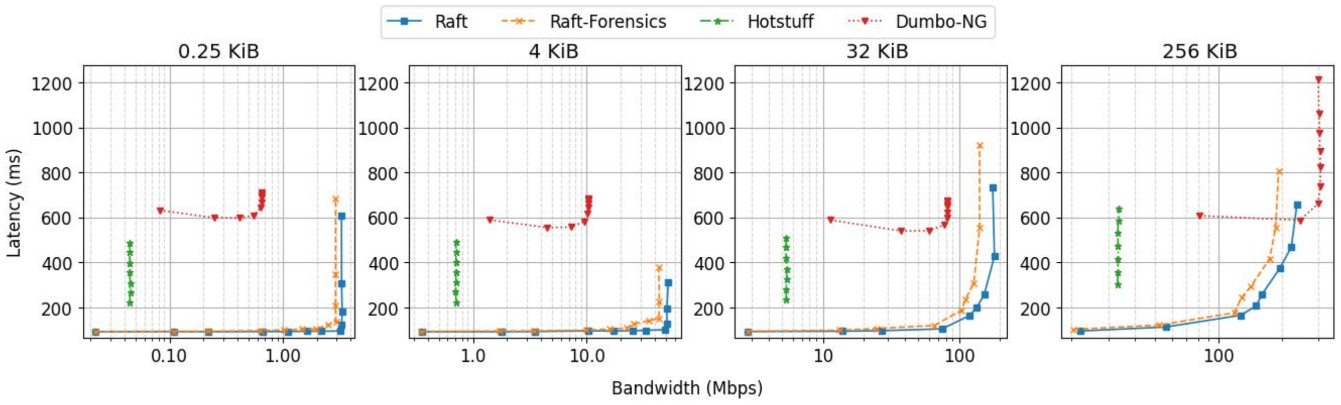


Figure 5: Latency-bandwidth tradeoff

Implementation. We implement the auditing algorithm in Python[3], which can be tested along with a lightweight Raft simulator that achieves better control that the fully-implemented Raft-Forensics in C++ over the leader elections, the adversarial nodes' behavior and race conditions in general. In particular, it is capable of assigning the adversary to a node and simulating the fork and bad vote attacks in Examples 4.2 and 4.3. It ensures that the adversary generates legitimate data to prevent it from being caught before consistency checks. For the best performance in memory usage, it writes the data into chunked files that are available for auditing. In Appendix 7.2, we run benchmarks on the performance of both the data legitimacy and consistency checks of the auditing algorithm. The benchmarks are consistent with our complexity analysis, and demonstrate a significant advantage in chunking data.

Based on the backend software above, we also implement a visualizer based on [48] that demonstrates the attacks and the outputs of the auditing algorithm, including the identity of the culprit and the irrefutable evidence. Fig. 9 shows a screenshot of the visualizer.

[3]URL redacted for double-blind review.

Item	Complexity
Data Legitimacy Check (Alg. 4)	$O(H + \Lambda)$
Global legitimacy Check	$O(n(H + \Lambda))$
Pairwise consistency check (Alg. 1)	$O(\Lambda)$
Global consistency check (Alg. 2)	$O(n\Lambda)$
Total time	$O(n(H + \Lambda))$
Total space (memory)	$\Theta(n(H + \Lambda))$

Table 4: Computational complexity of auditing.

8 LIVE-RAFT-FORENSICS

So far, we provided accountability for Byzantine faults that compromise safety. In this section, we ask how to account for liveness attacks. Although we use a partially synchronous network model that guarantees message arrivals in bounded time, the GST is typically unknown. Consequently, any method that tries to distinguish a late message and a dropped message is unable to achieve 100% accuracy. Therefore, it is impossible to provide irrefutable evidence of Byzantine faults where the adversary fails to send a message or claims failure in receiving a message. This means it is impossible to accurately identify the faulty party if two parties experience a message delivery failure, and both parties accuse each other of

```
Input: Log chain $L_w$, election list $E_w$ and commitment certificate
       $C_w$ for all $w \in \mathcal{V}$, nodes $u$ and $v$
Output: If $u$ and $v$ are consistent, $\varnothing$; otherwise a non empty set of
        Byzantine nodes
/* $u, v$ must pass DataLegitimacy checks in Alg. 4 */
Function PairConsistency($L, E, C, u, v$):
    for $t \in E_u$.keys $\cap$ $E_v$.keys
        if $E_u[t]$.request.leader $\neq$ $E_v[t]$.request.leader
            return $E_u[t]$.voters $\cap$ $E_v[t]$.voters    // Case 0
    $i_v$, $i_u \leftarrow C_v$.index, $C_u$.index
    if $L_v[\min\{i_v, i_u\}] = L_u[\min\{i_v, i_u\}]$
        return $\varnothing$    // Chains are consistent
    if $C_v$.term $= C_u$.term
        return $\{E_v[C_v$.term$]$.request.leader$\}$    // Case 1
    $h \leftarrow \mathrm{argmax}_{u,v}\{C_u$.term, $C_v$.term$\}$
    $\ell \leftarrow \mathrm{argmin}_{u,v}\{C_u$.term, $C_v$.term$\}$
    $\tau \leftarrow \min\{t | t \in E_h$.keys, $t > C_\ell$.term$\}$
    if $C_\ell$.term $\in E_h$.keys
        $j \leftarrow E_h[\tau]$.request.freshness.index
        if $C_\ell[j] \neq C_h[j]$
            return $\{E_\ell[C_\ell$.term$]$.request.leader$\}$    // Case 2
    return $E_h[\tau]$.voters $\cap$ $C_\ell$.voters    // Cases 3 & 4
```

Algorithm 1: The auditing algorithm for pairwise consistency checks. Time complexity $O(n + \Lambda)$.

```
Input: Log chain $L_w$, election list $E_w$ and commitment certificate
       $C_w$ for all $w \in \mathcal{V}$, node set $\mathcal{V}$
Output: Byzantine nodes $A \subset \mathcal{V}$
Function AuditAll($L, E, C, \mathcal{V}$):
    for $v \in \mathcal{V}$
        if not DataLegitimacy($L_v, E_v, C_v$)
            return $\{v\}$
    $w \leftarrow \mathrm{argmax}_{v \in V} L_v$.length
    for $v \in \mathcal{V} - \{w\}$
        $A_v \leftarrow$ PairConsistency(L, E, C, v, w)
        if $A_v \neq \varnothing$
            return $A_v$
    return $\varnothing$
```

Algorithm 2: The early-exiting auditing algorithm of Raft-Forensics. A full version can be found in Alg. 5.

the fault. For example, in a censorship attack, an adversarial leader ignores all the requests from a client. In an alternative world, an adversarial client falsely accuses an honest leader of censoring its requests. In both cases, liveness is damaged, but an auditor can never find out which node is adversarial.

Instead of providing accountability for liveness attacks, we aim to tolerate them. In other words, we aim to modify Raft-Forensics such that the liveness of the system is guaranteed even if there are Byzantine faults. In the meantime, it should still account for Byzantine faults that damage safety as Raft-Forensics does. We name the new consensus algorithm **Live-Raft-Forensics** to emphasize its difference from Raft-Forensics. We define liveness in Def. 8.1 and specify the design of Live-Raft-Forensics in §8.2. Then, we formally prove that Live-Raft-Forensics achieves the liveness property. Since Live-Raft-Forensics makes no changes to log replication compared to Raft-Forensics, its performance on average (without leader changes) is the same as in §7.

8.1 Model

We use the same assumptions as in §3, except that we assume the total number of faulty nodes does not exceed f. This is motivated by the fact that if there exists a node with Byzantine faults and f *other* honest nodes with crash faults, we can account for safety threats but we cannot tolerate liveness threats that inevitably paralyze the system without enough live honest nodes.

We define liveness from the perspective of request processing.

Definition 8.1 (Liveness). In a partially-synchronous network, a consensus algorithm is *live* if within bounded time after a client submits a request, the client will receive irrefutable confirmation that request is committed.

8.2 Design

Table 5 shows a list of additional parameters that are used in the system design.

Symbol	Description	Requirement
$\Omega_{\heartsuit}$	Timeout of leader's heartbeats or RPC responses	$\geq \Delta$
$\Omega_{\texttt{Cand}}$	Timeout of candidate during election	$\geq 2\Delta$
$\Omega_{\texttt{Voter}}$	Timeout of voters during election	$\geq \Omega_{\texttt{Cand}} + 2\Delta$
$\Omega_{\texttt{Client}}$	Timeout of client's first trial of request submission	$\geq 4\Delta$
$\Omega_{\texttt{Request}}$	Timeout of receiving request commitment certificate	$\geq 2\Delta$

Table 5: Table of Parameters of Live-Raft-Forensics. Recall that we assume a partially-synchronous network with max message delay Δ after GST.

8.2.1 Motivation. Live-Raft-Forensics can be divided into three major components – client dispute, prevote and round-robin election. Although they are highly coupled in the system to achieve liveness, we list three examples of liveness attacks that motivate each of them. In detail, Example 8.2 shows why clients must be able to issue complaints about the leader. Example 8.3 motivates a prevote procedure that prevents a single Byzantine node from using new elections to block an honest leader. Example 8.4 motivates a round-robin leader election to prevent adversarial nodes from gaining an indefinite advantage in winning elections.

Example 8.2 (Censorship). When a client sends a request to the leader, the leader ignores the request.

Example 8.3 (Anarchy). Whenever a new leader is elected, an adversarial node claims that the new leader timed out and starts a new election.

