Prosecutor: An Efficient BFT Consensus Algorithm with Behavior-aware Penalization Against Byzantine Attacks

Gengrui Zhang
Department of Electrical & Computer Engineering
University of Toronto, Canada
gengrui.zhang@mail.utoronto.ca

Hans-Arno Jacobsen
Department of Electrical & Computer Engineering
University of Toronto, Canada
jacobsen@eecg.toronto.edu

ABSTRACT

Current leader-based Byzantine fault-tolerant (BFT) protocols aim to improve the efficiency for achieving consensus while tolerating failures; however, Byzantine servers are able to repeatedly impair BFT systems as faulty servers launch attacks without costs. In this paper, leveraging Proof-of-Work and Raft, we propose a new BFT consensus protocol called Prosecutor that dynamically penalizes suspected faulty behavior and suppresses Byzantine servers over time. Prosecutor obstructs Byzantine servers from being elected in leader election by imposing hash computation on new election campaigns. Furthermore, Prosecutor applies message authentication to achieve secure log replication and maintains a similar message-passing scheme as Raft. The evaluation results show that the penalization mechanism progressively suppresses and marginalizes Byzantine servers if they repeatedly launch malicious attacks.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**.

KEYWORDS

blockchains; consensus protocols; Byzantine fault tolerance

ACM Reference Format:
Gengrui Zhang and Hans-Arno Jacobsen. 2021. Prosecutor: An Efficient BFT Consensus Algorithm with Behavior-aware Penalization Against Byzantine Attacks. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Québec city, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3464298.3484503

1 INTRODUCTION

Studies of Byzantine fault-tolerant (BFT) protocols have gradually arisen to provide solutions that improve the throughput and latency for trending permissioned blockchain platforms such as Hyperledger Fabric [6], R3 Corda [10], and Libra [35]. Practical Byzantine fault tolerance (PBFT) [15] and its variants are widely used as a foundation for obtaining consensus [13, 14, 18, 43]; under normal operation, PBFT commits a value by engaging five message-passing phases with a message transmission complexity of $\mathcal{O}(n^2)$ [15, 36]. The complexity could reach $\mathcal{O}(n^3)$ when f+1 replicas multicast view-change messages to replace a faulty leader. Therefore, it is commonly believed that large-scale PBFT-based blockchain systems are impractical [18, 43].

More efficient protocols can be found in the category of protocols tolerating benign failures. For example, Raft [39] has been widely adopted in private blockchain implementations and in-cluster data replications [6, 43]. Compared with Paxos [31, 32], Raft elects a designated leader in a given logical time to conduct log replication in which entries flow only from the leader to the other servers, requiring $\mathcal{O}(n)$ message deliveries [25, 26].

Optimizations for BFT algorithms have been made from various angles. For example, BFT-SMaRt [9] develops fine-tuned PBFT state machine implementations; HoneyBadger BFT [36] proposes a leaderless approach; SBFT [24] amortizes coordination workloads into different leader servers; and HotStuff [44] rotates leadership for transactions. Additionally, Aardvark [16] requires a minimum throughput from correct leaders, and RBFT [7] runs consensus in parallel on redundant replicas to enhance robustness. However, none of the above approaches penalizes Byzantine servers, imposing costs to faulty behavior that is suspected of attacking the system.

In contrast to these approaches, in this paper, we propose a new BFT consensus protocol called Prosecutor, which comprises two key components: Proof-of-Commit leader election (PoC_{LE}), which suppresses the ability of Byzantine servers to usurp leadership over time, and secure log replication (SLR), which uses message authentication to achieve consensus with linear message complexity.

PoC_{LE} adapts some features from Raft (e.g., terms, and server states) and leverages the notion of Proof-of-Work [38] to impose computational work on suspected faulty servers, suppressing Byzantine servers from taking over leadership. In particular, a server that initiates a leader election request has to perform mandatory computations incurred by a hash function; the difficulty of the hash function increases dramatically if a server repeatedly initiates election requests, potentially resulting in exorbitant energy and time costs. If Byzantine servers continuously usurp the leadership, the difficulty of future leader elections rapidly becomes unaffordable for Byzantine servers to perform.

After being elected through PoC_{LE}, the leader begins conducting the consensus process and the system enters the SLR stage. In general, SLR contains five leader-to-others message-passing phases, and all the messages are signed by the sender to preserve server identity. Thus, armed with message authentication, SLR ensures that a faulty leader cannot manipulate the messages in all the phases of SLR and tolerates up to f Byzantine failures with $3f$+1 servers in total. Under normal operation, messages flow only from the

Middleware '21, December 6–10, 2021, Québec city, QC, Canada

ACM ISBN 978-1-4503-8534-3/21/12…$15.00
https://doi.org/10.1145/3464298.3484503

leader and other servers, achieving consensus with a linear message transmission cost.

Equipped with PoC$_{LE}$, Prosecutor ultimately suppresses Byzantine servers, gradually pushing leader duties to correct servers. If an elected server cannot conduct log replication, the server will be suspected as faulty and subjected to more difficult computational work associated with the development of future leaders.

Thus, Byzantine servers are gradually suppressed and ultimately cannot afford the computation imposed by the penalization scheme. Prosecutor protects correct leaders from being usurped by Byzantine servers over time.

The contributions of this paper are threefold:

(1) Proof-of-Commit leader election (PoC$_{LE}$). Mandatory and resilient hash computations are imposed on new election requests. PoC$_{LE}$ makes it unaffordable for Byzantine servers to continuously attack the system by taking over leadership from correct servers.
(2) Secure log replication (SLR). The log replication scheme with message authentication guarantees that faulty servers cannot manipulate proposed values, and SLR tolerates f Byzantine failures with $3f$+1 servers in total.
(3) Implementation and evaluation comparisons with HotStuff and PBFT. We implemented the Prosecutor protocol and evaluated its performance in terms of throughput and latency as well as leader election time under simulated Byzantine attacks.

The remainder of this paper is organized as follows: Section 2 presents Byzantine failures in Raft; Section 3 introduces the design details of Prosecutor and discusses its correctness properties; Section 4 analyzes the process for vanishing Byzantine proposers and discusses potential limitations; Section 5 presents the evaluations of Prosecutor; and Section 6 discusses related work.

2 BYZANTINE FAILURES IN RAFT

Raft ensures log safety under non-Byzantine conditions such as crash failures, network delays, packet loss and duplication [39]. Although Raft is not designed to tolerate Byzantine failures, Raft is efficient in terms of replicating log entries, and some concepts can be adapted to design a new BFT consensus protocol. Thus, it is crucial to understand the impact of Byzantine attacks before leveraging Raft into our new algorithm. In this section, we present some background about Raft and a brief overview of possible erroneous scenarios under the attacks from Byzantine servers.

Raft divides time into *terms* numbered with consecutive integers. The term value acts as a logical clock [30] and can be increased in two cases:

- If server $\mathcal{S}_i$ receives a higher-term value sent from another server $\mathcal{S}_{k<i}$, $\mathcal{S}_i$ sets its $term_{\mathcal{S}_i}$ to $term_{\mathcal{S}_k}$.
- If its timer expires, S_i triggers a timeout and increments its term ($term_{\mathcal{S}_i}$) by one.

In Raft, servers take on one of three roles: leader, candidate, or follower. Under normal conditions, only one leader interacts with clients and other servers through *heartbeats* in fixed intervals. Each server has a timer, and the timer is reset upon receiving *heartbeat* RPCs. If the leader crashes, followers transition to the candidate state after their timer expires and initiate leader election requests to solicit votes by issuing *RequestVote* RPCs. If votes from a majority of servers are collected, the candidate becomes the leader that conducts further consensus processes. The leader replicates entries to servers using *AppendEntries* RPCs, and followers passively respond to requests and synchronize entries.

Although Raft is not designed to tolerate Byzantine failures, it helps us to design a Byzantine fault-tolerant algorithm by understanding how Byzantine faulty servers jeopardize safety and liveness of Raft leader election and log replication. Figure 1 illustrates four examples in both phases.

Failures during leader election. Byzantine failures undermine Raft's leader election mechanism. For example, in Figure 1a, a faulty follower can usurp the leadership from a correct leader. The Byzantine server S_4 deliberately increases its term higher than that of the leader S_1 and initiates a new election. Servers vote for S_4, and S_4 replaces the correct leader. Furthermore, a faulty leader can undermine elections from candidates. For instance, in Figure 1b, the faulty leader S_2 sabotages an ongoing election by sending a premeditated higher-term command after receiving the election request from S_4. Consequently, S_2 remains as the leader, whereas S_4 aborts the current election campaign and transitions back to a follower.

