MSc thesis in computer science

Liveness checking of Streamlined Blockchain Consensus

Yanzhuo Zhou
2023

TU Delft

LIVENESS CHECKING OF STREAMLINED BLOCKCHAIN CONSENSUS

A thesis submitted to the Delft University of Technology in partial fulfillment
of the requirements for the degree of

Master of Science in Computer Science

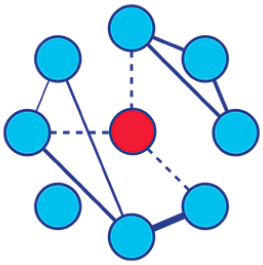
by

Yanzhuo Zhou

4993918

August 2023

The work in this thesis was made in the:



Distributed Systems (DS) group
Department of Computer Science
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Delft University of Technology

Thesis committee: Dr. Johan Pouwelse, Advisor
Dr. Jérémie Decouchant, Supervisor
Dr. Burcu Kulahcioglu Ozkan, Co-Supervisor
Dr. Kaitai Liang

ABSTRACT

Byzantine consensus protocols are designed to build resilient systems to achieve consensus under Byzantine settings, maintaining safety guarantees under any network synchrony model and providing liveness in partially or fully synchronous networks. However, several Byzantine consensus protocols have been shown to violate liveness properties under certain scenarios. Existing testing methods for checking the liveness of consensus protocols check for time-bounded liveness violations, which generate a large number of false positives.

In this thesis, for the first time, we check the liveness of Byzantine consensus protocols by the temperature and lasso detection methods and precisely detail ad-hoc system state abstractions that should be used to test these algorithms. We provide a theoretical analysis of the recently-published safety and liveness attacks and whether the existing test method is able to detect them. To investigate the issues, we focus on the streamlined blockchain consensus, particularly the HotStuff protocol family, which has been recently developed for blockchain consensus. Among these protocols, the HotStuff protocol itself is both safe and live under the partial synchrony assumption, whereas 2-Phase Hotstuff and Sync HotStuff protocols can violate liveness in subtle fault scenarios. We implemented our liveness checking methods on top of the Twins automated unit test generator to test the HotStuff protocol family and explored the scenarios with message delay. Our results indicate that our methods successfully detect all known liveness violations and produce fewer false positives than the bounded liveness checks.

Keywords. Liveness checking, Byzantine consensus, HotStuff protocols, Twins, Lasso detection, Temperature checking.

ACKNOWLEDGEMENTS

Thanks to my supervisors Jérémie Decouchant, Burcu Kulahcioglu Ozkan, and Johan Pouwelse. I couldn't finish this thesis without their help. Thanks to my previous supervisor Jan Rellermeyer and ING coordinator Elvan Kula to give me a chance to participate in the research team of ING. Thanks to my family and friends for supporting me through this particularly tough time in my life.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

1 INTRODUCTION

Fault tolerance is a crucial component required for providing reliable large-scale distributed services. Significant work has been done to ensure high-performance systems resilient to crash bugs and robust to be deployed. Nevertheless, distributed systems that guarantee high security against malicious Byzantine faults [40], continue to pose intriguing challenges to academia and enterprise-level applications.

Byzantine Fault Tolerance (BFT) is a paradigm that gives distributed systems the ability to tolerate a limited proportion of arbitrary faults (i.e., Byzantine faults) such as equivocation (i.e., sending conflicting messages to different nodes) and loss of internal state. Most BFT protocols utilize State Machine Replication (SMR) on multiple replicas, enabling them to tolerate up to a limited proportion of arbitrary faults. In particular, BFT consensus protocols aim at solving the consensus problem among n nodes that might include up to f faulty nodes. BFT consensus protocols aim at ensuring the safety and liveness properties. Safety ensures that the correct nodes always decide on the same value, while liveness ensures that the protocol always eventually progresses.

Practical Byzantine Fault Tolerant (PBFT) [17] has been long-termly regarded as a cornerstone of Byzantine protocols. It is a voting-based algorithm similar to Paxos [38]. PBFT has inspired numerous follow-up protocols aimed at improving performance and security [66]. Zyzzyva [37] achieves better performance and throughput by utilizing speculative execution. Tendermint [11] features a communication pattern similar to PBFT but employs gossip-based communication, reducing overhead. HotStuff [65] achieves a similar linear view change property that permits an honest leader to drive a decision within linear time complexity, attaining optimistic responsiveness, where any correct designated leader only needs the first quorum of responses to progress to the next proposal.

Guaranteeing the liveness of a BFT consensus protocol is a difficult and error-prone process. For example, 2-phase HotStuff is a consensus protocol that Yin et al. discuss for pedagogical purposes and that could initially be considered live [65]. However, a particular scenario is shown to prevent the system from making progress, as nodes alternatively vote on two conflicting blocks. A similar attack called the force-locking attack [46], breaks both the safety and the liveness of a preliminary version of the Sync HotStuff protocol by maliciously delaying messages. These examples of liveness violations call for effective testing methods that will assist researchers and developers in detecting and tracing them.

However, testing and verifying the correctness of a consensus protocol remains a complex and challenging task, particularly given the dynamic and intricately distributed nature of such systems. Previous testing works on consensus have mostly focused on analyzing crash-tolerant protocols. For example, Jepsen testing tool [43] simulates network partitions for distributed systems, and it has detected several consensus violations [33, 34, 31]. Twins [7] is one of the few testing systems that have been specifically designed to test the safety of BFT consensus protocols under Byzantine scenarios. Twins can detect safety violations using scenarios that involve only a few communication rounds. However, most existing testing systems do not check for liveness violations, which require the generation of infinite executions.

A common approach to finding liveness violations is to check for *bounded* liveness, i.e., checking whether the properties are satisfied within a bounded amount of time[32, 48]. To do so, the programmer sets some bounds for an event to happen and reports the executions that exceed the specified thresholds. In the case of

consensus protocols, correct processes should accept the same value within a certain delay or within a given number of execution steps. However, it is difficult for developers to correctly estimate adequate bound values, in particular in real-world production-level consensus applications. Low bound values lead to false negatives while using very large bound values incurs high running times.

Specific liveness testing methods have been proposed for distributed systems. *Temperature-based* detection algorithms [32, 47] maintain a temperature variable that is increased each time the system transitions to a hot state, whose definition is system-specific. The *lasso detection* approach [47] relies on state caching to identify whether a system reaches the same hot state multiple times and therefore discover potential liveness violations. Little work has been done to effectively apply these techniques to test liveness violations in the blockchain consensus.

In this thesis, we investigate Byzantine attacks in blockchain consensus protocols that have been found so far. Our purpose is to analyze whether these attacks can already be tested by existing techniques, and if not, how can we apply and extend existing techniques to detect these challenging attacks.

This thesis applies temperature and lasso detection methods to test the liveness of BFT consensus algorithms. We focus on the HotStuff family of protocols that have been designed for Blockchain consensus, which are sometimes called streamlined as they rely on a leader to reach a linear communication complexity and use a block-locking approach. We believe that our approach can be generalized to all protocols that rely on views, and in particular to all partially-synchronous protocols that rely on a leader.

1.1 RESEARCH QUESTIONS

The main research questions we explore and investigate in this thesis are:

RQ1 What are recently-detected BFT attacks? Can testing methods be used to detect some interesting and challenging attacks?

RQ2 How can we adapt existing liveness checking techniques for testing blockchain systems?

RQ3 Can the Twins testing system be extended to incorporate these liveness checking techniques? Can timing violations due to Byzantine faults be detected?

RQ1 investigates what attacks exist in the latest blockchain consensus protocols and which can not yet be tested. To answer **RQ2**, we are going to explore how can we adapt some liveness problem-checking methods for detecting existing violations, and investigate what kind of protocols can our methodology be applied to. **RQ3** technically explores if the Twins testing system can successfully use the techniques in **RQ2** to accomplish the goals in **RQ1**.

1.2 CONTRIBUTIONS

In this thesis, we contribute solutions to the above research questions. In summary, this work makes the following contributions:

- We present a summary of recently published BFT attacks in blockchain consensus protocols and analyze their types, causes, and feasibility of testing.
- We define the notion of *partial state* and *hot state* for the HotStuff family of protocols, which are required for the implementation of the temperature and lasso detection methods.

- Then, we present a variant of the lasso detection approach, which, differently from previous works, does not use a controlled scheduling environment to check for the existence of lassos.
- We also describe how to extend the Twins framework to support the temperature and lasso detection methods for the liveness testing of BFT consensus algorithms.
- We evaluate the performance of our testing methods on state-of-the-art protocols from the HotStuff protocol family and compare their accuracy to one of the classical bounded-liveness checking methods.

1.3 THESIS OUTLINE

This thesis is structured as follows:

Chapter 1 provides brief background knowledge of the topic, the aim of our research, and contributions that we have made. Chapter 2 introduces some basic concepts and the network models of BFT protocols. Chapter 3 reviews previous research work on BFT consensus that has been conducted so far, including details of some related protocols in this thesis and recently-published attacks. Chapter 4 overviews the temperature and lasso-detection based liveness checking methods. Chapter 5 provides the definition of hot states for streamlined protocols and describes our extensions to the liveness checking methods for these protocols. Chapter 6 evaluates our methodology's performance through experiments and analyzes the results. The final Chapter 7 concludes the thesis.

2 BFT CONCEPTS

2.1 OVERVIEW OF BYZANTINE FAULT TOLERANCE

2.1.1 Byzantine generals problem

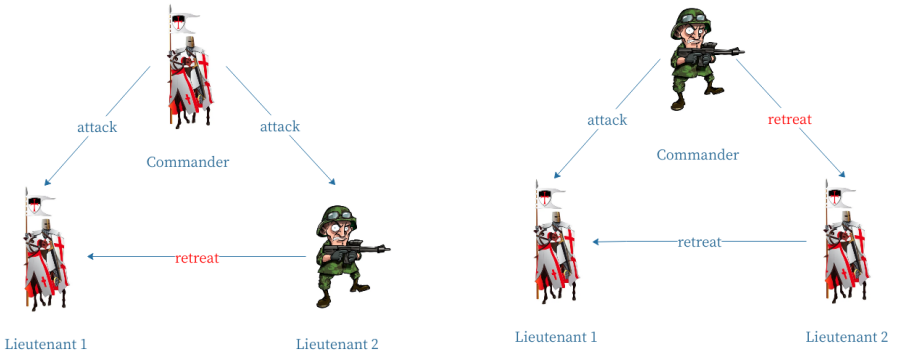
The Byzantine generals problem [40], first formalized by L. Lamport et al., highlights the challenges of coordinating a decision among generals who communicate through messengers. As is shown in Figure 2.1a and Figure 2.1b, traitors may send fake votes, and when a lieutenant is betrayed, it may mislead another lieutenant to retreat. Similarly, a betrayed commander equivocates messages to different lieutenants. The solution to this problem is to add more generals to at least $3f+1$, as illustrated in Figure 2.2a and 2.2b. When a lieutenant is malicious, other lieutenants can still receive 2 correct messages to make a decision. Similarly, when the commander has betrayed, the lieutenants cannot agree on a decision. They realize that the current commander might be a traitor and need to re-elect a reliable one.

In real-world distributed systems, a leader is a commanding general, and a replica is a lieutenant. Behaviors like message lying, deliberate message delay, and message faking are known as Byzantine behaviors. The goal of a BFT system is to help distributed systems maintain resiliency against malicious Byzantine general attacks.

2.1.2 The $3f + 1$ bound

The case of a common node abruptly crashing is called a non-Byzantine fault or Crash Fault Tolerance (CFT). Consider a scenario with two different requests. As f replicas may be faulty, the client requires at least $N - f$ replies as a Quorum Certificate (QC) to proceed. The intersection between the two quorums is $(2n - 2f) - n = n - 2f$. If there are only crash failures, we have to ensure that at least one intersected replica survives ($n - 2f > 0$). Hence, at least $2f + 1$ replicas are required against CFT.

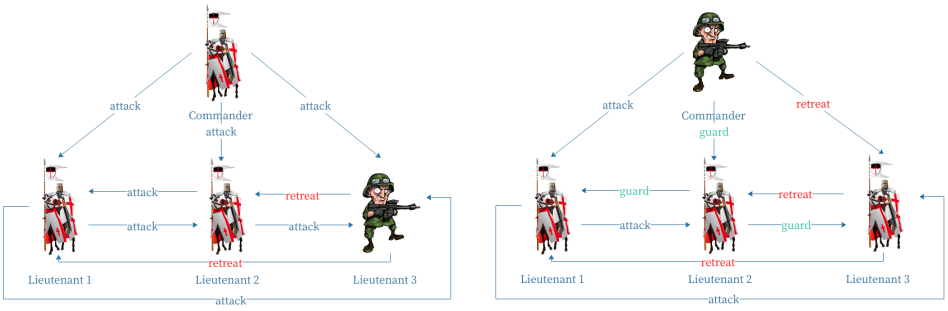
However, scenarios with Byzantine faults are complicated because it is hard for the system to determine whether faulty nodes crash or act maliciously. At most f replicas may be Byzantine. In such cases, we must exclude a quorum of honest



(a) Lieutenant 2 is a traitor.

(b) Commander is a traitor.

Figure 2.1: Scenarios with $2f+1$ generals (f=1).



(a) Lieutenant 3 is a traitor.

(b) Commander is a traitor.

Figure 2.2: Scenarios with $3f+1$ generals (f=1).

nodes and ensure that at least one honest replica survives against Byzantine nodes ($n - 2f > f$). Thus, at least $3f + 1$ replicas are needed for BFT systems.

Note that we do not consider the use of trusted components in this work, which would lower the necessary number of replicas to $2f + 1$ [22].

2.1.3 State machine replication

State machine replication (SMR) is a technique used in distributed computing to ensure that a group of nodes agree on a consistent state. Each node maintains a replica of the state machine and executes the same sequence of operations. The nodes communicate with each other to exchange information and ensure agreement on the current state. Byzantine SMR protocols typically consist of three components: consensus, checkpoint, and view change. The consensus module is based on a reliable total order broadcast protocol where requests are executed and committed by honest replicas in a determined order. The checkpoint is used to verify log information performed by replicas. The view change mechanism is triggered when faults are detected to elect a new leader. We assume a static membership and consider dynamic adaptation to be out of the scope of this work [55].

2.1.4 Properties

In order to check the correctness of BFT systems, distributed BFT protocols can be characterized into these properties:

1. *Safety*: A safety property guarantees that the correct nodes always decide on the same value.

2. *Liveness*: A liveness property guarantees that the protocol always eventually progresses.

3. *Validity*: The decided value is a valid proposal.

The validity condition can be referred from the notation proposed by Cachin et.al [13]. This property ensures the validity of the blocks and the decided value. Specifically, it guarantees that a blockchain implementation can accept a block signed by valid replicas and decide on a block containing legitimate transactions.

2.2 NETWORK MODEL

2.2.1 Synchronous system

A synchronous consensus system requires a group of nodes must agree on a value or decision within a bounded time interval, denoted as Δ. This assumes that message delivery time and node processing times are known and bounded and that the network is reliable and predictable. However, it is challenging to ensure network synchrony in real-world distributed systems due to network partitions and other practical constraints. It may not be a practical solution for high-performance applications because a large number of lock-step rounds between nodes leads to high latency.

2.2.2 Asynchronous system

An asynchronous system allows for arbitrary manipulation of message delivery timing, which enables adversaries to arbitrarily delay their messages as long as they guarantee eventual delivery. In practice, consensus verification is challenging in large asynchronous systems, especially in convoluted network environments with intertwined message delivery, loss, and delay. The FLP impossibility result [28] proves that deterministic asynchronous consensus cannot achieve liveness, even with a single crash failure. This restriction can be mitigated by extended features such as timing manipulation, randomization, and other approaches in practice.

