Gemini: BFT Systems Made Robust

Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, Dahlia Malkhi

Novi

Abstract

This paper presents Gemini, a principled strategy for effectuating Byzantine attack scenarios at scale in Byzantine Fault Tolerant (BFT) systems and examining their behavior.

Gemini builds a thin wrapper over an existing distributed system designed for Byzantine tolerance. To emulate material, interesting attacks by a Byzantine node, it instantiates *twin* copies of the node instead of one, giving both twins the same identities and network credentials. To the rest of the system, the twins appear indistinguishable from a single node behaving in a "questionable" manner. This approach generates many interesting Byzantine behaviors, including equivocation, double voting, and losing internal state, while forgoing uninteresting behaviors that are trivially rejected by honest nodes, such as producing semantically invalid messages. Building on this idea, Gemini can systematically generate Byzantine attack scenarios at scale, execute them in a controlled manner, and examine their behavior.

To systematically generate Byzantine attacks, Gemini scenarios iterate over protocol rounds and vary the communication patterns among nodes. Despite this being inherently exponential, one new attack and several known attacks were materialized by Gemini in the arena of consensus protocols. In all cases, protocols break within fewer than a dozen protocol rounds, hence it is realistic for the Gemini approach to expose the problems. In two of these attacks, it took the community more than a decade to discover protocol flaws that Gemini would have surfaced within minutes. Additionally, Gemini has been incorporated into a production setting in which it can execute 44M Gemini-generated scenarios daily. Whereas the system at hand did not manifest errors, subtle safety bugs that were deliberately injected for the purpose of validating the implementation of Gemini itself were exposed within minutes.

1 Introduction

Byzantine Fault Tolerant (BFT) protocols introduced in the seminal work of Lamport et al. [19] are designed to withstand attacks or arbitrary malfunction of internal nodes. However, creating Byzantine attacks in order to validate a BFT system is challenging: (*i*) Byzantine behavior is unconstrained and (*ii*) developers may be tainted by what they think that the system is designed to tolerate. Last, as a pragmatical consideration, developing code that implements Byzantine attacks might be risky.

This paper introduces Gemini, a principled approach for effectuating Byzantine attacks on BFT systems and examining their behavior. Instead of coding incorrect behavior, Gemini runs faulty nodes in two (or generally, k) parallel universes in tandem. Both instances have the same credentials/signing-keys and run autonomously. Thus, for example, both nodes can send messages in the same protocol round, but these messages will carry conflicting proposals or votes; to the rest of the system, this twins behavior will appear indistinguishable from an equivocating behavior by a single node. In another example, one twin may send a vote in one round, and its twin will "forget" it has voted in the next round; again, to the rest of the system, this will appear indistinguishable from a single node violating safety rules.

Gemini is based on the insight that most interesting Byzantine attacks are internal and leverage knowledge of the expected behavior of participants, hence they go unnoticed. In particular, Gemini foregoes trivial attacks such as sending semantically invalid messages, or sending a message without justification. Thus, leveraging existing code, Gemini can automatically cover material Byzantine behaviors. Indeed, Section 3 demonstrates one new, and several known, attacks on BFT protocols materialized as Gemini attacks.

Crucially, in all cases, protocols break within fewer than a dozen protocol steps, hence Gemini successfully exposes them. Note that Gemini scenarios systematically iterate over protocol rounds and vary the communication patterns among nodes. While inherently exponential, in the above attacks, it took Gemini only minutes to discover protocol flaws that in some cases, took the community decades to surface.

Gemini has been incorporated into a production setting in which Gemini can execute 44M Gemini-generated scenarios

daily. Whereas the system at hand did not manifest errors, subtle safety bugs that were deliberately injected for the purpose of validating the implementation of Gemini itself were exposed within minutes

Gemini & attacks on BFT replication. Our work on Gemini arises in the context of BFT replication protocols. In this domain, several worrisome safety and liveness vulnerabilities were exposed recently [1, 22] in both known protocols [17, 21] and in new ones [2].

One reason that BFT replication lends itself well to analysis via Gemini is as follows. A common paradigm underlying practical BFT replication protocols is a view-by-view design. Each view is driven by a designated leader proposing to the nodes and going through voting rounds by the nodes. If a leader is successful, a consensus decision is reached in the view. If not, nodes give up after a timeout and move to the next view. Transitioning to the new view/leader is tricky: A new leader must discover if the previous leader was successful, but it may be able to communicate only with a subset of the nodes. The transition logic turns out to be the source of problems in all the above cases, hence exposing the flaw requires only one or two leader rotations.

Gemini implementation.

Gemini effectuates a Byzantine attack by a Byzantine node via instantiating *twin* copies of the node instead of one, giving both twins the same identities and network credentials. To the rest of the system, the twins appear indistinguishable from a single node behaving in a "questionable" manner. Gemini minutely interacts with existing code to control message delivery and schedule various coarse-steps such as protocol rounds. It is practical to deploy in real systems as it uses existing node code, easily keeping up with an evolving software project.

We built an attack generator based on the Gemini approach in the LibraBFT open-source project, the BFT replication core of the Diem payment system [13]. Implementing Gemini in LibraBFT consists of two principal parts.

The first is a *scenario executor* that deploys a network configuration where some nodes have twins. The scenario executor hides twins behind a thin multiplexing wrapper; to the rest of the system, each pair of twins appear as a single entity. The scenario executor controls the scheduling of message deliveries according to a prescribed scenario. This is accomplished through a transport emulator in the LibraBFT repository called *Network Playground*.

The second part is a *scenario generator*. The scenario generator enumerates scenarios by varying the number of nodes and the message delivery schedule, then feeding the scenarios to the scenario executor. We describe in the paper several strategies for drastically reducing the number of scenarios through aggressive trimming of symmetrical scenarios. Among these strategies, one minimally "opens" the LibraBFT implementation and lets the scenario executor determine when a node acts as a leader in the consensus protocol. This removes duplicate scenarios that differ only in their leaders. Section 7 reports on our experience with Gemini in LibraBFT.

Coverage. What attacks does the Gemini approach capture? Developing a rigorous theory that answers this question is an intriguing question left for future work. Here, we provide anecdotal evidence of coverage in three forms:

(*i*) Section 2 brings intuition and experience of several decades of work in the field. There are only a handful of ways in which a Byzantine attacker can materially deviate from the safety rules imposed by its protocol. For example, it can equivocate and send different proposals to different groups of recipients, or it can pretend it did not send/receive a message and propose or vote in a manner that conflicts with such a message.

(*ii*) Evaluating within the LibraBFT production system Section 7 provides compelling validation of the Gemini approach. Whereas the system at hand did not manifest errors, self-injected subtle safety bugs—for the purpose of validating the implementation of Gemini itself—were exposed within minutes. In particular, we created a simple safety-violating setting by deploying $f+1$ (instead of f) nodes with Gemini, which led to an expected consistency violation within seconds. We further injected three subtle logical bugs, which only slightly deviated from the original specification. In all three cases, with only f twins (faults), Gemini successfully exposed safety violations.

(*iii*) Section 3.1 shows how Gemini can instantiate a safety violation in a new protocol described in a recent manuscript [15]. This highlights the importance of systematically analyzing the properties of BFT protocols using Gemini to expose subtle flaws. In Section 3 several known attacks on BFT consensus protocols are reinstated by the Gemini approach. These attacks cover a broad spectrum of vulnerabilities, e.g., safety, liveness, timing, and responsiveness.

In some protocol steps, a node may wait for messages to determine its next action. Under Gemini, the node is forced to act according to the messages it received, as if the node provided a justification for each step in form of the history of messages it received. Deviating from this behavior was not required to reinstate any of the attacks discussed in Section 3, though in principle, various deviating behaviors would not be covered by Gemini. Another coverage challenge emerges in synchronous protocols because a node behavior may be based on real time. In such protocols, Gemini essentially forces a node to behave in a timely manner. We tackle this case in one of the attacks investigated in Section 3 and demonstrate that nonetheless, a slight adaptation of the original attack reinstates the attack in Gemini. However, we do not know yet which timing attacks may not be covered. We discuss some concrete future directions in Section 9 for extending Gemini in the settings we explore as well as others.

2 Motivating the Gemini Approach

We open this section with a quick primer on the Byzantine Fault Tolerant (BFT) replication problem, and describe the notation that will be used to describe attacks through the rest of this paper. We then provide high-level intuition on why Gemini is a viable approach by showing the different kinds of Byzantine behaviors that can be captured by Gemini. (Concrete attacks using Gemini are described later in Section 3 and Section 7.1.)

BFT Replication. The goal of BFT replication is for a group of nodes to provide a fault-tolerant service through redundancy. Clients submit requests to the service. These requests are collectively sequenced by the nodes; this enables all nodes to execute the same chain of requests and hence agree on their (deterministic) output.

Except when specifically noted, we consider protocols that maintain safety against arbitrary delays in message transmissions. That is, we assume an *asynchronous network* setting. The main challenge is to drive *agreement* on a chain of requests (and their output) among all non-faulty nodes despite node failures. It is common to rely on leaders to populate the network with a unique proposal. During periods in which the leader is non-faulty and communication among the leader and non-faulty nodes is timely, this regime can drive consensus quickly. This approach is called *partial synchrony*, indicating that it maintains safety at all times and progress only during periods of synchrony.

In the Byzantine fault model, a node may crash or arbitrarily deviate from the protocol. In this setting, a BFT replication system implements a fault tolerant service via n nodes, of which a threshold $f < n/3$ may be Byzantine. As Byzantine behavior is defined rather vaguely, there is no principled way to evaluate BFT systems. Gemini is a new approach to systematically generate Byzantine attacks. The main idea of Gemini is the following: running two (generally, up to k) autonomous instances of a node that both use correct code and share the same identity, allows us to emulate most interesting Byzantine attacks. Two nodes share the same identity when they share the same credentials and signing keys.

Notation. Nodes are represented by capital alphabets (e.g., A) and the twin of a node is represented by the same alphabet with the prime symbol (e.g., A'). When referring to a set of nodes, we enclose them in parentheses e.g., (A, B, B'). We underline a node that is serving as the leader, e.g., $\underline{A}$. The adversary can delay and filter messages between nodes. We denote partitions of nodes by enclosing them in braces, e.g., $P_1 = \{A, B, C, D\}$ and $P_2 = \{E, F, G\}$, and reserve the capital letter P to denote them. Additionally, to show messages allowed in a given direction, we use the symbols $\rightarrow$ and $\leftrightarrow$. For example, $A \rightarrow (B, C)$ means A can send messages to B and C; similarly, $A \leftrightarrow P_2$ means A can send messages to and receive messages from any node of the partition P_2.