Failures during log replication. Faulty followers have no impact on correctness if they remain a minority because the leader dominates the log replication process. In Figure 1c, a correct leader S_1 receives a majority of replies and commits the proposed entry, regardless of the faulty reply from S_4. By contrast, a faulty leader, shown in Figure 1d, can commit arbitrary entries and force followers to synchronize. Followers cannot perceive this situation because they are isolated and communicate only with the leader; therefore, the log replication is no longer safe.

To conclude, Raft cannot defend against attacks from a Byzantine server in leader election, and the system is vulnerable during log replication if the leader is faulty. However, by having a correct leader, Raft can proceed normally during log replication. Therefore, detecting and suppressing the ability of Byzantine servers from taking over leadership can assist the system in operating normally.

3 PROSECUTOR DESIGN

In this section, we introduce the design of Prosecutor in detail. First, we discuss the system model, including failure, network, and cryptography assumptions. Next, we present an overview of Prosecutor and PoC$_{LE}$, the leader election that penalizes suspected faulty behavior. Then, we describe the client interaction and secure log replication. Finally, we discuss some correctness properties.

3.1 System Model

The basic failure assumption is that a typical Prosecutor system that consists of $3f$+1 servers can tolerate up to f Byzantine servers, which may collude to exhibit arbitrary failures, forming sets of faulty messages, to undermine safety. Prosecutor does not limit the number of faulty clients.

To preserve liveness, Prosecutor assumes a partially synchronous system. Compared with asynchronous systems, where consensus is not guaranteed [21], partially synchronous systems allow the arrival of messages with a bounded network delay [19]. Specifically,

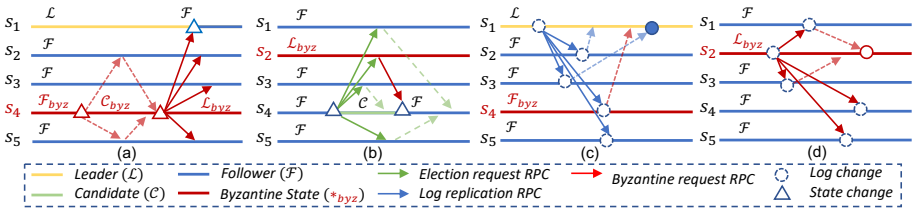


Figure 1: Byzantine failures in Raft. In (a) and (b), Raft cannot defend takeover attacks. In (c) and (d), minority faulty followers do not cripple log replication.

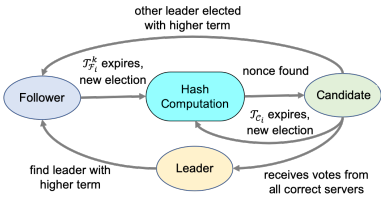


Figure 2: State transitions in Prosecutor.

a known bounded network delay δ applies to each execution but is guaranteed to hold starting only at some unknown time T, i.e., the global stabilization time; after time T, the message transmission delay among servers is bounded by δ. Although partial synchrony is required for preserving liveness, safety does not rely on any network assumptions.

Prosecutor utilizes threshold signatures [34, 41] to enable distributed cosigning on messages, which has been used by numerous state-of-the-art BFT protocols [24, 36, 44, 45]. An established threshold signature, denoted by σ, allows the combination of t individually signed messages into one signature with a threshold t. For example, if a server S sends a (t, n) threshold signature for message $\mathcal{M}$, recipients are able to verify that there were at least t out of n servers with signed $\mathcal{M}$.

The adversary model allows faulty servers and faulty clients to exhibit arbitrary faults excluding garbage data pouring (i.e., proposing meaningless data to increase server workload and saturate the network). For example, adversaries may send conflicting messages to different servers, stop responding at any given time, or issue election requests without following the protocol. By requiring message authentication, Byzantine servers are not able to steal the identities from correct servers. Thus, Prosecutor can construct a Byzantine quorum of $2f$+1 messages from different servers to tolerate the erroneous messages sent from f Byzantine servers. Furthermore, with the assumption of partial synchrony, Prosecutor can implement timers with a period of t ($t \gg \delta$) to detect non-responding servers. Similarly to PBFT [15], Prosecutor cannot tolerate adversaries that pour garbage data into the system. However, the system's intolerance of this behavior can be mitigated by applying access control. The system can dynamically change access permissions and limit the sending rate to identify and deny requests from garbage-pouring clients.

Since the penalization mechanism involves computation works, the system model requires no super-computation-power participants. While participating servers may have various computation abilities, super-computation-power servers may obliterate the effectiveness of penalization that aims to suppress Byzantine servers. In our evaluation, servers all have the same configuration with the same computation power.

3.2 Protocol Overview

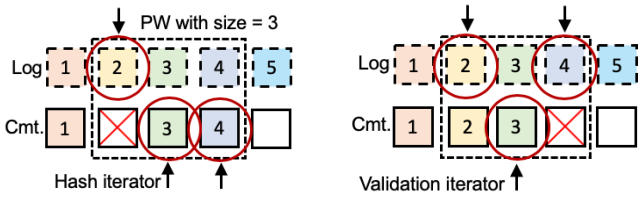
Prosecutor is a leader-based BFT protocol with two vital components: Proof-of-Commit leader election (PoC$_{LE}$) and secure log replication (SLR). PoC$_{LE}$ detects the absence of correct leadership and suppresses Byzantine servers by penalizing faulty behavior, and SLR ensures the safety of log replication under all circumstances. To simplify the basic setup, we adopt some elements from Raft [39] as follows:

- The three server states. Each server operates in one of the three states: leader ($\mathcal{L}$), candidate ($\mathcal{C}$), or follower ($\mathcal{F}$). Under normal operation, the system has only one leader, and all the other servers assume a follower role.
- The term concept. Prosecutor divides time into terms, which increase monotonically. The term properties described in Section 2 also apply to our approach; however, term increments are subjected to performing computational work (details in Section 3.3).

In contrast to Raft, instead of using a single election timer, Prosecutor deploys two exclusive types of timers for followers and candidates:

(1) A commit-sensitive timer, denoted by $\mathcal{T}_{\mathsf{F}_i}$. This timer is employed to detect the absence of correct leadership and is active only when the server state is a follower. Each follower $\mathcal{F}_i$ initializes $\mathcal{T}_{\mathsf{F}_i}$ with a time period of t_c; $\mathcal{T}_{\mathsf{F}_i}$ starts to count down if f+1 *secondComplain* messages are received (introduced in Section 3.4) and resets to t_c when a value is committed.
(2) A self-wait timer, denoted by $\mathcal{T}_{\mathsf{C}_i}$. This timer is active only when the server is in the candidate state. $\mathcal{T}_{\mathsf{C}_i}$ is initialized with a time period of t_s and starts to count down when a candidate $\mathcal{C}_i$ begins to solicit votes from all the other servers. If $\mathcal{C}_i$ cannot complete the current election campaign before $\mathcal{T}_{\mathsf{C}_i}$ expires, $\mathcal{C}_i$ aborts the current election and launches a new leader election campaign.

With commit-sensitive timers, faulty leaders can be terminated early through the client interaction. Followers consider the current leader is faulty upon the expiration of their commit-sensitive timers and begin new leader election campaigns, whereas the self-wait timer limits the completion time for one election campaign. After a follower obtains a qualified hash computation result, the follower transitions to the candidate state and broadcasts requests to solicit votes; the broadcast also starts the self-wait timer. If a candidate cannot collect enough votes during its election campaign, the candidate abandons the current campaign and starts the hash process again. This backward state transition ensures that if concurrent candidates split votes in a given election, candidates are able to restart new election campaigns, avoiding provisional livelocks stranding them in the candidate state. Furthermore, both the commit-sensitive timer and self-wait timer are initialized with random periods within



(a) The PW of server S_i. (b) The located PW of server S_j.

Figure 3: Proof window with size = 3.

a certain range (e.g., 300–600 ms), aiming to reduce the probability that different servers trigger timeouts simultaneously, downgrading the probability of concurrent election campaigns.