Example 8.4 (Vote Monopoly). When a leader election begins, the adversarial node increments its term after an overly-brief timeout period (instead of randomly sampling between 150-300 ms, as honest nodes do). If the adversary receives a quorum, instead of claiming it, the node remains silent, such that each honest node will timeout and revote at a new term. By protocol, an honest node will not vote for any other node than itself or the adversary. The election never ends, so the system cannot process client requests.

8.2.2 Design Details. There are two types of liveness disputes: client-leader, and follower-leader. We deal with client-leader disputes through a robust broadcast-type protocol. We next deal with follower-leader disputes through a prevote protocol.

Dispute Between a Client and a Leader. A client submits a request to the system by sending the request to arbitrary nodes. A leader processes the request directly, while a follower forwards the request to the current leader. Suppose the client resubmits a client request periodically if the request is not processed in time. This may happen when the leader is faulty, and it causes a timeout. Formally, we let the client's timeout be $\Omega_{\texttt{Client}}$(i.e., the time it waits after submitting a request before assuming a fault).

Once the client's timeout timer expires, the client executes a second try by broadcasting its request to all the nodes, where each follower node forwards the request to the leader. Upon seeing a second-trial message, the follower starts a timer that can be stopped only upon receiving both a replication of the log entry and a *commitment certificate* from the leader. If a follower does not receive the certificate before timeout, it immediately starts a prevote, announcing that it no longer regards the current leader as legitimate. When there exists no more than $f-1$ nodes that still collaborate with the leader, the leader will not be able to commit any request and fall by more complaints from other clients.

On the other hand, this try-again mechanism is safe when the client is adversarial and the leader is honest. Each honest follower will receive the commitment certificate in time from the honest leader, so the honest leadership authority will remain.

Prevote. After dealing with client-leader disputes, we resolve follower-leader disputes over leader's liveness by prevotes. In vanilla Raft and Raft-Forensics, a follower can initiate a leader election by simply claiming the leader has timed out, which can be exploited to stop an honest leader from working. We use prevotes to prevent adversarial nodes from denying an honest node leadership.

From the perspective of a follower, when the leader fails to send a heartbeat, respond to RPCs or process a client request in time, the follower determines that the leader is faulty and broadcasts a signed prevote message for the next term $t+1$. A full list of conditions for prevote is listed in §B.4.1.

After broadcasting the message, v hangs and collects prevote messages from other nodes. We say that v is *waiting for* term $t+1$. When another node u receives such a prevote message, it will store the prevote but not react if u does not meet the prevote conditions locally. On the other hand, if u meets one of the conditions, it will generate another prevote message signed by itself. Then combines v's prevote with its own and broadcasts the prevotes.

Let w be an arbitrary node (can be u or v) who obtains 2 valid prevotes. w will broadcast them and enter the election of term $t+1$. Meanwhile, w starts a timer that times out after $\Omega_{\texttt{Voter}}$. We say w is *inside* term $t+1$ before the timeout.

Round-Robin Election. Once the multi-signed prevote is acquired, v broadcasts the multi-signed prevote message to all the other nodes, including the candidate c_{t+1} of the term $t+1$. v starts a timer of duration $\Omega_{\texttt{Voter}}$ that times out after an honest c_{t+1} could sufficiently react to the election. c_{t+1} is the only available candidate defined by the protocol that can receive votes and win the election at term $t+1$. We choose $c_{t+1} = (t+1) \mod n$, such that the candidacy moves to every node round-robin as the term number increments.

When c_{t+1} enters term $t+1$, it broadcasts a vote request to all the nodes. The vote request contains a tuple of log status (T, J), which denotes the term number and height of c_{t+1}'s latest log entry, respectively. Given this tuple, an honest node v agrees to vote for c_{t+1} if and only if c_{t+1} is as fresh (Definition 4.1) as itself. Upon agreement, it signs the request and returns it to the candidate c_{t+1}. Note that, however, when c_{t+1} enters term $t+1$, it sets its timer with a different duration $\Omega_{\texttt{Cand}}$, satisfying

$$\Omega_{\texttt{Cand}} \geq 2\Delta, \quad \Omega_{\texttt{Voter}} \geq \Omega_{\texttt{Cand}} + 2\Delta.$$

Once the candidate c_{t+1} receives $f+1$ votes, it generates a *leader certificate* which is a multi-signature of these voters, and establishes authority by broadcasting the certificate. If the candidate fails to do so, nodes time out at term $t+1$, and broadcast prevote messages to start the election of term $t+2$.

The procedure of the leader election is specified by Algorithm 6.

8.3 Theoretical Liveness Guarantee

The following theorem shows that Live-Raft-Forensics achieves liveness as defined in Def. 8.1. The proof is in Appendix A.3.

Theorem 8.5. *We assume all clients will resubmit a request to the system within Δ time after one node admits a leader with a valid leader certificate. After a client submits a request to Raft-Forensics at time $t=0$, one of the two events will occur.*

i. The request will be committed at time $t=t'$ for some $t' < \infty$.
ii. The adversarial node can be accused by an auditor with irrefutable proof.

Whether to implement Live-Raft-Forensics depends on the internal threat model. First of all, in client-leader dispute, an adversarial client may launch distributed denial-of-service (DDoS) attacks to spam follower nodes with dispute messages. However, DDoS threats are not specific to Live-Raft-Forensics; clients can already spam requests in vanilla Raft. The prevote design also increases the communication complexity and affects performance during leader elections. However, leader elections do not occur often in practice and the size of broadcast prevote messages is constant. Finally, for round-robin election, the expected time complexity of an election is a constant (§A.5) in n, which is asymptotically no worse than than in vanilla Raft. On the other hand, a Byzantine node can increase the time complexity of leader election up to linear in n, but vanilla Raft and Raft-Forensics are much worse – their elections cannot even proceed under the attack in Example 8.4. The major weakness of Live-Raft-Forensics is its inability to generalize to multiple Byzantine nodes. Understanding how to tolerate such faults is an interesting direction for future work.

9 CONCLUSION

We present Raft-Forensics, a CFT consensus algorithm that provides accountability for Byzantine faults that damage safety. In the presence of a mismatch of ledgers between two honest nodes, an auditor is guaranteed to find at least one Byzantine node with irrefutable proof generated by a polynomial-time algorithm. We implement Raft-Forensics and evaluate its performance. While Raft-Forensics maintains similar performance in both throughput and latency to Raft, it proves a significant overall advantage compared to Hotstuff and Dumbo-NG, two high-performance BFT algorithms. In addition, the auditing procedure of Raft-Forensics has computational complexity linear in the total data size, which is optimal. Since it is impossible to account for some Byzantine faults that only damage liveness, we present Live-Raft-Forensics alternatively to not only preserve the features of Raft-Forensics, but tolerate faults that only threaten liveness as well. We show that Live-Raft-Forensics guarantees the execution of transactions in bounded time under a partially-synchronous network.

ACKNOWLEDGEMENTS

We wish to thank Chris Meiklejohn and Heather Miller for their valuable insights and advice on this project. This work was supported in part by the Air Force Office of Scientific Research under award number FA9550-21-1-0090, as well as the generous support of Chainlink Labs, Ripple Labs, and IC3 industry partners.

REFERENCES

[1] Kyle Banker, Douglas Garrett, Peter Bakkum, and Shaun Verch. 2016. *MongoDB in action: covers MongoDB version 3.0.* Simon and Schuster.

[2] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2019. SoK: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies.* 183–198.

[3] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation.* 335–350.

[4] Christian Cachin and Marko Vukolić. 2017. Blockchain consensus protocols in the wild. *arXiv preprint arXiv:1707.01873* (2017).

[5] Vittorio Capocasale, Danilo Gotta, and Guido Perboli. 2023. Comparative analysis of permissioned blockchain frameworks for industrial applications. *Blockchain: Research and Applications* 4, 1 (2023), 100113.

[6] Apache Cassandra. 2014. Apache cassandra. *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra* 13 (2014).

[7] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 398–461.

[8] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OsDI*, Vol. 99. 173–186.

[9] Ben Christensen. 2012. Fault Tolerance in a High Volume, Distributed System. Netflix Blog. (2012). https://netflixtechblog.com/fault-tolerance-in-a-high-volume-distributed-system-91ab4faae74a.

[10] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 189–204.

[11] Team Cloudify. 2021. Geo Redundancy Explained, Cloudify. Cloudify Blog. (2021). https://cloudify.co/blog/geo-redundancy-explained/.

[12] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[13] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[14] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems.* 34–50.

[15] Sisi Duan, Michael K Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 2028–2041.

[16] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[17] Scott Dynes, Eric Goetz, and Michael Freeman. 2008. Cyber security: Are economic incentives adequate?. In *Critical Infrastructure Protection 1.* Springer, 15–27.

[18] eBay. 2017. NuRaft. https://github.com/eBay/NuRaft/tree/v1.3. (2017). Accessed on April 19, 2023.

[19] etcd. 2023. etcd. https://etcd.io/. (2023). Accessed on April 19, 2023.

[20] fascy. 2022. Dumbo-NG. https://github.com/fascy/Dumbo_NG.git. (2022). Accessed on April 19, 2023.

[21] Brian Fung. 2021. Cyberattacks are the number-one threat to the global financial system, Fed chair says. *CNN Business* (2021).

[22] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.* 1187–1201.

[23] Nishant Garg. 2013. *Apache kafka.* Packt Publishing Birmingham, UK.

[24] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers.* Springer, 296–315.