2.2.3 Partially synchronous system

Partially synchronous systems, as defined in [26], strike a balance between synchronous and asynchronous systems by assuming an unknown Global Stabilization Time (GST) and a finite upper bound Δ on message delivery times. The adversary can arbitrarily delay, drop, or reorder messages until the GST, but once the GST is reached, the network becomes fully synchronous, and message delivery is guaranteed. The system can tolerate long delays, such as those due to network congestion or overload, as any message sent at time t has to be delivered by time $\Delta + max(t, GST)$.

Partially synchronous consensus protocols can achieve high performance during periods of full synchrony while still maintaining a high degree of fault tolerance during periods of partial synchrony. This allows them to scale to large distributed systems while still providing the safety and liveness guarantees necessary for consensus.

2.2.4 P2P communication

Peer-to-peer (P2P) networks are a communication model in that data are stored and shared collectively by peer nodes, enabling all clients to provide resources. The distributed nature of P2P networks increases the robustness of the network by replicating data across multiple nodes and eliminates the need for a central indexing server to discover data. In blockchain consensus, P2P networks offer better security and resiliency to malicious attacks, making them immune to Denial-Of-Service (DOS) attacks and ensuring fault tolerance. P2P gossip is widely used to disseminate data to all members of the network, preventing centralization and ensuring network resilience [16, 15]. We assume that all replicas are directly connected to each other, which simplifies the implementation of distributed computing primitives [10].

3 BFT PROTOCOLS AND ATTACKS

3.1 PARTIALLY SYNCHRONOUS BYZANTINE CONSENSUS

3.1.1 PBFT

PBFT is a seminal practical BFT protocol proposed by Castro and Liskov in 1999 [17]. It is a typical state machine replication algorithm that tolerates Byzantine malicious behaviors in a partially synchronous system.

A normal standard PBFT view, is completed in five phases: REQUEST, PRE-PREPARE, PREPARE, COMMIT, and REPLY, as shown in Figure 3.1:

1. When the process is invoked, the leader node receives a REQUEST message sent by the client and verifies the message's validity. The leader then assigns a sequence number n to the current request message and broadcasts a PRE-PREPARE message to all other backup replicas.

2. The backups authenticate the received PRE-PREPARE message and broadcast PREPARE messages along with a digest of the requests in the previous message. These two phases ensure that every non-faulty replica agrees on the order of requests in the same view.

3. Once any replica receives a quorum of $2f + 1$ PREPARE messages, it broadcasts a COMMIT message to all other replicas. This signals that a quorum of replicas has completed the persistence and promises to commit the request in its local history.

4. Finally, any replica that receives a quorum of COMMIT messages caches the client's current request and replies with a REPLY message back to the client. When the client receives $f + 1$ REPLY messages, a decision can be made. Otherwise, the request is re-transmitted.

In the event that the leader node is down, the view change phase is initiated when backups are waiting for the leader to respond over a specified timeout duration. The process comprises three phases. Each backup replica broadcasts a VIEW-CHANGE message, which contains the set of stable checkpoints and PREPARE messages. If the new leader replica receives $2f$ VIEW-CHANGE messages, it broadcasts a NEW-VIEW message. The new leader continues to execute any unfinished request from the previous view. After the other replicas verify the NEW-VIEW message, they synchronize and enter a new view.

3.1.2 Tendermint

Tendermint [58] propagates messages over a wide-area P2P gossip network. It employs a novel view change strategy in which the leadership automatically rotates to another replica based on a predefined leader selection algorithm. It applies a locking mechanism that instructs how the current replica votes for previously locked values, defending against some safety issues.

As illustrated in Figure 3.2, a leader relays the request and broadcasts a new PROPOSAL message. Other replicas echo the proposal and broadcast a PREVOTE message. Once a majority of votes are received, the node locks this value and gossips a

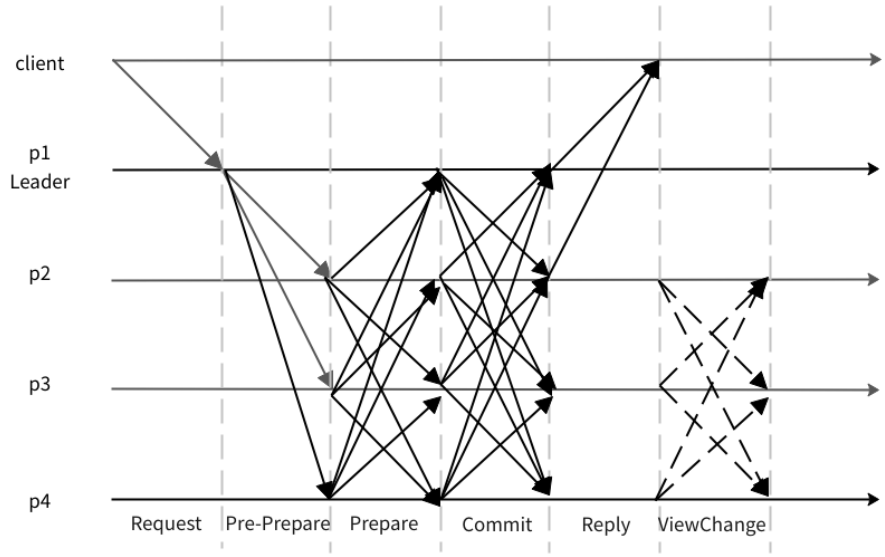


Figure 3.1: PBFT protocol.

PRECOMMIT message. After a majority of PRECOMMIT messages are received, they decide on the current value. Then, a new leader is rotated. It is important to note that in each phase, Tendermint schedules a local timeout to progress.

3.1.3 HotStuff

HotStuff [65] is a leader-based BFT replication protocol whose message complexity is linear with the number of processes instead of quadratic as in PBFT. To achieve this goal, HotStuff's normal case consists in four communication phases that involve leader-to-replicas or replicas-to-leader communication, and its view-change procedure is embedded in its normal case. Classically, HotStuff uses $n = 3f + 1$ processes to tolerate f faults and guarantees responsiveness because the leader initiates the next phase when it receives $n - f$ equal votes and because an unresponsive leader leads the replicas to initiate a view-change. HotStuff also mentions using threshold signatures and pipelining its operations to further improve its performance. Note that for simplicity, we focus on the non-pipelined version of HotStuff, which is called Basic HotStuff. The liveness and safety properties of HotStuff have been proven in partially synchronous networks, and testing our liveness checking tools on HotStuff allows us to evaluate possible false positives.

HotStuff processes maintain and extend a chain of blocks that contain user transactions that is initialized with a genesis block. All processes maintain the latest prepared and locked blocks they know of. In a nutshell, HotStuff proceeds according to five phases, which are illustrated in Figure 3.3. These phases can be described as follows.

(1) New-view 1/2-phase. All processes send the latest prepared block they know of to the leader.

(2) Prepare phase. The leader waits for $2f + 1$ identical prepared blocks and sends a propose message to all processes that contain a block that extends it. All processes are expected to vote for this new block by sending to the leader their signature on it. In this phase, a process votes on a block if it extends the latest block it locked (for safety) or if it originates from a more recent view (for liveness).

(3) Pre-commit phase. The leader gathers $2f + 1$ votes and aggregates them into a quorum certificate, which it sends to all processes. Upon receiving a quorum

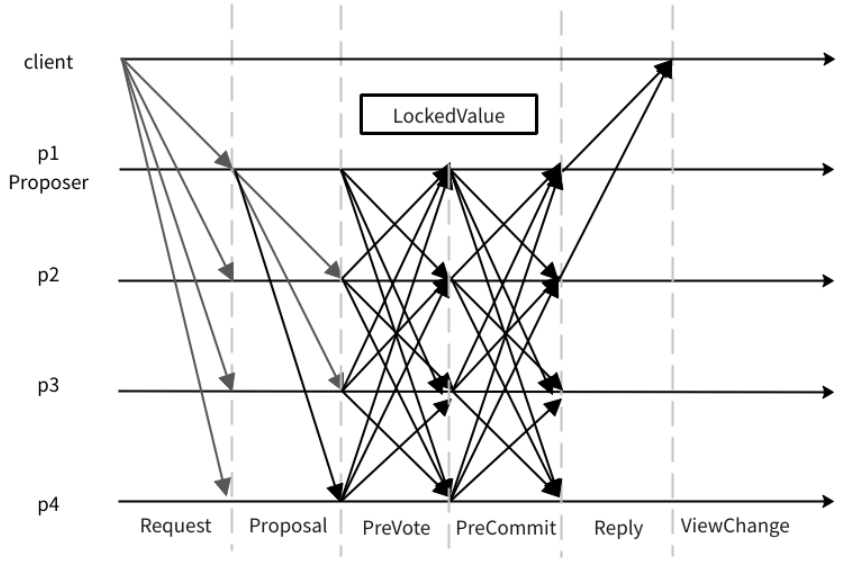


Figure 3.2: Tendermint protocol.

certificate, a process marks the block as being prepared and sends its vote for this block to the leader.
(4) Commit phase. The leader gathers a quorum certificate on a prepared block and forwards it to all processes. Upon receiving a quorum certificate in this phase all processes mark this block as being locked and send their vote for it to the leader.
(5) Decide 1/2-phase. The leader assembles a quorum certificate on a locked block and forwards it to all processes. Upon receiving this QC all processes execute the block.

3.1.4 2-Phase HotStuff

2-phase HotStuff is a variant of Basic HotStuff that Yin et al. discuss for pedagogical reasons in the original HotStuff paper [65]. 2-Phase HotStuff is very similar to HotStuff and only differs from it by the fact that it combines the Precommit and Commit phases into a single phase. In 2-Phase HotStuff, a process can lock on a block once it receives a quorum certificate in the Prepare phase.

3.1.5 Sync HotStuff

Sync HotStuff [3] is a variant of HotStuff for synchronous networks. Sync HotStuff uses a minimum of $n = 2f + 1$ processes to tolerate f Byzantine processes.

Assuming that the communication latency is bounded by Δ, Sync HotStuff's latency is bounded by 2Δ. In the steady case, upon entering a new view, all processes send their highest locked block to the new leader. After waiting for an initial 2Δ period where it receives the highest locked blocks from all correct processes, an honest leader broadcasts a new block proposal that extends over the highest locked block along with a quorum certificate for the highest locked block to all processes. All honest processes subsequently broadcast their vote on the leader's proposal during the following round and initialize a local 2Δ commit timer associated with this proposal. If a process does not detect a conflicting block when its commit timer expires, then it commits the block and all its ancestors and otherwise drops it. While waiting for commit timers to expire, the leader keeps proposing blocks, and processes keep voting on blocks, which is shown with blocks B_1 and B_2 in Figure 3.4a. In the

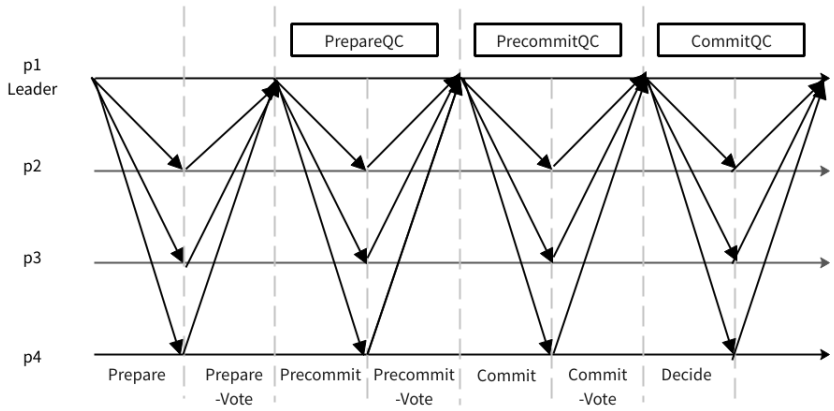


Figure 3.3: HotStuff protocol.

steady case, an honest leader, therefore, keeps proposing and committing blocks every 2Δ.

Whenever it votes on a block, a process resets a local blame timer to 3Δ. A view change (Figure 3.4b) is triggered if a process's timer expires, if it refuses to vote on a block, or if detects that the leader broadcasts conflicting blocks, then it broadcasts a blame message. Upon receiving $f + 1$ blames, a process broadcasts them and stops voting in view r, waits for 2Δ, and finally moves to view $r + 1$.

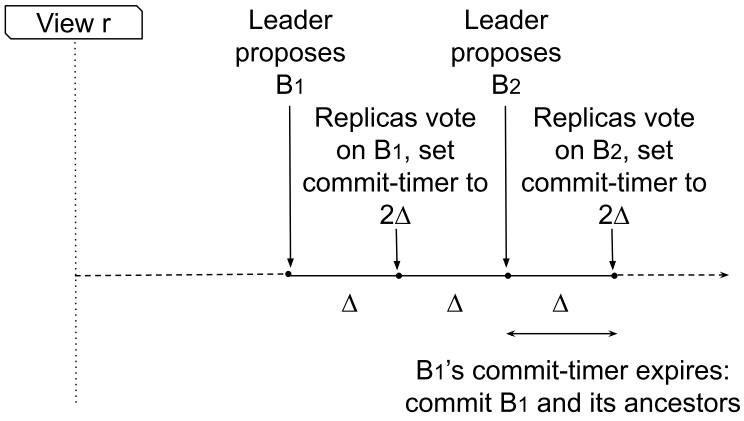
Sync Hotstuff supports different models to enhance its robustness against safety violations. The mobile sluggish model adds an extra PRECOMMIT phase to tolerate network delays for messages. The optimistic responsive model increases the quorum size to $3n/4$. These models can share some utility modules, such as a common event queue and network library.

3.1.6 Casper & Gasper

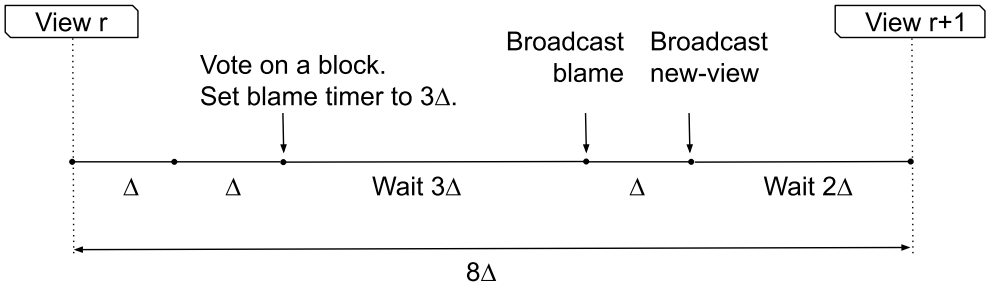
Casper [12] aims to reduce message complexity by merging PREPARE and COMMIT messages into a single VOTE message [60]. This feature allows Casper to be essentially considered a variant of Chained Tendermint, with a Two-Chain commit rule. It is designed to ensure strong accountability and penalize the misbehavior of faulty validators to defend against long-range attacks and catastrophic crashes.

As shown in Figure 3.5a, Casper defines its checkpoints using common tree structures like height, leaf, and child. A checkpoint refers to an on-chain block, and a supermajority link is a connection between two checkpoints that a higher checkpoint has accepted a quorum of votes from the lower one, e.g. $G \Longrightarrow B_1$. A checkpoint B is called justified if it has a supermajority link $A \Longrightarrow B$ from a justified checkpoint A. If this supermajority link is directly extended from a parent block, e.g.$B_2 \Longrightarrow B_3$, then we call the parent (B_2) finalized.

Gasper [4] combines Casper with an LMD-GHOST fork choice rule [57, 4]. Figure 3.5b provides an example of this algorithm. It considers the latest block messages to determine the heaviest observed subtree. This decision rule provides Gasper with network partition tolerance and the ability to finalize or validate, blocks, which can protect against protocol violations.

