

# Twins: BFT Systems Made Robust

Shehar Bano ✉

Facebook Novi, United Kingdom

Alberto Sonnino ✉

Facebook Novi, United Kingdom

Andrey Chursin ✉

Facebook Novi, United States

Dmitri Perelman ✉

Facebook Novi, United States

Zekun Li ✉

Facebook Novi, United States

Avery Ching ✉

Facebook Novi, United States

Dahlia Malkhi ✉

Facebook Novi, United States

---

## Abstract

This paper presents Twins, a principled strategy for effectuating Byzantine attack scenarios at scale in Byzantine Fault Tolerant (BFT) systems and examining their behavior. Twins 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, Twins can systematically generate Byzantine attack scenarios at scale, execute them in a controlled manner, and examine their behavior. Twins 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 Twins in the arena of BFT consensus protocols. In all cases, protocols break within fewer than a dozen protocol rounds, hence it is realistic for the Twins approach to expose the problems. In two of these attacks, it took the community more than a decade to discover protocol flaws that Twins would have surfaced within minutes. Additionally, Twins has been incorporated into a production setting (DiemBFT [13]) in which it can execute 44M Twins-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 Twins itself were exposed within minutes.

**2012 ACM Subject Classification** Security and privacy → Distributed systems security

**Keywords and phrases** Distributed Systems, Byzantine Fault Tolerance, Real-World Deployment

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

**Supplementary Material** All artifacts presented in this paper are made publicly available. Specifically, this includes: (i) the Rust implementation of LibTwins, the Twins framework we implemented for LibraBFT (Section 5); (ii) the artifacts (the AWS orchestration scripts, and microbenchmarking scripts and data) used to evaluate LibTwins (Section 6); and (iii) the Python simulator and Twins instantiation of safety flaw in Fast-HotStuff (Section 3).

**Funding** This work is funded by Novi, a subsidiary of Facebook.

**Acknowledgements** The authors would like to thank Ben Maurer, David Dill, Daniel Xiang, Kartik



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

licensed under Creative Commons License CC-BY 4.0

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:29

Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

46 Nayak, and Ling Ren for feedback on late manuscript, and George Danezis for comments on early  
 47 manuscript. We also thank the Novi Research and Engineering teams for valuable feedback.

## 48 **1 Introduction**

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

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

64 Twins is based on the insight that most interesting Byzantine attacks are internal and  
 65 leverage knowledge of the expected behavior of participants, hence they go unnoticed. In  
 66 particular, Twins foregoes trivial attacks such as sending semantically invalid messages,  
 67 or sending a message without justification. Thus, leveraging existing code, Twins can  
 68 automatically cover material Byzantine behaviors. Indeed, Section 3 demonstrates one new,  
 69 and several known, attacks on BFT protocols materialized as Twins attacks. Crucially, in all  
 70 cases, protocols break within fewer than a dozen protocol steps, hence Twins successfully  
 71 exposes them. Note that Twins scenarios systematically iterate over protocol rounds and  
 72 vary the communication patterns among nodes. While inherently exponential, in the above  
 73 attacks, it took Twins only minutes to discover protocol flaws that in some cases, took the  
 74 community decades to surface.

75 Twins has been incorporated into a production setting in which Twins can execute 44M  
 76 Twins-generated scenarios daily. Whereas the system at hand did not manifest errors, subtle  
 77 safety bugs that were deliberately injected for the purpose of validating the implementation  
 78 of Twins itself were exposed within minutes

79 **Twins & attacks on BFT replication.** Twins arises in the context of BFT replication  
 80 protocols. In this domain, several worrisome safety and liveness vulnerabilities were exposed  
 81 recently [1, 22] in both known protocols [21, 17] and in new ones [2]. One reason that  
 82 BFT replication lends itself well to analysis via Twins is as follows. A common paradigm  
 83 underlying practical BFT replication protocols is a view-by-view design. Each view is driven  
 84 by a designated leader proposing to the nodes and going through voting rounds by the nodes.  
 85 If a leader is successful, a consensus decision is reached in the view. If not, nodes give up after  
 86 a timeout and move to the next view. Transitioning to the new view/leader is tricky: A new  
 87 leader must discover if the previous leader was successful, but it may be able to communicate  
 88 only with a subset of the nodes. The transition logic turns out to be the source of problems  
 89 in all the above cases, hence exposing the flaw requires only one or two leader rotations.