[25] Mike Graf, Ralf Küsters, and Daniel Rausch. 2020. Accountability in a permissioned blockchain: Formal analysis of hyperledger fabric. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE, 236–255.

[26] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Speeding dumbo: Pushing asynchronous bft closer to practice. *Cryptology ePrint Archive* (2022).

[27] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security.* 803–818.

[28] Suyash Gupta, Sajjad Rahnama, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. 2022. Dissecting BFT Consensus: In Trusted Components we Trust! *arXiv preprint arXiv:2202.01354* (2022).

[29] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review* 41, 6 (2007), 175–188.

[30] Moin Hasan and Major Singh Goraya. 2018. Fault tolerance in cloud computing environment: A systematic survey. *Computers in Industry* 99 (2018), 156–172.

[31] HashiCorp. 2023. Consul. https://www.consul.io/. (2023). Accessed on April 19, 2023.

[32] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems.. In *USENIX annual technical conference*, Vol. 8.

[33] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. 2020. Fast-hotstuff: A fast and resilient hotstuff protocol. *arXiv preprint arXiv:2010.11454* (2020).

[34] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems.* 295–308.

[35] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. 2010. Accountability: definition and relationship to verifiability. In *Proceedings of the 17th ACM conference on Computer and communications security.* 526–535.

[36] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport.* 277–317.

[37] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. 2016. XFT: Practical Fault Tolerance beyond Crashes.. In *OSDI.* 485–500.

[38] James Lovejoy, Madars Virza, Cory Fields, Kevin Karwaski, Anders Brownworth, and Neha Narula. 2023. Hamilton: A {High-Performance} Transaction Processor for Central Bank Digital Currencies. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23).* 901–915.

[39] Nancy A Lynch. 1996. *Distributed algorithms.* Elsevier.

[40] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. 2018. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018.* 393–407.

[41] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.* 31–42.

[42] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14).* 305–319.

[43] relab. 2020. Hotstuff. https://github.com/relab/hotstuff. (2020). Accessed on April 19, 2023.

[44] Mohammad Roohitavaf, Jung-Sang Ahn, Woon-Hak Kang, Kun Ren, Gene Zhang, Sami Ben-Romdhane, and Sandeep S Kulkarni. 2019. Session guarantees with raft and hybrid logical clocks. In *Proceedings of the 20th International Conference on Distributed Computing and Networking.* 100–109.

[45] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/Ispa*, Vol. 1. IEEE, 57–64.

[46] Ermin Sakic and Wolfgang Kellerer. 2017. Response time and availability study of RAFT consensus in distributed SDN control plane. *IEEE Transactions on Network and Service Management* 15, 1 (2017), 304–318.

[47] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. 2021. BFT protocol forensics. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*. 1722–1743.

[48] simplespy. 2020. DiemForensics. https://github.com/simplespy/DiemForensics. (2020). Accessed on April 19, 2023.

[49] Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 729–730.

[50] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[51] The Diem Team. 2021. DiemBFT v4: State Machine Replication in the Diem Blockchain. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf. (2021). Accessed on April 19, 2023.

[52] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 1–36.

[53] Weili Wang, Sen Deng, Jianyu Niu, Michael K Reiter, and Yinqian Zhang. 2022. ENGRAFT: Enclave-guarded Raft on Byzantine Faulty Nodes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2841–2855.

[54] Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael K Reiter. 2023. Communication-Efficient BFT Using Small Trusted Hardware to Tolerate Minority Corruption. In *26th International Conference on Principles of Distributed Systems (OPODIS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[55] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.

A PROOF

A.1 Proof of Thm. 6.1

When state machine safety is breached, there must exist two honest nodes u and v with conflicting *committed* log lists, where neither chain is a prefix of another. Namely, u's last committed entry has term and index (t_u, j_u) and v's entry has term and index (t_v, j_v). u must have the CC of entry (t_u, j_u) and v must have the CC of (t_v, j_v). We note that by the protocol, if an honest node owns a raw or committed entry (t, j), it must own the LC of the leader at term t.

We do not assume, however, that when an auditor finds the conflict between u and v, it ensures or presumes that u and v are honest. Honesty of u or v is unnecessary in a sense that even if one of them is Byzantine, it cannot falsify or tamper with the log list, the leader signatures, the CC's and the LC's it provides for the auditor. The only action that is adversarial while undetectable is to submit an incomplete log list, which must be a prefix of the actual list it maintains. In addition, we may assert that a Byzantine node will honestly provide the CC associated with the log list, as a proof that the list is committed. This guarantees that an auditor is able to get two CC's from u and v.

To analyze a conflict, we should first check whether each term is used by only one leader. Case 0 applies when a term is used by two different leaders.

Case 0. u and v conflict on leaders at a same term. We can track the voter who voted for both leaders by inspecting the voter intersection between the leader certificates maintained by u and v, respectively. This voter can be accused because it is forbidden to vote twice in a same term.

If the election lists of u and v do not conflict, we proceed to cross-checking the log lists. In this step, our tactics rely on whether the logs have conflicting entries at the same term. For the rest of the cases, we assume $t_u \geq t_v$ without loss of generality.

Case 1. There is a conflict within term t_v. In detail, u and v own two entries ε_u and ε_v, respectively. Both ε_u and ε_v are at term t_v, but neither is an ancestor or descendant of another. An auditor is able to accuse the leader of term t_v, where the evidences are the leader's signatures on term t_v's last entries, which are different for nodes u and v.

Case 2. There is no conflict within term t_v. We can assert that $t_u > t_v$, because otherwise we must have $t_u = t_v$ and all entries of u and v at term t_v are in the same branch. This denies our very first assumption – u and v have entries on different branches.

With $t_u > t_v$ established, we define $T_u^+ \triangleq \{t | t \in T_u,\ t > t_v\}$. We pick $\tau \triangleq \min T_u^+$, which exists because T_u^+ has at least one element t_u. We consider term τ's leader ℓ_τ and its freshness (t_ℓ, j_ℓ) when it requested for vote. By the minimality of τ, $t_\ell \leq t_v$. Because there exists a conflict between u's chain and v's chain, (t_ℓ, j_ℓ) must be staler than (t_v, j_v). Otherwise, we must have $t_\ell = t_v$ and $j_\ell \geq j_v$. In conjunction with the case assumption "no conflict within term t_v", we deduce that v's chain is a prefix of u's chain.

Hence, every honest voter of leader ℓ_τ must not have voted after signing the fresher log entry (t_v, j_v).

On the other hand, every honest voter of ℓ_τ must not have voted **before** signing the log entry at (t_v, j_v), either. This is because $t_v < \tau$, which implies the entry is from a previous term for the CC signers. By the protocol, the only circumstance where an honest node may sign such a stale entry is during a log synchronization. However, an honest node can only accept synchronization up to entry (t_ℓ, j_ℓ), the freshest entry before term τ's first entry. Signing any entry with freshness in-between is illegal.

Therefore, it is impossible for an honest node to sign both the CC of log entry at (t_v, j_v) and the LC of term τ, regardless of the temporal order. An auditor is able to acquire the CC from u and v in order to find the conflict. Since ℓ_τ has nodeted an entry to u, either ℓ_τ or u is honest and will provide the LC to the auditor. Then, by the pigeonhole principle, there exists at least one node who signed both the CC and the LC. This node must be Byzantine and can be accused after examining the log list.

In the above discussion, we have proven that if a chain fork exists on two different nodes, at least one adversarial node can be accused with irrefutable proof by an auditor with access to their log lists. ■

A.2 Proof of Thm. 6.3

Suppose the client's request is reported with index i and a descendent CC of block B_c at index $j \geq i$ and term t_c. When client safety is violated, there exists an honest node with a different committed log entry at index i and term t_n. Suppose the node maintains a CC' of block B_n at index $k \geq i$. Since entries of CC and CC' have a different ancestor at the same index i, they must be on conflicting branches. We discuss two cases.

Case 1. $t_c = t_n$. The leader of this term can be accused for violating the append-only rule. The corresponding leader signatures serve as the evidence.

Case 2. $t_c > t_n$. Any node who received B_n must not accept the leader who proposes B_c with a conflicting ancestor at index i. Any node who accepted B_c's proposer must not agree to replicate B_n, an entry of a lower term. Hence, we can find at least one adversarial node in the intersection of voters of CC' and the LC of B_c's leader.

Case 3. $t_c < t_n$. On the node's blockchain, we find the first term τ that is higher than t_c. Similarly to Case 4 in §A.1, we can find at least one adversarial node in the intersection of voters of CC and the LC of τ.

In all cases, the exposure of a Byzantine-faulty node is guaranteed. ■

A.3 Proof of Thm. 8.5

First of all, we present the following lemma.

Lemma A.1 (Consistency During Election). *Assume that the system is stabilized (post-GST). Let the timeout of voters and the candidate be Ω_{Voter} and Ω_{Cand}, satisfying $\Omega_{\mathsf{Cand}} \geq 2\Delta$ and $\Omega_{\mathsf{Voter}} \geq 2\Delta + \Omega_{\mathsf{Cand}}$. After an honest node first collects 2 prevotes of term τ that denies the leader or candidate of term $\tau - 1$, one of the two events will happen within time $(\Omega_{\mathsf{Voter}} + \Delta)$ if c_τ is non-faulty:*

(i) c_τ is admitted as a leader for term τ by every non-faulty node. This always happens when c_τ has log chain as fresh as at least f other non-faulty nodes.