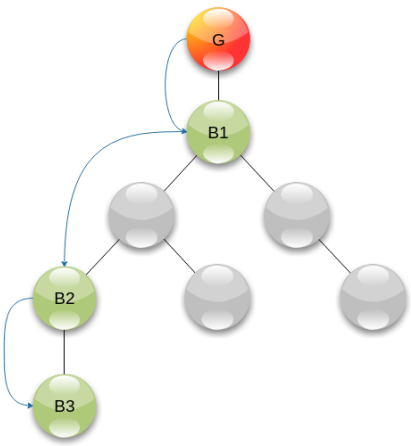


(a) Steady case with 2 successive blocks B_1 and B_2. Block B_1 is committed by a process 2Δ after it votes on it.

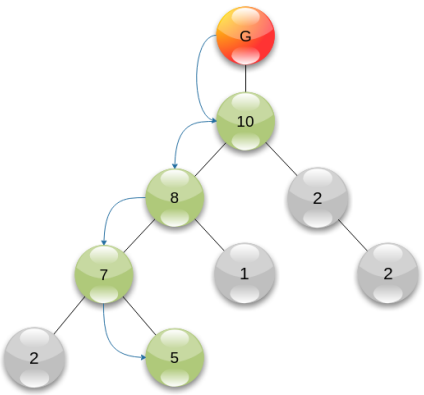


(b) View change.

Figure 3.4: Sync HotStuff's steady case and view change procedures.



(a) Casper protocol.



(b) LMD-GHOST fork choice rule.

3.2 BYZANTINE BEHAVIORS

Equivocation

Equivocation is a most common Byzantine behavior where a replica proposes conflicting proposals to different replicas, leading to safety violations if different sets of replicas commit different values. This behavior involves signing multiple votes in the same round at a given height.

Amnesia

Amnesia is a scenario where a faulty replica forgets its prior vote for a proposal in the current round and double-votes or double-signs, leading to equivocated votes in the same round.

Timing attack

Timing attacks involve distorting the timing of message transmission by delaying or reordering messages to disrupt the consensus protocol's execution. Attacks that impact message delays, for example, can increase the view change timeout or instruct honest replicas to withhold and delay their votes to a certain time slot.

Forking attack

The forking attack is a common chain-based attack that overwrites previous blocks that have not yet been committed. A Byzantine leader can deliberately generate conflicting forks from the honest leader. This attack can delay commitment, making it hard to build a direct parent-child chain and resulting in wasted resources spent on the forked blocks.

3.3 EXISTING ATTACKS

Overview

The table 3.1 presents an overview of existing attacks to the best of our knowledge in BFT consensus protocols. We list the Byzantine attacks along with their paper names, authors, attack type, whether they have been detected, and whether they have been fixed. For those attacks that we speculate can already be tested with Twins, we call it "Twins-able". As a result, this table aims to provide a comprehensive overview of the potential vulnerabilities that can undermine the safety and liveness of BFT consensus protocols, thereby informing potential improvements and refinements for these protocols. A detailed discussion of testing will be in the following Chapter 4.

3.3.1 Safety attacks

Zyzzyva, hBFT, EZBFT's safety attack

A safety attack [1] is described to violate the safety of Zyzzyva [37], which is later fixed [2]. Similarly, hBFT and EZBFT [53, 54] break their safety under a similar attack scenario. Here is an example scenario in Figure 3.6. This figure includes 4 replicas, p_1,p_2,p_3,p_4 of which p_1 is a faulty leader.

Round 1: Leader p_1

Protocol	Year	Violation	Type	Detected?	Fixed?
Zyzzyva [1]	2017	safety	Equivocation	Twins-able	Yes
hBFT [54]	2019	safety	Equivocation	Twins-able	-
EZBFT [53]	2019	safety	Equivocation	Twins-able	-
Tangaroa [14]	2017	safety	Network delay (loss)	Twins-able	-
dBFT [62]	2021	safety	Network delay	Twins-able	Yes
Algorand [63]	2020	safety	Fork	-	Yes
2-phase HotStuff [65]	2019	liveness	Non-Responsiveness attack	Not yet (Twins)	Yes
Tendermint [6]	2018	safety & liveness	Silence & Malicious delay	Not yet (Twins)	Yes
Sync HotStuff [46]	2019	safety & liveness	Timing attack	Not yet (Twins)	Yes
Casper [49]	2019	liveness	Malicious delay	Not yet (Twins)	Yes
Gasper [50]	2021	safety & liveness	Equivocation & Malicious delay	Not yet (Twins)	Yes
FaB [1]	2017	liveness	Equivocation	Twins-able	Yes
PBFT [8]	2021	liveness	Isolation	Not yet (Injection)	Yes
PBFT [45]	2016	liveness	Malicious delay	Yes (Miller et al. [45])	-
PoS Ethereum [51]	2022	safety & liveness	Fork attack	Not yet (Twins)	-
DBFT [21]	2018	liveness	Algorithm design problem	-	-

Table 3.1: Attacks in BFT consensus protocols

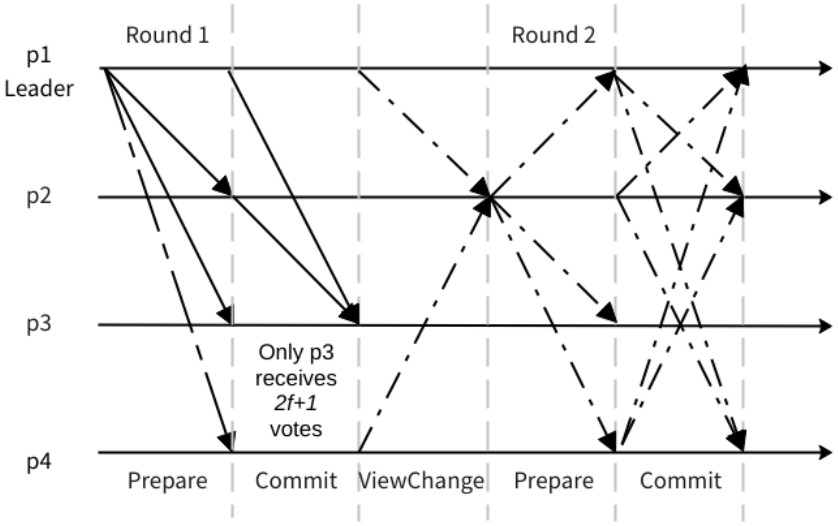


Figure 3.6: Zyzzyva's safety attack.

1. p_1 proposes a PREPARE message for value a to other replicas in a solid line and equivocates its vote for value b to p_4 in a dotted line.

2. p_2,p_3,p_4 receive PREPARE messages and broadcast COMMIT messages to other replicas. However, only p_3 receives $2f + 1$ COMMIT messages for value a and commits value a at sequence number 1. For simplification, we omit extra lines.

Round 2: Leader p_2

1. In the next round, p_1,p_2,p_4 blame VIEW-CHANGE messages: p_1,p_4 carry value b (p_1 equivocates). $2f + 1$ VIEW-CHANGE messages elect p_2 as the new leader. p_2 accepts $f + 1$ votes for value b and proposes it.

2. p_2 broadcasts PREPARE messages, and further these replicas except p_3 normally commit conflicting value b at the same height. Hence, safety breaks.

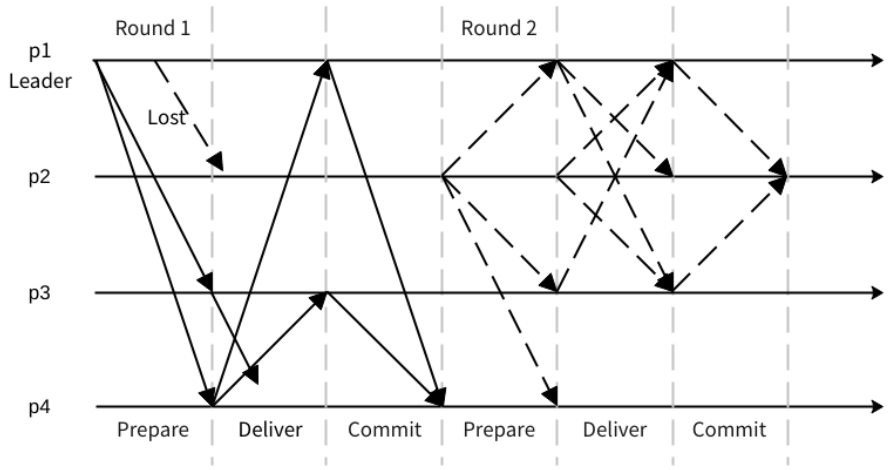


Figure 3.7: Tangoroa's safety attack.

Tangoroa's safety attack

IBM Research group points out a Tangoroa's [20] safety issue in [14]. Tangoroa extends Raft's [52] messaging structure but it is not Byzantine resilient.

In Figure 3.7, we demonstrate this attack scenario with 4 nodes: p_1,p_2,p_3,p_4 of which p_1 is a faulty leader initially. The malicious leader broadcasts a payload a to its peers while it deliberately omits to send p_2 the "AppendEntries" message either due to message loss or delay. p_3 echos the proposal while it does not complete delivery and further commit. Only p_4 delivers and commits the payload a. Then the leader rotates to p_2, p_2 has no prior knowledge of a and delivers another payload b that does not equal a. Then the two correct nodes decide on different payloads and violate the safety.

dBFT's safety attack

Wang et.al [62] present a theoretical security analysis on dBFT, demonstrating attack scenarios that violate safety, which is caused by network delays and partitions. Figure 3.8 shows the details of this scenario. The initial setting includes 7 replicas, of which p_2 is faulty. p_1 can also be seen as compromised with message delay.

Round 1: Leader p_1

1. p_1 proposes a PREPARE message for value a to other peers. Only half of the replicas (p_5,p_6,p_7) receive and accept value a, while the other half (p_2,p_3,p_4) know nothing about the proposal and time out.
2. The number of votes is undoubtedly lower than a majority, thus the VIEW-CHANGE messages are broadcast.

Round 2: Leader p_3

1. In the next round, a new leader p_3 proposes another value b. The delayed messages are received by p_2,p_3,p_4. The malicious node p_2 accepts the message and fakes a RESPONSE for the previous value a. Then, p_2 collects a quorum of responses for value a from p_1,p_5,p_6, and p_7, which is sufficient for deciding on this block.
2. Meanwhile, other replicas p_3,p_4,p_5,p_6, and p_7 execute normally and decide on value b. So far, the system has decided on two conflicting blocks in the same round at the same height, hence safety breaks.

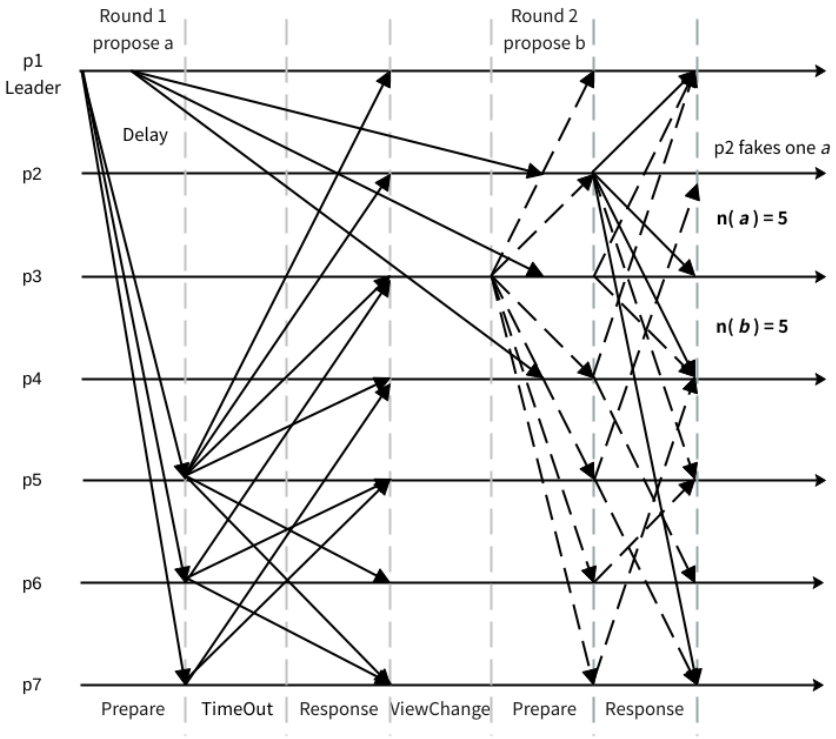


Figure 3.8: dBFT's safety attack with f=1.

Algorand's fork attack

In [63], Wang points out Algorand can easily create forks. Typically, the possibility for the Algorand to fork is relatively low (at most $1/10^9$). Suppose that there is a chain with r blocks, and we define r_1 where $3 * r_1 < r$. The adversary can corrupt all r_1 corresponding user sets, and then construct a new chain $r + 1$ long forked from the position r_1. The longer chain is legal and successful, thus for this kind of proof-of-stake blockchain, the users are able to create a conflicting chain easily.

3.3.2 Liveness attacks

2-phase HotStuff's liveness attack

The lack of a Precommit phase prevents 2-phase HotStuff from making progress in some particular scenarios, even though it remains safe. In these problematic scenarios, different processes lock on conflicting blocks and never get to execute any or update their locks. More precisely, in each view, a subset of processes locks on the block that is proposed while others reject it, and view changes that are triggered by network asynchrony prevent sufficiently enough processes from adopting the newest proposed block. Under these circumstances, the system fails to progress and is stuck in an infinite loop.

Figure 3.9 illustrates one of these problematic scenarios with four processes P_1, P_2, P_3, and P_4. Process P_1 is faulty, while the other processes are correct. The scenario this Figure illustrates is the following. Originally, all processes are locked on the same block B_0 (e.g., the genesis block). In the first view, process P_1 is the leader and proposes a new block B_1 only to P_3 and P_4 in the Prepare phase, and then only sends the quorum certificate it assembles to P_3 in the Commit phase. The messages that P_1 omits to send are shown using dashes. At the end of the first view, P_3 is the only correct process to lock on B_1. In the next view, P_2 is the leader. Processes P_1 and P_4 send block B_1 on which they are locked to P_2, but P_3's message, which

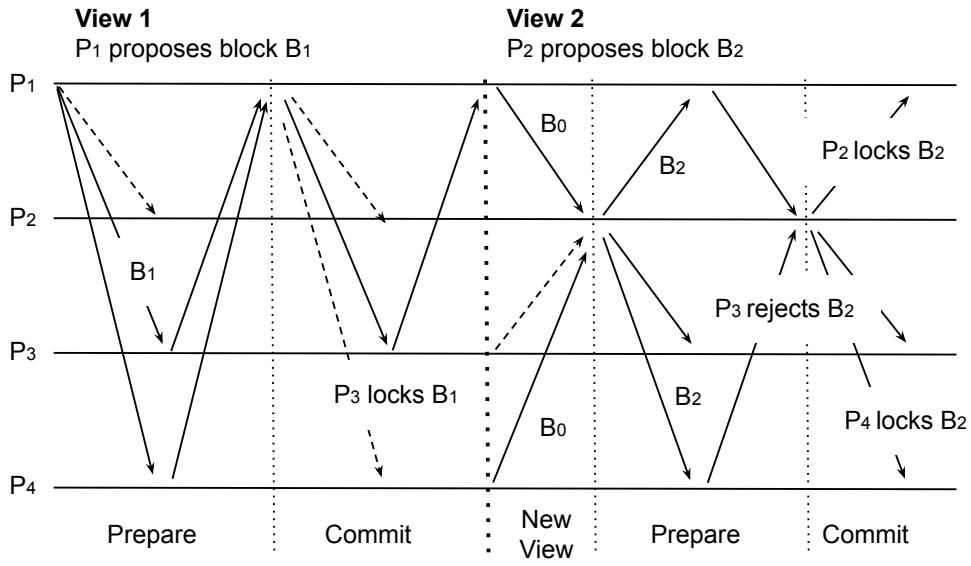


Figure 3.9: 2-phase HotStuff's liveness attack.

contains block B_1, is delayed (shown using dashes). Consequently, P_2 proposes a block B_2 that extends over B_0 but is in conflict with B_1 in the Prepare phase to all processes. Eventually, processes P_2 and P_4 lock on B_2 while P_3 rejects it and remains locked on B_1. The system is then deadlocked in future views if P_1 remains silent because no quorum of $2f + 1$ processes can be assembled by any leader.