90 **Twins implementation.** Twins effectuates a Byzantine attack by a Byzantine node via  
 91 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. Twins 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 Twins approach in the DiemBFT open-source project, the BFT replication core of the Diem payment system [13]. Implementing Twins in DiemBFT 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 DiemBFT 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 DiemBFT 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 6 reports on our experience with Twins in DiemBFT.

**Coverage.** What attacks does the Twins 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 DiemBFT production system Section 6 provides compelling validation of the Twins approach. Whereas the system at hand did not manifest errors, self-injected subtle safety bugs—for the purpose of validating the implementation of Twins itself—were exposed within minutes. In particular, we created a simple safety-violating setting by deploying  $f + 1$  (instead of  $f$ ) nodes with Twins, 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), Twins successfully exposed safety violations.

(iii) Section 3.1 shows how Twins 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 Twins to expose subtle flaws. Section 3 reinstates several known attacks on BFT protocols using the Twins 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 Twins, 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 Twins. Another coverage challenge emerges in synchronous protocols because a node behavior may be based on real

time. In such protocols, Twins 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 Twins. However, we do not know yet which timing attacks may not be covered. We discuss some concrete future directions in Section 8 for extending Twins in the settings we explore as well as others.

## 2 Motivating the Twins 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 Twins is a viable approach by showing the different kinds of Byzantine behaviors that can be captured by Twins. (Concrete attacks using Twins are described later in Section 3 and Section 6.1.)

**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. Twins is a new approach to systematically generate Byzantine attacks. The main idea of Twins 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.

Twins 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  $P_1 = \{A, B, C, D, \underline{E}\}$ ,  $P_2 = \{C, D, E, \underline{F'}, G\}$ . 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. Twins 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. Twins 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 Twins

In this section, we demonstrate one new, and several known, attacks on BFT replication protocols, expressed as Twins 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 Twins 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 Twins (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 Twins to expose subtle flaws. More details are provided in Appendix F.

We implemented the Fast-HotStuff BFT consensus algorithm in a Python simulator which we release as open source<sup>1</sup>. The simulator then executes Twins scenarios over the algorithm.

<sup>1</sup> <https://github.com/asonnino/twins-simulator/tree/master/fhs>

## 226 3.2 Reinstated Attacks

227 We present several known attacks on BFT protocols, expressed as Twins scenarios. In all  
 228 cases, exposing vulnerabilities requires only a small number of nodes, partitions, rounds  
 229 and leader rotations. It is worth noting that later, our evaluation (Section 6) of LibTwins,  
 230 Twins implemented for DiemBFT, shows that running an automated scenario generator  
 231 (Section 4.2) with these configurations would cover the described attacks within minutes. We  
 232 did not undertake to re-implement all these protocols and apply a Gemini scenario generator  
 233 to them; our implementation covers only DiemBFT [13].

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

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

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

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

---

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



### 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  $v_2$ .

- 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:  $P_1 = \{\underline{D}, E\}$ ,  $P_2 = \{\underline{D'}, F\}$ ,  $P_3 = \{\underline{G}\}$ .  $D$  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.,  $P = \{\underline{D}, \underline{D'}, \underline{E}, \underline{F}, \underline{G}\}$ .
- 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  $P_1 = \{\underline{D}, E\}$ ,  $P_2 = \{\underline{D'}, F\}$ ,  $P_3 = \{\underline{G}\}$ .
- $F$  broadcasts  $QC(v_2)$ , which only reaches  $P_2$ .

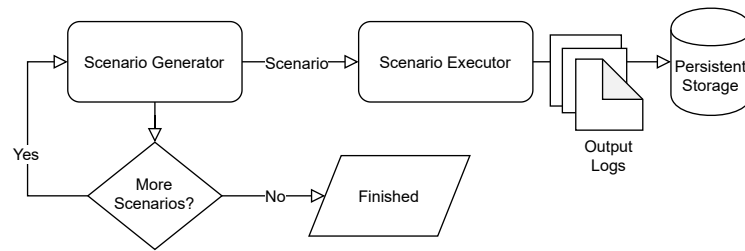
**View 3:** Drop all proposals from  $F$  until View 3 starts.

- Create the partitions  $P_1 = \{\underline{D}, \underline{E}, \underline{G}\}$ ,  $P_2 = \{\underline{D'}, F\}$ .
- 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 Twins attack scenarios, this section presents a framework for systematically generating such scenarios.