State transitions. Figure 2 illustrates the state transitions. Initially, all servers run as followers. Suppose a server $\mathcal{S}_i$ in Prosecutor runs as a follower, $\mathcal{F}_i$. If its timer ($\mathcal{T}_{\mathsf{F}_i}$) expires, $\mathcal{F}_i$ invokes algorithm Election-Request to start an election campaign, engaging a hash computation process to obtain a hash result that meets certain requirements; the hash result represents the computation that $\mathcal{F}_i$ must perform to proceed. Then, $\mathcal{F}_i$ transitions to $\mathcal{C}_i$ and issues election requests to solicit votes from all the other servers. At this time, $\mathcal{C}_i$ becomes a follower again if a higher-term leader is found, it remains in the candidate state if $\mathcal{T}_{\mathsf{C}_i}$ expires and executes algorithm Election-Request again to start a new election campaign, and finally, it becomes a leader ($\mathcal{L}_i$) if votes are collected from f+1 servers. $\mathcal{L}_i$ takes the leader duty until another valid leader with a higher term is discovered and transitions back to serve as a follower.

3.3 Proof-of-Commit Leader Election, PoC$_{LE}$

PoC$_{LE}$ imposes computation work on servers that initiate leader election campaigns. Generally, followers whose timers trigger a timeout initiate an election campaign by executing algorithm Election-Request to perform computations tailored to the server's behavior.

Inspired by Proof-of-Work (PoW) [38], the amount of computation can be adjusted by requiring different prefixes of the hash result. Algorithm Election-Request combines a *proof window* with a random string, called *nonce*, and repeatedly hashes the combination until a result meets the *threshold* requirement. These concepts are explained next, along with a discussion of the algorithm.

A *proof window* (PW) shows a server's log and commit states. To keep the log consistent, a newly elected leader must have the most up-to-date commit state. If the system waits for one value to be committed before engaging in the next one, a PW containing the last committed value is able to reflect the server's commit state. However, Prosecutor allows the consensus processes for committing values to operate in parallel so that at the time when a timeout is triggered, servers may have arbitrary commit states at a given time, as some servers may have committed a value while some may have logged the value but still wait for the commit instruction from the leader. In this case, a PW should also contain logged but uncommitted values to represent a server's commit state. Specifically, when $\mathcal{T}_{\mathsf{F}_i}$ expires, a PW starts with the first uncommitted value (the first slot in PW) and ends at the last committed value (the last slot in PW), and the proof window size is denoted by k.

Algorithm 1: Election-Request.

```
Input: term, Id, PW, threshold
 •PW is the current proof window
1  flag ← false
2  hpw ← HASH(PW)                    •PW is hashed iteratively
3  term ← term + 1
4  while flag < true do
5      nonce ← GENERATE(rand)
       •generate random strings for hashing
6      res ← HASH(term, Id, hpw, nonce)
7      pass ← VALIDATE(res, threshold)
8      if pass then
9          flag ← true
10 args ← CONCAT(term, Id, hpw, res, nonce)
11 return args
```

Figure 3a shows an example of a PW with size=3, and a value v in the slot m of server S_i's PW is denoted by v_i^m. Server S_i has one logged but uncommitted value (v_i^2) and two committed values (v_i^3 and v_i^4), and the information of the PW will be sent to all other servers through election requests. Assume another server, $\mathcal{S}_j$, allocates its own PW in Figure 3b based on the information of the received PW in Figure 3a. As long as the values in the same slot are identical, regardless of log or commit, the two PWs are considered to be identical. In this example, although v_j^4 is in $\mathcal{S}_j$'s log, the two PWs in Figure 3a and 3b are still considered to have the same elements and have the same hash results.

After hashing the PW, algorithm Election-Request increments the term and begins the hash computation for obtaining a qualified result that must meet a threshold requirement. The threshold is an integer representing the number of identical consecutive characters the result should prefix. For example, if applying SHA256, the hash result of string "tx#1:SCRBGGLMFI" is "eeeee14630ddb57...", so it meets a threshold requirement of *threshold*=5. Therefore, by adjusting the threshold requirement, the amount of computation changes; namely, the higher the threshold is, the more computation an election requester has to perform.

In each loop, function Generate() generates a random string called *nonce*, which is hashed together with *term*, *Id*, and *hpw* to obtain a result, *res* (Line 6). The loop continues until *res* meets a certain threshold requirement (Line 7), and the algorithm returns arguments for issuing election requests. Theoretically, given a hashing space $\mathcal{S}$ and a threshold th, the runtime complexity of algorithm Election-Request is $\mathcal{O}(\mathcal{S}^{th})$.

Threshold credit. The adjustment of thresholds is imperative for penalizing suspected faulty behavior since higher thresholds incur costlier computation. The threshold for a server is dynamically determined and related to a server's behavior in both leader election and log replication. Each server stores a map containing the IDs and thresholds of all the other servers and updates them at runtime.

A server $\mathcal{S}_i$ increases the threshold for server $\mathcal{S}_j$ following Equation (1) if a valid leader election request sent by $\mathcal{S}_j$ is received, in which *newTerm* is the term received from $\mathcal{S}_j$, *term* is the current term of $\mathcal{S}_i$, and *reqCtr* is the recorded number of election requests

Algorithm 2: GRANT-VOTE.

Input: *args*, *myPW*, *myTerm*
•*myPW is located in the proof window on voters*
1 $rTerm$, rId, $rHpw$, $rRes$, $rNonce \leftarrow args$
2 **if** $rTerm$ Ÿ $myTerm$ **then**
3 | **return** `false`
4 $myTerm \leftarrow rTerm$ •*always update to higher term*
5 $hpw \leftarrow$ HASH($myPW$)
6 $pass \leftarrow$ COMPARE($rHpw$, hpw)
7 **if** $pass$ < `true` **then**
8 | **return** `false`
•*Identical hash results of two PWs are obligatory*
9 $res \leftarrow$ HASH($rTerm$, rId, hpw, $rNonce$)
10 $pass \leftarrow$ EQUALS($rRes$, res)
11 **if** $pass$ < `true` **then**
12 | **return** `false`
•*check whether the hash result complies with the nonce*
13 $threshold \leftarrow$THRESHOLD($rServer.Id$)
14 $pass \leftarrow$ VALIDATE(res, $threshold$)
15 **if** $pass$ < `true` **then**
16 | **return** `false`
•*check whether the hash result meets required threshold*
17 **return** *pass*

initiated by $\mathcal{S}_j$.

$$threshold = (newTerm - term) \times reqCtr, \tag{1}$$

The threshold is increased along with the number of initiated election requests, dynamically inhibiting Byzantine faulty servers from launching takeover attacks. First, each time a faulty server initiates an election request but does not lead the consensus after being elected, the threshold for the faulty server is increased, resulting in dramatic cost for this server to initiate the next election campaign. Second, Equation (1) restricts the increment of terms, protecting against potential attacks aimed at maxing out the term value. In practice, if terms reach the maximum capacity of their data structure, a consensus for `refresh-term` is proposed to refresh the term value based on $2f$+1 votes for agreement.

However, a server $\mathcal{S}_i$ decrements the threshold for server $\mathcal{S}_j$ by one ($threshold$=$max(0, threshold - 1)$) in a given term if k values have been committed under the leadership of $\mathcal{S}_j$, in which k is a predefined parameter representing the minimum number of values that are expected to be committed under correct leadership. The threshold decrement reduces the computational burden imposed on correct leaders to initiate future leader election campaigns.

Finally, if a follower obtains a qualified hash result, the follower transitions to the candidate state and broadcasts signed messages $voteMe$=$\sigma_{c_i}\langle args\rangle$ to solicit votes from all the other servers. The candidate votes for itself and starts its self-wait timer, $\mathcal{T}_{C_i}$, to limit the current election time.

After receiving a *voteMe* message, followers examine the signatures and extract *args* according to algorithm GRANT-VOTE, which has an $\mathcal{O}(1)$ runtime complexity since the hashing function needs to be executed only once. To grant a vote, there are three validation checks:

(1) *Term validation.* The term (*rTerm*) received from the requesting candidate must be higher than the follower's term (*myTerm*); if not, a value of false is returned.
(2) *Commit validation.* The follower allocates its own PW (*myPW*) based on the received PW. The hashes of the two PWs, *rHpw* and *hpw*, must be identical; if not, the algorithm returns a value of false.
(3) *Threshold validation.* The hash result *res* should meet the threshold for the candidate. Function THRESHOLD() returns the current threshold for the requesting servers by checking its *Id*. If the result does not comply with the required threshold, it returns a value of false.