Sync HotStuff's force-locking attack

Momose and Cruz's force-locking attack has shown that an adversary that controls the faulty processes and the network delays can break both the safety and the liveness of a preliminary version (eprint 20190312:115828) of Sync HotStuff [46].

Sync HotStuff processes keep processing the messages they receive during the 2Δ period that follows the reception of $f + 1$ blames. During this period of time, some correct processes might receive a quorum certificate from a leader. In that case, these processes are then forced to update their locked blocks, while other honest processes might not be able to do so. If different honest processes are led to lock on conflicting blocks, then the system may never be able to make progress in the future, in particular, if faulty processes subsequently remain silent since no locked block can ever collect enough votes.

The safety attack on Sync HotStuff builds on the situation where honest processes have locked on different blocks. In subsequent views, the adversary is assumed to be able to leverage network delays and use the votes of the Byzantine processes to lead different subsets of correct processes to commit different blocks.

Tendermint's liveness attack

A liveness violation [14] in the preliminary version of Tendermint is highlighted. This vulnerability is demonstrated by Amoussou-Guenou et al. in [6] with a complex 7-round scenario where two different replicas alternately lock their own proposed value at different heights. This attack is caused by the leader's silence and message delay, which allows the faulty leader to remain silent until the end of the view and delay message delivery to specific replicas.

We illustrate it in Figure 3.10. We drop PRECOMMIT messages to shorten the illustration.

Round 2: Leader p_2

1. In round 1, p_1 has already proposed and locked a value v_1.

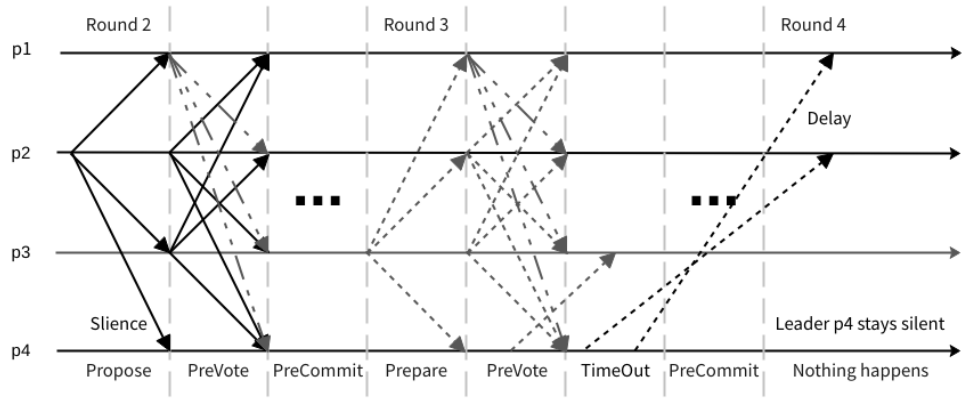


Figure 3.10: Tendermint's liveness attack.

2. The leader p_2 proposes its value v_2 but p_1 rejects the proposal and votes for its locked value v_1. The faulty replica p_4 remains silent and only p_3 echoes p_2, so no quorum is formed in this round.

Round 3: Leader p_3

1. The leader is rotated to p_3 and it proposes v_3. Still, p_1 rejects this proposal again. p_2,p_3 send their PREVOTE messages for v_3 to other replicas, while p_4 only prevotes v_3 to p_3 before the timeout, delaying its PREVOTE messages to p_1 and p_2.
2. Only p_3 collects a quorum and locks its proposed value v_3.

Round 4: Leader p_4

1. The faulty leader p_4 remains silent so nothing happens.
2. Although delayed messages from round 3 are received, they do not change the situation.

Later, p_1 can update its lock again but other replicas do not change their locked values, leading to an infinite scramble for control over the proposal.

Casper's bouncing liveness attack

A bouncing attack on Casper FFG has been identified, which causes damage to liveness under the partially synchronous model. Casper has two tiers, and the fork choice should always prioritize the justified checkpoints currently at the highest depth. The bouncing attack is caused by inconsistency between the latest justification layer and the fork-choice layer.

Assume that there are two candidate chains collecting votes from validators. One chain has initially justified a block. In the next round, the attacker deliberately delays its votes to justify another chain's block. This situation can repeat infinitely, with both chains growing, preventing a chain from being committed on a direct child block.

Gasper's balancing liveness attack

A Gasper vote contains a Casper vote and a GHOST vote. However, a liveness violation has been detected by Neu et al. in [50]. This attack is not only similar to Casper's bouncing attack, but it also causes faulty replicas to split votes between two chains, breaking safety even without a network partition. The adversary should

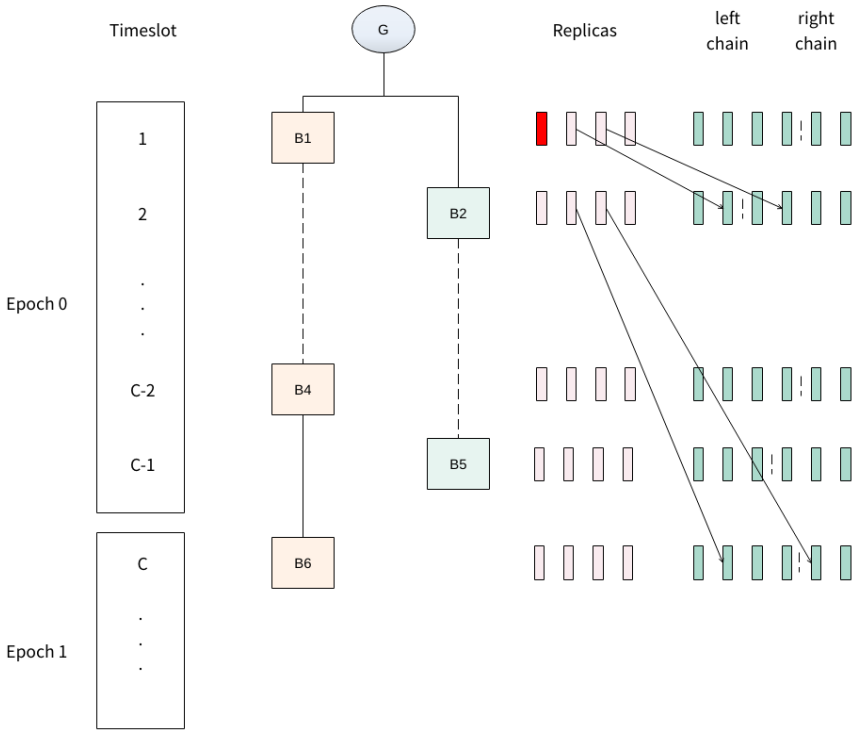


Figure 3.11: Gasper's balancing attack.

be able to know when fork choice is executed, deliver a message to specific validators at a specific time, and prevent honest validators from exchanging their recently received messages with each other.

Figure 3.11 depicts the details of this attack. Gasper splits progress into C timeslots in each epoch. In the attack scenario, the adversary proposer kicks off the attack by equivocating two conflicting blocks with the 'left' and 'right' chains in slot 1 of epoch 0. For each following slot except the last one in epoch 0, a pair of validators delay their current votes to the next slot in order to selectively ensure the equality between the 'left' and 'right' votes in slot $i+1$. For each following slot in epoch 0, another pair of validators delay their current votes to the next epoch in order to selectively ensure equality between the 'left' and 'right' votes in slot $C+i$. If there is a vacancy for honest validators, the faulty proposer assigns a fake node to fill the gap.

In Epoch 1, the adversary releases the additional votes backlog in the previous epoch to keep splitting honest validators into two chains. During this Epoch, the adversary is still instructing honest validators to believe that the chain they previously voted on is still leading the execution. This scenario can continue infinitely, with neither chain able to reach a consensus.

FaB's liveness attack

A liveness attack [1] is presented on FaB [44], a precursor of Zyzzyva with a two-phase commit fast track. This attack is simple and similar to Zyzzyva's safety violation. It also starts with a faulty leader splits the honest replicas into two groups with conflicting proposals value a and b: (1) group a: $f+1$ votes for value a and (2) group b: f votes for value b. Only one replica A of the group a receives $2f+1$ votes for value a and form a QC. During the next view change phase, there is a QC for value a and $f+1$ VIEW-CHANGE messages for value b including one equivocated view change blame from the faulty leader. The progress can stop when this replica A and $2f$ replicas in group b do not create a new QC.

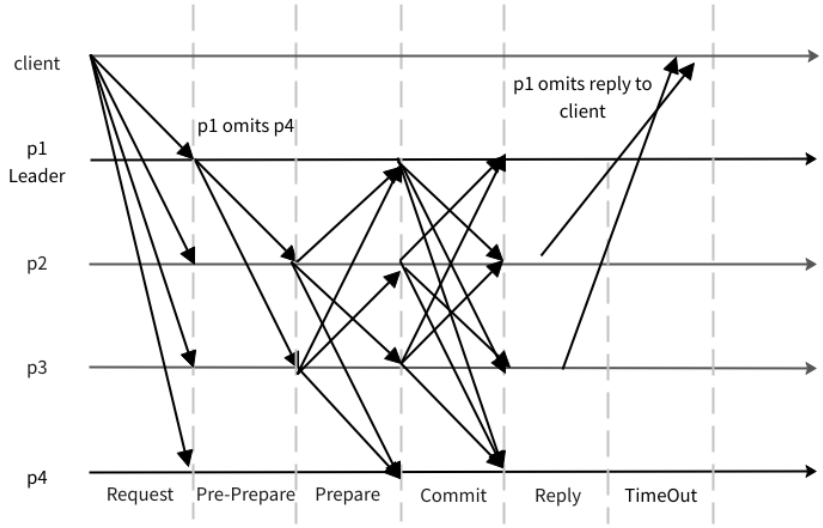


Figure 3.12: PBFT's read-only optimization liveness attack

PBFT's read-only optimization liveness attack

Berger et al. [8] present a liveness violation in PBFT when the read-only optimization request is applied. All operations including reads and updates have to wait for a quorum of replies from the client. If a malicious leader selectively omits to send PREPREPARE messages to up to f correct replicas, such replicas lack information about the client request and cannot participate in the further commit phase, violating liveness. The authors conjecture that this attack can be extended to similar protocols, such as HQ, PBFT-CS, and BFT-SMaRt.

From Figure 3.12 we can see an example scenario: the Byzantine leader p_1 deliberately omits to send the PREPREPARE message with the proposed value v to p_4, so p_4 is unable to prepare for v in the next phases and can not take part in the further PREPARE and COMMIT phases. According to the reasons above, the liveness breaks when the system times out.

PBFT's delaying liveness attack

In the HoneyBadger [45], a delay attack is described that can thwart the progress of PBFT by simply delaying the scheduler's message delivery to the current leader. This forces the leader to withhold its PREPREPARE message until the timeout triggers a view change. Moreover, the scheduler delays the receipt of VIEW-CHANGE messages from the last round, making it difficult for the new leader to catch up with the latest progress. This cycle can be repeated, breaking liveness.

PoS GHOST's balancing liveness attack

Neu et al. [51] present a generic attack discovered on PoS GHOST variants. This attack is illustrated with $k = 5$ number of initial honest blocks in Figure 3.13. Honest nodes on the left build a chain, while faulty blocks on the right fork the same blocks and equivocate a sub-tree grown from the genesis. Honest nodes create another new chain on top of node 2; however, the faulty blocks again fork another sub-tree to replace honest nodes. Finally, all honest nodes can be replaced. If the total number k of initial honest blocks is large enough, the adversary can replace $\Theta(k^2)$ honest blocks.

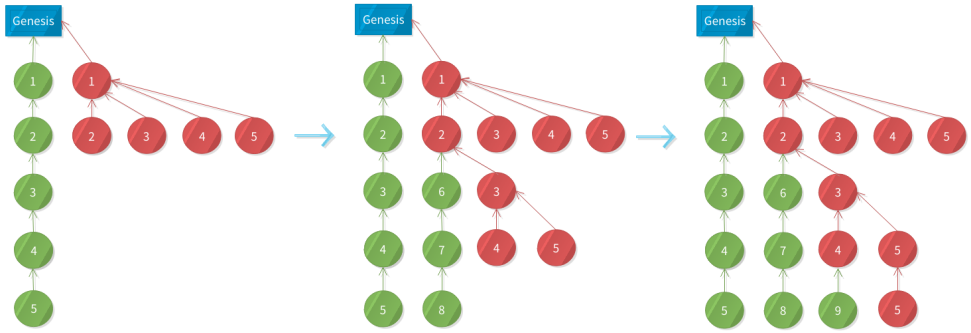


Figure 3.13: PoS GHOST's avalanche attack.

DBFT's non-deciding liveness attack

DBFT [21] is a leaderless Byzantine consensus for blockchain where each node shares a similar priority in the decentralized execution of the consensus. DBFT supports an asynchronous communication network, and the authors provide a binary Byzantine consensus (BBC) using a strategy called BV-broadcast, which is quite similar to randomized consensus [45]. In each round, BV-broadcast proposes an estimated value and then decides upon the estimated convergence to the round number modulo 2. The decision process still loops because it may help other peers to converge in the next two rounds. The authors have addressed this issue in the same paper and proposed a fix.

4 BFT TESTING SYSTEMS AND ALGORITHMS

4.1 CONSENSUS TESTING

4.1.1 Twins

Facebook proposed Twins [7] based on their own product, DiemBFT [23], which is a highly scalable and secure protocol based on HotStuff. It systematically generates test case scenarios with Byzantine faults and explores them. It models Byzantine behaviors using twin copies of the processes, i.e., processes with the same identities and credentials. It runs the cluster with twin replicas and network partition, where the twin replicas exhibit Byzantine behaviors such as equivocation, double voting, and loss of internal state, which causes them to forget their voted values.

Figure 4.1 presents a Twins system architecture with its components. The scenario generator employs a round-by-round generation style. Given that the number of parameters (nodes, leaders, rounds) is determined, it produces various scenarios and feeds them into the executor. The generation can be summarized into several steps:

1. The scenario generator first produces all the possible node partitions.
2. Then the scenario generator assigns each partition to all possible leaders.
3. The scenario generator generates all possible ways to arrange the leader-partitions scenario pairs over a given number of rounds.
4. Finally, the generated scenarios are sent to the scenario executor for execution. The execution results are saved to logs, especially the information on buggy scenarios.

The research team claimed that several famous BFT attacks have been reinstated and successfully detected with only a few nodes and a few minutes. However, it is open work to rigorously characterize the attacks that Twins can cover. To the best of our knowledge, Twins has not yet been extended to detect liveness bugs.

4.1.2 Other testing methods

There is a large body of work that propose new BFT consensus algorithms, make them robust against Byzantine faults [19], or model-check the correctness of consensus algorithms [39, 36, 35]. In this section, we focus on the most related work on testing BFT consensus implementations.

Several existing methods for testing consensus systems focus on analyzing crash-fault tolerant protocols and exercise different executions of the systems under asynchrony, network faults, and crash process faults [5, 24]. e.g.*Jepsen* [43] is an effort to test the safety of distributed databases and consensus protocols. It simulates network partitions for distributed systems, and it has detected several violations in the consensus systems [33, 34, 31]. However, Jepsen is not designed for Byzantine consensus.