Systematically and efficiently generating Twins scenarios that provide good coverage requires tailoring to the specific BFT protocol settings. We develop the Twins framework which generates and executes *scenarios* that describe the node and network configurations. Specifically, the Twins 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



■ **Figure 1** Twins high-level design.

that are fed to the scenario executor to check for violations. The following design goals underlie the Twins framework:

- **Generic & Modular.** Twins 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.** Twins 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.** Twins 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 Twins—the scenario executor and the scenario generator—in detail.

## 4.1 Scenario Executor

In every Twins scenario, a threshold of the nodes are ‘misconfigured’ to have a twin instance with identical transport endpoint credential and secret keys. The Twins 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):

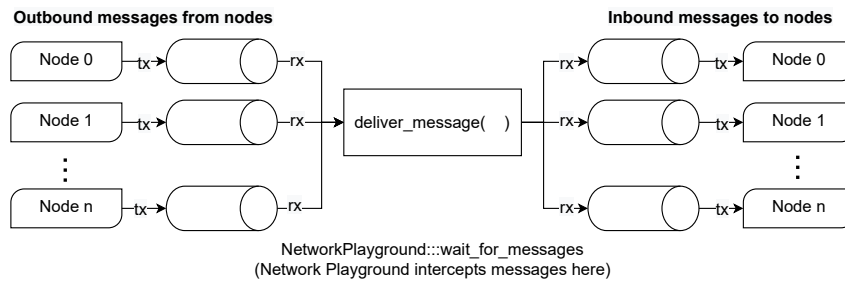
- **Step 1.** The scenario generator first produces the set of all possible partitions of nodes (called *partition scenarios*). For example, for a network of 4 nodes  $(A, B, C, D)$ , possible partition scenarios  $(P)$  include  $\{P_1 = \{A, D\}, \{B, C\}\}$ , and  $P_2 = \{\{A\}, \{B, C, D\}\}$ . 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 Twins 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<sup>3</sup> rotation) and generate only materially different scenarios. The implementation we describe in the Evaluation section of this paper (Section 6) 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

<sup>3</sup> Nodes can have designated roles in the protocol, referred to as *node labels*. Twins 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.



■ **Figure 2** Design of DiemBFT's *Network Playground*.

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 8).

## 5 Implementation

We implemented the Twins framework for DiemBFT, which we call LibTwins. Appendix A provides an overview of DiemBFT. 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 DiemBFT 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 LibTwins, `execute_scenario` and `scenario_generator`, are provided in Appendix B. We are open sourcing the Rust implementation of LibTwins<sup>4</sup>.

### 5.1 Scenario Executor

The LibTwins scenario executor leverages the network emulator of DiemBFT, *network playground*<sup>5</sup>. Network playground provides an apparatus for running single-host DiemBFT 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 messages) and observe node response. Figure 2 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 DiemBFT, 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 LibTwins, 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 LibTwins is built on top of network playground. This required the following modifications and extensions to the original library:

<sup>4</sup> <https://github.com/diem/diem>

<sup>5</sup> [https://github.com/diem/diem/blob/master/consensus/src/network\\_tests.rs](https://github.com/diem/diem/blob/master/consensus/src/network_tests.rs)

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

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

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

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

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

## 447 5.2 Scenario Generator

448 The scenario generator produces scenarios in three main steps. First, it generates all the  
 449 possible ways in which a set of  $N$  nodes can be split into  $P$  partitions (partition scenarios).  
 450 Second, it generates all possible ways in which  $L$  leaders can be combined with the partitions  
 451 generated in the previous step. Finally, it generates all the possible ways in which the partition-  
 452 leader combinations can be permuted over  $R$  rounds of consensus protocol execution. The  
 453 scenario generator can operate in online or offline modes. In the online mode, scenarios  
 454 are generated on the fly and fed to the scenario executor. The scenario generator can be  
 455 configured to write the scenarios to a file. In the offline mode, the scenario generator reads  
 456 previously generated scenarios from a file and feeds them to the scenario executor.