In fact, *term validation* (Lines 1-4) guarantees that followers never vote for a candidate with a lower term. When a server receives a message with a higher term, the server synchronizes to this term and refuses to accept lower-term commands. Next, *commit validation* (Lines 5-8) checks the log consistency in which the hash results of the received PW and local PW must be identical, confirming that the requesting candidate is up-to-date. Then, *threshold validation* (Lines 9-16) verifies the tailored computational work for the candidate by examining the hash result. After comparing the two hash results, function THRESHOLD() returns the threshold of the candidate, and VALIDATE() inspects the computation by checking the prefix of *rRes* accordingly. Finally, if message *voteMe* passes algorithm GRANT-VOTE and the follower has not voted in the current term, the follower replies with a vote message $vote$=$\sigma_{f_i}\langle t, id, th\rangle$ to the requesting candidate, where t and id are the follower's term and Id, respectively, and th is the current threshold for the candidate.

A candidate becomes the new leader after receiving f+1 votes in the same term and broadcasts the collected votes to all the other servers for verification. However, if the candidate cannot collect f+1 votes before the self-wait timer expires, the candidate aborts the current election campaign and initiates a new one by executing algorithm ELECTION-REQUEST to obtain a new hash result.

In addition, a temporary network partition may cause legitimate leader election requests to be unsuccessful, as no candidate can collect enough votes to claim the next leadership. Since partial synchrony assumes temporary asynchrony during the global stabilization time, Δ_t, message passing between servers is suspended. Thus, candidates may not be able to complete an election campaign and initiate new leader election requests by performing the hash computation with an increased threshold; after Δ_t, when the network recovers, unnecessary computational work may be inflicted on correct servers. Prosecutor avoids such consequences by inspecting the thresholds a candidate has accrued. In particular, if the self-wait timer expires, a candidate periodically broadcasts `Inquire-Threshold` messages to inquire about its current threshold that a nonce should prefix. Then, the requesting candidate adjusts the threshold accordingly after other servers reply. Therefore, after Δ_t, the candidate still performs the computational work specified by its past behavior, regardless of the consequences entailed during the partition period.

3.4 Client Interaction

The design of client interaction in normal operation follows the philosophy of that in PBFT [15]. Clients are able to communicate with all the servers. To propose a value, a client broadcasts a signed proposal $propo=\{\sigma_{c_i}\langle d(v_k), ts, p, id\rangle, v_k\}$. Timestamp ts specifies the logical order of client requests, p indicates the message is a proposal, and id distinguishes client C_i. Clients reserve a timer to limit the period of time for a proposed value to be committed. The timer begins after the proposals are sent. If sufficient announcements (see Section 3.5) from the servers cannot be collected before the timer expires, the client stops waiting for announcements and abandons the proposal. Then, the client resets the timer and re-sends the proposal with a new timestamp.

Moreover, different from PBFT, Prosecutor empowers a novel complaint-based client-server interaction to detect leader failures. Clients report unsuccessful commitment of the proposed value to disseminate speculations that the current leader has failed. Particularly, if the timer keeps expiring, clients send *clientComplain* messages for the value to the connected servers. Next, the servers that have received the *clientComplain* messages forward the proposal (*propo*) to the leader and broadcast *secondComplain* messages to all the servers (including itself). Then, if a server receives f+1 of *secondComplain* messages for the value, the server, serving as a follower, starts to count down its commit-sensitive timer, $\mathcal{T}_{\mathsf{F}_i}$; when $\mathcal{T}_{\mathsf{F}_i}$ expires, the server starts a leader election campaign.

The complaint-based scheme assists the system in detecting a faulty leader. A faulty leader can deliberately defer or stop responding to the requests sent from clients. With this scheme, a new leader election is triggered by f+1 of *secondComplain* messages, which means at least one correct server (say S_i) has forwarded the value to the leader. If the leader is correct, the consensus for committing the value should proceed despite the connection failure of clients, and S_i resets its timer ($\mathcal{T}_{\mathsf{F}_i}$) after the value is committed. However, if the leader is faulty, the correct server initiates a new election after its timer expires, and the system rotates the leadership away from the faulty servers.

In addition, faulty clients are not able to inflict unnecessary election campaigns so that the thresholds of correct servers will not be increased. A server starts its timer after receiving f+1 *secondComplain* messages of a proposed value. A correct client should always be able to connect to $2f$+1 servers so that f faulty servers can initiate only f fake *secondComplain* messages, which cannot trigger a new election among correct servers. With a correct leader, even if f faulty servers collude with faulty clients, there is always one correct server that forwards the value to the leader so that the leader can conduct the consensus and all the other correct servers stay in the follower state. Therefore, a correct leader cannot be replaced through the complaint-based scheme. Though faulty servers can initiate new elections to launch takeover attacks, they will be penalized by PoC_{LE} and rapidly marginalized in leader election (see Section 4).

3.5 Secure Log Replication

Secure log replication (SLR) is designed to safely replicate the proposed values; that is, no faulty value can be committed even if the leader is faulty. To tolerate Byzantine failures, followers need to compare what values the other servers have prepared to commit. SLR takes advantage of the leadership in Prosecutor by using the leader as a bridge of communication among servers. In general, SLR consists of five phases, with each phase involving a message transmission complexity of $\Theta(n)$. Figure 4 shows the workflow under normal operation starting with the above client interaction.

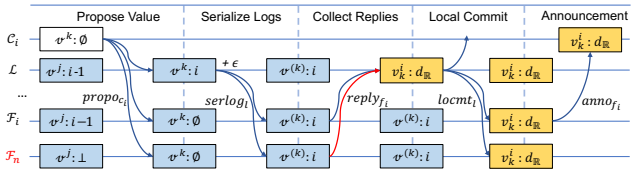


Figure 4: Normal operation of SLR in Prosecutor.

Serialize logs phase. After receiving the *propo* message, the leader waits for time ϵ ($\epsilon \gg$ *network latency*) and broadcasts message $serlog=\sigma_l\langle d(propo), i, t, s, id\rangle$, in which i is the index assigned to the value, t is the term, s indicates the serialize logs operation, and id is the leader's Id. If the *propo* message is forwarded by a *secondComplain* message from a server, *serlog* also piggybacks the message so that servers can obtain and verify the value according to its digest. In addition, Prosecutor is able to handle disordered requests through SLR since the design of the proof window supports parallel proposed values. ϵ is added for practical considerations, which lets servers receive messages from the clients before receiving instructions from the leader; a similar concept is introduced in Tendermint [11] for validators to wait for messages from proposers.

Collect Replies phase. After verifying the signature, followers respond to the leader if the received *serlog* messages satisfy the following three requirements simultaneously: 1) t is no less than the follower's term; 2) $d(propo)$ is the digest of the *propo* message; and 3) index i is available. Then, followers reply to the leader by issuing $reply=\sigma_{f_i}\langle d(serlog), i, t, r, id\rangle$, where i is the received index, t is the follower's term, and r indicates the "collect replies" operation.

Local Commit phase. The leader cumulatively collects replies into a collection set R until $|\mathsf{R}|=2f+1$, which reaches the threshold of $2f$ received replies. Thereafter, the leader combines R into a threshold signature and prepares a local commit message $locmt=\sigma_l\langle d(\mathsf{R}), i, t, l, id\rangle$, signing the digest of R, index (i), term (t), and identifier (l) of the local commit operation. Then, the leader broadcasts the *locmt* messages to all the other servers. In the meantime, the leader sends an announcement message $anno=\sigma_l\langle d(locmt), i, t, a, ts, id\rangle$ to the requesting client (parameters are explained below).

Announcement phase. After receiving the *locmt* message, followers examine the collection set to ensure that the signature has a threshold of $2f$+1. Next, followers commit the value proposed in *propo* and reset their timer if the timer was enabled by f+1 received *secondComplain* messages for the proposal. Then, followers send a confirm announcement message $anno=\sigma_{f_i}\langle d(locmt), i, t, a, ts, id\rangle$ to the requesting client, where i is the assigned index for the committed value, a indicates the announcement operation, and ts is the received timestamp for the value. Finally, the client considers the proposed value as committed when $2f$+1 announcement messages are received.

3.6 Correctness Argument

We now present some correctness properties held by Prosecutor in terms of PoC$_{LE}$ and SLR. First, we show that SLR operations are safe with or without a correct leader.