The scenarios described below use a network configuration of 7 nodes, (A, B, C, D, E, F, G). Byzantine nodes have twins denoted with $'$, as in F', G'. To experiment with any of the deviating behaviors described below, one can increase the number of Byzantine faults to $f + 1$ (say E, F, G have twins E', F', G') and expect to see conflicting commits.

Equivocation. A quintessential Byzantine behavior is for a node to *equivocate*. That is, in the same step, a Byzantine node might send different messages to different recipients.

Gemini covers equivocation by splitting honest nodes between two partitions, each one communicating with only one twin of each pair. For example, we can split the system into $P_1 = \{A, B, C, D, \underline{F}\}, P_2 = \{C, D, E, \underline{F'}, G\}$. The leader(s) F and F' execute correct leader code but nevertheless may generate conflicting proposals due to different inputs or randomness seeds. If there is a protocol flaw then these conflicting proposals could respectively commit in P_1 and P_2, hence safety breaks.

Amnesia. An important role that nodes have in agreement protocols is *vote* for a single proposal per view. However, a Byzantine node might vote for a proposal and then 'forget' that it has voted and vote again. Gemini covers amnesia by letting one of the twins vote on one proposal. Since the other twin is oblivious to the vote happening, it may nevertheless—albeit executing correct code—vote on a different proposal.

More concretely, as in the scenario above, we can split the nodes into two partitions, $P_1 = \{A, B, E, F, G\}, P_2 = \{C, D, E, F', G'\}$. If there is a protocol flaw then this double-voting behavior may result in conflicting commits in P_1 and P_2, hence safety breaks.

Losing internal states. Another notable deviation for Byzantine nodes is to lose their internal state, particularly a *lock* that guards a value they voted for. Gemini covers this deviation by letting one of the twins get locked on a value in one view, but in some subsequent view, bring the other twin who is ignorant that a lock exists.

More concretely, we can split the nodes into two partitions $P_1 = \{A, B, E, F, G\}, P_2 = \{C, D, E, F', G'\}$. In one view, the adversary relays messages only among P_1. In the next view, it switches to P_2, causing F', G'—albeit executing correct code—to ignore their 'previous' actions. This can repeat any number of times. If there is a protocol flaw then conflicting proposals may commit in different views, hence safety breaks.

3 Attacks Materialized in Gemini

In this section, we demonstrate one new, and several known, attacks on BFT replication protocols, expressed as Gemini scenarios. We provide insight into the attacks and defer the details of all but the linear leader-replacement attack to an appendix, due to space constraints.

3.1 New Attack

Fast-HotStuff [15] is a new protocol, described in a recent manuscript. It is similar to HotStuff [31], except with a 2-phase commit rule. The safety violation we reveal using Gemini is possible because Fast-HotStuff does not require consecutive rounds in order to commit. Specifically, QCs formed by some of the (partitioned) nodes do not reach the other nodes, resulting in two parallel branches that eventually commit two conflicting blocks. We instantiate this safety violation with Gemini (using only network partitions in a network with 4 nodes and within 11 rounds). This highlights the efficacy of systematically analyzing the properties of BFT protocols via Gemini to expose subtle flaws. More details are provided in Appendix 14.

We implemented the Fast-HotStuff consensus algorithm in a Python simulator which we release as open source[1]. The simulator then executes Gemini scenarios over the algorithm.

3.2 Reinstated Attacks

We present several known attacks on BFT protocols, expressed as Gemini scenarios. In all cases, exposing vulnerabilities requires only a small number of nodes, partitions, rounds and leader rotations. It is worth noting that later, our evaluation (Section 7) of LibGemini, Gemini implemented for LibraBFT, shows that running an automated scenario generator (Section 4.2) with these configurations would cover the described attacks within minutes. We did not undertake to re-implement all these protocols and apply a Gemini scenario generator to them; our implementation covers only LibraBFT [13].

Safety attack on Zyzzyva. Zyzzyva broke new ground in BFT replication with the introduction of an optimistic single phase "fast track" commit. Eleven years elapsed from its publication until a safety flaw in Zyzzyva was discovered [1], during which numerous research project and systems were built on it. Gemini generates a scenario that exposes the flaw with 4 nodes and two leader rotations: the first leader equivocates via a twin, and the next two leaders drop messages to/from some nodes. The details of this attack using Gemini is described in Appendix 11.

Liveness attack on FaB. FaB [21], a precursor to Zyzzyva, is a view-based protocol with an optimistic fast track. Not surprisingly, a similar problem arises in FaB due to a flawed leader replacement protocol [1], albeit manifesting as a liveness bug. Gemini exposes this bug in a short scenario with $n = 4$ and three leader rotations, leading to a complete absence of leader proposals. The detailed attack using Gemini is described in Appendix 12.

Timing attack on Sync HotStuff. *Force-Locking Attack* [22] is a timing attack on a preliminary version of a synchronous BFT protocol named Sync HotStuff [2] (which was subsequently updated to resist the attack). As before, Gemini captures this attack with only a small system size, $n = 5$, and two leader rotations. However, in order to create timing attacks, Gemini needs to be aware of timing information for protocol steps and messages deliveries. Extending Gemini with timing data is left for future work. In the specific attack at hand, course-grain timing at fixed intervals—fewer than ten—suffice to reinstate the attack. The detailed attack using Gemini is described in Appendix 13.

Non-Responsiveness attack on linear leader-replacement. Practical Byzantine Fault Tolerance (PBFT) [9] is a seminal work that was designed to work efficiently in the asynchronous setting. Carrying the classical PBFT solution to the blockchain world, Tendermint [7] and Capser [8] introduced a simplified *linear* strategy for leader-replacement. However, it has been observed [6, 30] that this strategy forgoes an important property of asynchronous protocols—*Responsiveness*—the ability of a leader to advance as soon as it receives messages from $2f + 1$ nodes.[2] Indeed, bringing linear leader-replacement approach into PBFT, we demonstrate a liveness attack using a Gemini scenario. Lack of progress is detected by observing that two consecutive views with honest leaders whose communication with a quorum is timely do not produce a decision. We present the details of this attack using Gemini in the next section.

3.3 Non-Responsiveness Attack

We now describe in more detail the non-Responsiveness attack above on linear leader-replacement. The seminal PBFT solution operates two-phase views. A simplified, linear leader-replacement works as follows. A leader proposes to extend the highest *quorum certificate* (QC) it knows. A QC is formed on a proposed value if it gathers $2f + 1$ votes from nodes. Nodes vote on the leader proposal if it extends the highest QC they know. A commit decision on the leader proposal forms if $2f + 1$ nodes form a QC, and then $2f + 1$ nodes vote for the QC. Progress is hinged on leaders obtaining the highest QC from the system, otherwise liveness is broken.

Using the notation from Section 2, the liveness attack here uses 4 replicas (D, E, F, G), where D has a twin D'. In the first view, D and D' generate equivocating proposals. Only D, E receive a QC for D's proposal. The next leader is F who proposes to re-propose the proposal by D', which E and D do not vote for because they already have a QC for that height. Only F and D' receive a QC for F's proposal. This scenario repeats indefinitely, resulting in loss of liveness. More specifically, this attack works as follows:

View 1: Initialize D and D' with different inputs v_1 and v_2.

[1] Link removed for blind review.

[2] Tendermint is a precursor to HotStuff [31] and LibraBFT [13] which operates in two-phase views, but has no Responsiveness. HotStuff/LibraBFT solve this by adding a third phase.

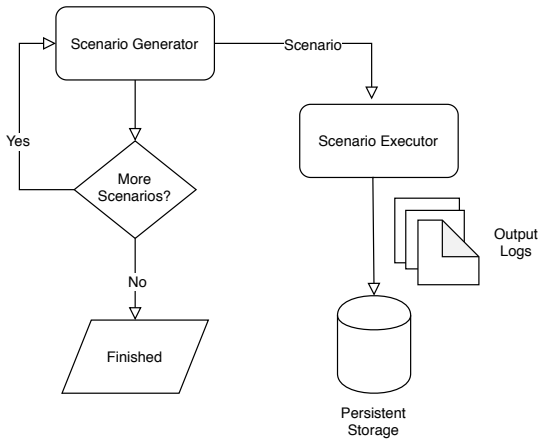


Figure 1: Gemini high-level design.

- Create the partitions $P_1 = \{\underline{D}, E, G\}$, $P_2 = \{\underline{D'}, F\}$.
- Let D and D' run as leaders for one round. D proposes v_1 to P_1 and gathers votes from P_1 creating $QC(v_1)$. D' proposes v_2 to P_2 and gathers votes but not a QC.
- Create the following partitions: $P_1 = \{D, E\}$, $P_2 = \{\underline{D'}, F\}$, $P_3 = \{\underline{G}\}$. D broadcasts $QC(v_1)$, which only reaches P_1 i.e., (D, E).

View 2: Drop all proposals from D and D' until View 2 starts.

- Remove all partitions, i.e., $P = \{D, D', E, \underline{F}, G\}$.
- Let F run as leader for one round. F re-proposes v_2 (i.e., D''s proposal in the previous round) to P. (D, E) do not vote as they already have $QC(v_1)$ for that height. F gathers votes from the other nodes and forms $QC(v_2)$.
- Create partitions $P_1 = \{D, E\}$, $P_2 = \{\underline{D'}, F\}$, $P_3 = \{\underline{G}\}$.
- F broadcasts $QC(v_2)$, which only reaches P_2.

View 3: Drop all proposals from F until View 3 starts.

- Create the partitions $P_1 = \{D, \underline{E}, G\}$, $P_2 = \{\underline{D'}, F\}$.
- Let E run as leader for one round. E proposes v_3 which extends the highest QC it knows, $QC(v_1)$. As before, E manages to form $QC(v_3)$, but as a result of a partition, the QC will only reach (D, E). Next, there is a view-change, F is the new leader, and there are no partitions. F proposes v_4 which extends $QC(v_2)$, the highest QC it knows. However, (D, E) do not vote because v_4 does not extend their highest QC i.e., $QC(v_3)$. This scenario can repeat indefinitely, resulting in the loss of liveness.

4 Systematic Scenario Generation

Whereas the previous section demonstrated manually crafted Gemini attack scenarios, this section presents a framework for systematically generating such scenarios.