457 **Pruning scenarios.** A naïve enumeration of all combinations of  $P$  partitions,  $L$  leaders, and  
 458  $R$  rounds may explode quickly (see Table 1). In order to constrain the number of generated  
 459 scenarios in a particular run, we provide hooks to control the number of  $P$  partitions,  
 460 the number of  $L$  leader-partition pairs, and the number of leader-partition configuration

461 assignments to rounds. For all three cases, we specify whether the selection is deterministic—  
 462 first  $X$ —or randomized—an  $X$  sample. In the third case—configuration assignment to  
 463 rounds—the total combination space to select from is large. Therefore, the scenario generator  
 464 allows randomizing the per-round configuration selection, rather than sampling over the  
 465 entire space of assignments.

## 466 6 Evaluation

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

471 All our evaluations correspond to 4–7 nodes, 4–7 rounds and 2–3 partitions. Intuitively,  
 472 these configurations seem sufficient to expose any safety violations. Indeed, the known  
 473 attacks on BFT protocols described in Section 3 were exposed with only a small number  
 474 of nodes, partitions and leader rotations. A recent work [24] on the coverage of random  
 475 scenarios to detect crash faults shows that coverage depends on the number of partitions and  
 476 node labels (in our case, the leaders), but not on the number of nodes. For Jepsen [16], all  
 477 the bugs that provide meaningful coverage have a small number of rounds, and 2–3 partitions  
 478 and roles [24]. Using higher values for these parameters leads to a very large number of  
 479 scenarios, which cannot be feasibly executed without some sort of filtering (Section 5.2). It  
 480 is an interesting open question whether increasing the value of these parameters has a higher  
 481 chance of exposing safety violations.

### 482 6.1 Validation

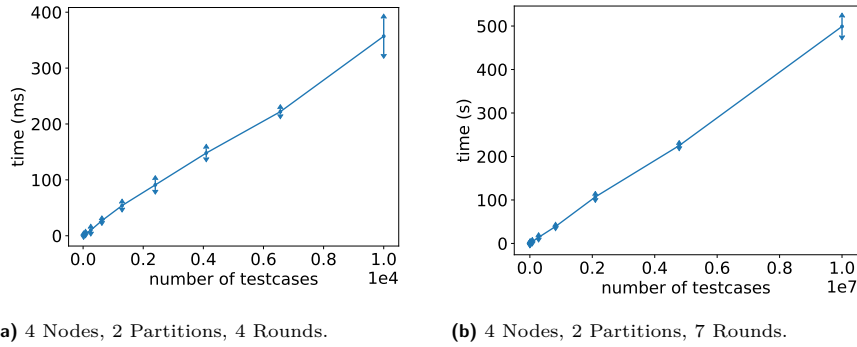
483 We deliberately introduce bugs to DiemBFT, and validate that LibTwins is able to model  
 484 and detect attacks that exploit the injected vulnerabilities. This approach is similar to  
 485 *mutation testing*, a well-known technique to evaluate the quality of existing tests in terms of  
 486 whether they can detect programs with deliberately injected modifications (called “mutants”).  
 487 While approaches such as automated mutation testing can help us to exhaustively introduce  
 488 mutants, this is computationally expensive and not practical for large, complex systems.  
 489 We select bugs to inject into DiemBFT based on their ability to compromise the program’s  
 490 functional correctness. We note that this choice is based on our intuition and experience, and  
 491 does not provide any coverage guarantees. The validation approach we use is to: (i) inject  
 492 the bug into DiemBFT; and (ii) generate scenarios using the LibTwins scenario generator,  
 493 checking for any safety violations. We instantiate the scenario generator with different  
 494 configurations and vary them until a safety violation is exposed.

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

500 **Changing quorum size to  $2f$ .** BFT protocols consider a state transition safe if it receives  
 501 votes from an honest majority of nodes (i.e., quorum). We change DiemBFT’s quorum size

---

<sup>6</sup> <https://github.com/libra/libra>



■ **Figure 3** Time taken by the scenario generator to produce LibTwins scenarios. Each data point is the average of 10 runs; error bars represent one standard deviation.

from  $2f+1$  to  $2f$ . LibTwins 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, LibTwins 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.** Upon receiving a proposal, nodes vote for it only if the *block\_round* is greater than the *last\_voted\_round* (Safety Rule 1, Appendix A). We introduce a subtle bug to DiemBFT 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*. LibTwins 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  $A'$  and quickly end up committing two different proposals for the same round.

**Forgetting to update preferred round.** Upon receiving a proposal, nodes vote 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, Appendix A). 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 A). 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 LibTwins 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 DiemBFT 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.