(ii) Every non-faulty node initiates prevote and collects 2 prevotes of term $\tau + 1$.

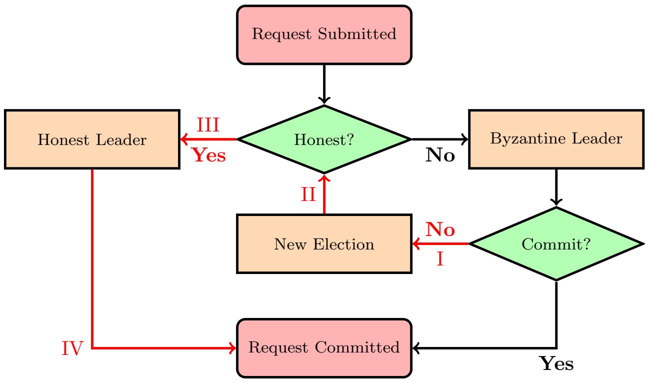


Figure 6: Flow chart of request processing in Raft-Forensics.

The proof of this lemma is given in Appendix A.4.

We consider the following four liveness properties.

(I) **Punishment of Failure:** When a leader fails to commit a client request in time, every honest node will end its leadership and start a leader election.

(II) **Election Liveness:** During a leader election, a fresh candidate will be elected within finite time.

(III) **Honest Opportunity:** When a leader election starts, there does not exist an adversarial strategy, such that the Byzantine node avoids committing at least one request in time, while no honest node can be elected in the future.

(IV) **Honest Durability:** When an honest node is the leader, the adversary cannot force at least f honest nodes to deny the leadership.

First of all, we show that if these properties are all satisfied, a client request will eventually be committed. In Figure 6, we assume the request is submitted when a leader is available, i.e., one node is admitted as a leader by at least $f+1$ nodes. This is always feasible in finite time because the system cannot remain leaderless indefinitely by Property (II).

Property (I) prevents a Byzantine leader from occupying leading power to censor requests. Property (III) prevents the adversary from keeping the system inside the adversarial loop. Property (IV) ensures that an honest leader is able to commit the request.

We now argue that in Raft-Forensics, all the 4 liveness properties are achieved.

Punishment of Failure: This is guaranteed by the dispute between a client and a leader in §5.1.2. Suppose a Byzantine leader keeps a client from receiving the commitment certificate of a particular request. After detecting a timeout of the first trial, the client informs every node of its second trial. This procedure takes $\Omega_{\texttt{Client}}$ time.

Subsequently, the nodes forward the request to the leader, and wait for the leader to commit the request and send the commitment certificate. Under the existence of $f+1$ non-faulty followers, if the Byzantine leader commits the request honestly and sends the certificate to one of the follower in time, the follower will relay the certificate to the client and there is no problem with liveness. Otherwise, all $f+1$ non-faulty followers will initiate a prevote to overthrow the Byzantine leader as a punishment of failure in committing the request before timeout. This judgement takes $\Delta + \Omega_{\texttt{Request}}$ time.

Therefore, it takes at most $\Omega_{\texttt{Client}} + 2\Delta + \Omega_{\texttt{Request}}$ time for a client to overthrow a Byzantine leader which refuses to commit its request.

Election Liveness & Honest Opportunity:

Case 1: If the previous leader is Byzantine, the election will traverse through every non-faulty node before the Byzantine node. By assumption, at least $f+1$ nodes are non-faulty during the election. Hence, there exists one non-faulty node $\widetilde{v}$ as fresh as f other non-faulty nodes. By Lemma A.1, for each round with a non-faulty candidate, all non-faulty nodes will either admit the candidate together, or proceed to the next candidate together. Additionally, either $\widetilde{v}$ or one of $\widetilde{v}$'s predecessor will be elected with admission by all non-faulty nodes. We discuss two sub-cases.

Case 1.1 The leader is not fresh enough, i.e., there exists one log entry with CC received by a non-faulty client or node (and hence the auditors) which is not present in the new leader's log list. As a consequence, the Byzantine node must appear on both this CC and the LC of the new leader, and an auditor will be able to detect it by the conflict between these certificates. This corresponds to Event ii. of the theorem.

Case 1.2 The leader is fresh enough. Then, the leader is able to commit the request within timeout $\Omega_{\texttt{Client}}$. In the worst case, it takes $f+1$ rounds for the leader to be finally elected. By Lemma A.1, each round takes at most $\Delta + \Omega_{\texttt{Voter}}$ time. Hence, the leader is able to commit the request at most $\Omega_{\texttt{Client}} + (f+1)(\Delta + \Omega_{\texttt{Voter}}) + \Delta$ time after one honest node collects enough prevotes against the Byzantine leader. This corresponds to Event i. of the theorem.

Case 2: If the previous leader is honest, we consider the case where every candidate between the previous leader and the Byzantine node is unable to obtain $f+1$ votes, since the other case can be discussed in a same way as Case 1. The election will take at most $(f+1)(\Delta + \Omega_{\texttt{Voter}})$ time. For the same reason, we also assume that the Byzantine node is fresh enough to obtain f votes from the non-faulty nodes. The Byzantine node (denoted by a) may cause a division between the non-faulty nodes, such that all non-faulty nodes in set P admit a's authority, while all nodes in Q broadcast prevotes for the next term and wait for enough prevotes to go forward.

If $|P| < f$, a cannot commit any client request because it is unable to obtain $f+1$ signatures. Then, a will lose leadership due to Property (I) property and we return to Case 1. As the client resubmits the request at most Δ time after the Byzantine leader is admitted, each non-faulty node sends prevote against the leader at most $(2\Delta + \Omega_{\texttt{Client}} + \Omega_{\texttt{Request}})$ time after the leader is first admitted.

If $|P| \geq f$ and a fails to respond a client request with commitment proofs, a will still lose leadership for the same reason and we return to Case 1. This also takes at most $(2\Delta + \Omega_{\texttt{Client}} + \Omega_{\texttt{Request}})$ time after the leader is first admitted. Otherwise, a commits every client request and leads the entire system to work as an honest node.

In both cases, Properties (II) and (III) are satisfied.

Honest Durability: In order to disable the leadership, the adversary must collect 2 prevotes. By assumption, the adversary is unable to affect the communication between any pair of honest nodes. The only way to attack the leadership is through a colluding client, who attempts to overthrow the leader by testimonies. Suppose a Byzantine client attempts to falsely accuse an honest leader under assistance of a Byzantine follower. The only approach is to broadcast a request (of second trial) to a subset of the honest followers. Because the leader is honest, it will commit each request and send the corresponding certificate to the followers before timeout $\Omega_{\texttt{Client}}$, which makes it impossible for an honest follower to initiate a prevote. Therefore, there can only exist one prevote from its accomplice, which does not suffice to disable the leadership.

In total, the time t' it takes a client to get its request committed is upper bounded by

$$\begin{aligned}\bar{t} \triangleq&(f+1)(\Delta+\Omega_{\texttt{Voter}})+\Delta+\Omega_{\texttt{Client}}+\Omega_{\texttt{Request}}+\Omega_{\texttt{Client}}+\\ &(f+1)(\Delta+\Omega_{\texttt{Voter}})+\Delta\\ =&2(f+1)(\Delta+\Omega_{\texttt{Voter}})+2\Delta+2\Omega_{\texttt{Client}}+\Omega_{\texttt{Request}}. \qquad \blacksquare\end{aligned}$$

A.4 Proof of Lemma A.1

Without loss of generality, we let $t = 0$ be the time when an honest node first collects $f + 1$ prevotes of term τ. We point out that every honest node v will receive $f + 1$ prevotes at $t_v^0 \in [0, \Delta]$ and start the election for term τ. We aim to prove that each event happens before $t = \Omega_{\texttt{Voter}} + \Delta$.

We spilt the proof into two parts. In Part 1, we show that when c_τ has log list as fresh as at least f other non-faulty nodes, (i) happens. In Part 2, we prove that in other cases, either one of (i) and (ii) happens.

Part 1. If the candidate $c = c_\tau$ is non-faulty, it broadcasts request for vote to every node at t_c^0. It takes the request t_v^1 time to reach a non-faulty node v, and t_v^2 time for v's vote agreement or rejection to reach c. Let ζ_v denote the time when c receives response from v. It satisfies that

$$\zeta_v = \max\left\{t_c^0 + t_v^1,\ t_v^0\right\}+t_v^2 \leq \max\{t_c^0+\Delta,\ \Delta\}+\Delta = t_c^0+2\Delta \leq t_c^0+\Omega_{\texttt{Cand}}.$$

Therefore, by the time $t = t_c^0 + \Omega_{\texttt{Cand}}$, the candidate $c = c_\tau$ will be able to collect responses from *all* non-faulty nodes. If c_τ is as fresh as at least f non-faulty nodes, each of these nodes will vote for c_τ. Then, c_τ can establish its authority by broadcasting the multi-signed vote as the leader certificate. Suppose the certificate travels to an arbitrary non-faulty node v for time t_v^3. Let ξ_v denote the time it arrives at v (with respect to $t = 0$ when the prevotes are broadcast). It satisfies that

$$\xi_v \leq t_c^0 + \Omega_{\texttt{Cand}} + t_v^3 \leq \Omega_{\texttt{Voter}} - 2\Delta + 2\Delta \leq t_v^0 + \Omega_{\texttt{Voter}}.$$

In other words, v will be able to acknowledge the leadership of c_τ before it could timeout at term τ. Possibly, v receives the certificate even during the previous term $\tau - 1$. However, it can process it immediately after it receives the multi-signed prevote of term τ.