Systematically and efficiently generating Gemini scenarios that provide good coverage requires tailoring to the specific BFT protocol settings. We develop the Gemini framework which generates and executes *scenarios* that describe the node and network configurations. Specifically, the Gemini framework is comprised of two components as shown in Figure 1: (*i*) the scenario executor, and (*ii*) the scenario generator. The scenario executor runs a single scenario and generates output logs, while the scenario generator produces various scenarios that are fed to the scenario executor to check for violations. The following design goals underlie the Gemini framework:

- **Generic & Modular.** Gemini is modular with respect to the particular BFT protocol implementation being analyzed, imposes as little complexity as possible on the development, and easily keeps up with code changes.
- **Parametrizable.** The network setup (i.e., the number of nodes, leaders per round, and network configuration per round) and adversarial assumptions (i.e., how many Byzantine faults are tolerated) is configurable.
- **Feasible.** Gemini allows pruning duplicate scenarios in order to provide coverage of material attacks.
- **Customizable Coverage.** The coverage of scenarios, i.e., the subset of all possible scenarios to choose for execution, is configurable by randomly sampling scenarios to run among all possible enumeration.
- **Reproducible.** Gemini writes logs to persistent storage, containing sufficient information to detect and reproduce any safety violations.

Next, we describe the two main components (Figure 1) of Gemini—the scenario executor and the scenario generator—in detail.

4.1 Scenario Executor

In every Gemini scenario, a threshold of the nodes are 'misconfigured' to have a twin instance with identical transport endpoint credential and secret keys. The Gemini scenario executor gets as input a scenario consisting of a node-set, a subset of which are marked *compromised* (representing Byzantine nodes); and a round-by-round message delivery schedule. The scenario executor sets up a network of nodes with a given number of compromised nodes and per round partitions and leaders. The compromised nodes correspond to the nodes for which the scenario executor creates twins (i.e., identical instances with the same credentials and signing keys), thereby emulating misbehavior.

As mentioned above, we address BFT replication protocols that proceed in rounds initiated by a designated leader, each round representing a state transition in the protocol's state machine replicated on each node. For each round, the scenario executor creates a given network partition and assigns given leaders to the round. The scenario executor runs the BFT protocol among nodes for a pre-specified number of

rounds, at the end of which, the scenario executor checks for violations. Specifically, protocol guarantees can be violated in two principal ways, safety and liveness. A safety violation is detected if two nodes commit to conflicting decisions. A liveness violation can be detected if the protocol fails to commit within a certain number of steps or a certain duration bound.

4.2 Scenario Generator

We build a scenario generator of round-by-round scenarios: for each round, the scenario generator enumerates possible leaders and message delivery schedules among nodes. The scenario generator produces various scenarios to be fed into the scenario executor. Each scenario represents a unique instance of executor configuration parameters, i.e., the compromised nodes and per round network partitions and leaders. Scenarios are generated systematically as follows (see notations in Section 2):

- **Step 1.** The scenario generator first produces the set of all possible partitions of nodes (called *partition scenarios*). For example, for a network of 4 nodes (A,B,C,D), possible partition scenarios (P) include $\{P_1 = \{A,D\},\{B,C\}\}$, and $P_2 = \{\{A\},\{B,C,D\}\}$. This problem relates to the *Stirling Number of the Second Kind* [26] which enumerates the ways in which a set of N nodes can be divided up into P non-empty partitions, where P ranges from N (i.e., each node is self-isolated) to 1 (i.e., fully connected network without partitions).
- **Step 2.** Next the scenario generator assigns each partition scenario to all possible leaders i.e., the set of N nodes assuming any of those can be a potential leader. For example, for the example partition scenario above $\{P_1 = \{A,D\},\{B,C\}\}$ for a network of nodes (A,B,C,D), possible leader-partition combinations include $\{\underline{A},P_1\}$, $\{\underline{B},P_1\}$, $\{\underline{C},P_1\}$, $\{\underline{D},P_1\}$. Each leader-partition combination fully describes the Gemini configuration required for each round.
- **Step 3.** The scenario generator lists scenarios by enumerating all possible ways in which the leader-partition pairs generated in the previous step can be arranged over R rounds (i.e., permutation, with or without replacement).

The scenario generator iterates over the generated scenarios linearly, and invokes the scenario executor for each scenario. For safety analysis, usually a small number of rounds ($<$ 10) suffices to expose logical bugs in the protocol. Scenario generators therefore need to enumerate a reasonable number of combinations.

Pruning scenarios. Important to the success of the approach is for the scenario generator to avoid duplicate scenarios (e.g., in symmetry or node label[3] rotation) and generate only materially different scenarios. The implementation we describe in the Evaluation section of this paper (Section 7) employs aggressively such pruning. Certain heuristics further substantially reduce the number of scenario configurations. For example, in most safety violations the set of honest parties is split into two, hence it suffices to play with two or three partitions per round. These optimizations make it feasible to cover a broad range of meaningful scenarios. For analyzing liveness, many scenarios will obviously fail to make progress because there does not exist a super-majority quorum that has reliable and timely communication among its members. Hence, for liveness analysis the scenario generator must guarantee that eventually such a quorum exists.

Message delays and timeouts. We note that the scenario generator does not address message delays and timeouts, only the dropping of messages and their relative delivery order. Because the BFT protocol may employ timers, the dropping of messages implicitly implies that relevant endpoint incur a violation of presumed bounds on transmission delays. Future work may incorporate explicit message delays into the scenario generator to check specific timing violations and also to analyze BFT protocols in the synchronous model (Section 9).

5 Overview of LibraBFT

We now shift our attention to utilizing Gemini for validating BFT replication in LibraBFT [13]. We discuss our implementation and evaluation of Gemini for LibraBFT in Sections 6 and 7. In this section, we provide an overview of LibraBFT (for details, see the technical report [27]).

LibraBFT operates in a round-by-round manner, electing leaders in each round among the nodes to balance node participation. Rounds are slightly different from conventional "views" because it takes multiple rounds to reach a decision, but leaders are rotated in each round. The leader protocol is quite simple. A leader proposes an extension to the longest chain of requests that it knows already. Usually leaders collect batches of requests to propose, referred to as blocks, hence the LibraBFT protocol forms a chain of blocks (or a blockchain). Nodes vote for a proposed block, unless it conflicts with a longer chain that they believe may have reached consensus already. Nodes send their votes to the next leader to help the leader learn the longest safe chain. If there are three consecutive blocks in the chain, B_k, B_{k+1}, B_{k+2}, which are proposed in consecutive rounds, r_k, r_{k+1}, r_{k+2}, and each block has votes from $2f+1$ nodes (gathered in a data structure called the *quorum certificate*, or QC), then the protocol has reached consensus on block B_k.

If $2f+1$ send votes to the next leader in a timely manner, a QC is formed by the leader and it sends the next proposal. Nodes maintain a timer to track progress. When the timer expires and a node still has not received a proposal, it broadcasts

[3] Nodes can have designated roles in the protocol, referred to as *node labels*. Gemini incorporates the label 'leader', which is the case for standard BFT protocols. Extensions of these protocols might have further hierarchy e.g., primary and secondary leaders. This is currently not supported, but the scenario generator can be easily extended to support different node labels.

a timeout vote on a Nil block. When a node gathers enough timeout votes to form a timeout certificate, it advances its round. Every time a round fails, timeout periods are increased, allowing lagging nodes to catch up and enabling the protocol to eventually reach a decision.

As briefly alluded to in the Introduction, the trickiest part of BFT replication is to manage leader transition. LibraBFT maintains four parameters to ensure safety, and at the same time facilitate progress: (*i*) *current_round*, the node's current round; (*ii*) *last_voted_round*, the last round for which the node voted; (*iii*) *parent_round*, the round of the block certified by the QC attached with the block being processed; (*iv*) *grandparent_round*, the parent of the block certified by the QC; and (*iv*) *preferred_round*, the highest known grandparent round. Note that as a QC serves as a pointer to the previous certified block, *parent_round* and *grandparent_round* do not need to be explicitly tracked; these can be derived from the QC carried by a block.

Upon Receiving a Proposal. Upon receiving proposal for a block, a node processes the certificates it carries, and votes for the proposed block if it satisfies a simple voting rule: If a node voted for B_{k+2}, it *prefers* the sub-tree of proposals rooted at block B_k (regardless of round numbers). A node will not vote for a block B that does not belong to its preferred sub-tree rooted at B_k, unless B's parent has votes from $2f+1$ nodes at a higher round than r_k. Concretely:

- **Safety Rule 1.** The *block_round* is greater than *last_voted_round*.
- **Safety Rule 2.** The block's *parent_round* is greater than or equal to *preferred_round*.

If the node decides to vote for the proposed block, it updates its state as follows:

- **Update Rule 1.** Update *last_voted_round* to round of the proposed block.
- **Update Rule 2.** Update the node's *preferred_round* to the proposed block's *grandparent_round* if the latter is higher.
- **Update Rule 3.** Update the node's *current_round* to the *parent_round* $+1$, if the latter is higher.

Upon Receiving a Vote. For every round, the nodes send their votes to the leader of the next round. When the leader receives a vote, it performs the following safety checks:

- **Safety Rule 3.** If a vote from the same node was previously received for the *same* block and round, the leader rejects the vote and generates a 'duplicate vote' warning.
- **Safety Rule 4.** If a vote from the same node was previously received for a *different* block but same round, the leader rejects the vote and generates an 'equivocating vote' warning.

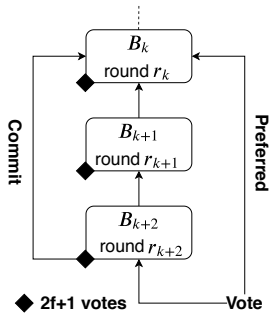


Figure 2: Consensus and preferred sub-trees in LibraBFT.

If a vote passes both these checks, the leader considers it as valid and checks if it has enough votes to form a QC. When a QC has been formed, the leader generates a new round event, broadcasts a new block proposal and updates its state.

- **Update rule 4.** When a leader gathers enough votes to form a QC, it broadcasts a new proposal and increments *current_round*.

Spoiler alert: In our evaluation in Section 7.1, we are going to deliberately modify the above rules. We will see that this enables safety violations that the Gemini framework will expose.

6 Implementation

We implemented the Gemini framework for LibraBFT, which we call LibGemini. As described in Section 4, an implementation consists of two principal ingredients, a scenario generator and an scenario executor (Figure 1). We first describe the scenario executor implementation which leverages a network emulator in LibraBFT referred to as the *network playground*. We then proceed to describe the scenario generator implementation. For completeness, the Rust code and interfaces for the main functions of LibGemini, `execute_scenario` and `scenario_generator`, are provided in Appendix 10. We are open sourcing the Rust implementation of LibGemini[4].