Nodes	Twins	Partitions	Rounds	Step 1	Step 2	No Repl.	Step 3 Repl.	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 LibTwins 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. In **Static** configurations, each partition-leader pair is statically configured for all the  $R$  rounds.

## 6.2 Microbenchmarks

We present microbenchmarks for the two main components of LibTwins: scenario generator (Section 5.2) and scenario executor (Section 5.1). 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 5.2), we find the filter at Step 2 to be most useful. We use this filter to make our at-scale Twins analysis (Section 6.3) feasible. Note that this inevitably comes at the cost of completeness of coverage—a trade-off that we cannot completely eliminate. Figure 3 shows how long the scenario generator takes to produce scenarios for the same number of nodes (4) and partitions (2), and 4 (Figure 3a) and 7 (Figure 3b) 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  $\sim 1.2$ s.

The variation observed above in execution times is expected because of how DiemBFT handles timeouts (Appendix A). For each scenario, LibTwins runs DiemBFT until it has observed a given number of messages (proposals and votes), which roughly corresponds to



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.

the number of rounds. In some scenarios, LibTwins 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. 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.

### 6.3 Running Scenarios at Scale

We evaluate LibTwins at scale, by running it against the correct code of DiemBFT. 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 and execute scenarios, LibTwins can be scaled up arbitrarily via sharding. We execute the sharded scenarios over 20 parallel instances of LibTwins 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.

## 7 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 [28, 11, 10, 20] 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) focused on crash faults. Twins 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. Twins advances state-of-the-art by providing 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 that Twins is suitable for evaluating real-world systems, and can be scaled up arbitrarily for larger scenario coverage. Twins automatically generate scenarios that modify the interaction of components with the environment, without opening the code.

## 8 Future Work & Conclusion

Twins is a novel approach to systematically analyze BFT systems. It 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 Twins for DiemBFT that exposes misconfiguration and purposely injected logical bugs within minutes. Many directions are left open for future extensions.

**Theory of Twins coverage.** As mentioned in the Introduction, it is left open to rigorously characterize the attacks that Twins can cover. In particular, we conjecture that Twins 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 Twins since the only possible attack by Byzantine nodes is to select different subsets of messages to report to different targets. Similarly, we conjecture that Twins can cover timing violations in a class of ‘lock-step’ synchronous protocols. Increasing coverage of Twins 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 Twins scenarios. While this paper focused squarely on safety of the core consensus protocol, the Twins approach can be extended to validate ancillary components of BFT systems. For example, DiemBFT 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 Twins can cause a safety violation by creating an inconsistent node change (i.e., parts of the network believe in different nodes). Similarly, DiemBFT’s smart contract execution engine is re-instantiated via a similar mechanism, and can be subjected to a similar Twins-based attack.

**Extending Twins implementation.** With respect to the concrete DiemBFT Twins implementation presented in Section 5, 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.

## 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.
- 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/LPgoW018TCeMIggJVakt\\_tendermint.pdf](https://cdn.relayto.com/media/files/LPgoW018TCeMIggJVakt_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. DiemBFTBFT. <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.

- 693 25 Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar.  
694 Plankton: Scalable Network Configuration Verification Through Model Checking. In *USENIX*  
695 *Symposium on Networked Systems Design and Implementation*, 2020.
- 696 26 Basil Cameron Rennie and Annette Jane Dobson. On Stirling Numbers of the Second Kind.  
697 *Journal of Combinatorial Theory*, 7(2):116–121, 1969.
- 698 27 The Diem Team. State Machine Replication in the Libra  
699 Blockchain. [https://developers.libra.org/docs/assets/papers/  
700 libra-consensus-state-machine-replication-in-the-libra-blockchain/2019-11-08.  
701 pdf](https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain/2019-11-08.pdf), 2019.
- 702 28 Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing  
703 Missing Events in Distributed Systems with Negative Provenance. *ACM SIGCOMM Computer*  
704 *Communication Review*, 44(4):383–394, 2014.
- 705 29 Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. Predicting and  
706 Preventing Inconsistencies in Deployed Distributed Systems. *ACM Transactions on Computer*  
707 *Systems (TOCS)*, 28(1):1–49, 2010.
- 708 30 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff:  
709 BFT Consensus in the Lens of Blockchain. arXiv preprint arXiv:1803.05069, 2018.
- 710 31 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff:  
711 BFT Consensus with Linearity and Responsiveness. In *ACM Symposium on Principles of*  
712 *Distributed Computing*, 2019.