Therefore, all the non-faulty nodes will admit c_τ, which signals the ending of a leader election. Note that for all v, the admission of leader happens at

$$t = \xi_v \leq t_v^0 + \Omega_{\texttt{Voter}} \leq \Omega_{\texttt{Voter}} + \Delta.$$

Part 2. We consider circumstances where c_τ is not as fresh as at least f other non-faulty nodes, where there exists at least one node with a more fresh log among f other non-faulty nodes. As an honest node, c_τ timeouts at $t = t_c^0 + \Omega_{\texttt{Cand}}$. This guarantees that by time $t = t_c^0 + \Omega_{\texttt{Cand}} + t_v^3$, an honest node v must have received from the candidate c_τ either a prevote for the next term, or a leader certificate (which may happen in the existence of the Byzantine node) . Because

$$t_c^0 + \Omega_{\texttt{Cand}} + t_v^3 \leq \Delta + \Omega_{\texttt{Voter}} - 2\Delta + \Delta \leq t_v^0 + \Omega_{\texttt{Voter}},$$

v must not have timed out. Therefore, all the non-faulty nodes will act consistently towards the result broadcast by c_τ. Hence, c_τ is either elected as the next leader, or starts a prevote for the next term $\tau + 1$. In the latter case, each non-faulty node will time out and eventually collect $f + 1$ prevotes for term. $\tau + 1$ before 5Δ time since it entered term τ Again, we point out that each honest node makes a decision at time

$$t = t_c^0 + \Omega_{\texttt{Cand}} + t_v^3 \leq \Omega_{\texttt{Voter}} + t_v^0 \leq \Omega_{\texttt{Voter}} + \Delta. \qquad \blacksquare$$

A.5 Performance of Round-Robin Election

Proposition A.2. *Assume that in Live-Raft-Forensics, there is no Byzantine faults and at the start of an election, the freshness order of the followers is uniformly distributed. Then, the expected time complexity of a round-robin election is a constant that is irrelevant to the number of nodes n.*

Proof. In §A.3, we have shown that a round of election is guaranteed to terminate within bounded time, where the bound is only relevant to the timeout parameters in Table 5. To prove the statement, we only need to show the expected number of rounds is a constant to the number of nodes n. By the protocol, an election ends if the candidate is at least as fresh as f other nodes. This effectively requires its freshness rank (from freshest to stalest) to be no greater than $f + 1$. Subsequently, a round election iterates over randomly permuted ranks and ends at the first rank that is no greater than $f + 1$. Without loss of generality, we assume the freshnesses of the nodes are distinct, so that each rank from 1 to n is taken by exactly one node.

Now we calculate the expectation $\mathbb{E}[R]$ of the position R of the first rank that is no greater than $k \in [1, n]$, which we denote by $\pi_{n,k}$. Clearly,

$$\pi_{n,1} = \sum_{r=1}^{n} \frac{1}{n} \cdot r = \frac{n+1}{2}, \quad \pi_{n,n} = 1.$$

For $k \geq 2$, we discuss the rank of the first position. With probability k/n, it is less than or equal to k, and $R = 1$. With probability $(n-k)/n$, it is greater than k, so the problem reduces to finding the first rank among the remaining $n - 1$ positions that is no greater than k. We can describe this nature with

$$\pi_{n,k} = \frac{k}{n} + \frac{n-k}{n}(1 + \pi_{n-1,k}), \quad \forall n \in \mathbb{N},\ k \in [1, n-1].$$

Given the first term $\pi_{k,k} = 1$, we can derive by deduction that $\pi_{n,k} = \frac{n+1}{k+1}$. Recall that we aim to find $\pi_{n,f+1}$, which can be computed by plugging $n = 2f + 1$ in:

$$\pi_{n,f+1} = \frac{n+1}{f+2} = 2 - \frac{2}{f+2} < 2.$$

Therefore, a round-robin election is expected to end within 2 rounds, regardless of the number of nodes n. This concludes the proof. ■

B PSEUDO-CODE FOR ALGORITHMS

B.1 Raft-Forensics

We formalize Raft-Forensics consensus algorithm in pseudo-code in Algorithms 3a, 3b and 3c. They compose a simple example of Raft-Forensics, where each log entry contains one single transaction, and each log replication starts with replicating one single entry. The algorithm consists of a main thread and several message receiver threads. The main thread controls the role transitions of the node, and the receiver threads define how a node handles each type of message. We highlight with red text the parts that are specific to Raft-Forensics.

For simplicity, we used black-box functions including **`VerifyCert`** and **`VerifySig`**. In both functions, we assume the algorithm can derive the public key of any node given its identity. **`VerifyCert`** takes a CC or a LC as input and outputs the verification result. Specifically, both certificates have a list of voters, a list of signatures and a message to sign, which corresponds to the `pointer` field of a CC and the hash of the `request` field of a LC. **`VerifyCert`** checks if the number of voters reaches the quorum ($f+1$), and if all the signatures are valid. On the other hand, **`VerifySig`** takes a signature, a hashed message and an LC as input. The LC contains the signer's identity, from which the algorithm can find its public key and verify the signature. The output is a predicate of whether the signature is valid.

Although the framework of the algorithm is made clear, we do not specify some implementation details at low levels. For example, in **`HandleCommit`** and **`HandleAppendEntries`**, h_i is read as the hash pointer of a local entry at a given index i. This indexing is not trivial because if no other state is maintained, h_i needs to be derived all the way from h_0 which is expensive. Thus, a system engineer needs to consider trading space for time. Similarly, the maintenance of the log list `logs`should also be tactical as its size is prohibitive for RAM but acceptable for permanent storage in practice. In addition, a system engineer should carefully treat the conditions of rejecting or accepting a request, which may not be exhaustive in the pseudo-code. They also need to handle the rejection messages properly to optimize the performance.

```
Initialize secret_key, logs, pointer, myCC, leader_sig,
  election_list, timer_leader, timer_follower
term ← 0, leader ←⊥, role ← FOLLOWER
Function MainThread():
    while true
        As Follower:
            role ← Follower
            timer_follower.tick(UniformSample($\Omega_{Lo}$, $\Omega_{Hi}$))
            timer_follower.wait()     // timer can be restarted
        As Candidate:
            role ← Candidate, leader ←⊥
            term ← term + 1
            broadcast VoteRequest(self, term,
                logs.tail.freshness, pointer)
            wait for $f$ verified Vote's on Hash(VoteRequest)
        As Leader:
            role ← Leader, leader ← self
            broadcast LC
            while true
                timer_leader.tick($\Omega_\heartsuit$)
                timer_leader.wait()    // timer can be restarted
                broadcast Heartbeat
Thread HandleVoteRequest(req):
    if req.term ≤ term
        return                        // discard low-term requests
    (In MainThread) goto As Follower
    term ← req.term
    if req.freshness is staler than logs.freshness
        reply "reject" and return
    reply Vote($\sigma_{secret\_key}$(req)) and return
Thread HandleLC(LC):
    if LC.request.term < term or not VerifyCert(LC)
        return   // discard low-term/illegal leadership claims
    (In MainThread) goto As Follower
    term ← LC.request.term
    leader ← LC.request.leader
    election_list[term] ← LC
```

Algorithm 3a: Raft-Forensics (Part 1), including the main loop, and the receiver threads handling vote requests and LC's. Ω_{Lo}, Ω_{Hi} and $\Omega_\heartsuit$ are system parameters already introduced by vanilla Raft.

B.2 Chain Integrity Algorithm

For each node v, the audited data consists of 4 components: the log list L_v, the commitment certificate C_v, the election list E_v and leader signatures S_v. We list the requirements for the data of a node v to be *legitimate*.

1. L_v starts with a dummy entry `InitLog` and lasts with the corresponding entry of the commitment certificate C_v.
2. For each entry in L_v,
   i) The `index` field equals the number of its ancestors.
   ii) The `term` field is no less than that of its predecessor.
   iii) There exists a leader certificate in E_v keyed by `term`.
   iv) If L_v is the last entry of its term, the corresponding signature in S_v can be verified with its hash pointer and the leader's public key.
3. The signatures in commitment certificate C_v are correctly signed by the voters in C_v. Numbers of voters and signatures should equal and be at least $f+1$.
4. For each leader certificate with term t in E_v,
   i) The signatures are correctly signed by the voters. Numbers of voters and signatures should equal and be at least $f+1$.
   ii) If there exists at least one entry at term t in L_v, the predecessor of first entry at term t must have the same freshness as the certificate.
   iii) (Ensured by Alg. 4) A log entry with term t exists in L_v.

```
Thread HandleClientTransaction(tx):
    if role ≠ Leader
        return
    entry ← new entry(term, logs.tail.index + 1, tx)
    logs.append(entry)
    pointer ← temp_ptr ← hash(pointer||entry)
    sig ← $\sigma_{\text{secret\_key}}$(pointer)
    leader_sig[term] ← sig
    broadcast AppendEntries([entry], sig,
     election_list[term])
    timer_leader.restart()
    wait for $f$ verified Acknowledgement's on temp_ptr
    myCC ← assembled CC from signatures and temp_ptr
    broadcast Commit(myCC)
    timer_leader.restart()
Thread HandleLogSync($t$, $i$):
    if role ≠ Leader
        return
    if $t$ > term or $i$ > logs.tail.index
        reply Reject and return
    if logs[$i$].term ≠ $t$
        reply Reject and return
    entries ← logs[$i + 1$ : end]
    sigs ← Dict({}), LCs ← Dict({})
    for $\varepsilon$ : entries
        if $\varepsilon$.term ∉ sigs.keys
            sigs[$\varepsilon$.term] ← leader_sig[$\varepsilon$.term]
            LCs[$\varepsilon$.term] ← election_list[$\varepsilon$.term]
    if $t$ < entries.head.term
        sigs[$t$] ← leader_sig[$t$]
    reply AppendEntries(entries, sigs, LCs) and return
Thread HandleCommit(CC):
    if role ≠ Follower or sender ≠ leader
        return
    $i$ ← CC.index, $t$ ← CC.term
    if $i$ ≤ myCC.index or $t$ < myCC.term
        reply Reject and return
    if $i$ > logs.index or $t$ > logs[CC.index].term
        reply LogSync(myCC.term, myCC.index) and return
    if $t$ < logs[CC.index].term
        reply Reject and return
    if CC.pointer ≠ $h_i$ or not VerifyCert(CC)
        reply Reject and return
    myCC ← CC
    reply Committed and return
Thread HandleHeartbeat():
    if role = Follower And leader = sender
        timer_follower.restart()
```