6.1 Scenario Executor

The LibGemini scenario executor leverages the network emulator of LibraBFT, *network playground*[5]. Network playground provides an apparatus for running single-host LibraBFT deployments, emulating a network and intercepting all messages exchanged between nodes. Scenarios can be written to manipulate the intercepted messages (e.g., by dropping certain

[4] https://github.com/libra/libra

[5] https://github.com/diem/diem/blob/master/consensus/src/network_tests.rs

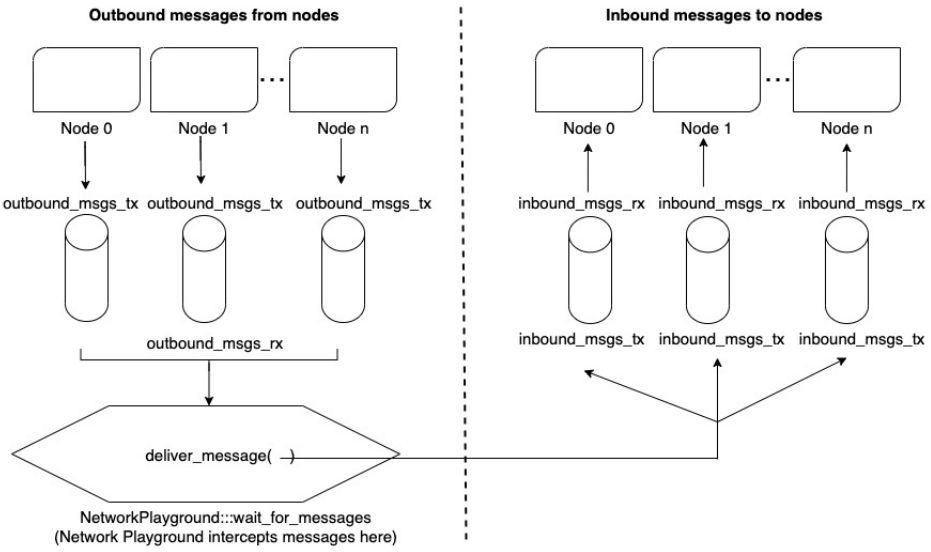


Figure 3: Design of LibraBFT's *Network Playground*.

messages) and observe node response. Figure 3 shows the design of the network playground. Nodes are represented by processes run on different threads (that run the full consensus protocol), and network links between them are expressed as Rust channels that provide asynchronous unidirectional communication between threads. In LibraBFT, nodes are identified by their *Account Address* (a public key that uniquely identifies a node). Channels are associated with their respective account addresses (nodes). When a node starts a new round, it checks whether it is leader for this round; if yes, then it generates on the fly a block to propose using a mock block generator. Each call to the mock block generator produces a different block. This has important implication for LibGemini, as we require a node and its twin to propose different blocks at the same round to emulate equivocation.

The scenario executor component (Section 4) of LibGemini is built on top of network playground. This required the following modifications and extensions to the original library:

- **Adding twins.** We wrote a new method to add nodes to the network that supports twins. The method takes 'compromised nodes' as a parameter to refer to the nodes for which to create twins. For each target node, a duplicate instance is created with the same credentials and signing keys. Consequently, in the eyes of the other nodes the compromised node and its twin are indistinguishable.

- **Inferring rounds.** LibGemini requires to apply a number of filtering policies at the round level. Network playground does not have a notion of rounds—it only supports static configurations that remain unchanged throughout protocol execution. There is no global notion of rounds in a distributed system with partial synchrony; instead, nodes have their own view of which round they are in, which they include in their messages. We enable network playground to extract round from intercepted messages and accordingly apply filtering criteria.

- **Round-based message filtering.** Network playground allows writing rules to drop intercepted messages that meet certain criteria, i.e., messages to or from specified nodes and messages of specified types e.g., votes or proposals. LibGemini extends network playground to drop intercepted messages *per round*, which allows emulating different network partitions per round. The message dropping rules treat compromised nodes and their twins differently—the rules apply to account addresses (which uniquely identify nodes), not public keys (which are the same for a target node and its twins).

- **Deterministic multi-leader election.** LibraBFT currently uses a non-deterministic leader election algorithm. LibGemini requires leader election at a finer granularity, i.e., assigning a specified leader to each round, potentially assigning multiple leaders to a round (because if a compromised node is elected as a round leader, its twins becomes leader too). We wrote a new leader election algorithm for LibraBFT that supports these requirements.

To emulate running the protocol for a given number of rounds, we approximate rounds by the number of messages emitted by nodes. Note that in a system with partial synchrony, we can only make guesses about rounds as there is no global notion of rounds. Using message-count per-round (without partitions) as an 'over-guesstimate', we let the nodes vote for 3 extra rounds. Over-running a scenario has no consequence on the results of LibGemini (other than longer scenario execution time) because any safety violations would have already been detected in earlier rounds.

6.2 Scenario Generator

The scenario generator produces scenarios in three main steps. First, it generates all the possible ways in which a set of N nodes can be split into P partitions (partition scenarios). Second, it generates all possible ways in which L leaders can be combined with the partitions generated in the previous step. Finally, it generates all the possible ways in which the partition-leader combinations can be permuted over R rounds of consensus protocol execution. The scenario generator can operate in online or offline modes. In the online mode, scenarios are generated on the fly, and fed to the scenario executor. The scenario generator can be configured to write the generated scenarios to a file. In the offline mode, the scenario generator reads previously generated scenarios from a file and feeds them to the scenario executor. For debugging purposes, the scenario generator can also operate in a 'dry run' mode—scenarios are generated with the given parameters, without running them, and statistics are printed at the end.

Pruning scenarios. A naïve enumeration of all combinations of P partitions, L leaders, and R rounds may explode quickly (see Table 1). In order to constrain the number of generated scenarios in a particular run, we provide hooks to control the number of P partitions, the number of L leader-partition pairs, and the number of leader-partition configuration assignments to rounds. For all three cases, we specify whether the selection is deterministic—first X—or randomized—an X sample. In the third case—configuration assignment to rounds—the total combination space to select from is large. Therefore, the scenario generator allows randomizing the per-round configuration selection, rather than sampling over the entire space of assignments.

7 Evaluation

We validate the capability of LibGemini to model and detect attacks, present microbenchmarks for the main components of LibGemini, and describe our experiments at scale using Amazon Web Services (AWS) [4]. We are open sourcing the Rust implementation of LibGemini, AWS orchestration scripts, and microbenchmarking scripts and data to enable reproducible results[6].

All our evaluations correspond to 4–7 nodes, 4–7 rounds and 2–3 partitions. Intuitively, these configurations seem sufficient to expose any safety violations. Indeed, the known attacks on BFT protocols described in Section 3 were exposed with only a small number of nodes, partitions and leader rotations. A recent work [24] on the coverage of random scenarios to detect crash faults shows that coverage depends on the number of partitions and node labels (in our case, the leaders), but not on the number of nodes. For Jepsen [16], all the bugs that provide meaningful coverage have a small number of rounds, and 2–3 partitions and roles [24]. Using higher values for these parameters leads to a very large number of scenarios, which cannot be feasibly executed without some sort of filtering (Section 6.2). It is an interesting open question whether increasing the value of these parameters has a higher chance of exposing safety violations.

[6] https://github.com/libra/libra

7.1 Validation

We deliberately introduce bugs to LibraBFT, and validate that LibGemini is able to model and detect attacks that exploit the injected vulnerabilities. This approach is similar to *mutation testing*, a well-known technique to evaluate the quality of existing tests in terms of whether they can detect programs with deliberately injected modifications (called "mutants"). While approaches such as automated mutation testing can help us to exhaustively introduce mutants, this is computationally expensive and not practical for large, complex systems. We select bugs to inject into LibraBFT based on their ability to compromise the program's functional correctness. We note that this choice is based on our intuition and experience, and does not provide any coverage guarantees. The validation approach we use is to: (*i*) inject the bug into LibraBFT; and (*ii*) generate scenarios using the LibGemini scenario generator, checking for any safety violations. We instantiate the scenario generator with different configurations, starting with small parameter values that we increase until a safety violation is exposed.

We begin with the base case: can LibGemini generate a scenario that violates safety when the BFT threshold is exceeded (i.e., $> f$ Byzantine nodes)? We discovered a safety violation with 4 nodes and 2 twins $(\underline{A}, B, C, D, \underline{A'}, B')$, 7 rounds, and static scenario configuration (i.e., each partition-leader combination is run for all R rounds). LibGemini executed 62 scenarios of which 8 led to safety violation within 86s.

Changing quorum size to $2f$. BFT protocols consider a state transition safe if it receives votes from an honest majority of nodes (i.e., quorum). We change LibraBFT's quorum size from $2f+1$ to $2f$. LibGemini detects a safety violation with 4 nodes and 1 twin $(\underline{A}, B, C, D, \underline{A'})$, 7 rounds, and static scenario configuration (i.e., where each partition-leader combination is run for all the R rounds). Within 20s, LibGemini executes 14 scenarios of which 6 lead to safety violation. These scenarios have the same pattern: Nodes are split into two partitions of size 2 and 3, with A in one partition and A' in the other. As nodes in the two partitions can form quorum, oblivious to each other they continue to generate quorum certificates on blocks proposed by A and A', respectively. Ultimately, nodes in the two partitions commit two different blocks.

Accepting conflicting votes. When a node receives a block proposal, it votes for the block only if the *block_round* is greater than the *last_voted_round* (Safety Rule 1, Section 5). We introduce a subtle bug to LibraBFT by changing this rule,

so that a node votes for a block if the *block_round* is greater than *or equal to* the *last_voted_round*. LibGemini detects the safety violation within a few seconds, with 4 nodes and 1 twin $\{\underline{A}, B, C, D, \underline{A'}\}$, and 7 rounds. This safety bug was detected in one-shot, with 0 partitions. Nodes vote on proposals from both A and A'—after a few rounds, they end up committing two different proposals for the same round.

Forgetting to update preferred round. When a node receives a block proposal, it votes for the block if the *block_round* is greater than *last_voted_round*, and the block's *parent_round* is greater than or equal to *preferred_round* (Safety rules 1 and 2, Section 5). We disable the first check, and bypass the second check by never updating *preferred_round* so it permanently remains at 0 (Update rule 2, Section 5). The main ingredient of an attack that exploits the bug described above is to propose a block in an old round, and get the nodes to *over-write* committed blocks (safety violation). The challenge for LibGemini is that as a twin node runs correct code, it cannot be made to propose blocks in arbitrary rounds. One option is to partition the twin node in an old round, and bring it back up in a later round, so it starts proposing blocks from where it left. This is, however, not possible in a 'full disclosure' protocol like LibraBFT where each quorum certificate (or timeout certificate) contains the full history of previous messages that led to the certificate. That is, as soon as A' recovers from the partition, it receives a quorum certificate (or timeout certificate) from other nodes and advances its round.