## 713 **A Overview of DiemBFT**

714 We now shift our attention to utilizing Twins for validating BFT replication in DiemBFT [13].  
715 We discuss our implementation and evaluation of Twins for DiemBFT in Sections 5 and 6. In  
716 this section, we provide an overview of DiemBFT (for details, see the technical report [27]).

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

729 If  $2f + 1$  send votes to the next leader in a timely manner, a QC is formed by the leader  
730 and it sends the next proposal. Nodes maintain a timer to track progress. When the timer  
731 expires and a node still has not received a proposal, it broadcasts a timeout vote on a Nil  
732 block. When a node gathers enough timeout votes to form a timeout certificate, it advances  
733 its round. Every time a round fails, timeout periods are increased, allowing lagging nodes to  
734 catch up and enabling the protocol to eventually reach a decision.

735 As briefly alluded to in the Introduction, the trickiest part of BFT replication is to manage  
736 leader transition. DiemBFT maintains four parameters to ensure safety, and at the same time  
737 facilitate progress: (i) *current\_round*, the node’s current round; (ii) *last\_voted\_round*, the  
738 last round for which the node voted; (iii) *parent\_round*, the round of the block certified by  
739 the QC attached with the block being processed; (iv) *grandparent\_round*, the parent of the  
740 block certified by the QC; and (v) *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.

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.

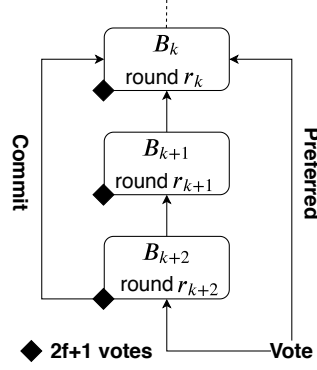
■ **Update rule 4.** When a leader gathers enough votes to form a QC, it broadcasts a new proposal and increments *current\_round*.

*Spoiler alert:* In our evaluation in Section 6.1, we are going to deliberately modify the above rules. We will see that this enables safety violations that the Twins framework will expose.

## **B LibTwins Implementation of Scenario Executor and Scenario Generator**

This section provides the Rust code for the two main functions of Twins, `execute_scenario` and `scenario_generator`. The code listings in Figure 5 and Figure 6 present simplified Twins 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



■ **Figure 4** Consensus and preferred sub-trees in DiemBFT.

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 (*twi* $n0$ ). We split the network into two partitions  $\{n0, \textit{twi}n0, n1\}$  and  $\{n1, n3\}$ . For each round  $n0$ , *twi* $n0$  (in partition 1) and  $n3$  (in partition 2) are leaders. We then run the protocol for enough rounds (at least 3 in DiemBFT) to get a commit on a block. In partition 1, both  $n0$  and *twi* $n0$  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 A). The second partition does not have enough nodes to form quorum. Consequently, no blocks are committed.

## C Detailed Safety Attack on Zyzzyva

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

### C.1 Summary of Zyzzyva

Zyzzyva is an SMR protocol in the same settings as DiemBFT (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 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:



```

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.

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

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

## 818 C.2 Safety Attack on Zyzzyva

819 The Zyzzyva flawed scenario safety demonstrated in [1] goes through a succession of three  
 820 views. In the first view, a faulty leader generates conflicting proposals  $v_1, v_2$  and splits honest  
 821 nodes between  $f + 1$  that vote for  $v_1$  and  $f$  that vote for  $v_2$ . The faulty leader gathers a

## 23:22 Twins: BFT Systems Made Robust

```

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.

822 *CC* on  $v_1$  but does not send it to other nodes. In the second view, a good leader adopts  
823  $v_2$  and drives agreement in the fast track. In the third view,  $f$  faulty nodes join the  $f + 1$   
824 honest nodes that voted for  $v_1$  in the first view. They send the leader a *CC* for  $v_1$ , hence  
825 the protocol proceeds with  $v_1$ , in conflict with the  $v_2$  commit. The attack on Zyzzyva needs  
826 only  $n = 4$  nodes, of which  $f = 1$  is faulty, and it is fairly easy to re-instate using the Twins  
827 framework. There are four nodes,  $(D, E, F, G)$ . To model the case that  $D$  is Byzantine, it  
828 has a twin  $D'$  initialized with different input. We drive the execution creating partitions and  
829 electing leaders at each step, according to the attack described above. We describe below  
830 the detailed attack using Twins.