Algorithm 3b: Raft-Forensics (Part 2), including the receiver threads handling client transactions, log synchronizations, commits and heartbeats.

```
Thread HandleAppendEntries(entries, sigs, LCs):
    if role ≠ Follower or sender ≠ leader
        return
    timer_follower.restart()
    /* Freshness checks of entries                        */
    if entries.tail is not fresher than logs.tail
        reply Reject and return
    if entries.head.index ≤ myCC.index or
     entries.head.term < myCC.term
        reply Reject and return
    LogSyncMessage ← LogSync(myCC.term, myCC.index)
    if entries.head.index > logs.tail.index + 1
        reply LogSyncMessage and return
    /* Check if head entry fits when it initiates new term   */
    $i$ ← entries.head.index − 1, $t$ = logs[$i$].term, $h \leftarrow h_i$
    fit ← $t$ ≤ entries.head.term, verified ← false
    if $t$ < entries.head.term
        if $t$ ∈ sigs.keys And
         VerifySig(sigs[$t$], $h$, election_list[$t$])
            verified ← true
        fit ← verified
    if not fit
        if $i$ = myCC.index
            reply Reject and return
        reply LogSyncMessage and return
    /* Check if the entries and signatures are legitimate   */
    $h_{\text{prev\_term}} \leftarrow h$
    for $\varepsilon$ : entries
        if $i + 1 \neq \varepsilon$.index or $t > \varepsilon$.term
            reply Reject and return
        $i \leftarrow i + 1$, $t \leftarrow \varepsilon$.term, $h$ ← Hash($h||\varepsilon$)
        if $\varepsilon$ = entries.tail or $t < \varepsilon$.successor.term
            if $t$ ∉ sigs.keys or $t$ ∉ LCs.keys
                reply Reject and return
            if VerifySig(sigs[$t$], $h$, LCs[$t$])
                verified ← true
            else if verified
                reply Reject and return
            else
                reply LogSyncMessage and return
            if not VerifyCert(LCs[$t$])
                reply Reject and return
            if $t$ ∈ election_list
                if election_list[$t$] ≠ LCs[$t$]
                    reply Reject and return
            else if LCs[$t$].request.pointer ≠ $h_{\text{prev\_term}}$
                reply Reject and return
            $h_{\text{prev\_term}} \leftarrow h$
    pointer ← $h$
    for $t$ : sigs.keys ∩ LCs.keys
        leader_sig[$t$] ← sigs[$t$]
        election_list[$t$] ← LCs[$t$]
    logs[$i$ : $i$ + entries.length] ← entries
    reply Acknowledgement($\sigma_{\text{secret\_key}}(h)$) and return
```

Algorithm 3c: Raft-Forensics (Part 3), including the receiver thread handling log replication (AppendEntries).

Alg. 4 describes the verification procedure of a node's data integrity. If the data is not legitimate, the node is considered Byzantine-faulty and will not participate in the pairwise consistency checks in Alg. 1. In other words, all nodes in Alg. 1 are guaranteed to have legitimate data. Note that it is possible that an honest node accepts a leader, but does not nodete any entry from it for various reasons. For simplicity in Alg. 1, we remove all the leader certificates of terms of which no entries exist in the log list.

```
Input: log list $L_v$, election list $E_v$, commitment certificate $C_v$ and
       leader signatures $S_v$ for $v \in \mathcal{V}$
Output: A predicate indicating whether the data of $v$ is valid, an
        updated election list $E_v$, and an auxiliary election list $E'_v$
Function DataLegitimacy($L_v, E_v, C_v, S_v$):
    fail ⟵ (false, ⊥, ⊥)
    /* Check log list */
    if $L_v[0] \neq$ InitLog
        return fail
    AllTerms ⟵ ∅
    $h$ ⟵ zero
    for $i = 1, 2, \cdots,$ len($L_v$) $- 1$
        if $L_v[i]$.index $\neq i$
            return fail                    /* Ensure index correctness */
        $t$ ⟵ $L_v[i]$.term
        AllTerms ⟵ AllTerms ∪ {$t$}
        if $t < L_v[i-1]$.term
            return fail                    /* Ensure non-decreasing indexs */
        if $t \notin E_v$.keys or $t \notin S_v$.keys
            return fail                    /* Ensure log creator is traceable */
        $h$ ⟵ hash($h \| t \| L_v[i]$.index$\| L_v[i]$.payload)
        if $i =$ len($L_v$) $- 1$ or $t < L_v[i+1]$.term
            if not VerifySig($S_v[t], h, E_v[t]$)
                return fail                /* Ensure authenticity */
            if $i <$ len($L_v$) $- 1$
                $t'$ ⟵ $L_v[i+1]$.term
                if $t' \notin E_v$.keys or $h \neq E_v[t']$.request.pointer
                    return fail            /* Ensure LC correctness */
    /* Check Commitment Certificate */
    if $C_v$.pointer $\neq h$ or
      not VerifyCert($C_v$)
        return fail
    /* Check election list */
    $E'_v$ ⟵ Dict({})
    for term $\in E_v$.keys
        if not VerifyCert($E_v$[term])
            return fail                    /* Ensure LC correctness */
        if term $\notin$ AllTerms
            $E'_v$[term] ⟵ $E_v$[term]
            $E_v$.pop_by_key(term)
            continue                       /* Remove unused terms */
    return (true, $E_v, E'_v$)
```

Algorithm 4: Chain integrity check of Raft-Forensics

B.3 The Full Auditing Algorithm

In Alg. 5, we specify the full version of auditing algorithm which does not exit early after detection of one Byzantine node. It exhausts the available data and detects as many Byzantine nodes as possible. As long as a Byzantine node participates in forking the consensus, it will be exposed by this algorithm. Hence, in comparison with the early-exit version (Alg. 2), the full version is recommended, because unlike the former, it does not miss the Byzantine nodes whose accomplice is exposed earlier (e.g., with illegitimate data). However, we address that if multiple Byzantine nodes take part in the same attack (for example, voting for two different leaders at the same term), both full and early-exit variants of the auditing algorithm can find all these nodes at the same time.

As expected, it suffers from a slightly higher complexity. Suppose the number of conflicting branches equals β. The time complexity of global consistency check (Table 4) will be raised from $n\Lambda$ to $n\beta\Lambda$. However, compared to the high complexity of global legitimacy check, which is linear to the size of the log list, this change is negligible.

```
Input: Log chain $L_w$, election list $E_w$ and commitment certificate
       $C_w$ for all $w \in \mathcal{V}$, node set $\mathcal{V}$
Output: Byzantine-faulty nodes $A \subset \mathcal{V}$
Function AuditAll($L, E, C, \mathcal{V}$):
    $A, V$ ⟵ ∅, ∅
    for $v \in \mathcal{V}$
        if DataLegitimacy($L_v, E_v, C_v$)
            $V$ ⟵ $V \cup \{v\}$
        else
            $A$ ⟵ $A \cup \{v\}$
    while $|V| > 1$
        $U, w$ ⟵ ∅, $\mathrm{argmax}_{v \in V} L_v$.length
        for $v \in V - \{w\}$
            $A_v$ ⟵ PairConsistency(L, E, C, v, w)
            $A$ ⟵ $A \cup A_v$
            if $A_v \neq \emptyset$
                $U$ ⟵ $U \cup \{v\}$
        $V$ ⟵ $U$
    return $A$
```

Algorithm 5: The full auditing algorithm of Raft-Forensics.

B.4 Live-Raft-Forensics

We first outline the precise prevote conditions. Then we provide pseudocode for the new leader election process.

B.4.1 List of Prevote Conditions.

A. As a follower admitting a leader.
   A1. The leader fails to send a heartbeat before timeout $\Omega_{\heartsuit}$.
   A2. The leader fails to respond within RPC before timeout $\Omega_{\heartsuit}$.
   A3. The leader sends invalid data during RPC calls, such as unauthenticated requests, invalid certificates, and logs of invalid format.
   A4. The leader fails to send a valid commitment certificate before timeout $\Omega_{\mathsf{Request}}$ since the request was forwarded by self.
B. As a follower during election.
   B1. The candidate fails to send a valid leader certificate before timeout Ω_{Voter}.
   B2. The candidate sends invalid data, such as ill-formatted vote requests and leader certificates.
C. As a candidate: it receives $f+1$ vote rejections before timeout Ω_{Cand} or fails to receive f votes at timeout Ω_{Cand}.