To emulate A' going back in time and proposing a block for an older round, we let it run as leader for a few rounds, crash it, and then recover it again as leader. When A' comes back up again it starts from round 0, proposing a block that builds on the *genesis* block (the first committed block). Because of our modifications to the *preferred_round* and *last_voted_round* checks, the nodes re-write history.

7.2 Microbenchmarks

We present microbenchmarks for the two main components of LibGemini: scenario generator (Section 6.2) and scenario executor (Section 6.1). The microbenchmarks are run on an Apple laptop (MacBook Pro) with a 2.9 GHz Intel Core i9 (6 physical and 12 logical cores), and 32 GB 2400 MHz DDR4 RAM.

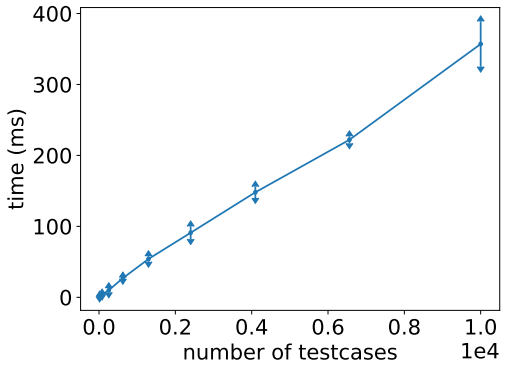
Scenario generator microbenchmarks. The scenario generator incurs a one-time computational cost—once the scenarios are generated, the scenario generator feeds them one by one to the scenario executor. Table 1 shows the number of scenarios generated with different configurations. We observe that the number of nodes and the number of rounds significantly increase the output of Step 1, which increases proportionally in the number of twins (as we only configure nodes with twins to become leaders). We find that non-static configurations in Step 3 cause the number of scenarios to explode. Therefore, of the various filters implemented for the scenario generator (Section 6.2), we find the filter at Step 2 to be most useful. We use this filter to make our at-scale Gemini analysis (Section 7.3) feasible. Note that this inevitably comes at the cost of completeness of coverage—a trade-off that we cannot completely eliminate. Figure 4 shows how long the scenario generator takes to produce scenarios for the same number of nodes (4) and partitions (2), and 4 (Figure 4a) and 7 (Figure 4b) rounds. We observe that while it expectedly takes longer to generate scenarios for 7 rounds vs. 4 rounds due to a larger number of permutations, for each case the time taken increases linearly in the number of scenarios. We observe a similar linear trend in our microbenchmarks for other configurations with varying number of nodes and partitions (figures not included due to space constraints).

Scenario executor microbenchmarks. Table 2 shows the time the scenario executor takes to execute a scenario. We repeat each measurement over 100 randomly selected scenarios from a configuration with 2 partitions, and varying number of nodes (4 and 7) and rounds (4–12). We observe that for 4 nodes, the execution time ranges from 234–465ms for 4–12 rounds, with a maximum standard deviation of 314ms. For 7 nodes, the execution time ranges from 547–748ms for 4–12 rounds, with a maximum standard deviation of $\sim$ 1.2s.

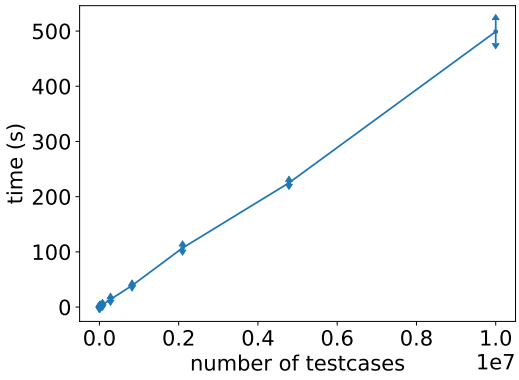
The variation observed above in execution times is expected because of how LibraBFT handles timeouts (Section 5). For each scenario, LibGemini runs LibraBFT until it has observed a given number of messages (proposals and votes), which roughly corresponds to the number of rounds. In some scenarios, LibGemini can quickly pull out the given number of messages and finish the scenario in a timely manner. In other scenarios, we might end up with partitions where the nodes are not able to make progress and advance rounds, due to frequent round failures and increased timeout values. The implication of this for LibGemini is that some scenarios may take longer to run, waiting for the network to emit enough messages to conclude the scenario. The execution of scenarios has negligible ($< 0.1\%$) memory and CPU footprints.

7.3 Running Scenarios at Scale

We evaluate LibGemini at scale, by running it against the correct code of LibraBFT. Specifically, we executed 44M scenarios which were randomly selected from the 200M scenarios corresponding to the third row of Table 1 (that is, with 4 nodes, 2 partitions, 7 rounds, permuted with replacement). We first generated all the 200M scenarios and randomly selected 44M samples. We ran the scenario generator in offline mode so the scenarios are written to file rather than being passed to the scenario executor. We then split the generated scenarios into 20 shards. The scenarios can be easily sharded, as the scenarios are independent of each other—this implies that subject to the availability of computing power to generate



(a) 4 Nodes, 2 Partitions, 4 Rounds.



(b) 4 Nodes, 2 Partitions, 7 Rounds.

Figure 4: Time taken by the scenario generator to produce LibGemini scenarios. Each data point is the average of 10 runs; error bars represent one standard deviation.

Nodes	Twins	Partitions	Rounds	Step 1	Step 2	Step 3		
						Without Replacement	With Replacement	Static
4	1	2	4	15	15	$\sim 3 \times 10^4$	$\sim 5 \times 10^4$	15
4	1	3	4	25	25	$\sim 3 \times 10^5$	$\sim 4 \times 10^5$	25
4	1	2	7	15	15	$\sim 3 \times 10^7$	$\sim 2 \times 10^8$	15
4	1	3	7	25	25	$\sim 2 \times 10^9$	$\sim 6 \times 10^9$	25
7	2	2	4	255	510	$\sim 7 \times 10^{10}$	$\sim 7 \times 10^{10}$	510
7	2	3	4	3,025	6,050	$\sim 1 \times 10^{10}$	$\sim 1 \times 10^{15}$	6,050
7	2	2	7	255	510	$\sim 9 \times 10^{18}$	$\sim 9 \times 10^{18}$	510
7	2	3	7	3,025	6,050	$\sim 3 \times 10^{26}$	$\sim 3 \times 10^{26}$	6,050

Table 1: The number of LibGemini scenarios generated for various configurations. Steps 1, 2 and 3 correspond to the scenario generation pipeline described in Section 4. **Step 1:** The number of ways in which N nodes can be distributed among P partitions. **Step 2:** The number of ways in which the partitions generated in Step 1 can be combined with leaders. **Step 3:** The number of ways in which the partition-leader pairs generated in Step 2 can be permuted (with and without replacement) over R rounds. **Static** configuration refers to the case where each partition-leader pair is statically configured for all the R rounds.

Rounds	4 Nodes		7 Nodes	
	Mean (ms)	Std. (ms)	Mean (ms)	Std. (ms)
4	239	314	547	1,286
5	250	87	555	1,059
6	284	88	555	802
7	296	87	559	752
8	334	209	647	810
9	363	175	643	557
10	398	222	653	539
11	433	168	718	570
12	465	179	748	223

Table 2: The time scenario executor takes to execute a scenario for 4 and 7 nodes, over varying number of rounds and fixed partitions (=2). Each measurement is repeated for 100 randomly selected scenarios.

and execute scenarios, LibGemini can be scaled up arbitrarily via sharding. We execute the sharded scenarios over 20 parallel instances of LibGemini on AWS. We use `t3.2xlarge` instances with 8 vCPUs, 2.5 GHz, Intel Skylake P-8175; 32 GB of RAM, and 300 GB of SSD storage. All machines run a fresh installation of Ubuntu 18.04. We did not observe any safety violations.

8 Related Work

There are two typical approaches to validate distributed systems. The first approach is to offer strong guarantees by building a fully verified system from the ground up [18, 25], or to show the absence or presence of bugs [10, 11, 20, 28] by exhaustively enumerating the space of system behaviors [5, 29] under systematically injected faults [3].

Fully verified systems do not scale to systems deployed in the real world. Model checking and exhaustive enumeration of distributed system faults (especially, Byzantine arbitrary behavior) leads to state explosion (despite partial order reduction techniques [14]), resulting in low performance. This motivates the second approach of random validation, which underlies the discipline of *Chaos Engineering*, exemplified by systems like Chaos Monkey [23]. The main idea is to analyze the resiliency of a distributed system by randomly injecting faults (e.g., terminating processes). Jepsen [16] is a blackbox analysis framework that runs processes with a random, auto-generated workload and randomly injected network partitions. A related approach is to subject the system being evaluated to *trials by fire* such as Cosmos Game of Stakes [12], i.e.,

financially incentivizing the community to attack the 'mock' network, and analyzing successful attacks to harden the network. Random validation is effective and scalable—but it is not comprehensive or reproducible, and cannot be used to evaluate distributed systems in an ongoing fashion.

Prior work (with the exception of Jepsen, a random validation framework) has focused on crash faults. Gemini is a new, principled approach to validate BFT systems by emulating Byzantine behavior via twins—copies of 'compromised' nodes that can send duplicate or conflicting messages. Gemini advances state-of-the-art in two ways. First, it provides a framework to systematically generate scenarios with configurable coverage, and only modeling correct executions (thus avoiding the state explosion problem associated with formal methods). We show with extensive evaluations (Section 7) that Gemini is suitable for evaluating real-world systems, and can be scaled up arbitrarily for larger scenario coverage. The second contribution of Gemini is to automatically generate scenarios that modify the interaction of components with the environment, without opening the code.

9 Future Work & Conclusion

This paper presented Gemini, a novel approach to systematically analyze BFT systems. The new approach provides coverage for many, but not all, Byzantine attacks. The paper demonstrated anecdotal evidence of coverage with respect to several known Byzantine attacks, and an implementation of Gemini for LibraBFT that exposes misconfiguration and purposely injected logical bugs within minutes. Many directions are left open for future extensions.

Theory of Gemini coverage. As mentioned in the Introduction, it is left open to rigorously characterize the attacks that Gemini can cover. In particular, we conjecture that Gemini covers all Byzantine behaviors in a class of protocols that have 'full disclosure': each message includes a reference to its entire causal past and any source of non-determinism (such as local coin flips), and nodes act deterministically according to their causal past. It would seem that this class of protocols is fully covered by Gemini since the only possible attack by Byzantine nodes is to select different subsets of messages to report to different targets. Similarly, we conjecture that Gemini can cover timing violations in a class of 'lock-step' synchronous protocols. Increasing coverage of Gemini in the settings we explore as well as others, and providing a formal treatment of coverage remain interesting open challenges.