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

832 **Step 2** During View 1:

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

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

- 840   – Create the following partitions:  $P_1 = \{D', E, \underline{G}\}$ ,  $P_2 = \{D, F\}$ .
- 841   – Run  $G$  as leader, and let it collect (NEW-VIEW) messages from  $D'$  and  $E$ . Using
- 842   Validity Rule 2 (Appendix C.1),  $G$  decides to propose for  $v_2$ .
- 843   – Remove all partitions, i.e.,  $P = \{D, D', E, F, \underline{G}\}$ .
- 844   –  $G$  proposes  $v_2$ , and collects votes from everyone. This leads to a commit of  $v_2$ .

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

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

## 852 **D Detailed Liveness Attack on FaB**

853 We present a summary of FaB, and use Twins to reinstate a known liveness attack on FaB [1].

854 We use the notation described in Section 2.

### 855 **D.1 Summary of FaB**

856 FaB is a single-shot consensus protocol for the partial synchrony setting with  $n = 3f + 1$ .<sup>7</sup>

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

### 862 **D.2 Liveness Attack on FaB**

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

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

872 **Step 2** During View 1:

<sup>7</sup> 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  $t = 0$ .

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

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

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

## 888 **E Detailed Liveness Attack on Sync HotStuff**

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

### 892 **E.1 Summary of Sync HotStuff**

893 The preliminary version of Sync HotStuff [2] is an SMR solution in the synchronous model  
 894 with  $n = 2f + 1$  parties.<sup>8</sup>

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

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

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

---

<sup>8</sup> 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 Twins approach.

## E.2 Implementing Synchrony Attacks in Twins

Due to the synchronous settings and the nature of the attack which heavily leverages synchrony assumptions, in this case a Twins 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 Twins scheduler to deliver messages to specific parties at specified times. While this is captured by the Twins approach, our current implementation (Section 5) does not support this feature (this will be implemented in future Twins extensions).

Generally, we expect that the granularity of the scheduler timing can be fairly coarse. In particular, there is a known parameter  $\Delta$ , 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  $0.5\Delta$  increments, e.g., at times  $0, 0.5\Delta, \Delta, 1.5\Delta, 2.0\Delta, \dots$ . Therefore, a Twins network emulator could operate in discrete lock-step at  $0.5\Delta$  increments. With this capability in place, the force-locking attack can be re-instated in the Twins approach as described below.

## E.3 Safety Attack on Sync HotStuff

We now rebuild the force-locking attack on the preliminary version of Sync HotStuff using Twins. 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 Twins 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,  $\mathbf{S}_t \xrightarrow{v} S'$  means that  $S$  sends  $v$  to  $S'$  (message arrival time is not known).

To reinstate this attack with Twins, 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 Twins.

**At time  $1.5\Delta$  :**

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

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

**At time  $2.5\Delta$  :**

- $C$  receives  $V_1$ , and broadcasts its vote.

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

**At time  $3\Delta$  :**

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

$$\begin{aligned} 958 & (D, A', B')_{3\Delta} \xrightarrow{blame(A)} \star \\ 959 & (D, A', B')_{3\Delta} \xrightarrow{blame(A)} (D, A', B')_{3\Delta} \end{aligned}$$

960 **At time  $3.5\Delta$  :**

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

$$963 (C)_{2.5\Delta} \xrightarrow{vote(v_1)} (D)_{3.5\Delta}$$

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

$$\begin{aligned} 966 & (A, B)_{3.5\Delta} \xrightarrow{vote(v_1)} \star \\ 967 & (A, B)_{3.5\Delta} \xrightarrow{vote(v_1)} (C)_{3.5\Delta} \end{aligned}$$

968 **At time  $4\Delta$  :**

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

$$\begin{aligned} 971 & (D, A', B')_{3\Delta} \xrightarrow{blame(A)} (C)_{4\Delta} \\ 972 & (C)_{4\Delta} \xrightarrow{blame(A)} \star \end{aligned}$$