B.4.2 Leader Election. In Alg. 6, we precisely define the protocol of the round-robin leader election in Live-Raft-Forensics (§8.2).

```
Input: Term $t$, node $v$
Output: New Term, New Leader
Function LeaderElection($t$, $v$):
    $\tau \leftarrow t$
    while true
        Prevote:
        broadcast $\sigma_v$(prevote($\tau$))
        Wait indefinitely for 1 prevote message
        Vote:
        $\tau \leftarrow \tau + 1$
        Start listener of $\sigma_U$(prevote($\tau + 1$)) with $|U| \geq 2$ (Jump
          to PrevoteInterrupt)
        if $c_\tau = v$
            Start timed event (Jump to Timeout after $\Omega_{\mathsf{Cand}}$)
            broadcast RequestVote($v$, $(T, J)$)
            Wait for signed votes $\sigma_{\cdot}$(RequestVote($v$, $(T, J)$))
              from $f + 1$ nodes
            broadcast LC = $\sigma_U$(RequestVote($v$, $(T, J)$))
            return $\tau, v$
        else
            Start timed event (Jump to Timeout after $\Omega_{\mathsf{Voter}}$)
            Wait for RequestVote($c_\tau$, $(T, J)$)
            if By $(T, J)$, $c_\tau$ is as fresh as $v$
                Vote for $c_\tau$ by sending RequestVote($c_\tau$, $(T, J)$)
            else
                Inform $c_\tau$ of rejection
            Wait for multi-signed $\sigma_U$(RequestVote($c_\tau$, $(T, J)$))
            return $\tau, c_\tau$
        Timeout:
        continue
        PrevoteInterrupt:
        continue and skip Prevote of next iteration
```

Algorithm 6: Round-Robin Leader Election in Live-Raft-Forensics

C COMPARISON AGAINST PEERREVIEW

We present the differences in efficiency between Raft, Raft-Forensics and PeerReview. In details, we study the communication complexity and spatial complexity of the following five communication activities of Raft.

- **Single append:** the follower is originally synchronized with the leader, and receives one single entry from the leader that can directly append to its log list.
- **Log synchronization:** the leader sends an entry that may not directly append to the follower's log list, and the follower requests for all the entry since last committed.
- **Commit:** the leader informs a follower to commit an entry.
- **Heartbeat:** the leader sends heartbeat messages as a liveness signal.
- **Vote Request:** a candidate requests a follower for vote and eventually becomes the leader.

The total complexities are combinations of the activities which depend on the workload of the system. Typically, an *idle* workload has extremely low frequency of transactions, and heartbeat is the major activity. In a *light* workload, single append is the major activity because transactions are submitted at a frequency low enough for the leader to finish replicating the previous entry to all followers before handling the next one. In a *heavy* workload, log synchronization is the major activity because the leader has to replicate multiple entries that include all queued transactions. Note that we do not focus on the remaining two activities. The frequency of commit activity can be chosen at a low frequency compared to others. Vote requests is rare in practice because it is triggered by the leader's crash fault.

Considering the differences in auditor setups between PeerReview and Raft-Forensics, a direct comparison may be unfair. Hence, we derive HubReview, which is a simple, centralized variant of PeerReview. Briefly, instead of letting the nodes auditing (or witnessing) each other in PeerReview, we let one single trusted auditor audit all the nodes in HubReview like Raft-Forensics. All other protocols of HubReview are the same as PeerReview. Consequently in HubReview, nodes no longer need to forward authenticators (digital signatures) to witnesses, and thus save some communication complexities.

We present the complexities in Table 6. For clarity, we list the relevant notations and their example values in Table 8 and the resulting relative overhead complexities in Table 7. Under high workloads, Raft-Forensics's overhead communication complexity is roughly 1/4 of HubReview and 1/6 of PeerReview. Under low workloads, Raft-Forensics's overhead communication comlexity is roughly 1/2 of HubReview and 1/3 of PeerReview. Under idle workloads, Raft-Forensics has no communication overheads. For all workloads, Raft-Forensics has no amortized spatial overheads. In summary, Raft-Forensics has a significant advantage over PeerReview in efficiency.

D OTHER EVALUATION RESULTS

D.1 Effect of Transaction Size

We extend our evaluation results in §7.1 by inspecting the effects of *transaction size* on both latency and throughput. To observe these effects, we fix the client concurrency level and extend our choices for transaction sizes to a geometric sequence starting with 256 Bytes to 2 MiB. Each single experiment is configured by client concurrency and transaction size in the same way as in §7.1, and yields the throughput and latency of transactions. Because the concurrency levels in Hotstuff and Dumbo-NG configurations are not comparable against Raft and Raft-Forensics, we only run the experiments for the latter.

Figures 7a and 7b show the effects of transaction size on the system performance. In both figures, Raft-Forensics performs similarly to Raft – over all experiments, its latency is at most 25% higher than Raft, while its throughput is at least 93% of Raft. When the number of concurrent clients is no greater than 10, the throughput of Raft-Forensics reaches 98% of Raft at least, which makes them almost identical. These figures are also helpful for system designers to choose a suitable transaction size to maximize bandwidth usage without sacrificing latency. For instance, with 50 clients, 16 KiB is the most suitable transaction size, while with 10 clients, 128 KiB is optimal.

	Communication Complexity				Amortized Spatial Complexity			
	Raft	Raft-Forensics (ours)	PeerReview	HubReview	Raft	Raft-Forensics (ours)	PeerReview	HubReview
Single Append	b	$2s$	$(2w+2)s+2p$	$2s+2p$	b	0	$2s+2b$	$2s+2b$
Log Sync	$(m+1)b$	$(t+2)s$	$(6w+6)s+6p$	$6s+6p$	mb	$(t-1)s$	$6s+(2m+2)b$	$6s+(2m+2)b$
Commit	CONST	$(f+1)s$	$(2w+2)s+2p$	$2s+2p$	0	0	$2s$	$2s$
Heartbeat	CONST	0	$(2w+2)s+2p$	$2s+2p$	0	0	$2s$	$2s$
Vote Req	CONST	$p+(f+2)s$	$(6w+6)s+6p$	$6s+6p$	0	$p+(f+2)s$	$6s$	$6s$

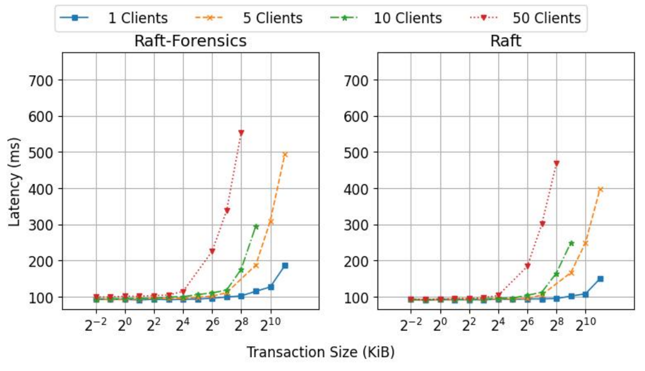
Table 6: Comparison of complexities between Raft, Raft-Forensics, PeerReview and HubReview. For Raft-Forensics, PeerReview and HubReview, we list the *overhead* complexities comparing against Raft.

	Communication Complexity			Spatial Complexity		
Workload	Raft-Forensics	PeerReview	HubReview	Raft-Forensics	PeerReview	HubReview
High	**26.5%**			**0%**		
Low	**52.9%**	153.4%	100%	**0%**	100%	100%
Idle	**0%**			**0%**		

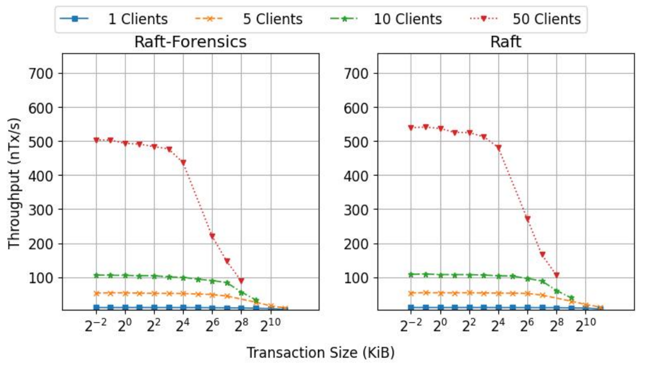
Table 7: Relative overhead complexities of Raft-Forensics, PeerReview and HubReview. Parameter values are listed in Table 8.

Notation	Description	Value
b	Size of a log entry	
p	Size of a hash pointer	64
s	Size of a digital signature	72
w	Number of witnesses in PeerReview	1
m	Number of entries in log synchronization	
t	Number of unique terms in log synchronization	1

Table 8: List of notations and their example values in Table 6.



(a) Relation between latency and transaction size.



(b) Relation between throughput and transaction size.

Figure 7: Relation between latency/throughput and transaction size.

D.2 Performance of Auditing Algorithm

In addition to theoretical complexity bounds, we simulate and benchmark the audit process. The audit process consist of two steps – data generation and evaluation of auditing performance.

To benchmark the audit process, we need to control: 1) the adversarial nodes' behavior, 2) the number of requests, which approximately equals the length of log chains, and 3) the chunk size of log chains. In our C++ implementation (as well as vanilla nuRaft) it is time-consuming and difficult to add different varieties of adversarial agents to generate desired test cases, particularly during elections (e.g., electing a leader that is favorable to the adversary). Moreover, the full nuRaft implementation introduces a vast range of race conditions, which are challenging to control.