Lemma 1. *Faulty leaders cannot mutate proposed values.*

Proof. We prove by contradictions. Say the value sent from a client is v, and the faulty leader intends to conduct the consensus for committing value v'. If v' is committed, we claim that $2f$+1 servers have sent announcements for v' to the client. The claim requires servers to receive a *locmt* message containing $2f$+1 identical digests of *reply* messages in the collection set, denoted by $\mathsf{R}_{v'}$. However, servers hash the *propo* message received from the client and validate the hash with $d(propo)$ received from the leader. Then, servers send *reply* messages for v, and the leader constructs a collection set R_v, where $|\mathsf{R}_v|$=$2f$+1. Since messages are signed by the senders, the leader cannot modify servers' replies in R_v. If the leader colludes with f faulty servers and constructs another set $\mathsf{R}_{v'}$, the maximum number of messages in $\mathsf{R}_{v'}$ is $2f$, in which f of them are from the faulty servers and another f of them are from the remaining $n-|\mathsf{R}_v|$ servers. Therefore, no correct servers will commit the value v' since $\mathsf{R}_{v'}$ does not contain enough *reply* messages for v', which contradicts the claim.

With Lemma 1, another way for a faulty leader to attack the system is to stop responding. Since only the leader arranges the assignment of proposed values, the system will experience an out-of-service (OTS) condition. Nonetheless, the impact of this stop-responding attack is temporary. If the leader stops issuing *serlog* messages, the followers that receive complain messages from the client broadcast *secondComplain* messages and a new leader will be elected through PoC$_{LE}$. However, if the leader stops handling *locmt* messages to hinder the consensus process, the timers of followers cannot be reset and trigger timeouts when they expire; consequently, new elections will be campaigned for replacing the current leader. Therefore, a faulty leader cannot manipulate the proposed value or permanently occupy the leadership by faulty behavior.

Theorem 1. *(fault tolerance) To tolerate f Byzantine failures, Prosecutor requires at least n=$3f$+1 servers in total.*

Proof. The leader can be faulty or correct. No consensus can be reached with a faulty leader, and as discussed above, a faulty leader will be rotated out of the leader position by PoC$_{LE}$, so we discuss scenarios in which the system has a correct leader. We assume that f Byzantine faulty servers exist in the system. In the worst case, faulty servers behave correctly in the collect replies phase and undermine the process in the following local commit phase. In particular, the collection set R must collect $2f$+1 replies, and f of them are from Byzantine servers; in the next phase, f Byzantine servers start to behave incorrectly; i.e., the Byzantine server will not obey the protocol. We prove that Prosecutor outputs correct results, delivering sufficient announcements, tolerating f Byzantine servers with $3f$+1 servers in total.

We denote a validation set as $\mathbb{S}$ (initially $\mathbb{S}\leftarrow\varnothing$), a set of f Byzantine faulty servers as $\mathcal{N}_f$, and a set of correct servers as $\mathcal{N}_c$. In the worst case, the collection set R contains $2f$+1 replies from both $\mathcal{N}_f$ and $\mathcal{N}_c$; therefore, $\mathbb{S}\leftarrow\{\mathcal{N}_f, \mathcal{N}_c\}$, where $|\mathcal{N}_f|$=f and $|\mathcal{N}_c|$=f+1. In the next phase, correct servers in $\mathcal{N}_c$ send announcements to clients indicating the committed value. In the meantime, Byzantine servers could stop responding to others; consequently, the proposing client receives only f+1 announcements from $\mathcal{N}_c$, which is insufficient because a client requires at least $2f$+1 announcements to confirm that the proposed value has been committed. Thus, $\mathbb{S}$ requires a correct set $|\mathcal{N}_{c'}|$ with f correct servers to have a quorum for consensus. In total, $\mathbb{S}\leftarrow\mathbb{S}\bigcup\mathcal{N}_{c'}$ and $|\mathbb{S}|$=$3f$+1. Therefore, Prosecutor requires $3f$+1 servers in total to tolerate f Byzantine failures.

Given Lemma 1 and Theorem 1, we now argue for the safety property of Prosecutor.

Safety Argument. *If a value is committed at index i by SLR, the value on correct servers will never be overwritten at index i through the consensus processes led by leaders elected by PoC$_{LE}$ in all higher-numbered terms.*

Proof. (Sketch) Assume a value v is committed with an index i by SLR. Then, v is locally committed at index i by $2f$+1 servers, denoted by two sets $\mathcal{N}_a$ and $\mathcal{N}_b$, where $|\mathcal{N}_a|$=f+1 and $|\mathcal{N}_b|$=f, and the f remaining servers are denoted by a set $\mathcal{N}_c$. In addition, if v was locally committed by $2f$+1 servers, v had been logged by $2f$+1 servers at index i in the previous phase, where v can be presented in the log of the servers from set $\mathcal{N}_a\bigcup\mathcal{N}_b$ or $\mathcal{N}_a\bigcup\mathcal{N}_c$.

If the current leader is correct, according to the replication scheme in SLR, the correct servers maintain the same commit status, and their PWs are identical. In this case, the safety argument holds: 1) if the next leader is correct, it proceeds consensus by assigning further proposed values starting from index i+1; 2) if the next leader is faulty, although it can assign i to a new value, correct servers refuse to reuse i in the serialize logs phase since it was occupied by v.

However, if the current leader is faulty, the situation is more complicated. The faulty leader can collude with faulty followers and selectively send messages to correct servers. Suppose $\mathcal{N}_c$ is the set of faulty servers, and there are four possible scenarios. First, $\mathcal{N}_a\bigcup\mathcal{N}_b$ logged and committed v at index i. Similar to the above discussion, the safety argument holds. Second, $\mathcal{N}_a\bigcup\mathcal{N}_b$ logged but $\mathcal{N}_a\bigcup\mathcal{N}_c$ committed v at index i. Conversely, the third scenario is where $\mathcal{N}_a\bigcup\mathcal{N}_c$ logged but $\mathcal{N}_a\bigcup\mathcal{N}_b$ committed v at index i. In both cases, if the next leader reuses index i, $\mathcal{N}_a$ and $\mathcal{N}_b$ refuses to follow the reuse, as index i was occupied in log or commit slots. The fourth scenario is where $\mathcal{N}_a\bigcup\mathcal{N}_c$ logged and committed v at index i. If the next leader reuses index i, $\mathcal{N}_b$ will accept because $\mathcal{N}_b$ is not aware that index i has been used. However, the consensus needed for reusing index i cannot be obtained because $\mathcal{N}_a$ will reject it, so $|\mathcal{N}_b\bigcup\mathcal{N}_c|$=$2f$, and a quorum cannot be formed. Thus, the safety argument also holds.

Theorem 2. *(linearity) Prosecutor has a linear message transmission complexity under normal operation.*

Proof. During normal operation, to commit a proposed value, SLR uses five phases, with each phase involving a message transmission cost of $\Theta(n)$. There is no direct communication among followers in normal operation since messages flow only between

the leader and other servers. Each worker transmits an $O(1)$ message to the leader, and the leader transmits an $O(1)$ message to $n-1$ of the other servers. Phases do not overlap for the completion of one consensus instance. Consequently, by summing up the transmission cost in each phase, the total message transmission complexity is $O(n)$ for committing a proposed value.

Theorem 3. *Proof-of-Commit leader election involves a message transmission cost of* $\mathcal{O}(n^2)$.

Proof. To initiate a PoC$_{LE}$ request, a server needs to receive $f+1$ messages of *secondComplain* to start its timer. In the best case, only $f+1$ complaint messages are sent to all the other servers $(n-f)$, and in the worst case, $2f+1$ complaint messages are sent since $2f+1$ servers are connected to a correct client. Since $f=\lfloor \frac{n}{3} \rfloor + 1$, on average, PoC$_{LE}$ engages a message transmission cost of $\mathcal{O}(n^2)$.

4 SUPPRESSING BYZANTINE SERVERS

In this section, we specify the effect of the penalization mechanism in PoC$_{LE}$. First, we analyze how it becomes costly for Byzantine servers to launch attacks to usurp leadership. Then, we introduce the process of Byzantine servers vanishing from the competition for leadership with rapidly increasing computations. Finally, we discuss some potential limitations that may reduce the efficiency of the protocol.

While a faulty leader is active, no consensus can be processed since only the leader arranges the proposed values. Thus, the system experiences an out-of-service condition, reducing the system availability. This problem is common for all leader-based BFT consensus algorithms such as PBFT [15], BFT-SMaRt [9], SBFT [24], HotStuff [44], and LibraBFT [35], whereas the dynamic-penalization election scheme of PoC$_{LE}$ mitigates this problem by reducing the out-of-service time.