Targeting BFT systems, *BFTSim* [56] explores the system's behavior under unexpected network conditions and faults using a network simulator. *Turret* [41] detects performance attacks on BFT systems by generating Byzantine attack scenarios with

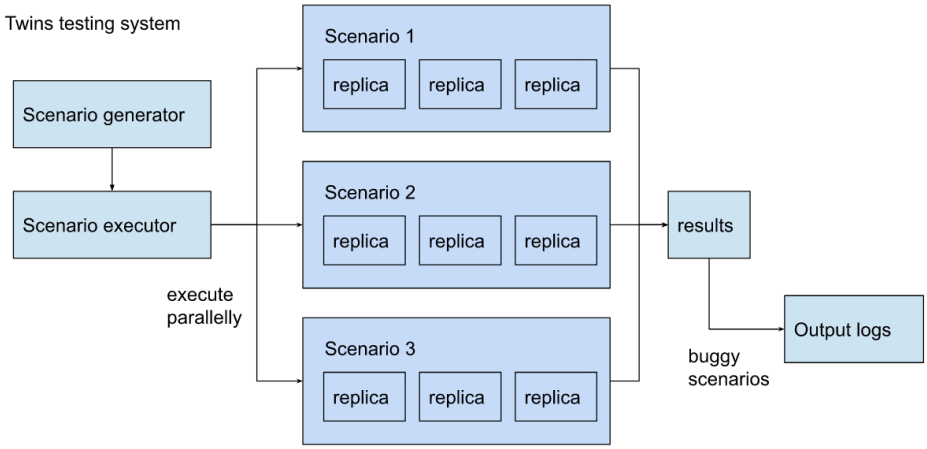


Figure 4.1: Twins system.

malicious message deliveries, including message dropping, delay, duplication, and diversion. *Apollo* [61], is designed as SBFT's [29] testing framework. The main design of the testing strategy is that it provides reusable functions, specific components, and methods applicable to its specific instances. Several works [42, 27, 25] provide testing frameworks that can model and inject network and Byzantine faults into the executions of BFT protocols.

Netrix [25] provides a domain-specific language and a controlled networking environment that allows programmers to specify restrictions on the generated executions or implement their unit tests with network and Byzantine faults.

Recent work *ByzzFuzz* [64] generates test executions with randomly sampled network and Byzantine process faults. It models Byzantine faults using small-scope mutations to the original contents of the protocol messages and randomly injects a parameterized number of mutations.

4.1.3 Discussion

In this thesis, we primarily use Twins as our test system. First, Twins is an innovative approach specific for BFT consensus testing, while prior testing work focused more on crash faults. Second, Twins is easy to implement and adopt, which gives us sufficient space for extension. Third, Twins has already theoretically proved that it is able to reinstate some known attacks presented in our Table 3.1. Although Twins hasn't yet implemented the detection of Zyzzyva's safety violation, we believe Twins can easily detect it within several rounds, and further extend to detect similar safety violations caused by equivocation or network delay in Table 3.1. Finally, Twins does not implement any strategies or algorithms to detect liveness bugs. All the discussions of liveness issues are that Twins can theoretically reproduce FaB's liveness issue and Twins is unable to resolve timing attacks in Sync HotStuff. It is calling for some open work such as detecting liveness violations and tackling multiple pairs of Twins, which is highly related to our topic.

To the best of our knowledge, we do not find a more suitable BFT testing system or method to start our thesis than Twins.

4.2 LIVENESS CHECKING

In this section, we introduce two approaches that aim at identifying liveness violations. The first one is bounded liveness checking (also referred to as *temperature*

checking) [32, 47] and *lasso detection with partial state caching* [47]. These methods build on the notion of a *hot state*, which is a system-wide state in which the system does not satisfy some of its properties or produce useful results. Intuitively, the temperature-checking method reports a violation if the system remains in a hot state for a long time. On the other hand, lasso detection methods detect the existence of a cycle of states (i.e., a lasso) where the system can possibly get stuck forever following its transitions.

4.2.1 Temperature method

Temperature checking checks for *bounded* liveness, which means that it reports a violation if an execution does not produce a useful result (i.e., produce a new block in the case of blockchain consensus) for a specified amount of time. The method maintains a temperature variable $temp$, which is equal to the number of successive hot states the system remained in. It reports a violation if the temperature reaches a certain threshold value TT provided by the programmer.

Algorithm 1 provides the pseudocode of temperature checking. The checking temperature method is called whenever the system reaches a state s after executing a sequence of system events *trace*. This function increments the system's temperature $temp$ if state s is a hot state (line 2), and resets it to 0 otherwise (line 7). Violations are detected when the temperature reaches the temperature threshold (line 3), and therefore directly depend on the value of parameter TT that is provided by the user. A low threshold value might result in a high number of false positives, while a high threshold value leads to longer execution traces that are more difficult to interpret.

Algorithm 1 CheckTemp

Input: Current state s
Input: Current trace *Trace*
Input: Current temperature *Temp*
Input: Threshold TT
Output: Updated temperature value

```
1 if Hot(s) then
2 |   Temp ← Temp + 1
3 |   if Temp = TT then
4 |   |   REPORT-LIVENESS-BUG(Trace)
5 |   end
6 else
7 |   Temp ← 0
8 end
9 return Temp
```

4.2.2 Partial-state caching method

Liveness checking based on lasso-detection aims at finding a cycle of states (i.e., a lasso) where the system might get stuck and repeat its state infinitely often.

Detecting lassos in the executions of distributed systems is challenging because it is impractical to register the entire state of complex software systems. However, one can rely on the partial-state caching method [47]. This method captures only part of the system state to check whether a partial state is repeated during an execution. Since the state caching is only partial, repeating the same partial state does not ensure repeating the same state in the execution. The existence of the cycle is then verified by replaying the execution of the detected trace using a controlled scheduler that enforces the execution of certain events and traces.

Algorithm 2 and Algorithm 3 details the partial-state caching algorithm. Given an execution of a trace *Trace* that has reached a system state s, it checks whether the current execution may cause a liveness violation. To do so, it uses a hash function *Hash* to hash the partial state information, which ideally maps each partial state to a different hash value. For each new state s that is reached during the execution, it then checks if $Hash(s)$ has been seen earlier. If it is the case, then it means that a potential cycle in the state transition system has been identified. The cycle forms a liveness violation if the states in the traces do not satisfy the system's properties, i.e., if all the events $e \in trace$ are in hot states. If the algorithm finds a cycle with a hot trace, it then verifies the existence of a real cycle by checking each state's enabled processes and, if so, reports a liveness violation.

The partial caching method is clearer for developers to analyze than the temperature method because it is easy to target the position where the system fails.

Algorithm 2 CheckLasso

Input: Current state s
Input: Current trace *Trace*
Input: Threshold value *RT*
Output: Updated trace
Output: New current trace

```
for i ← 0 to Len(Trace) do
    if Hash(s) = Hash(Trace[i]) then
        C ← Trace[i..len(Trace)]
        if Hot(C) ∧ Fair(C) then
            return ReplayCycle(s, C, Trace, RT)
        end
    end
end
```

Algorithm 3 RepeatCycle

Input: Current state s
Input: Current trace *Trace*
Input: Potential cycle C
Input: Threshold *RT*
Output: New current state
Output: Updated trace

```
for j ← 0 to RT × Len(C) − 1 do
    i ← j mod len(C)
    m ← scheduled(C[i])
    if m ∉ Enabled(s) then
        return (s,Trace)
    end
    s′ ← T(m, s)
    Trace ← Trace + (m, Enabled(s), Hot(s), Hash(s))
    s ← s′
    if ¬IsHot(s) ∨ Enabled(s) ≠ enabled(C[i + 1 mod len(C)]) then
        return (s,Trace)
    end
end
REPORT-LIVENESS-BUG(Trace)
```

4.2.3 Other liveness checking work

There are some other works focusing on liveness checking. Biere et al. [9] present a transition from checking liveness violations to safety properties (LTS). This paper discusses lasso-shaped state traces and counter-based liveness checking, which is a prior work similar to the partial-state caching algorithm. *k-LIVENESS* [18] checks the liveness of finite-state systems by counting and bounding the number of a fairness constraint. This method is simpler than LTS yet performs better with fairness constraints. Tsuchiya and Schiper [59] uses model checking to verify asynchronous consensus algorithms. They reduce the challenging and infinite consensus problems to finite bound checking scenarios.

In our thesis, we do not adopt these methods because they are either too old, complicated, or impractical for a blockchain system.

5 APPLY LIVENESS CHECKING TECHNIQUES

OVERVIEW

Based on the analysis of the attacks listed in Chapter 3.3, it is clear that Twins can quickly capture most existing safety violations. Checking safety properties in the execution of a protocol is straightforward since we can check whether they are violated in the states that the system reaches.

The liveness attacks listed in the Table 3.1 are caused by conflicting locks or disturbing message orders. For streamlined protocols such as Tendermint, 2-Phase HotStuff, and Sync HotStuff, their liveness breaks when two chains lock conflicting blocks and therefore do not vote for each other. Casper's liveness violations are triggered by delaying votes, while it also enters a similar scenario that replicas are not allowed to progress conflicting votes. Gasper's liveness attack is more complicated as its adversary controls several pairs of replicas to withhold their votes and balance the number of votes between two chains across different epochs. It requires tackling multiple pairs of twins and more precise message control over multiple views. Detecting these liveness violations is more difficult than detecting safety properties since it requires finding an infinite execution that the system actually does not progress.

While the temperature and partial state caching methods provide a practical solution for checking the liveness of software systems, they are not directly applicable to blockchain consensus systems for several reasons. First, there does not exist a common notion of a partial state that captures relevant information during the execution of blockchain systems. Defining partial states is a delicate task. On the one hand, a partial state that overly abstracts the system information may fail at capturing essential state information and therefore suffer from a high rate of false positives. On the other hand, a partial state that would include too much information would not be impractical with large software systems. Second, the notion of *hot state* has not been defined for streamlined blockchain systems and is required by the temperature and lasso detection methods, which we aim to use. Finally, the lasso detection method requires a controlled scheduler to check whether a detected potential cycle is replayable. More specifically, it uses the scheduler to enforce the system to run the sequence of events that produced the detected cycle of system states; it checks if the cycle is replayable and only reports a violation if it is replayable. This makes the lasso detection method difficult to apply for systems that do not have a controlled event scheduler.

In this work, we address these issues by: (i) formulating a *partial-state* definition that captures the essential state information during the execution of a streamlined BFT consensus algorithm; (ii) defining *hot states*, which are states that model bad states, for streamlined BFT consensus; and (iii) using the execution state space for checking the existence of lassos (i.e., state cycles).

5.1 DEFINITIONS

5.1.1 Partial State in the HotStuff Protocol Family

The liveness of the HotStuff protocols in Table 3.1 can be violated in the presence of conflicting locks among the processes of a system. Essentially, our partial process state encapsulates essential information about the state of a process, which is modified through the various phases of the protocol execution. The HotStuff authors have provided various variables to describe the protocol state, such as the locked block, the last executed block, the height of the last voted block, the prepared block, the current view id, and the leader id. After having listed the variables that a process maintains, we realized that storing prepared blocks in process states is not necessary to detect the known liveness bugs in HotStuff protocols. However, locked blocks are instrumental in known liveness violations. In addition, executed blocks allow us to identify situations where two locked blocks exist in the system, which is a necessary condition for liveness bugs, but one has been executed, which indicates progress. We, therefore, define partial process states as follows.

Definition 1 (Partial process state)**.** *We define the partial state s of a process p as a tuple $\left\langle H(b_{prepared}), H(b_{lock}), H(b_{exec}) \right\rangle$ where $H(.)$ is a hash function, $b_{prepared}$ is the last block that the process prepared, b_{lock} is the block that is locked by the process and b_{exec} is the last block it executed.*

In the context of the HotStuff protocol, the system's state is defined as the set of states of all processes within the system. It is worth noting that, for the purpose of defining hot states, the state of the network channels may not be necessary to include.

Definition 2 (Partial system state)**.** *The partial system state is a tuple $\langle stateMap \rangle$ where $stateMap : P \mapsto S$ maps each process p to its partial state $S_p \in S$.*

We calculate the hash of a system state once and store it to test whether two states are equal based on their hashes. This approach can reduce the computational burden and improve the efficiency of the protocol. It is important to note that if the locked blocks stem from the same blockchain, we do not need to compute a new hash to avoid introducing too many distinct states. This strategy can also help improve the performance of the system by minimizing the number of necessary computations to check for system state equality.

5.1.2 Hot State in the HotStuff Protocol Family

We define hot states for streamlined consensus protocols based on their *locking* phase, where a node locks on a block once it learns that a Byzantine quorum has committed to it.

Definition 3 (Hot state for streamlined protocols)**.** *We say that a streamlined blockchain system is in a* hot state *if it satisfies these conditions: (i) the correct processes hold at least two locks on conflicting blocks, (ii) there is no locked block on which a quorum certificate could be generated if all correct processes that have not locked on a block decided to lock on it, and (iii) a correct process has not executed a block.*

The first condition (i) checks for conflicting locked blocks in the system, where two different processes have simultaneously locked distinct blocks. Condition (ii) states that the processes are not able to generate a quorum certificate on one of the existing locked blocks to reach consensus in this view, i.e., no locked block

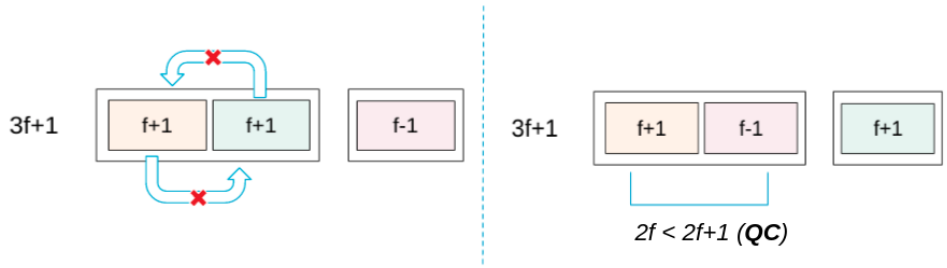


Figure 5.1: Two locks in the system.

can accumulate enough votes from processes that either already locked on it or could lock on it (because they have not locked a conflicting block). Condition (iii) guarantees that the system does not execute a block or respond to client requests, preventing any further updates until the conflicting state is resolved. In Figure 5.1, we present a scenario that there are some honest replicas locking on two conflicting blocks, and $f-1$ replicas that do not or have not yet locked any block. According to HotStuff's voting rule, these two groups of replicas never vote for each other because their proposal does not extend their previous block. If $f-1$ replicas vote for one group, all the $2f$ votes are not enough to create a new quorum.

Given Definition 3, which provides a general definition of hot states for streamlined blockchain protocols, it is possible to adapt the general hot definition for HotStuff, 2-Phase HotStuff, and Sync HotStuff given their quorum certificate sizes (i.e., $2f+1$ for HotStuff and 2-Phase HotStuff, and $f+1$ for Sync HotStuff).

Hashing

To check whether the system is in a hot state, we use a liveness monitor, which keeps track of the current state of the processes. In particular, we save the hashes of the prepared ($b_{prepared}$), locked (b_{lock}), and the last executed block (b_{exec}) for each process, and we ignore other variables. The probability of two hashes colliding is extremely low at $4.3 * 10^{-60}$, and it is believed that hash collision never occurs during testing.

Execution cases

In real executions, there can be three main situations that could occur. They are labeled to be *Safe*, *Hot* and *Recover*. A *Safe* execution means the system never enters a hot state and successfully commits a decision. A *Hot* execution is when the system remains in hot states until the end of the execution. A *Recover* execution is when the system experiences a finite number of hot states but eventually recovers from them.