As a result, for more fine-grained control over our audit experiments, we used the simulator to generate the data required for auditing. We summarize the functionalities of the simulator as follows.

- It is able to fully simulate log replications with different pairwise node delays.
- It simplifies leader election so the choice of the new leader is controllable and suitable for testing auditing performance. For instance, we can elect a leader that is most favorable for the adversary.
- It simulates a client who submits requests periodically and a configurable number of nodes who replicate and commit log entries in response.

- It is also capable of assigning the adversary to a node and simulating the fork and bad vote attacks in Examples 4.2 and 4.3. It can launch the attack once at any time during request submissions.
- The simulator ensures the adversary is able to generate legitimate data to prevent it from being caught before consistency checks.
- In the simulator, the nodes write their committed log entries in order into chunked files. They also write their commitment certificate and leader certificates.

Using this simulator, we set up five server nodes where one of them is Byzantine. We generate transaction traffics of sizes 10,000, 40,000, 90,000 and 250,000. This results in approximately the same number of entries on the log chain of each node. We let the Byzantine node launch two types of attacks – fork (Example 4.2) and bad vote (Example 4.3). For each attack, we set two different chunk sizes ∞ and 100, which corresponds to the max number of log entries in each file. Under each configuration of these parameters, we let the Byzantine node launch the attack at various positions during the transaction traffic. Specifically, the position can be expressed by the proportion of the time of attack to the time of the last transaction, which is selected from 0.1 to 0.9. We plot the time consumption of **DataLegitimacy** in Fig. 8a and the time consumption of **AuditAll** in Fig. 8b.

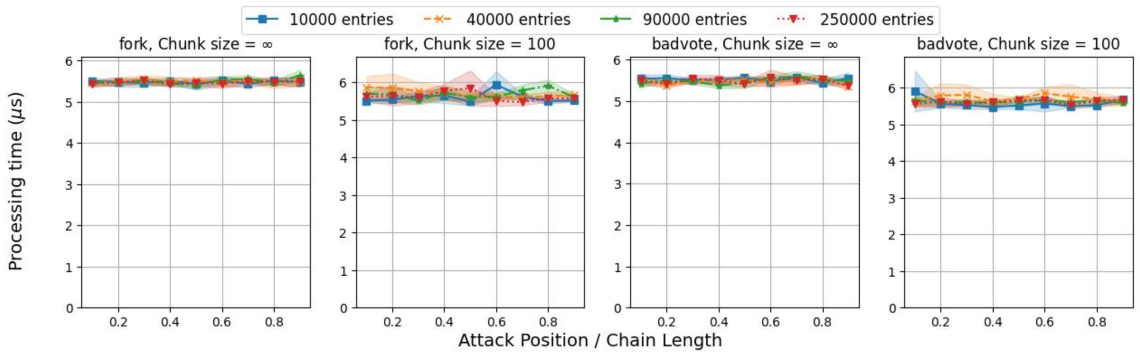
By Fig. 8a, the time consumption of **DataLegitimacy** is constant over each log entry, which implies a linear time complexity over all entries. This constant is irrelevant of the attack position or the size of the log list. Notably, the time consumption is improved when we choose chunk size 100, instead of saving everything in the same file.

By Fig. 8b, the time consumption of **AuditAll** is constantly low when the Byzantine node launches the fork attack or the nodes use chunked files for storage. In the fork attack, it is easy for the auditor to discover from the last entries that the log lists conflict at the same term, and then expose the leader of that term without further looking into the depths of the lists. In the bad vote attack, however, the auditor must search for log entries by a given term. Specifically, when two nodes u and v have different ending terms t_u and t_v where $t_u > t_v$ without loss of generality, the auditor must find the first entry in u's log list that has a higher term than t_v. The earlier the attack, the deeper this entry is, and the longer it takes for the auditor to scan from rear to front as we do in this experiment. This explains the third subfigure of Fig. 8b. When the log entries are stored in chunks, the auditor can derive the file that stores this entry and directly search within the file which has no more than 100 entries. This technique accelerates consistency checks by up to 4000× by comparing the third and fourth subfigures.

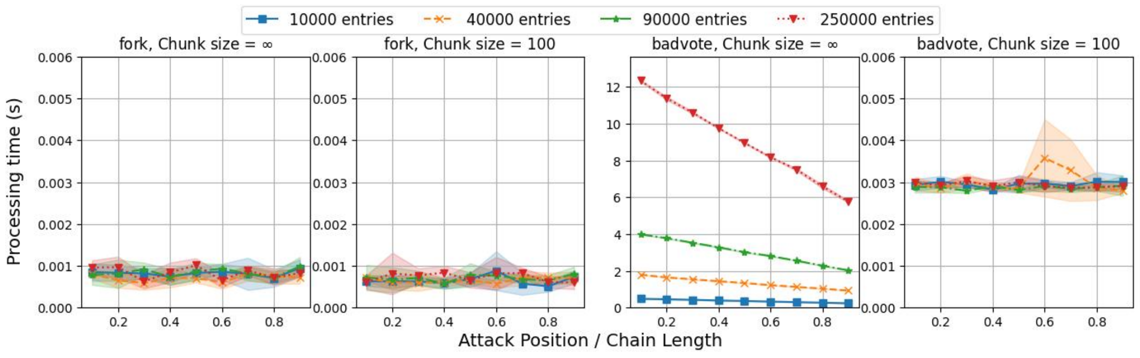
With the total processing time reduced to milliseconds, the **AuditAll** algorithm for consistency checks spends much shorter time than the **DataLegitimacy** algorithm for data legitimacy checks, which takes tens or hundreds of milliseconds. This is consistent with our complexity analysis in §7.2.

D.3 Dashboard of Auditing Algorithm

Fig. 9 shows the screenshot of the dashboard mentioned in §7.2.

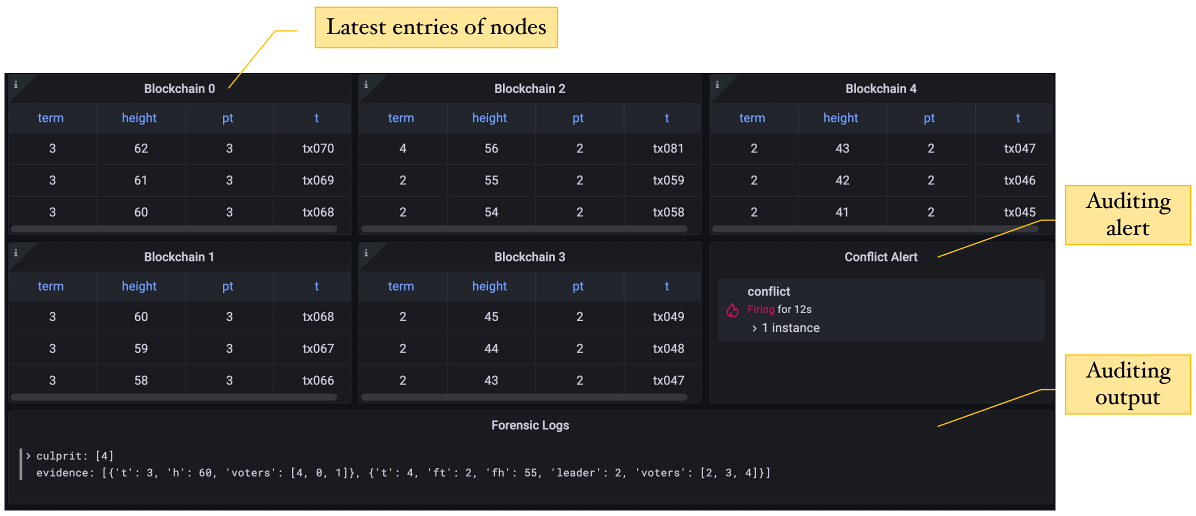


(a) Processing time of `DataLegitimacy`, the first stage of auditing.



(b) Processing time of `AuditAll`, the second stage of auditing.

Figure 8: Evaluation of different stages of the auditing algorithm.



Blockchain 0

term	height	pt	t
3	62	3	tx070
3	61	3	tx069
3	60	3	tx068

Blockchain 2

term	height	pt	t
4	56	2	tx081
2	55	2	tx059
2	54	2	tx058

Blockchain 4

term	height	pt	t
2	43	2	tx047
2	42	2	tx046
2	41	2	tx045

Blockchain 1

term	height	pt	t
3	60	3	tx068
3	59	3	tx067
3	58	3	tx066

Blockchain 3

term	height	pt	t
2	45	2	tx049
2	44	2	tx048
2	43	2	tx047

Conflict Alert

conflict
Firing for 12s
› 1 instance

Forensic Logs

```
› culprit: [4]
  evidence: [{'t': 3, 'h': 60, 'voters': [4, 0, 1]}, {'t': 4, 'ft': 2, 'fh': 55, 'leader': 2, 'voters': [2, 3, 4]}]
```

Figure 9: Dashboard of auditing in Raft-Forensics. This figure demonstrates a simulation of a blockchain system, consisting of five nodes with leader election occurring every 20 transactions. One of the nodes, node 4, is a Byzantine node that launches a bad vote attack at 70% progress of the simulation, resulting in a safety violation. Upon detecting the conflict, the auditor process issues an alert and initiates an auditing algorithm to identify the culprit and extract evidence. In this particular scenario, node 4 is successfully detected, and the extracted evidence shows that it voted for a CC in term 3 and later voted for a conflicting LC in term 4.