4.1 Vanishing Byzantine Proposers

Prosecutor limits the ability of servers to initiate new elections by imposing hash computation on leader election proposers. Under normal operation, correct servers initiate leader election campaigns when their timer expires, whereas Byzantine servers can usurp the leadership from a correctly operating leader and perform takeover attacks by proposing new election campaigns regardless of timer expiration. To mitigate the impact, PoC$_{LE}$ dynamically increases and keeps track of the servers' thresholds based on the frequency of corresponding initiated election campaigns.

Lemma 2. *The hash computation incurred by PoC$_{LE}$ grows exponentially for continuously initiated election campaigns.*

Proof. Assume that the hashing space is $\mathcal{S}$, the length of hash results is L, and the threshold for granting a vote for server n_i is η. The probability $P(T_\eta)$ that n_i finds a qualified nonce is as follows:

$$P(T_\eta) = \mathcal{S}^{L-\eta} \cdot \mathcal{S}^{-L} = \mathcal{S}^{-\eta}.$$

If n_i has continuously sent α election requests, its threshold increases to $\eta + \alpha$, where $\alpha \in \mathbb{N}_{>0}$; thus, the probability decreases.

$$P(T_{\eta+\alpha}) = \mathcal{S}^{L-(\eta+\alpha)} \cdot \mathcal{S}^{-L} = \mathcal{S}^{-\eta-\alpha}.$$

Therefore, after initiating α new elections, a Byzantine server is required to perform $\frac{P(T_\eta)}{P(T_{\eta+\alpha})} = \mathcal{S}^{\alpha}$ times the number of computations required for a correct server.

In addition, with the exponentially increasing hash computations required by PoC$_{LE}$, each election request ramps up the time cost for obtaining the next qualified nonce. Although Byzantine faulty servers may collude to deceive, there is a time, denoted by TC_b, when a cohort of f Byzantine faulty servers that are involved in performing the increasingly taxing computation can no longer defeat correct servers in further election campaigns.

Theorem 4. *After TC_b, Byzantine servers vanish from leadership competition while a correct leader is guaranteed.*

Proof. Correct servers always vote for legitimate leader election requests sent from a candidate that is up-to-date and has obtained the required hash result. By performing the computation required by a higher threshold, faulty servers can usurp leadership from correct servers, which leads to a suspended out-of-service condition. However, the suspension is temporary because a new election campaign will always take place after a correct server's timer expires. Because SLR ensures values cannot be manipulated, a faulty leader, from the Byzantine cohort, will be replaced by a correct server that collects $f+1$ votes in the next election campaign. However, the cost for launching takeover attacks rapidly becomes unaffordable, and the out-of-service time of the system narrows while faulty servers are penalized for each attack. In particular, if a server behaves maliciously, the increase in its threshold results in a positive feedback loop in which every new election campaign requires exponentially increased computational time than its previous election; ultimately, the server reaches its upper computational limit and cannot find a corresponding nonce in a timely manner. After TC_b, the computation required to obtain a nonce surpasses the computation capability of the cohort of the Byzantine servers.

In addition, the periodic `Inquire-Threshold` messages avoid unnecessary hash computations if the network partition takes place; after recovering, servers resume election campaigns with the threshold adjusted only by their past behavior. In the worst case, in which f Byzantine servers collude as a cohort, faulty servers are marginalized and unable to participate in PoC$_{LE}$. Therefore, after TC_b, Byzantine servers vanish from the leadership competition and the system elects correct leaders.

In addition to the increasing computations performed for each election request, the changes of proof windows can also accelerate the process in which Byzantine servers vanish from the leadership competition. When proof windows change faster than the computational abilities to find a qualified nonce, an election campaign cannot be completed; that is, the nonce expires when the proof window moves to include newly committed values. If new values are committed, the proof window changes; then, a Byzantine server needs to abandon its current computational process and to admit a new proof window that includes the newly committed values. Specifically, in a high-throughput system with a correct leader, Byzantine leaders are coerced into repeatedly hashing proof windows in PoC$_{LE}$ if they start an infinite number of new election campaigns to usurp the leadership.

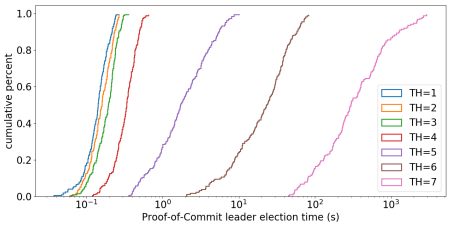


Figure 5: The cumulative percent of leader election time.

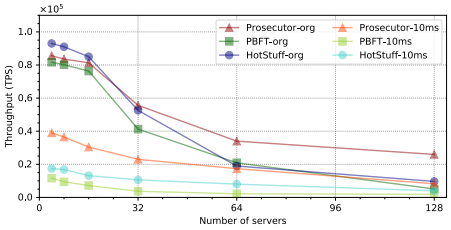


Figure 6: Throughput comparisons with different network latency (*batchSize*=500).

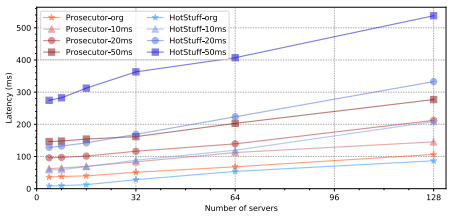


Figure 7: Latency comparisons for committing transactions with various network latencies.

4.2 Potential Limitations

In PoC$_{LE}$, the requirement for hash computation suppresses Byzantine servers from being elected as a leader, while the scheme may also have some negative effects on correct servers. Theoretically, correct servers may compete against each other if they initiate election campaigns simultaneously. Although Prosecutor intends to alleviate this negative effect by using randomized timers and engaging hash computation before initiating election campaigns, the probability of competition exists in theory. When such scenarios materialize, the candidate that wins the election is not affected if it behaves normally since its threshold will be decreased to the same level before the next election campaign, whereas the candidate that loses the election and goes back to the follower state is affected since it is considered as suspected faulty behavior and the decrements of its threshold will be deferred to future election campaigns. To avoid the competition among correct servers, the amount of randomness of timeouts should be configured considering the network latency and cluster size. For example, with 5 ms network latency, timeouts should range from 500 ms to 1000 ms in an 8-server cluster and from 500 ms to 1500 ms in a 16-server cluster. Additionally, the completion time for obtaining a qualified hash result varies significantly even under the same threshold (see Figure 5). Therefore, in practice, with appropriately randomized timers, the probability of candidates competing in a given election campaign is extremely low and can be neglected.

Another potential limitation of Prosecutor, which also exists in PoW [38], is that if Byzantine servers have much stronger computational abilities than correct servers, the system may spend more time struggling in elections because it takes longer for Byzantine servers to exceed their abilities to obtain the required hash result.

5 IMPLEMENTATION AND EVALUATION

We implemented the Prosecutor prototype using the Golang programming language and deployed the prototype on 4, 8, 16, 32, 64 and 128 VM instances on Compute Canada Cloud. Instances are located in the same data center, and the original network latency between two VMs is less than 2 ms; in the evaluation, we used `NetEm` [4] to implement higher network latencies. Each instance includes a machine with two virtual 2.40-GHz Intel Core processor (Skylake) CPUs with a cache size of 16 MB, 8 GB of RAM, and 20 GB of disk space running on Ubuntu 18.04.1 LTS. We measured the network bandwidth between servers using `iperf`, and the bandwidth is approximately 400 Megabytes/second. We used the original implementation of HotStuff [3] with the default pacemaker `rr` (round robin) and implemented PBFT [15] using Golang. Since issues of configuring HotStuff have been raised by the community [3], we also posted a docker container from which users can run HotStuff directly with all the required external tools, libraries, and dependencies [2].

In this evaluation, single and cohorts of Byzantine servers apply the following strategies to launch attacks: a Byzantine server launches takeover attacks whenever it is not the leader, and a cohort of Byzantine servers launch takeover attacks whenever the leader is not from the cohort. Cohorts of Byzantine servers collaborate to perform required computations faced by the one that has the lowest threshold. After being elected, Byzantine leaders send erroneous messages to all of the other servers to impede log replication.

5.1 Leader Election

The experiment aiming to measure leader election time was conducted in a 16-server cluster, with up to 5 of them being faulty. We forced one server to be an election requester and measured the election time as the period that begins when the timer expires and ends when the server is elected. The network latency among servers varied from 15 to 100 ms, and results are averaged over 200 independent runs.