Checking additional properties. A different dimension for extension is the type of guarantees which Gemini scenarios. While this paper focused squarely on safety of the core consensus protocol, the Gemini approach can be extended to validate ancillary components of BFT systems. For example, LibraBFT switches to a new set of nodes by committing a special block that includes the new set of nodes and signals the reconfiguration event. It would be useful to investigate if Gemini can cause a safety violation by creating an inconsistent node change (i.e., parts of the network believe in different nodes). Similarly, LibraBFT's smart contract execution engine is re-instantiated via a similar mechanism, and can be subjected to a similar Gemini-based attack.

Extending Gemini implementation. With respect to the concrete LibraBFT Gemini implementation presented in Section 6, several extensions are left for future work, including: (*i*) tackling more than a pair of twins; (*ii*) detecting liveness violations; and (*iii*) implementing process-level twins over TCP/IP.

Acknowledgments

This work is funded by Novi. The authors would like to thank Ben Maurer, David Dill, Daniel Xiang, Kartik Nayak, and Ling Ren for feedback on late manuscript, and George Danezis for comments on early manuscript. We also thank the Novi Research and Engineering teams for valuable feedback.

Availability

All artifacts presented in this paper are made publicly available.[7] Specifically, this includes: (*i*) the Rust implementation of LibGemini, the Gemini framework we implemented for LibraBFT (Section 6); (*ii*) the artifacts (the AWS orchestration scripts, and microbenchmarking scripts and data) used to evaluate LibGemini (Section 7); and (*iii*) the Python simulator and Gemini instantiation of safety flaw in Fast-HotStuff (Section 3).

References

[1] Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting Fast Practical Byzantine Fault Tolerance: Thelma, Velma, and Zelma. arXiv preprint arXiv:1801.10022, 2018.

[2] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *IEEE Symposium on Security and Privacy*, 2020.

[3] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-Driven Fault Injection. In *SIGMOD International Conference on Management of Data*, 2015.

[4] Inc. Amazon Web Services. AWS Whitepapers. https://aws.amazon.com/whitepapers, 2017.

[7] Links removed for blind review.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[6] Ethan Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. https://cdn.relayto.com/media/files/LPgoWO18TCeMIggJVakt_tendermint.pdf, 2016.

[7] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The Latest Gossip on BFT Consensus. arXiv preprint arXiv:1807.04938, 2018.

[8] Vitalik Buterin and Virgil Griffith. Casper the Friendly Finality Gadget. arXiv preprint arXiv:1710.09437, 2017.

[9] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[10] Ang Chen, W Brad Moore, Hanjun Xiao, Andreas Haeberlen, Linh Thi Xuan Phan, Micah Sherr, and Wenchao Zhou. Detecting Covert Timing Channels with Time-Deterministic Replay. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 541–554, 2014.

[11] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *ACM SIGCOMM Conference*, 2016.

[12] Cosmos. Cosmos Game of Stakes, 2018. https://github.com/cosmos/game-of-stakes.

[13] Diem. LibraBFT. https://github.com/diem/diem.

[14] Patrice Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and Pierre Wolper. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, 1996.

[15] Mohammad M Jalalzai, Jianyu Niu, and Chen Feng. Fast-hotstuff: A fast and resilient hotstuff protocol. arXiv preprint arXiv:2010.11454, 2020.

[16] Jepsen. Distributed Systems Safety Research. https://jepsen.io.

[17] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[18] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, May 1994.

[19] Leslie Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[20] Chia-Chi Lin, Virajith Jalaparti, Matthew Caesar, and Jacobus Van der Merwe. DEFINED: Deterministic Execution for Interactive Control-Plane Debugging. In *USENIX Technical Conference*, 2013.

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

[22] Atsuki Momose and Jason Paul Cruz. Force-Locking Attack on Sync Hotstuff. IACR Cryptology ePrint Archive, 2020.

[23] Netflix. Chaos Monkey. https://netflix.github.io/chaosmonkey/.

[24] Filip Niksic. *Combinatorial Constructions for Effective Testing*. Doctoral thesis, Technische Universität Kaiserslautern, 2019.

[25] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable Network Configuration Verification Through Model Checking. In *USENIX Symposium on Networked Systems Design and Implementation*, 2020.

[26] Basil Cameron Rennie and Annette Jane Dobson. On Stirling Numbers of the Second Kind. *Journal of Combinatorial Theory*, 7(2):116–121, 1969.

[27] The Diem Team. State Machine Replication in the Libra Blockchain. https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain/2019-11-08.pdf, 2019.

[28] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing Missing Events in Distributed Systems with Negative Provenance. *ACM SIGCOMM Computer Communication Review*, 44(4):383–394, 2014.

[29] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. Predicting and Preventing Inconsistencies in Deployed Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, 28(1):1–49, 2010.

[30] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT Consensus in the Lens of Blockchain. arXiv preprint arXiv:1803.05069, 2018.

[31] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT Consensus with Linearity and Responsiveness. In *ACM Symposium on Principles of Distributed Computing*, 2019.

10 LibGemini Implementation of Scenario Executor and Scenario Generator

This section provides the Rust code for the two main functions of Gemini, `execute_scenario` and `scenario_generator`. The code listings in Figure 5 and Figure 6 present simplified Gemini interfaces, i.e., we omit Rust-specific features such as explicit typing, details of error messages returned, de-referencing, and managing variable ownership.

The scenario executor, implemented by *execute_scenario* (Figure 5), executes scenarios generated by the scenario generator. This function takes as input the number of nodes and twins, and the leaders and partitions for each round. It creates a network with the given inputs, and starts running the protocol until the nodes have emitted a given number of messages, which approximate the number of rounds for which the protocol has been run.

The *execute_scenario* function exposes a simple interface, abstracting complex underlying network and SMR configurations. To demonstrate the simplicity and flexibility of *execute_scenario*, we show how to implement a simple scenario (Figure 6) where no quorum can be formed, and therefore no block gets committed. We set up a network with 4 honest nodes ($n0, n1, n2, n3$), and 1 twin ($twin0$). We split the network into two partitions $\{\underline{n0}, \underline{twin0}, n1\}$ and $\{n1, \underline{n3}\}$. For each round $n0$, $twin0$ (in partition 1) and $n3$ (in partition 2) are leaders. We then run the protocol for enough rounds (at least 3 in LibraBFT) to get a commit on a block. In partition 1, both $n0$ and $twin0$ propose different blocks for the same rounds. $n1$ will only vote for one of the two proposals because the second proposal is for a round that is not greater than its *last_voted_round* (Safety rule 1, Section 5). The second partition does not have enough nodes to form quorum. Consequently, no blocks are committed.

```
fn execute_scenario(
    num_nodes, // number of nodes
    target_nodes, // the nodes for which to create twins
    round_partitions, // Vector of partitions for each
        round
    round_leaders // Vector of leaders for each round
) {
    let runtime = consensus_runtime();
    let playground = NetworkPlayground::new(runtime.
        handle());

    // Start nodes and twins
    let nodes = SMRNode::start_num_nodes_with_twins(
        num_nodes,
        &target_nodes,
        &playground,
        round_proposers
    );

    // Create partitions
    create_partitions(&playground, round_partitions);

    // Start running the protocol and sending messages
    block_on(async move {
        let proposals = playground
            .wait_for_messages(2, NetworkPlayground::
                proposals_only::<Payload>)
            .await;

        // Pull enough votes to get a commit on the first
            block
        let votes: Vec<VoteMsg> = playground
            .wait_for_messages(num_nodes * num_of_rounds,
                NetworkPlayground::votes_only::<Payload
                >))
            .collect();
    });

    // Check that the branches are consistent at all
        heights
    let all_branches = vec![];

    for i in 0..nodes.len() {
        nodes[i].commit_cb_receiver.close();
        let node_commits = vec![];
        while let node_commit_id = nodes[i].
            commit_cb_receiver.try_next() {
            node_commits.push(node_commit_id);
        }
        all_branches.push(node_commits);
    }

    assert!(is_safe(all_branches));

    // Stop all nodes
    for each_node in nodes {
        each_node.stop();
    }
}
```

Figure 5: The `execute_scenario` function which executes scenarios.

11 Detailed Safety Attack on Zyzzyva

We present a summary of Zyzzyva, and use Gemini to reinstate a known safety attack [1] on Zyzzyva [17]. We use the notation described in Section 2.

11.1 Summary of Zyzzyva

Zyzzyva is an SMR protocol in the same settings as LibraBFT (partial synchrony and $n = 3f + 1$). It operates in a view-by-view manner. Each view has a designated leader. Nodes vote on the leader proposal if they consider it valid (we describe the validity criteria below, which has a flaw that enables the safety attack). A commit decision on the leader proposal forms in either of two tracks, fast and two-phase. In the fast track, all n nodes vote for the leader proposal to commit it. In the two-phase track, $2f + 1$ nodes form a commit-certificate (*CC*), then $2f + 1$ nodes vote for the *CC* to commit the proposal.

At the beginning of the view, nodes send the new leader a signed NEW-VIEW status message. The leader's first proposal carries the status of $2f + 1$ nodes at the beginning of the

```
fn twins_no_quorum_scenario() {
    let runtime = consensus_runtime();
    let playground = NetworkPlayground::new(runtime.
        handle());
    let num_nodes = 4;

    // 4 honest nodes
    let n0 = 0, n1 = 1, n2 = 2, n3 = 3;
    // twin of n0
    let twin0 = node_to_twin.get(n0);
    // twin of n1
    let twin1 = node_to_twin.get(n1);

    // Index #s of nodes for which we will create twins
    let target_nodes = vec![0];

    // Specify round leaders
    let round_leaders = HashMap::new();
    for i in 1..10 {
        // Insert (n0, twin0, n3) as leaders for round i
        round_leaders.insert(i, vec![n0, twin0, n3]);
    }

    // Specify round partitions
    let round_partitions = HashMap::new();
    for r in 0..10 {
        // Insert partitions for round r
        round_partitions.insert(
            r,
            vec![
                vec![n0, twin0, n1],
                vec![n2, n3],
            ],
        );
    }

    execute_scenario(
        num_nodes,
        &target_nodes,
        &round_partitions,
        &round_leaders
        );
}
```

Figure 6: Twins 'No Quorum' scenario.

view to prove the proposal validity. The (flawed) definition in Zyzzyva for a valid proposal upon view change is as follows. For each sequence slot:

- **Validity Rule 1** The leader picks among the states of $2f+1$ nodes, the *CC* from the highest view, if one exists.
- **Validity Rule 2** Otherwise, the leader picks a proposal that has $f+1$ votes from the highest view, if one exists.
- **Validity Rule 3** Finally, if none of the above exist, the leader creates a Nil proposal.