5.2 DETECT LIVENESS ISSUES IN 2-PHASE HOTSTUFF

5.2.1 Attack scenario

To describe the attack scenario when we reinstate the liveness violation in Twins, we set 4 processes and 1 twin as our default configuration. Nodes 1,2,3,4 are honest, and node $4'$ is the twin of node 4, with the system unable to distinguish between them. Suppose the twin node not only equivocates but also loses states (i.e., locked block and double voting), we present an example scenario and observe the state transition as shown in Figure 5.2:

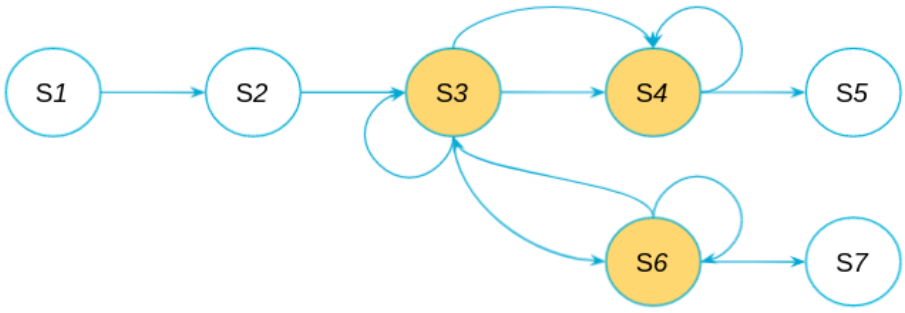


Figure 5.2: State transition.

View 1 Node 4 is selected as the first leader, with the partition {[1,2,4′], [3,4]}. Nodes 4 and its twin propose different blocks, but since the system can't distinguish between them, only QC([1,2,4′]) B_1 is created.

View 2 Node 1 is selected as the next leader, with the partition {[1,4′], [2,3,4]}. Node 1 and node 4′ receive messages and lock the previous QC B_1 although they can not collect any new QC. The system enters a state S_2.

View 3 Node 3 is selected as the next leader, with the partition {[1,4′], [2,3,4]}. Node 3 knows nothing about the previous QC due to the partition and creates a new QC([2,3,4]) B_2.

View 4 Node 4 is selected as the next leader, with the partition {[1,2,4′], [3,4]}. Nodes 3 and 4 lock the previous QC B_2, resulting in the system entering the hot state S_3. If honest node 2 is partitioned at the end of this view, no progress can be made since nodes [1,4′] and nodes [3,4] will never vote for each other, resulting in the system looping in hot state S_3.

View n After some views, the Byzantine node may lose its lock and return to S_3. If the system recovers from hot states, it will enter state S_5 or S_7.

5.3 DETECT SAFETY & LIVENESS ISSUES IN SYNC HOTSTUFF

In this section, we discuss how to detect the force-locking attack on the preliminary version of Sync HotStuff (discussed in section 3.3.2). This attack was summarized in the Twins paper, but it couldn't be detected due to the message delivery timing required.

5.3.1 Attack scenario

The force-locking attack involves an honest leader only collecting f votes when other $f+1$ honest replicas time out and blame the current view. The leader then receives a delayed vote from the twin node, creates a certified block, and is forced to update locks. Other replicas quit the current view, knowing nothing about the highest certified block.

5.3.2 Adaptation

Among the protocols on which we apply our liveness-checking methods, Sync HotStuff is the only one that cannot be directly simulated on top of Twins, since it does not rely on a lock-step process. More specifically, in the steady state, a leader in Sync HotStuff keeps proposing blocks every 2Δ until any equivocation or network delays prevent progress. However, a Twins scenario requires a new leader to be specified for each view, while Sync HotStuff's steady operating mode uses a leader until a view-chance occurs.

Therefore, we simulate Sync HotStuff on top of the view mechanisms that support the HotStuff protocols. In a nutshell, we allocate enough Δ periods of time per view so that messages generated by two successive views in Sync HotStuff never exist concurrently in the system, which would needlessly complicate implementation efforts. So the steady operating mode can also successfully proceed with a new leader and network partition like HotStuff.

5.3.3 Delay bounds

The force-locking attack on Sync HotStuff is possible when messages can suffer from network delays (i.e., outside of the strictly synchronous network model). However, Twins does not support test scenarios with message delays and cannot detect these attacks. We extended Twins to generate test execution scenarios that delay the delivery of messages at 0.5Δ increments.

In Sync HotStuff, there are four types of messages: PROPOSE, VOTE, NEW-VIEW, and BLAME. We do not delay NEW-VIEW and BLAME messages.

For PROPOSE messages, according to the view-change rule, proposals later than 3Δ are regarded as failed requests. It is meaningless to generate a case where an honest leader is completely unable to send a valid proposal and drop it. Therefore, we set the maximum *proposal-delay* bound to 3Δ.

For VOTE messages, according to Sync HotStuff's voting rule, after a proposal is received, replicas simultaneously send votes and start counting down a 2Δ commit-timer. If the *vote-delay* is longer than the commit-timer (2Δ), the votes arrive too late and the leader can never collect a quorum of votes. Thus, we bound the maximum *vote-delay* by 2Δ.

5.3.4 Scenario description (delaying and re-ordering)

To trigger the force-locking attack, we inject message delays into the scenario generator. We put up two abstractions, a coarse one that delays all messages uniformly, and a fine one that more precisely controls messages.

Coarse-abstraction

The coarse-abstraction scenario handles messages roughly. According to the delay bounds, we randomly delay Propose messages with delays that belong in $[0, \frac{1}{2}\Delta, 1\Delta, \frac{3}{2}\Delta, 2\Delta, \frac{5}{2}\Delta, 3\Delta]$ and delay all Vote messages with delays that belong in $[0, \frac{1}{2}\Delta, 1\Delta, \frac{3}{2}\Delta, 2\Delta]$. We use uniform distribution to make scenario generation simpler.

Fine-abstraction

The fine-abstraction scenario allows for more precise handling of messages, with the scheduler delivering messages to specific parties at specified timestamps. We must specify how each message is handled, including the kind of message delayed/re-ordered, the duration of delays, the timestamp a message starts delivering, the sequence of message orders, the sender and receiver, and even the priority to execute a specific message if it's re-ordered.

Generating this kind of scenario is more complicated, and many parameters must be randomized in each view. The scientific analysis of how a test case would be executed in practice can be difficult. We believe a coarse-abstraction scenario is sufficient to reveal the force-locking attack.

5.4 SCENARIO GENERATION

5.4.1 Generation algorithm

In Algorithm 4, we present an example of generating a coarse-abstraction scenario for Sync HotStuff to describe our scenario generation algorithm.

We start by enumerating all the possible combinations of different parameters, such as randomly generating delays and adding nodes to different partition sizes (line 9). Then, in each view, we use a uniform distribution to randomly select one from them as the configuration for that view (line 20). This selection is fair because there is no priority assigned to any specific case. In this way, the generated scenarios can also be viewed as shuffled test cases. To avoid generating duplicate cases, we record the generated test cases for deduplication (line 25).

5.4.2 Scenario prioritization

A potential issue for detecting liveness bugs with Twins is that its generator should generate long scenarios, which adds difficulty to testing due to redundant cases. In previous Twins, the focus is on safety violations, with each scenario being short (4-7 rounds). For liveness bugs, a few rounds are not sufficient. When testing 10 rounds, the number of scenarios that need to be generated is extremely large (10^{32}).

We set several rules to prune the number of scenarios that the Twins generator can produce.

Rule A: *Use two partitions with fixed sizes: {[2f+1], [n-(2f+1)]} .*

To detect liveness violations, we need at least one partition with a quorum size. We don't consider cases with more than two partitions because we only need to observe two conflicting partitions of locks. Furthermore, when the number of rounds is large enough, we can discard partitions with a super-quorum because they occupy a large proportion of scenarios that actually run duplicated scenarios.

Rule B: *Drop duplicated replacing cases.*

If we already have a partition such as {[1,4], [2,3,4′]}, we would consider partitions like {[2,4], [1,3,4′]}, {[3,4′], [1,2,4]} as duplicated cases because they just switch the places of honest replicas. Since we don't specifically distinguish between these honest replicas, we find that they have a lot of repetitions when tested, so we only need to keep one of them.

Rule C: *Twins are split into two partitions.*

This rule only removes the cases in which both twins are allocated into the partition. When the twin node is a leader, consider {[1,4,4′], [2,3]}. This case violates the rule that each node votes for only one proposal in a round, and only one proposal is valid after voting and view-change. We can drop this case if we require the twins to propose conflicting blocks at the same height.

Algorithm 4 Scenario generation

Input: Node set *Nodes*
Input: Partition sizes *Sizes*
Input: Number of nodes n
Input: Number of rounds r
Input: Number of scenarios s
Output: Generated scenarios *Generated*

```
// enumerate all the possible cases
cases ← []
for i ← 0 to n do
    for j ← 0 to len(Sizes) do
        // get random uniform-distributed delay from [0, 0.5Δ, ..., delayBound]
        pd ← getProposeDelay()
        vd ← getVoteDelay()
        p ← addNodes(Nodes, Sizes)
        append(cases, {L : Nodes[i], P : p, PreposeDelay : pd, VoteDelay : vd})
    end
end
// generate a certain number of distinct scenarios
casesLen ← len(cases)
used ← []
count ← 0
while count < s do
    idx ← []
    for i ← 0 to r do
        // get random uniform-distributed case from [0, casesLen]
        idx[i] ← randInt(o, casesLen)
    end
    if contains(used, idx) then
        continue
    else
        append(used, idx)
        scenario ← []
        for j ← 0 to r do
            case ← cases[idx[j]]
            append(scenario, case)
        end
        append(Generated, scenario)
        count ← count + 1
    end
end
return Generated
```

5.5 APPLICATION IN TWINS

5.5.1 Checking for temperature

For the temperature checking method, monitoring whether the system is in a hot state and maintaining the temperature variable to track the duration in which the system stays in a hot state is sufficient to detect potential liveness violations. The method requires two basic variables: (i) the temperature variable, denoted as *Temp*, to count the steps the system remains in the hot states, and (ii) the temperature threshold, denoted as *TT*, to determine which execution should be identified as a liveness bug. If *Temp* exceeds *TT*, a liveness bug is immediately reported, while a situation where the temperature does not reach the threshold or quickly recovers indicates a healthy case.

5.5.2 Checking for lassos

Typically, the standard approach for verifying the existence of a cycle is by replaying the detected cycle with a controlled scheduler. If the events' schedule in the detected cycle can be repeated, a violation could be confirmed and reported. However, one of the main challenges in replaying a cycle of system states is the need for a controlled event scheduler to enforce executing a particular event and reaching a particular state in the execution. Unfortunately, the majority of distributed systems and testing frameworks do not possess a controlled environment, making it difficult to confirm and report violations.

The existing Twins approach conducts test execution scenarios with specific Byzantine behaviors and network faults. However, it lacks the ability to control the execution of protocols at message granularity. Therefore, it cannot enforce the execution of a given schedule of events to check whether some detected cycle can be replayed. However, it is still possible to observe the states that the system reaches.

In this work, we check for lassos on the *state transition graph* of the system, which increases the likelihood of detecting potential cycles of states reachable in the executions and also does not require a controlled scheduler. We construct the state transition diagram using the information we collect in the test executions we run on the system.

During each test execution, we collect the state reachability information about the observed (partial) states and build a state transition graph. Starting from the initial system state, we observe the system states that are reached after running *a round of the protocol*. The execution of the protocol gives us a sequence of system states, where we transit from one state to another by running a protocol round. By recording these transitions, we can effectively build a graph that represents the possible system states and transitions that can occur during the execution of the protocol.

We maintain a single state transition graph $G = \langle S, T \rangle$ where S keeps the set of observed system states and T corresponds to the transitions between these states. The graph contains an edge from states s_1 to s_2 if we observed a transition from state s_1 to s_2 during one of the test executions. The state transition graph summarizes the set of states and the transitions between them that are encountered in a set of executions. We update the graph after each test execution with information about the new states and transitions. We made the choice of maintaining a single transition graph to increase the chances of observing a cycle. However, this choice implies that correctly identifying a cycle does not mean simply reaching a previously observed state since it does not necessarily create a cycle in the graph. Therefore, cycles have to be explicitly searched for in our state transition graph. It is sufficient to search for cycles after the state transition graph has been updated by all executions.

Figure 5.3 illustrates the state transition graph, its maintenance, and the appearance of a cycle in a simple example. In this example, a state transition graph G_1

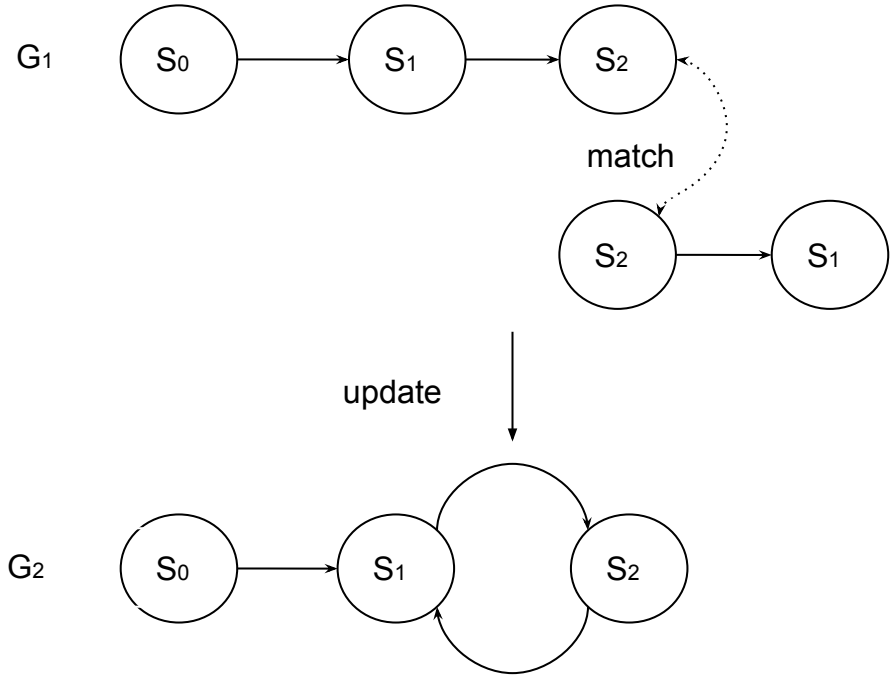


Figure 5.3: Update of the state transition graph.

keeps three system states S_0, S_1, and S_2, and their state transitions. Assuming that we run an additional test execution in which we observe that the system moves from S_2 to S_1, then we extend the graph by adding an edge from S_2 to S_3. The resulting graph G_2 contains a cycle between states S_1 and S_2, which is a potential lasso in the execution.

Note that differently from Algorithm 2, we do not check for fairness in Algorithm 5 because we only require to observe the occurrence of a lasso to report a liveness violation and do not need to check for a fair cycle or for starvation. The resulting state transition graph can then be analyzed for the presence of lassos, which represent potential cycles of system states. If a lasso is detected, we can then investigate further to confirm whether a violation exists. This technique has the advantage of not requiring a controlled scheduler, as the state transition graph can be built from the information obtained during test executions.

Algorithm 5 CheckLasso

Input: Current state *s*
Input: Current trace *Trace*
Output: Updated trace

```
1  for i ← 0 to len(Trace) do
2      if Hash(s) = Hash(Trace[i]) then
3          C ← Trace[i..len(Trace)]
4          if Hot(C) then
5              if checkCycle(s, ..) then
6                  REPORT-LIVENESS-BUG(Trace)
7              end
8          end
9      end
10 end
```

5.6 DISCUSSION

This section discusses the definitions and methods we use.