We evaluated the election time under increasing thresholds (1 to 7) as shown in Figure 5. The cumulative percentage curves for thresholds less than 4 have similar shapes, and their corresponding election times are all below 500 ms. When the $threshold$Ÿ4, compared with soliciting votes, the hash computation accounts for a small percentage of the election time. In contrast, as the threshold rises above 5, election time begins to diverge. When $threshold$=5, the election times of approximately 80% of the 200 runs exceed 1 second. Furthermore, the hash computation becomes extremely time-consuming when $threshold$ ¡ 7, and it takes more than 10^4 seconds to finish one election campaign. Consequently, if Byzantine servers continuously launch takeover attacks, winning an election becomes unrealistic due to the exponentially growing computation.

5.2 Throughput and Latency Comparisons

We compared the throughput of Prosecutor with PBFT and HotStuff. The latency is measured on clients, and throughput is measured on servers. We evaluated the throughput using different batch sizes and chose a batch size of 500 transactions to show the comparisons in which all of the three protocols approximately reach their peak throughput. We applied a dataset consisting of historical Bitcoin transactions [1] as the transactions to be replicated in the three

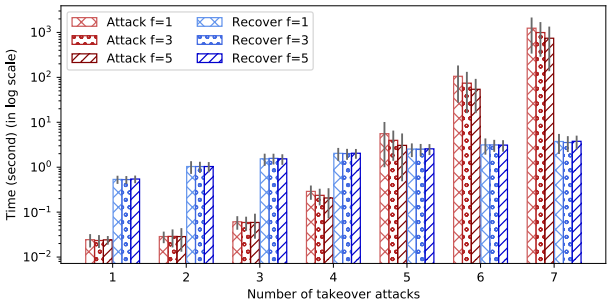


Figure 8: Time cost (in log scale) measured in terms of takeover attacks to recovery time under increasing numbers of attacks.

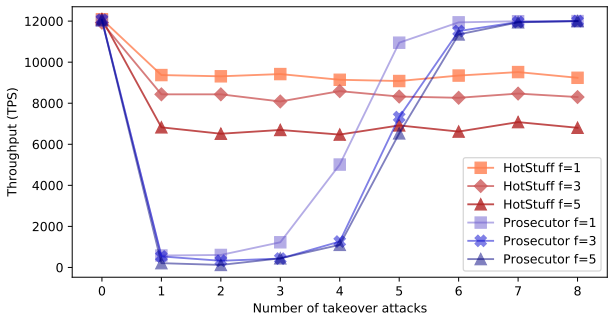


Figure 9: Throughput degradation comparisons under increasing failures with various Byzantine servers.

protocols. The evaluation is measured with emulated network latencies. In Figure 6 and 7, label `org` indicates that the network latency is original (less than 2 ms), and `10ms` represents a 10 ms additional emulated network latency in our experiment.

Figure 6 shows the throughput comparisons with two network latencies. HotStuff has a higher throughput without emulated latency when the system size is relatively small, but the throughput drops dramatically when the system scales up or has a higher network latency. PBFT also shows a quick throughput decline at large scales because of a quadratic growth of circulated messages. In contrast, Prosecutor stabilizes at one leader during the secure log replication. With a higher network latency, compared with HotStuff and PBFT, Prosecutor exhibits a higher throughput without suffering from many phases of communication. Additionally, Figure 7 shows the latency comparisons of Prosecutor and HotStuff with emulated network latencies of 10 ms, 20 ms, and 50 ms. As demonstrated in the figure, HotStuff is more sensitive to the change in network latency, whereas Prosecutor shows a relatively smaller increase in latency.

5.3 Recovery under Byzantine attacks

Since Prosecutor is able to recover from Byzantine attacks and penalize falsely-behaving servers accordingly, we evaluated the time costs of takeover attacks and corresponding recoveries as well as the throughput under increasing failures compared with HotStuff in a 16-server cluster, which can tolerate up to 5 Byzantine failures. Timeout periods are randomized between 500 ms and 1000 ms. To usurp correct leadership, Byzantine servers need to first obtain a qualified hash result that meets the threshold requirement and collect votes from $n-f$ servers. Since Byzantine leaders disseminate erroneous messages to impede consensus after being elected, correct servers initiate new election campaigns after their commit-sensitive timers expire; consequently, Byzantine leaders are replaced, and the system recovers. In our evaluation, Byzantine servers continuously launch a new attack when the system recovers from the previous attack, so the recovery time accumulates the time cost of all previous recoveries, including the time waiting for timeout.

Figure 8 shows that all the Byzantine cohorts (1, 3, and 5 servers) experienced an exponentially increased time spent on takeover attacks (time cost presented in log scale), whereas the recovery time grew linearly because they were waiting for more timeouts. When the Byzantine cohort contains more faulty servers, the recovery time is only extended by a constant factor. In particular, with 5 faulty servers, the time cost of the 7 th attack (743 s) is approximately 200x that of the corresponding recovery time (3.74 s).

In addition, we evaluated the throughput of Prosecutor and HotStuff under continuous attacks from various Byzantine servers. Both throughputs were measured in periods called *basePeriod*s, which begin at the start time of an attack and end at the start time of the next attack; the following equation defines the *basePeriod*.

$$basePeriod^{(n)} = Time^{(n+1)}_{attack\ starts} - Time^{(n)}_{attack\ starts}$$

Since attacks are continuous, in Prosecutor, the start time of the n+1 st attack is the same as the end time of the n th attack (i.e., $Time^{(n+1)}_{attack\ starts}$=$Time^{(n)}_{attack\ ends}$), which includes the "preparation time for launching attacks" (i.e., the time spent performing hash computation and soliciting votes). In addition, since HotStuff rotates leaders in a round-robin manner, with continuous attacks from cohorts of Byzantine servers, the *basePeriod* records the time between encountering faulty leaders. Thus, for both protocols, the throughput at the n th attack is the division of the number of committed transactions over *basePeriod*.

Figure 9 shows the throughput with 8 continuous attacks. We adjusted the sending rate from clients to stabilize both protocols at a throughput around 12 KTPS with a batch size of 100 before launching attacks. When failures materialize, the throughput of Prosecutor plummets initially and then recovers to the level without failures. This is because Prosecutor's penalization mechanism increases the threshold of faulty servers at each attack; soon afterward, servers in the Byzantine cohorts cannot afford the hash computation. The penalization mechanism pays for a throughput degradation before the system recovers to penalize faulty servers and avoids permanent throughput decline. In contrast, HotStuff experienced a sustained throughput downturn. Since HotStuff rotates leadership based on a quorum of votes, when experiencing failures, HotStuff can partially avoid faulty leaders. However, whenever faulty servers are assigned as leaders, HotStuff has to wait for a timeout period to proceed to the next leader. With 5 faulty servers, the throughput of HotStuff substantially decreases around 45% regardless of the number of launched attacks, whereas Prosecutor heals from the throughput decrease through the penalization mechanism, gaining higher throughput in the long run.

Protocols	Transmission complexity	Rounds for n transactions	Leader failure	Timer-based view change	Behavior penalty
PBFT [15], BFT-SMaRt [9]	$\mathcal{O}(n^2)$	$5n$	$\mathcal{O}(n^3)$	X	
HoneyBadgerBFT [36]	$\mathcal{O}(n^3)$	$6n$	NA†		
Tendermint [11], Casper [12]	$\mathcal{O}(n^2)$	$5n$	$\mathcal{O}(n^2)$		△
SBFT [24]	$\mathcal{O}(n)$	$5n$	$\mathcal{O}(n^2)$	X	
HotStuff [44]	$\mathcal{O}(n)$	$8n$	$\mathcal{O}(n)$	X	
Prosecutor	$\mathcal{O}(n)$	$5n$	$\mathcal{O}(n^2)$	X	X

† HoneyBadgerBFT is a leaderless protocol. △ Tendermint leaves behavior punishment for future work.

Table 1: Qualitative comparisons of selected state-of-the-art protocols.