The flaw is to prioritize Validity Rule 1 over Validity Rule 2, which causes the leader to prefer *CC* even if generated in a *lower view* than $f+1$ votes.

11.2 Safety Attack on Zyzzyva

The Zyzzyva flawed scenario safety demonstrated in [1] goes through a succession of three views. In the first view, a faulty leader generates conflicting proposals v_1, v_2 and splits honest nodes between $f+1$ that vote for v_1 and f that vote for v_2. The faulty leader gathers a *CC* on v_1 but does not send it to other nodes. In the second view, a good leader adopts v_2 and drives agreement in the fast track. In the third view, f faulty nodes join the $f+1$ honest nodes that voted for v_1 in the first view. They send the leader a *CC* for v_1, hence the protocol proceeds with v_1, in conflict with the v_2 commit. The attack on Zyzzyva needs only $n=4$ nodes, of which $f=1$ is faulty, and it is fairly easy to re-instate using the Gemini framework. There are four nodes, (D,E,F,G). To model the case that D is Byzantine, it has a twin D' initialized with different input. We drive the execution creating partitions and electing leaders at each step, according to the attack described above. We describe below the detailed attack using Gemini.

Step 1 Initialize D and D' with different inputs v_1 and v_2.

Step 2 During View 1:

- Create the following partitions: $P_1 = \{\underline{D}, E, F\}$, $P_2 = \{\underline{D'}, G\}$.
- Let D run as leader for one round. D proposes v_1 to P_1 and gathers votes from P_1 creating a *CC*.
- Create the following partitions: $P_1 = \{E, F\}$, $P_2 = \{\underline{D'}, G\}$, $P_3 = \{\underline{D}\}$.
- As a result, D does not get to share *CC* on v_1 with E and F.
- Similarly, for one round let D' propose v_2 to P_2 and gather votes from P_2.

Step 3 Delay all messages until a new view starts. View 2:

- Create the following partitions: $P_1 = \{D', E, \underline{G}\}$, $P_2 = \{D, F\}$.
- Run G as leader, and let it collect (NEW-VIEW) messages from D' and E. Using Validity Rule 2 (Section 11.1), G decides to propose for v_2.
- Remove all partitions, i.e., $P = \{D, D', E, F, \underline{G}\}$.
- G proposes v_2, and collects votes from everyone. This leads to a commit of v_2.

Step 4 Delay all further messages until new view starts. View 3:

- Create the following partitions: $P_1 = \{D, \underline{E}, F\}$, $P_2 = \{D', G\}$.
- Run E as leader, and collect (NEW-VIEW) messages from D and F. Note that D sends the *CC* on v_1 (from view 1) to E. Using Validity Rule 1 (Section 11.1), E decides to propose v_1.
- E proposes v_1 to P_1, and gathers votes from D, E and F (who empty their local logs, undoing v_2). This leads both E and F to commit v_1, a safety violation.

12 Detailed Liveness Attack on FaB

We present a summary of FaB, and use Gemini to reinstate a known liveness attack on FaB [1]. We use the notation described in Section 2.

12.1 Summary of FaB

FaB is a single-shot consensus protocol for the partial synchrony setting with $n = 3f + 1$.[8]

A precursor to Zyzzyva, FaB is a view-based protocol with an optimistic fast track. A leader drives a decision in the fast track if all nodes vote for it, and in the two-phase track if $2f + 1$ nodes vote for a $(2f + 1)$ commit-certificate (CC). When a new leader is elected, it picks a valid proposal that does not conflict with neither $f + 1$ votes nor a CC in the previous view.

12.2 Liveness Attack on FaB

The (flawed) selection criterion above leads an execution in the following scenario to become stuck. A faulty leader equivocates and proposes v_1, v_2 to $2f + 1$ and f honest nodes, respectively. In transitioning to the next view, there is a commit-certificate for v_1 and $f + 1$ votes for v_1 (including an equivocation by one faulty), hence neither is safe, and the new leader is stuck. The attack on FaB needs only $n = 4$ nodes, of which $f = 1$ is faulty, and it can be easily re-instated using Gemini. There are four nodes, (A, B, C, D) with D as a Byzantine node, for which we create a twin D' initialized with different input. We describe below the attack using Gemini.

Step 1 Initialize D and D' with different inputs v_1 and v_2.

Step 2 During View 1:

- Create the following partitions: $P_1 = \{A, B, \underline{D}\}$, $P2 = \{C, \underline{D'}\}$
- Run D as leader for one round. D proposes v_1 to P_1 which decides to vote on v_1.
- Insert the following rule in P_1: $(B, D) \rightarrow A$. That is, the only messages allowed are those from B and D, to A.
- D, A and B send their votes which only reach A. Thus, only A produces a CC for v_1.
- Meanwhile, the leader D' proposes v_2 to P_2.

Step 3 Delay all further messages until new view starts. Create the partitions: $\{\underline{A}, C, D'\}$, $\{B, D\}$. Let the new leader A collect NEW-VIEW status messages from P_1. These status messages block A from proposing both v_1 and v_2 due to the FaB proposal validity rule. The rule states that a proposal is valid if it does not conflict with neither $f + 1$ votes nor a CC in the previous view, which is not the case for v_1 (has a CC) and v_2 (has $f + 1$ votes) as described below:

- From A, the NEW-VIEW message contains the value v_1, and a CC for it.
- From C, the NEW-VIEW message contains the value v_2, and no CC.
- From D', the NEW-VIEW message contains the value v_2, and no CC.

13 Detailed Liveness Attack on Sync HotStuff

We present a summary of Sync HotStuff, and use Gemini to reinstate the force-locking attack [22] on a preliminary version of Sync HotStuff (which was fixed in an updated version). We use the notation described in Section 2.

13.1 Summary of Sync HotStuff

The preliminary version of Sync HotStuff [2] is an SMR solution in the synchronous model with $n = 2f + 1$ parties.[9]

In synchronous protocols like Sync HotStuff, nodes execute the protocol in terms of Δ, which is the known bound assumed on maximal network transmission delay. Sync HotStuff operates in a view-by-view regime —in each view there is a designated leader which proposes values to nodes. If a node accepts the proposed value, it broadcasts its vote. A node creates a commit certificate (CC) for a proposed value if it receives $f + 1$ votes on it. Nodes track the highest CC, and only vote on a proposed value if it: (i) extends the highest CC known to the node, and (ii) does not equivocate another value proposed for the same height.

A node creates and broadcasts a *blame* against a leader: (i) if the leader does not propose a value for 3Δ, or (ii) the leader proposes an equivocating value. If a node observes $f + 1$ blames against the leader in the current view, it broadcasts the $f + 1$ blames, then waits Δ (to allow the blames to reach all honest nodes), and moves to the new view. In the new view, it immediately sends the new leader the highest CC it knows of.

After a view change, the new leader waits for Δ to receive node status messages (carrying the highest CC known to them). The leader then proposes a value that extends the highest CC from among the received status messages. Nodes proceed in the new view as previously described.

[8] FaB is actually designed for a *parameterized* model with $n = 3f + 2t + 1$, with safety guaranteed against f Byzantine failures and fast track guaranteed against t. For brevity and uniformity, we ignore t here and set $t = 0$.

[9] The description here covers the first of three variants in that paper; two other variants are designed for slightly different synchrony assumptions, but the attacks on them are similarly covered by the Gemini approach.

13.2 Implementing Synchrony Attacks in Gemini

Due to the synchronous settings and the nature of the attack which heavily leverages synchrony assumptions, in this case a Gemini scheduler must control message delivery timing. More precisely, rather than only specifying whether a message is delivered to a party or dropped, attacks on synchronous protocols require the Gemini scheduler to deliver messages to specific parties at specified times. While this is captured by the Gemini approach, our current implementation (Section 6) does not support this feature (this will be implemented in future Gemini extensions).

Generally, we expect that the granularity of the scheduler timing can be fairly coarse. In particular, there is a known parameter Δ, the bound presumed by the algorithm on message transmission delays and hard-coded into it. Indeed, the force-locking attack needs to deliver messages at 0.5Δ increments, e.g., at times $0, 0.5\Delta, \Delta, 1.5\Delta, 2.0\Delta, ...$ Therefore, a Gemini network emulator could operate in discrete lock-step at 0.5Δ increments. With this capability in place, the force-locking attack can be re-instated in the Gemini approach as described below.

13.3 Safety Attack on Sync HotStuff

We now rebuild the force-locking attack on the preliminary version of Sync HotStuff using Gemini. The crux of the attack is for a faulty leader to generate a last-minute proposal that reaches only half of the honest nodes. The other half trigger a view change, and now the system becomes split. The first half continues to commit the first leader proposal with "help" from Byzantine nodes. The second half starts a new view and fork the chain. This attack can be reinstated with Gemini using 5 nodes (A, B, C, D, E), of which (A, B) are faulty and have twins (A', B').

Notation. We extend the notation described in Section 2 to capture message transmission in the synchronous setting as follows: $S_t \xrightarrow{v} S'_{t'}$ denotes the transmission of a value v from a set of nodes S that generate the value at time t, to a set of nodes S' that receive the value at time t'. If a value is broadcast, we use the $\star$ symbol instead of a set: For example, $S_t \xrightarrow{V} \star$ means that S broadcasts a value v at time t. Additionally, to highlight the 'send' or 'receive' action on a value, we use bold text on the left or right side of the arrow, respectively. For example, $\boldsymbol{S_t} \xrightarrow{v} S'$ means that S sends v to S' (message arrival time is not known).

To reinstate this attack with Gemini, we deploy 5 nodes (A, B, C, D, E), of which (A, B) are faulty and have twins (A', B'). Here, $n = 5$, $f = 2$, and quorum size is 3 (since synchronous BFT protocols tolerate f Byzantine nodes for $n = 2f + 1$). We describe below the detailed attack using Gemini.