For the streamlined protocols, our hot state checks if the system does not progress due to conflicting locks. This design is based on the published attack scenarios, and we believe this definition can be modified to detect any more challenging attack scenarios.

Specifically for Sync HotStuff, our attack scenario only extends message delays. In fact, we have provided a fine-abstraction scenario to theoretically support arbitrary message delaying and re-ordering, however, it is not fully explored.

The adaptation of the partial-state caching algorithm is not perfect. We believe that there are always better solutions to apply liveness checking techniques, which is left for future work.

6 PERFORMANCE EVALUATION

We check the liveness of HotStuff, Sync HotStuff, and 2-Phase HotStuff by running a set of test executions and using the temperature and lasso detection methods to detect liveness violations in these executions. Then, we compare the performance of the temperature and lasso detection methods to a baseline bounded liveness checking method.

6.1 EVALUATION SETUP

6.1.1 Twins Framework

DiemBFT[1] officially implements Twins with its testing modules and interfaces. However, Diem is a real-world decentralized application that is not easy to revise and extend its core when the codebase is complex with lots of intertwined components.

Instead, we built our experiments on top of the relab/hotstuff framework[2]. The framework implements a chained version of HotStuff and Twins testing suite, operating in a round-by-round manner similar to DiemBFT. In each round, an honest leader proposes a leaf block that extends the longest safe chain known to it. Replicas update their states and vote for the current proposal in order to generate a new committed QC.

6.1.2 Extension to Twins

To support the experiments presented in this thesis, we extended the Twins framework for our evaluation as follows.

First, we implemented the 2-phase HotStuff and Sync HotStuff protocols based on the consensus interfaces provided by the Twins framework. We rely on the implementation of HotStuff which is provided by Twins. Note that Fast HotStuff [30] is also implemented in Twins. We do not consider Fast HotStuff in our experiments because it would be redundant with the use of HotStuff's: the liveness of both protocols has been demonstrated. Second, we implemented additional functions to capture the system state and maintain the state transition diagram. Our implementation of the state transition diagram is thread-safe to support the parallel execution of test scenarios. Third, we extended the testing framework to generate and execute scenarios that can introduce message delays in the test executions. We used message-delaying test cases to test the executions of the Sync HotStuff protocol, whose correctness depends on the timing of the delivery of the messages. Finally, we implemented the temperature and lasso detection methods for checking liveness.

1 https://github.com/diem/diem
2 https://github.com/relab/hotstuff

Protocol	Rounds	Number of scenarios to generate
HotStuff & 2-Phase HotStuff	10	$\sim 1 \times 10^9$
	20	$\sim 1.2 \times 10^{18}$
Sync HotStuff	10	$> 1 \times 10^{32}$
	20	$> 1 \times 10^{32}$

Table 6.1: The number of Twins scenarios generated under various configurations.

Protocol	Rounds	T_{mean} (s)	T_{std} (s)
2-Phase HotStuff	10	0.759	0.351
	20	2.194	1.398
HotStuff	10	0.838	0.245
	20	2.813	0.919
Sync HotStuff	10	3.100	1.062
	20	4.156	2.151

Table 6.2: Execution time of a unit test scenario in seconds under various configurations.

6.1.3 Experimental setup

Our experiments run on an HP laptop (ZBook-Studio-G5) equipped with a 2.6 GHz Intel Core i7 (12 cores), 16 GiB memory, and a UHD Graphics 630.

Scenario generator. Table 6.1 presents the number of scenarios that can be generated for different protocols and rounds. The scenario generator runs the "one-time computational cost": it tries to enumerate all the ways in which all the nodes can be distributed among different network partitions and injected message delays. All the scenarios are configured with 2 network partitions, 4 honest processes, and 1 Byzantine (twin) process, and are executed for 10 or 20 rounds (blocks). In our experiments we do not explore tackling more than a pair of twins, so we do not increase larger permutations (e.g.7 processes with 2 twins) to present more complex scenarios. For Sync HotStuff, we only need 3 honest processes, but we have to additionally enumerate the message delays. The number of possible scenarios is much larger. If it is larger than 1×10^{32}, we did not record the exact number because it was already prohibitively large. Compared with benchmark results reported in [7], we were able to prune the number of scenarios to more than 10% of that with the same configuration.

Scenario executor. Table 6.2 provides the time required for Twins to execute a unit test that involves 10 or 20 blocks with HotStuff and Sync HotStuff. For each configuration, we randomly selected and executed 1,000 Twins unit test scenarios and report the average. For HotStuff, we apply a fixed network delay of 10 ms. For the execution of Sync HotStuff, we set Δ to 50 ms. The table lists the average execution times T_{mean} and the standard deviations T_{std} for each protocol and number of blocks. The times required to execute a scenario with Sync HotStuff are larger than those of 2-Phase HotStuff and HotStuff since Sync HotStuff requires the use of longer network delays and malicious delaying by the adversary. For example, with 20 blocks, Sync HotStuff requires 4.156 s to execute a scenario, while 2-Phase HotStuff and HotStuff respectively require 2.194 s (47% less) and 2.813 s (32% less).

Time-Bounded Liveness Checking

As a baseline, we checked the bounded liveness of the consensus algorithm executions using a time bound. This method takes a time bound parameter from the programmer and reports a potential liveness violation if the system does not reach a consensus within the given specified delay.

The effectiveness of the time-bounded liveness checking method depends on the actual value of the time-bound parameter chosen by the programmer. In our evaluation, we selected three representative bound values per protocol based on its expected normal case execution time, which we have presented in Table 6.2. We utilized different values for the time-bound parameter: i) a small bound value $T_{small} = T_{mean}$; ii) an intermediate bound value $T_{mid} = T_{mean} + T_{std}$, which covers 84% of the values of a normal distribution; iii) a large bound value $T_{large} = T_{mean} + 2T_{std}$, which covers 98% of the values of a normal distribution.

For a given time-bound parameter value, an execution is expected to reach consensus before the time-bound and, if not, will be associated with a potential liveness violation. Consequently, one can expect that increasing the value of the time-bound parameter decreases the number of false positive liveness violations, but it also increases the computational overhead. Selecting the right time-bound value is, therefore, a delicate process.

6.2 EXPERIMENTAL EVALUATION

All experiments in our research are measured by their performance differences and their ability to detect vulnerabilities. We do not discuss the baseline performance between the protocols or the performance between different experimental setups or benchmarks.

6.2.1 Parameters

There are several critical parameters to measure our approach:

1. Rounds: This parameter indicates the length of each scenario. In our experiments, we set the number of rounds to 10 or 20. We do not test scenarios that are less than 10 rounds in length since we think they are not long enough to demonstrate a standard liveness problem.

2. Threshold: This parameter indicates the temperature threshold for the temperature method and the time threshold for the time-bounded checking method when we report a liveness bug.

3. Time taken: This parameter measures the total runtime to run the tests (in seconds), including the execution of all generated scenarios, initialization of networking and configurations, and analysis of the execution results.

4. Trace length: This parameter measures the average number of rounds to report a violation (or a dash "-" if there are no violations).

5. Buggy rate: This parameter calculates the ratios of safety and liveness violations detected in the executions.

6. False positive rate: This parameter calculates the false positive rate over all the reported bugs.

Other relevant parameters are not explicitly revealed in the tables:

(1). Type of lassos: This parameter indicates the number of types of lassos that exist in our state transition cache.

(2). Types of states and hot states: This parameter measures the number of distinct states captured in our state transition cache after executing all generated scenarios. The number of total states counts in thousands. Thus we believe the system is safe from hash collisions (i.e.the possibility is $4.3 * 10^{-60}$).

(3). Command requests: This parameter measures the number of commands contained in each block. To simplify the testing, we default to one command per request for each view.

(4). Scenario sources and destinations: This parameter acts as the input and output for the experiments. We generate these scenarios into an input file and feed them into the executor. The executor outputs buggy scenarios (either liveness or

Rounds	Method	Threshold	Time	Trace length	% Safety violations	% Liveness violations	% False positives
10	Temperature	5	17 min 3 s	-	0	0	0
	Lasso detection	-	17 min 16 s	-	0	0	0
	Small-Timeout	0.8s	14 min 50 s	8	0	98.8	100
	Mid-Timeout	1.2s	20 min 23 s	9	0	96.4	100
	Large-Timeout	1.6s	31 min 43 s	10	0	94.3	100
20	Temperature	5	52 min 26 s	-	0	0	0
	Lasso detection	-	57 min 3 s	-	0	0	0
	Small-Timeout	2.8s	48 min 43 s	15	0	95.7	100
	Mid-Timeout	3.8s	1 h 5 min 12 s	18	0	94.1	100
	Large-Timeout	4.8s	1 h 19 min 34 s	20	0	90.6	100

Table 6.3: Liveness and safety violations detected with the temperature checking, lasso detection, and bounded liveness methods on the executions of the HotStuff protocol.

safety bugs) into a new JSON log file. This file should be configured the same as the input JSON file so that these scenarios can be re-evaluated.

6.2.2 False positives

False positive liveness violations can be detected based on the analysis of the replicas' local variables, such as their view number, prepared and locked blocks. However, the analysis of the remaining positives requires a thorough manual evaluation of the execution by developers. Indeed, this analysis is not straightforward, because network partitions and message drops might prevent the processes' view numbers to not be synchronized. We observe that the identification of false positives consists in distinguishing whether there are really two conflicting chains in the system so that it can no longer make progress. In our evaluation, we, therefore, consider an execution to be a false positive if we cannot identify two conflicting locked blocks in the process states it generates, or if there are two conflicting locked blocks such that one is extending the other for HotStuff. In the latter case, the SafeNode predicate in HotStuff would still allow progress to be made.

6.2.3 Results

Checking HotStuff's Liveness

Table 6.3 lists the results of testing the HotStuff protocol with 10,000 randomly selected Twins test scenarios and checking the liveness of the executions using the temperature and lasso-detection based methods, along with the baseline time-bound checking. For the safety property, we checked the agreement of the protocols by comparing their executed blocks. For the liveness property, we used time-bound and temperature methods to check whether the liveness properties are satisfied within a bounded duration of execution (bounded by temperature and time, respectively) and the lasso detection method to check whether the system can stay in a cycle of hot system states and therefore does not satisfy its property.

For the temperature method, we estimated a temperature threshold equal to 5 rounds, i.e., we report a violation if the system states in a hot state for 5 rounds. We use the same temperature threshold for 2-Phase HotStuff and Sync HotStuff protocols. With this temperature threshold value, the temperature-based and the lasso detection-based methods did not report any violations for HotStuff (i.e., 0 in the safety and liveness violations columns).

On the other hand, the time-bounded liveness checking baseline reported many potential liveness violations, where the executions could not reach a consensus in

Rounds	Method	Threshold	Time	Trace length	% Safety violations	% Liveness violations	% False positives
10	Temperature	5	16 min 25 s	9	0	0.23	0
	Lasso detection	-	16 min 19 s	8	0	0.42	0
	Small-Timeout	0.8s	14 min 21 s	8	0	77.8	98.2
	Mid-Timeout	1.2s	19 min 44 s	9	0	74.6	94.5
	Large-Timeout	1.6s	30 min 11 s	10	0	58.6	88.6
20	Temperature	5	51 min 13 s	12	0	1.92	0
		10	53 min 30 s	17	0	0.74	0
		15	54 min 2 s	20	0	0.17	0
	Lasso detection	-	52 min 26 s	13	0	2.04	0
	Small-Timeout	2.2s	38 min 52 s	12	0	78.8	97.6
	Mid-Timeout	3.6s	59 min 12 s	16	0	66.4	90.4
	Large-Timeout	5.0s	1h 20 min 33 s	20	0	46.5	77.9

Table 6.4: Liveness and safety violations detected with the temperature checking, lasso detection, and bounded liveness methods on the executions of the 2-Phase HotStuff protocol.

the given amount of time. The results for time-bounded liveness checking show that using a small timeout reports a lot of violations. For example, with 20 rounds and the small timeout value, the time-bounded liveness checking method reported that 78.8% of the executions contain liveness violations. However, 95.7% of those are false positives where consensus has not been reached within the allocated number of rounds because of network partitions. In fact, it is very likely that the Twins test generator produces scenarios that cannot gather a quorum of votes because of network partitions and lack of leader replacement. Most of such test scenarios have not even completed their executions before timing out, and then they are labeled as violations. Albeit fewer, using a larger timeout bound still leads to reporting a high number of false positive liveness violations in which the system does not even enter a hot state. For example, still with 20 rounds, using the large timeout value decreases the proportion of executions that contain liveness violations to 90.6%. Overall, time-bounded checking reports them as potential liveness violations since these scenarios cannot reach a consensus due to frequent network partitions.

Checking 2-Phase HotStuff's Liveness

Table 6.4 lists the results we obtained by executing randomly selected unit test scenarios on 2-Phase HotStuff, which is known to violate liveness in certain scenarios. Similar to the results for HotStuff, we observe that time-bounded liveness Similar to the results for HotStuff, we observe that time-bounded liveness checking methods provide a higher amount of false positives than temperature and lasso-based methods. To evaluate the effect of the temperature value on the amount of reported false positives, we ran the tests for a varying number of temperature bounds (5, 10, 15) and checked if the system execution can reach those temperature bounds in the execution of 20 rounds. We observed that increasing the temperature threshold increases the required time to complete the test executions while it reduces the amount of reported false positives. For example, replacing the small timeout by the large timeout with 20 rounds increased the computation time from 38 min 52 s to 1 h 20 min 33 s, and reduced the proportion of false positives from 97.6% to 77.9%. Similarly, the time taken for the time-bounded checking also increases with the time-bound. For the lasso detection method, the execution time does not depend on a predefined bound but on the execution time of the test case together with the time required for cycle detection.

Rounds	Method	Threshold	Time	Trace length	% Safety violations	% Liveness violations	% False positives
10	Temperature	5	6 min 54 s	7	3.1	1.8	2.2
	Lasso detection	-	6 min 38 s	6	2.6	1.3	2.6
	Small-Timeout	3.1s	5 min 49 s	7	0.3	33.6	96.4
	Mid-Timeout	4.1s	6 min 41 s	8	2.5	32.4	92.7
	Large-Timeout	5.1s	8 min 37 s	10	2.6	20.7	84.8
20	Temperature	5	8 min 36 s	9	1.7	1.9	3.6
		10	9 min 4 s	14	1.1	0.5	0
		15	9 min 12 s	18	0.7	0.1	0
	Lasso detection	-	8 min 42 s	8	2.1	1.6	3.4
	Small-Timeout	4.2s	7 min 2 s	7	0	34.2	95.3
	Mid-Timeout	6.4s	10 min 24 s	15	0.3	24.2	91.2
	Large-Timeout	8.6s	14 min 9 s	20	1.6	16.7	81.2

Table 6.5: Liveness and safety violations detected with the temperature checking, lasso detection, and bounded liveness methods on the executions of the Sync HotStuff protocol.

Checking Sync HotStuff's Liveness

Table 6.5 shows the execution results of testing Sync HotStuff (eprint version 20190312:115828) with Δ set to 50ms. Overall, our findings for testing Sync HotStuff are mostly similar to the observations we made with HotStuff (Table 6.3) and 2-Phase HotStuff (Table 6.4). One significant difference, however, is the fact that this version of Sync HotStuff can violate both safety and liveness in certain test scenarios. For example, we found that the lasso detection method identified that 2.1% and 1.6% of the executions respectively contained safety and liveness violations with 20 rounds. Additionally, we have observed slightly more false positive liveness violations in Sync HotStuff than with HotStuff and 2-Phase HotStuff. We believe that this higher false positive rate comes from the adaptations we had to make to run Sync HotStuff on top of Twins.