6 RELATED WORK

The consensus problem involves achieving agreement on a given value among a group of n processes in distributed systems. Paxos [31, 32] and Raft [39] achieve consensus tolerating benign failures (i.e., non-Byzantine failures) and efficiently replicate log entries by using a designated propose/leader. Under arbitrary failures [33], PBFT [15] provides a practical solution to tolerate f Byzantine failures by having $3f+1$ servers with an $\mathcal{O}(n^2)$ message transmission cost. Optimizations for more efficient BFT consensus are being developed, and Table 1 lists a qualitative comparison of selected protocols. BFT-SMaRt [9] implements a fine-tuned state machine for PBFT [15] to improve performance; Tendermint [11] and 700 BFT [23] suggest a primary-backup messaging pattern to avoid n-to-n broadcast; Zyzzyva [29] uses speculative execution and enables immediate response for leaders by pushing more duties to clients to detect inconsistent server logs; SBFT [24] and HotStuff [44] leverage threshold signatures to achieve linearity; and Pompe [45] introduces Byzantine-ordered consensus with a new BFT state machine replication architecture.

Leader-based BFT consensus protocols are efficient under normal operation; however, they are vulnerable under Byzantine failures since leadership can be assumed by faulty servers that deliberately degrade system performance. Some protocols aim to improve the robustness of BFT consensus protocols. For example, Prime [5] requires every replica to periodically broadcast messages sent by clients from the other replicas. If a leader becomes slower than an expected pace, the leader is replaced by new view changes. Furthermore, Aardvark [16] imposes a lower bound for a legitimate leader's performance: the throughput of the system under current leadership is supposed to be at least 90% of the maximum throughput achieved by the leaders of the last n (the number of total servers) views; otherwise, view changes occur and the system moves to the next leader. Similarly, Spin [42] uses a list to confine up to f servers that could not achieve consensus within a predefined period when they were leaders; RBFT [7] achieves consensus in parallel on redundant replicas to address server and client failures. Tendermint [11] is greatly inspired by Raft for constructing consensus and discusses the philosophy of punishing faulty behavior by using hybrid Proof-of-Stake [27] but leaves its design to future work. Compared with these optimizations, Prosecutor penalizes faulty servers and obstructs them from being elected as future leaders, alleviating the burden from Byzantine servers over time; Prosecutor can also terminate a faulty leader early by client interactions and replicate log entries without requiring more redundancy. In addition, the penalization in Prosecutor imposes computational costs on faulty servers, which differs from the previous approaches in which suspected faulty servers can be freely released after pretending to be correct servers.

Moreover, due to the popularity of cryptocurrencies, Proof-of-X-based protocols, in the permissionless category, have been successful, such as Proof-of-Work [38], Proof-of-Stake with hybrid mining and stake in PPCoin [27] and pure stake without mining for consensus in Algorand [22], Proof-of-Activity [8] and Proof-of-Luck [37]. Some approaches, such as PeerCensus [17] and Bitcoin-NG [20], suggest the election of a coordinator to increase efficiency, which embraces concepts in BFT algorithms as optimization. Additionally, Hybrid consensus proposes a joint classical- and blockchain-style consensus [40], and ByzCoin achieves Byzantine consensus with membership management [28]. Nonetheless, Prosecutor brings the mining concept (PoW) into permissioned blockchains for penalizing suspicious behavior, looking at BFT protocols in a new light by diagnosing Byzantine servers during consensus and then imposing costly hash computations to consequently suppress Byzantine servers.

7 CONCLUSIONS

In this paper, we introduced a BFT protocol called Prosecutor with Proof-of-Commit leader election and secure log replication. The leader election mandates hash computations for initiating election campaigns and suppresses Byzantine servers from launching takeover attacks. Furthermore, Prosecutor provides an efficient log replication process utilizing the elected leader and message authentications. The evaluation compares the throughput and latency with PBFT and Hotstuff. The experimental results show that Prosecutor penalizes faulty servers and heals from throughput decline under malicious attacks.

REFERENCES

[1] Bitcoin historical data from select exchanges, Jan 2012 to Dec 2020. https://www.kaggle.com/mczielinski/bitcoin-historical-data. Accessed: 2020.

[2] Hotstuff docker image with all external tools, libraries, and dependencies. Docker Hub: docker pull dopamineedward/hotterstuff:v1.

[3] Libhotstuff: A general-purpose bft state machine replication library with modularity and simplicity. https://github.com/hot-stuff/libhotstuff. Accessed: 2019.

[4] NetEm. https://www.linux.org/docs/man8/tc-netem.html.

[5] Amir, Y., Coan, B., Kirsch, J., and Lane, J. Byzantine replication under attack. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)* (2008), IEEE, pp. 197–206.

[6] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al.

Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (2018), ACM, p. 30.

[7] Aublin, P.-L., Mokhtar, S. B., and Quéma, V. Rbft: Redundant byzantine fault tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems* (2013), IEEE, pp. 297–306.

[8] Bentov, I., Lee, C., Mizrahi, A., and Rosenfeld, M. Proof of activity: Extending bitcoin's proof of work via proof of stake. *IACR Cryptology ePrint Archive 2014* (2014), 452.

[9] Bessani, A., Sousa, J., and Alchieri, E. E. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2014), IEEE, pp. 355–362.

[10] Brown, R. G., Carlyle, J., Grigg, I., and Hearn, M. Corda: an introduction. *R3 CEV, August* (2016).

[11] Buchman, E. *Tendermint: Byzantine fault tolerance in the age of blockchains.* PhD thesis, 2016.

[12] Buterin, V., and Griffith, V. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437* (2017).

[13] Cachin, C. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers* (2016), vol. 310.

[14] Cachin, C., and Vukolić, M. Blockchain consensus protocols in the wild. *arXiv preprint arXiv:1707.01873* (2017).

[15] Castro, M., Liskov, B., et al. Practical byzantine fault tolerance. In *OSDI* (1999), vol. 99, pp. 173–186.

[16] Clement, A., Wong, E. L., Alvisi, L., Dahlin, M., and Marchetti, M. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI* (2009), vol. 9, pp. 153–168.

[17] Decker, C., Seidel, J., and Wattenhofer, R. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking* (2016), ACM, p. 13.

[18] Dinh, T. T. A., Wang, J., Chen, G., Liu, R., Ooi, B. C., and Tan, K.-L. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 1085–1100.

[19] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM) 35*, 2 (1988), 288–323.

[20] Eyal, I., Gencer, A. E., Sirer, E. G., and Van Renesse, R. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), pp. 45–59.

[21] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM) 32*, 2 (1985), 374–382.

[22] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 51–68.

[23] Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems* (2010), pp. 363–376.

[24] Gueta, G. G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M., Seredinschi, D.-A., Tamir, O., and Tomescu, A. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)* (2019), IEEE, pp. 568–580.

[25] Howard, H. Arc: analysis of raft consensus. Tech. rep., University of Cambridge, Computer Laboratory, 2014.

[26] Howard, H., Schwarzkopf, M., Madhavapeddy, A., and Crowcroft, J. Raft refloated: Do we have consensus? *Operating Systems Review 49*, 1 (2015), 12–21.

[27] King, S., and Nadal, S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August 19* (2012).

[28] Kogias, E. K., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., and Ford, B. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th usenix security symposium (usenix security 16)* (2016), pp. 279–296.

[29] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 45–58.

[30] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[31] Lamport, L., et al. The part-time parliament. *ACM Transactions on Computer systems 16*, 2 (1998), 133–169.

[32] Lamport, L., et al. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

[33] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4*, 3 (1982), 382–401.

[34] Libert, B., Joye, M., and Yung, M. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Theoretical Computer Science 645* (2016), 1–24.

[35] Libra. An introduction to libra, 2019. https://libra.org/en-US/white-paper/.

[36] Miller, A., Xia, Y., Croman, K., Shi, E., and Song, D. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 31–42.

[37] Milutinovic, M., He, W., Wu, H., and Kanwal, M. Proof of luck: an efficient blockchain consensus protocol. In *Proceedings of the 1st Workshop on System Software for Trusted Execution* (2016), ACM, p. 2.

[38] Nakamoto, S., et al. Bitcoin: A peer-to-peer electronic cash system.

[39] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIXATC 14)* (2014), pp. 305–319.

[40] Pass, R., and Shi, E. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *31st International Symposium on Distributed Computing (DISC 2017)* (Dagstuhl, Germany, 2017), A. W. Richa, Ed., vol. 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 39:1–39:16.

[41] Shoup, V. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques* (2000), Springer, pp. 207–220.

[42] Veronese, G. S., Correia, M., Bessani, A. N., and Lung, L. C. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems* (2009), IEEE, pp. 135–144.

[43] Vukolić, M. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International workshop on open problems in network security* (2015), Springer, pp. 112–125.

[44] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G., and Abraham, I. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019), pp. 347–356.

[45] Zhang, Y., Setty, S., Chen, Q., Zhou, L., and Alvisi, L. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 633–649.