At time 1.5Δ :

- A is the leader, and broadcasts a proposal with $delay = \Delta$ for the value v_1 which extends v_0.

$$\boldsymbol{(A)_{1.5\Delta}} \xrightarrow{propose(v_1)} \star$$

At time 2.5Δ :

- C receives V_1, and broadcasts its vote.

$$(A)_{1.5\Delta} \xrightarrow{propose(v_1)} \boldsymbol{(C)_{2.5\Delta}}$$
$$\boldsymbol{(C)_{2.5\Delta}} \xrightarrow{vote(v_1)} \star$$

At time 3Δ :

- D blames A since it did not receive a proposal from A within 3Δ. Gemini (A', B') also did not receive a proposal from A, hence they also blame with A. (D, A', B') broadcast their blames with $delay = 0$, receive $f + 1$ blames from each other, and start waiting for Δ.

$$\boldsymbol{(D, A', B')_{3\Delta}} \xrightarrow{blame(A)} \star$$
$$(D, A', B')_{3\Delta} \xrightarrow{blame(A)} \boldsymbol{(D, A', B')_{3\Delta}}$$

At time 3.5Δ :

- D receives C's vote on v_1, but it cannot create a CC on v_1 since it has less than $f + 1$ votes.

$$(C)_{2.5\Delta} \xrightarrow{vote(v_1)} \boldsymbol{(D)_{3.5\Delta}}$$

- (A, B) broadcast their votes on v_1, which arrive at C with delay 0. As a result, C gathers $f + 1$ votes on v_1 and creates $CC(v_1)$.

$$\boldsymbol{(A, B)_{3.5\Delta}} \xrightarrow{vote(v_1)} \star$$
$$(A, B)_{3.5\Delta} \xrightarrow{vote(v_1)} \boldsymbol{(C)_{3.5\Delta}}$$

At time 4Δ :

- C receives $f + 1$ blame messages from (D, A', B'), broadcasts all blame messages, and starts waiting for Δ.

$$(D, A', B')_{3\Delta} \xrightarrow{blame(A)} \boldsymbol{(C)_{4\Delta}}$$
$$\boldsymbol{(C)_{4\Delta}} \xrightarrow{blame(A)} \star$$

- D has waited for Δ since it quit the old view w with leader A, so it starts the next view $w + 1$ and sends its highest commit certificate $CC(V_0)$ along with $f + 1$ blames on A to the next leader B, with $delay = 0$.

$$\boldsymbol{(D)_{4\Delta}} \xrightarrow{CC(v_0), blame(A)} (B)_{4\Delta}$$

- The new leader B receives $CC(v_0)$ from D and $f + 1$ blames on A, and broadcasts a proposal for value $v_1{}'$ extending V_0. Note that B does not know about $CC(v_1)$.

$$(D)_{4\Delta} \xrightarrow{CC(v_0), blame(A)} \boldsymbol{(B)_{4\Delta}}$$

$$(\boldsymbol{B})_{\boldsymbol{4\Delta}} \xrightarrow{propose(v_1{}')} \star$$

- D receives the proposal $v_1{}'$ from B, and broadcasts its vote with delay Δ, then it sets its commit timer to 2Δ and starts counting down.

$$(B)_{4\Delta} \xrightarrow{propose(v_1{}')} (\boldsymbol{D})_{\boldsymbol{4\Delta}}$$
$$(\boldsymbol{D})_{\boldsymbol{4\Delta}} \xrightarrow{vote(v_1{}')} \star$$

At time 4.5Δ :

- D receives votes on v_1 from (A,B); as it has now gathered $f+1$ votes on v_1 it creates $CC(v_1)$. However, this certificate is too late, as we will see in the following steps.

$$(A,B)_{3.5\Delta} \xrightarrow{vote(v_1)} (\boldsymbol{D})_{\boldsymbol{4.5\Delta}}$$

At time 5Δ :

- C has waited for Δ since it quit the old view with leader A, so it starts the next view $w+1$ and sends its highest certificate $CC(v_1)$ to the new leader B.

$$(\boldsymbol{C})_{\boldsymbol{5\Delta}} \xrightarrow{CC(v_1)} (B)$$

- C receives D's vote on $v_1{}'$ but does not vote since $v_1{}'$ (which extends $CC(v_0)$) does not extend its highest certificate $CC(V_1)$.

$$(D)_{4\Delta} \xrightarrow{vote(v_1{}')} (\boldsymbol{C})_{\boldsymbol{5\Delta}}$$

At time 6Δ :

- D commits $v_1{}'$ since it finished waiting for 2Δ and observed no equivocation or blame in the view $w+1$. However, D's highest certificate is $CC(v_1)$ (see time 4.5Δ).
- Now if the current leader B goes offline, this will result in a view change to view $w+2$ and the new leader will extend the blockchain from the highest certificate from the previous view, $CC(v_1)$. But D has committed $v_1{}'$ conflicting with v_1, hence safety is violated.

14 Attack on Fast-HotStuff

We present a safety attack against Fast-HotStuff [15] and express it using Gemini.

14.1 Summary of Fast-HotStuff

Fast-HotStuff is essentially HotStuff [31] with a 2-phase commit rule. In the happy-path, if the leader of round n is successful, the leader of round $n+1$ performs the same protocol as HotStuff, namely, it collects a QC from previous round and embeds it in the $n+1$ proposal. In the unhappy-path, if the leader of round n is unsuccessful, the protocol for leader

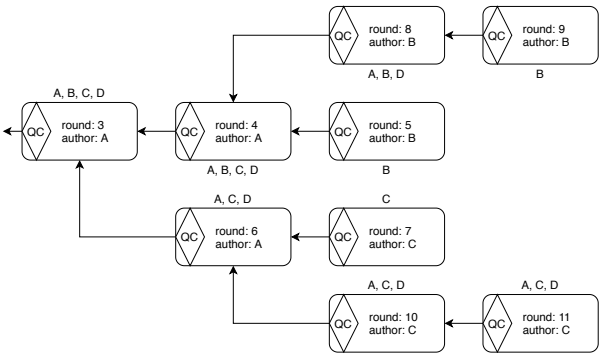


Figure 7: Example of safety attack on Fast-HotStuff.

$n+x+1$ $(x>0)$ provides a proof in the $n+x+1$ proposal that it is using the highest QC from $2f+1$ validators. This proof incurs quadratic communication complexity. Moreover, Fast-HotStuff claims it does not require consecutive rounds in order to commit.

The benefits of Fast-HotStuff are twofold. It provides a fast 2-phase track for HotStuff whenever the leader is successful in obtaining a QC for the previous round (happy-path). Fast-HotStuff is also faster both in phases (2 phases instead of 3) and in getting to a scenario that guarantees progress, namely, it requires 3 consecutive honest leaders (instead of 4). Requiring a leader proof for the unhappy-path prevents a proposal that conflicts with an uncommitted and unlocked tail of a chain that already has a QC. Thus, dishonest leaders cannot intentionally slow down progress by overriding the latest tail.

Fast-HotStuff is however flawed as explained in Section 14.2.

14.2 Safety Attack on Fast-HotStuff

Figure 7 illustrates the safety attack against Fast-HotStuff that we implement using Gemini. There are four nodes (A,B,C,D) all of which are honest—the safety attack can be executed leveraging only network partitions. Blocks are represented by rectangles (which are annotated with the nodes that receive the block). Block proposers are indicated as 'authors'. Diamonds refers to QCs (which are embedded into blocks). The arrows indicate the block that a QC refers to.

We execute the safety attack in 11 rounds starting at round 3 (rounds 2 carries QC for the genesis block).

Round 3: Initially there are no partitions, i.e., $P = \{\underline{A},B,C,D\}$.

- A proposes a block. Nodes send their votes on this proposal to the leader of the next round, node A.

Round 4:

- A gathers votes from the previous round, forms a QC, and includes the QC in a new block proposal. Nodes

send their votes on the new proposal to the leader of the next round, node B.

Round 5: Set node B as leader, i.e., $P = \{A, \underline{B}, C, D\}$.

- B gathers votes from the previous round, forms a QC, and includes the QC in a new block proposal.
- Create the following partitions: $P_1 = \{A, C, D\}$ and $P_2 = \{\underline{B}\}$.
- The partitions prevent B from broadcasting the new block (and the newly formed QC it embeds). B is thus the only node knowing the QC certifying the block of round 4.
- Nodes of P_1 time out, and send a NEW-VIEW message to the leader of the next round (node A) containing their highest known QC.

Round 6: Set node A as leader, i.e., $P_1 = \{\underline{A}, C, D\}$ and $P_2 = \{B\}$.

- A selects the highest QC from the NEW-VIEW messages (i.e., the QC certifying the block of round 3), and embeds it in a new block proposal. All nodes of P_1 vote on this proposal and send their votes to the leader of the next round (node C).

Round 7: Set node C as leader, i.e., $P_1 = \{A, \underline{C}, D\}$ and $P_2 = \{B\}$.

- C gathers votes from the previous round, forms a QC, and includes the QC in a new block proposal.
- Create partitions: $P_1 = \{A, B, D\}$ and $P_2 = \{\underline{C}\}$.
- These partitions prevent C from broadcasting the new block (and the newly formed QC it embeds). C is thus the only node knowing the QC certifying the block of round 6.
- Nodes of P_1 time out and send a NEW-VIEW message to the leader of the next round (node B) containing their highest known QC.

Round 8: Set node B as leader, i.e., $P_1 = \{A, \underline{B}, D\}$ and $P_2 = \{C\}$.

- B selects the highest QC from the NEW-VIEW messages (i.e., the QC certifying the block of round 4, presented by B), and embeds it in a new block proposal. All nodes vote on this proposal and send their votes to the leader of the next round (node B).

Round 9:

- B gathers all votes from the previous round, forms a QC, and includes the QC in a new block proposal.
- Create partitions $P_1 = \{A, C, D\}$ and $P_2 = \{\underline{B}\}$.
- The partitions prevent B from broadcasting its newly block(and the newly formed QC it embeds). B is thus the only node knowing the QC certifying the block of round 8 and committing the block at round 4.
- Nodes of P_1 time out and send a NEW-VIEW message to the leader of the next round (node C) containing their highest known QC.

Round 10: Set node C as leader, i.e., $P_1 = \{A, \underline{C}, D\}$ and $P_2 = \{B\}$.

- C selects the highest QC from the NEW-VIEW messages from the previous round (the QC certifying the block of round 6, presented by C), and embeds it in its new block proposal. The highest QC in the NEW-VIEW messages.
- All nodes of P_1 vote on this proposal and send their votes to the leader of the next round (node C).

Round 11: Set node C as leader, i.e., $P_1 = \{A, \underline{C}, D\}$ and $P_2 = \{B\}$.

- C assembles votes from the previous round into a QC certifying the block of round 10, thus committing the block of round 6.

The safety violation appears at round 11 when node C commits the block of round 6 while node B previously committed the block of round 4: both blocks have the same height and fork from the block of round 3.

14.3 Implementation of the Attack

We implemented a Python simulator of Fast-HotStuff using the discrete event simulator *simpy*. We demonstrate the safety violation by running a manually-crafted scenario in the simulator. We are open sourcing our Fast-HotStuff simulator as well as our Gemini scenario used for the attack[10].

[10] https://github.com/asonnino/twins-simulator