6.3 DISCUSSION

Our evaluation shows that time-bounded liveness checking introduces a significant number of false positives when applied to streamlined protocols, as compared to temperature-based and lasso-based techniques. We believe that most of the attacks examined for **RQ1** can be tested by flexibly adapting the abstractions and methods developed for **RQ2** and **RQ3**. We find that using a small timeout for liveness checking is not practical, as it fails to detect actual violations; rather, it detects a considerable number of false positives.

Even when increasing the timeout duration to a larger value, the rate of false positives remains high, and the execution time significantly increases. In contrast, temperature-based and lasso-based techniques utilize the concept of hot states to detect the occurrence of a violation, making them more precise and reducing the likelihood of false positives.

7 CONCLUSION

7.1 CONCLUSIONS

This thesis aims to answer the following research questions:

RQ1 What are recently-detected BFT attacks? Can testing methods be used to detect some interesting and challenging attacks?

RQ2 How can we adapt existing liveness checking techniques for testing blockchain systems?

RQ3 Can the Twins testing system be extended to incorporate these liveness checking techniques? Can timing violations due to Byzantine faults be detected?

In Chapter 4, we delve into BFT protocols, and undertake an in-depth analysis of their susceptibility to safety or liveness violations, in order to respond to **RQ1**. We meticulously examine various attacks existing in BFT consensus algorithms, and we believe that our findings cover most Byzantine behaviors.

Chapter 5 addresses **RQ2** by scrutinizing the existing liveness violations in streamlined Byzantine consensus protocols, specifically, the HotStuff protocol family. We found out that existing time bounded checking methods generate a high number of false positives. To overcome this limitation, we have adopted temperature and lasso-detection liveness-checking techniques to streamlined Byzantine consensus protocols, for the first time. We focused on the HotStuff protocol family and propose necessary definitions of system state abstractions (namely partial and hot states).

To address **RQ3**, we demonstrate that the implementation of our methods on top of the Twins testing framework successfully detects the liveness violations in 2-Phase HotStuff and in an early version of Sync HotStuff.

Both the temperature and the lasso-detection methods are shown to be practical for the liveness-checking of blockchain consensus algorithms. Our results indicate that these methods identify liveness violations with fewer false positives than the time-bounded liveness checking baseline. However, adequately selecting the threshold of the temperature-based method is a delicate task that can lead to false positives or high computational overhead, while lasso detection might be more memory intensive for complex systems.

7.2 LIMITATIONS AND FUTURE WORK

This section discusses the limitations of our designs and experiments and highlights potential areas for future work.

Extending attack coverage. We have only investigated a subset of the Byzantine behaviors, and liveness checking techniques have not been extended to other BFT testing systems. Our study focused on streamlined BFT blockchain protocols while non-lock-step asynchronous protocols have been excluded. Exploring and detecting liveness attacks in diverse protocols would be an exciting direction for future research.

Refining message timing. Our approach to message timing is limited to scheduling messages at a relatively coarse granularity. Further research could refine the approach by considering a more detailed message processing model and developing a

more precise algorithm to detect liveness violations in BFT protocols. For example, future work could explore more sophisticated attack scenarios where Gasper's adversary arbitrarily controls multiple Twins pairs to balance votes to specific replicas, which may require more fine-grained timing data and precise algorithms to capture the behavior.

BIBLIOGRAPHY

[1] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and Jean-Philippe Martin. "Revisiting fast practical byzantine fault tolerance". In: *ArXiv* (2017). URL: https://arxiv.org/abs/1712.01367.

[2] I. Abraham, G. Gueta, D. Malkhi, and Jean-Philippe Martin. "Revisiting Fast Practical Byzantine Fault Tolerance: Thelma, Velma, and Zelma". In: *ArXiv* (2018). URL: https://arxiv.org/pdf/1801.10022.pdf.

[3] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and Maofan Yin. "Sync HotStuff: Simple and Practical Synchronous State Machine Replication". In: *IEEE Symposium on Security and Privacy* (2020).

[4] V. Buterin et al. "Combining GHOST and Casper". In: *Preprint, arXiv:2003.03052* (2020).

[5] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. "Lineage-driven Fault Injection". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 331–346. DOI: 10.1145/2723372.2723711. URL: https://doi.org/10.1145/2723372.2723711.

[6] Y. Amoussou-Guenou, A. Del Pozzo, and M. Potop-Butucaru. "Correctness and Fairness of Tendermint-core Blockchains". In: *CoRR* 20.4 (2002), pp. 398–461.

[7] S. Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. "Brief Announcement: Twins – BFT Systems Made Robust". In: *35th International Symposium on Distributed Computing (DISC 2021)*. Ed. by Seth Gilbert. Vol. 209. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 46:1–46:4. ISBN: 978-3-95977-210-5. DOI: 10.4230/LIPIcs.DISC.2021.46. URL: https://drops.dagstuhl.de/opus/volltexte/2021/14848.

[8] C. Berger, H. P. Reiser, and A. Bessani. "Making Reads in BFT State Machine Replication Fast, Linearizable, and Live". In: *2021 40th International Symposium on Reliable Distributed Systems (SRDS)* (2021).

[9] A. Biere, C. Artho, and V. Schuppan. "Liveness Checking as Safety Checking". In: *Electronic Notes in Theoretical Computer Science* 66 (2002). URL: http://www.elsevier.nl/locate/entcs/volume66.html.

[10] Silvia Bonomi, Jérémie Decouchant, Giovanni Farina, Vincent Rahli, and Sébastien Tixeuil. "Practical Byzantine reliable broadcast on partially connected networks". In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2021, pp. 506–516.

[11] E. Buchman, J. Kwon, and Z. Milosevic. "The latest gossip on bft consensus". In: *Tech. rep., Tendermint* (2018). URL: https://arxiv.org/abs/1807.04938.

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

[13] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. "Secure and efficient asynchronous broadcast protocols". In: *CRYPTO 2001: Proceedings of the 21st International Conference on Cryptology* (2001), pp. 524–451.

[14] C. Cachin and M. Vukoli´c. "Blockchain consensus protocols in the wild". In: (2019). URL: https://ethresear.ch/t/analysis-of-bouncing-attack-on-ffg/6113.

[15] Tong Cao, Jérémie Decouchant, Jiangshan Yu, and Paulo Esteves-Verissimo. "Characterizing the impact of network delay on bitcoin mining". In: *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2021, pp. 109–119.

[16] Tong Cao, Jiangshan Yu, Jérémie Decouchant, Xiapu Luo, and Paulo Verissimo. "Exploring the monero peer-to-peer network". In: *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer. 2020, pp. 578–594.

[17] M. Castro and B. Liskov. "Practical Byzantine fault tolerance and proactive recovery". In: *ACM Transactions on Computer Systems* 20.4 (2002), pp. 398–461.

[18] K. Claessen and N. Sorensson. "A liveness checking algorithm that counts". In: *2012 Formal Methods in Computer-Aided Design (FMCAD)* (2012).

[19] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. Ed. by Jennifer Rexford and Emin Gün Sirer. USENIX Association, 2009, pp. 153–168. URL: http://www.usenix.org/events/nsdi09/tech/full%5C_papers/clement/clement.pdf.

[20] C. Copeland and H. Zhong. "Tangaroa: A Byzantine fault tolerant raft". In: *Class project in Distributed Systems, Stanford University* (2014). URL: http://www.scs.stanford.edu/14au-cs244b/labs/projects/copeland_zhong.pdf.

[21] T. Crain, V. Gramoli, M. Larrea, and M. Raynal. "Dbft: Efficient leaderless byzantine consensus and its application to blockchains". In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)* (2018), pp. 1–8.

[22] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. "DAMYSUS: streamlined BFT consensus leveraging trusted components". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 1–16.

[23] Diem. *Diem*. URL: https://github.com/diem/diem.

[24] C. Dragoi, C. Enea, B. Kulahcioglu Ozkan, R. Majumdar, and F. Niksic. "Testing consensus implementations using communication closure". In: *Proceedings of the ACM on Programming Languages* 210 (2020), pp. 1–29.

[25] Cezara Dragoi, Constantin Enea, Srinidhi Nagendra, and Mandayam Srivas. "A Domain Specific Language for Testing Consensus Implementations". In: *CoRR* abs/2303.05893 (2023). DOI: 10.48550/arXiv.2303.05893. arXiv: 2303.05893. URL: https://doi.org/10.48550/arXiv.2303.05893.

[26] C. Dwork, N. Lynch, and L. Stockmeyer. "Consensus in the presence of partial synchrony". In: *Journal of the ACM* 35.2 (1988), pp. 288–323.

[27] R. Fernandez, R. Martins, and J. Soares. "ZERMIA - A Fault Injector framework for testing Byzantine Fault Tolerance protocols". In: *NSS2021, 15th International Conference on Network and System Security* (2021).

[28] M. J. Fischer, N. A. Lynch, and M. S. Paterson. "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM* 32.2 (1985), pp. 374–382.

[29] G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. "SBFT: A Scalable and Decentralized Trust Infrastructure". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2019).

[30] M. M Jalalzai, J. Niu, and C. Feng. "Fast-hotstuff: A fast and resilient hotstuff protocol". In: *arXiv preprint arXiv:2010.11454* (2020).

[31] Jepsen. *Tendermint 0.10.2*. URL: https://jepsen.io/analyses/tendermint-0-10-2.

[32] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. "Life, death, and the critical transition: Finding liveness bugs in systems code". In: NSDI. 2007.

[33] Kyle Kingsbury. *Jepsen tests for Cassandra 2.0.0*. URL: https://aphyr.com/posts/294-call-me-maybe-cassandra.

[34] Kyle Kingsbury. *Jepsen tests for etcd 3.4.3*. URL: https://jepsen.io/analyses/etcd-3.4.3.

[35] Igor Konnov, Marijana Lazic, Ilina Stoilkovska, and Josef Widder. "Survey on Parameterized Verification with Threshold Automata and the Byzantine Model Checker". In: *Log. Methods Comput. Sci.* 19.1 (2023). DOI: 10.46298/lmcs-19(1:5)2023. URL: https://doi.org/10.46298/lmcs-19(1:5)2023.

[36] Igor Konnov and Josef Widder. "ByMC: Byzantine Model Checker". In: *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part III*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 11246. Lecture Notes in Computer Science. Springer, 2018, pp. 327–342. DOI: 10.1007/978-3-030-03424-5_22. URL: https://doi.org/10.1007/978-3-030-03424-5%5C_22.

[37] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. "Zyzzyva: speculative Byzantine fault tolerance". In: *Technical Report UTCS-TR-07-40, University of Texas at Austin, Austin, TX, USA* (2007).

[38] L. Lamport. "Paxos Made Simple". In: *ACM SIGACT News (Distributed Computing Column)* 121 (2001), pp. 51–58.

[39] L. Lamport. *The TLA home page*. URL: https://lamport.azurewebsites.net/tla/tla.html.

[40] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* (1982), pp. 382–401. URL: https://www.microsoft.com/en-us/research/publication/byzantine-generalsproblem/.

[41] H. Lee, J. Seibert, E.Hoque, C. Killian, and C. Nita-Rotaru. "Turret: A platform for automated attack finding in unmodified distributed system implementations". In: *2014 IEEE 34th International Conference on Distributed Computing Systems* (2014), pp. 660–669.

[42] Daniel LOK. "Modelling and Testing Composite Byzantine-Fault Tolerant Consensus Protocols". In: *Capstone Final Report for BSc (Honours) in Mathematical, Computational, and Statistical Sciences, YaleNusCollege* (2019).

[43] R. Majumdar and F. Niksic. "Why is random testing effective for partition tolerance bugs?" In: *Proceedings of the ACM on Programming Languages* 2.46 (2018), pp. 1–24.

[44] J-P Martin and L. Alvisi. "Fast Byzantine Consensus". In: *IEEE Transactions on Dependable and Secure Computing* 3 (2006), pp. 202–215.

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

[46] A. Momose and J. Cruz. "Force-Locking Attack on Sync Hotstuff". In: *IACR Cryptology ePrint Archive* (2020).

[47] R. Mudduluru, P. Deligiannis, A. Desai, A. lal, and S. Qadeer. "Lasso Detection using Partial-State Caching". In: *2017 Formal Methods in Computer Aided Design (FMCAD)* (2017).

[48] Madanlal Musuvathi and Shaz Qadeer. "Fair stateless model checking". In: *ACM SIGPLAN Notices* 43.6 (2008), pp. 362–371.

[49] R. Nakamura. *Apollo - the Concord BFT engine's system testing framework*. URL: https://github.com/vmware/concord-bft/tree/master/tests/apollo.

[50] J. Neu, E. Tas, and D. Tse. "Ebb-and-Flow Protocols: A Resolution of the Availability-Finality Dilemma". In: *IEEE Symposium on Security and Privacy 2021* (2021).

[51] J. Neu, E. Tas, and D. Tse. "Two Attacks On Proof-of-Stake GHOST/Ethereum". In: *Preprint* (2022). URL: https://arxiv.org/pdf/2203.01315.pdf.

[52] D. Ongaro and J. K. Ousterhout. "In search of an understandable consensus algorithm". In: *Proc. USENIX Annual Technical Conference* (2014), pp. 305–319.

[53] N. Shrestha and M. Kumar. "Revisiting hBFT: EZBFT: A Decentralized Byzantine Fault Tolerant Protocol with Speculation". In: *ArXiv* (2019). URL: https://arxiv.org/pdf/1909.03990.pdf.

[54] N. Shrestha and M. Kumar. "Revisiting hBFT: Speculative Byzantine Fault Tolerance with Minimum Cost". In: *ArXiv* (2019). URL: https://arxiv.org/pdf/1902.08505.pdf.

[55] Douglas Simoes Silva, Rafal Graczyk, Jérémie Decouchant, Marcus Völp, and Paulo Esteves-Verissimo. "Threat adaptive byzantine fault tolerant state-machine replication". In: *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2021, pp. 78–87.

[56] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. "BFT Protocols Under Fire". In: *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*. Ed. by Jon Crowcroft and Michael Dahlin. USENIX Association, 2008, pp. 189–204. URL: http://www.usenix.org/events/nsdi08/tech/full%5C_papers/singh/singh.pdf.

[57] Y. Sompolinsky and A. Zohar. "Secure high-rate transaction processing in bitcoin". In: *International Conference on Financial Cryptography and Data Security* (2015), pp. 507–527.

[58] Tendermint. *Tendermint*. URL: https://github.com/tendermint/tendermint.

[59] T. Tsuchiya and A. Schiper. "Verification of consensus algorithms using satisfiability solving". In: *Distributed Computing* (2011), pp. 341–358.

[60] Vitalik. *Casper FFG with one message type, and simpler fork choice rule*. URL: https://ethresear.ch/t/casper-ffg-with-one-message-type-and-simpler-fork-choice-rule/103.

[61] VMware. *Apollo - the Concord BFT engine's system testing framework*. URL: https://github.com/vmware/concord-bft/tree/master/tests/apollo.

[62] Q. Wang, R. Li, S. Chen, and Y. Xiang. "Formal Security Analysis on dBFT Protocol of NEO". In: (2021). DOI: arxiv-2105.07459. URL: https://arxiv.org/abs/2105.07459.

[63] Y. Wang. "Another Look at ALGORAND". In: (2020). URL: https://arxiv.org/pdf/1905.04463.pdf.

[64] Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. "Randomized Testing of Byzantine Fault Tolerant Algorithms". In: 7.OOPSLA1 (Apr. 2023). DOI: 10.1145/3586053. URL: https://doi.org/10.1145/3586053.

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

[66] Jiangshan Yu, David Kozhaya, Jeremie Decouchant, and Paulo Esteves-Verissimo. "Repucoin: Your reputation is your power". In: *IEEE Transactions on Computers* 68.8 (2019), pp. 1225–1237.