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

$$976 (D)_{4\Delta} \xrightarrow{CC(v_0), blame(A)} (B)_{4\Delta}$$

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

$$\begin{aligned} 979 & (D)_{4\Delta} \xrightarrow{CC(v_0), blame(A)} (B)_{4\Delta} \\ 980 & (B)_{4\Delta} \xrightarrow{propose(v_1')} \star \end{aligned}$$

- 981 ■  $D$  receives the proposal  $v_1'$  from  $B$ , and broadcasts its vote with delay  $\Delta$ , then it sets  
 982 its commit timer to  $2\Delta$  and starts counting down.

$$\begin{aligned} 983 & (B)_{4\Delta} \xrightarrow{propose(v_1')} (D)_{4\Delta} \\ 984 & (D)_{4\Delta} \xrightarrow{vote(v_1')} \star \end{aligned}$$

985 **At time  $4.5\Delta$  :**

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

$$\begin{aligned} 988 & \\ 989 & (A, B)_{3.5\Delta} \xrightarrow{vote(v_1)} (D)_{4.5\Delta} \end{aligned}$$

990 **At time  $5\Delta$  :**

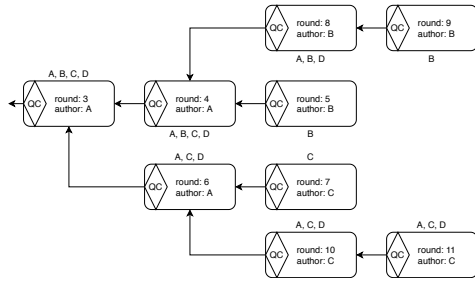
- 991 ■  $C$  has waited for  $\Delta$  since it quit the old view with leader  $A$ , so it starts the next view  
 992  $w + 1$  and sends its highest certificate  $CC(v_1)$  to the new leader  $B$ .

$$993 (C)_{5\Delta} \xrightarrow{CC(v_1)} (B)$$

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

$$996 (D)_{4\Delta} \xrightarrow{vote(v_1')} (C)_{5\Delta}$$





■ **Figure 7** Example of safety attack on Fast-HotStuff.

At time  $6\Delta$  :

- $D$  commits  $v_1'$  since it finished waiting for  $2\Delta$  and observed no equivocation or blame in the view  $w + 1$ . However,  $D$ 's highest certificate is  $CC(v_1)$  (see time  $4.5\Delta$ ).
- 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.

## F Attack on Fast-HotStuff

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

### F.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  $n + x + 1$  ( $x > 0$ ) 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 Appendix F.2.

### F.2 Safety Attack on Fast-HotStuff

Figure 7 illustrates the safety attack against Fast-HotStuff that we implement using Twins. 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.

1029 We execute the safety attack in 11 rounds starting at round 3 (rounds 2 carries QC for  
1030 the genesis block).

1031 **Round 3:** Initially there are no partitions, i.e.,  $P = \{\underline{A}, B, C, D\}$ .

- 1032 ■  $A$  proposes a block. Nodes send their votes on this proposal to the leader of the next  
1033 round, node  $A$ .

1034 **Round 4:**

- 1035 ■  $A$  gathers votes from the previous round, forms a QC, and includes the QC in a new  
1036 block proposal. Nodes send their votes on the new proposal to the leader of the next  
1037 round, node  $B$ .

1038 **Round 5:** Set node  $B$  as leader, i.e.,  $P = \{A, \underline{B}, C, D\}$ .

- 1039 ■  $B$  gathers votes from the previous round, forms a QC, and includes the QC in a new  
1040 block proposal.
- 1041 ■ Create the following partitions:  $P_1 = \{A, C, D\}$  and  $P_2 = \{\underline{B}\}$ .
- 1042 ■ The partitions prevent  $B$  from broadcasting the new block (and the newly formed QC  
1043 it embeds).  $B$  is thus the only node knowing the QC certifying the block of round 4.
- 1044 ■ Nodes of  $P_1$  time out, and send a NEW-VIEW message to the leader of the next round  
1045 (node  $A$ ) containing their highest known QC.

1046 **Round 6:** Set node  $A$  as leader, i.e.,  $P_1 = \{\underline{A}, C, D\}$  and  $P_2 = \{B\}$ .

- 1047 ■  $A$  selects the highest QC from the NEW-VIEW messages (i.e., the QC certifying the  
1048 block of round 3), and embeds it in a new block proposal. All nodes of  $P_1$  vote on this  
1049 proposal and send their votes to the leader of the next round (node  $C$ ).

1050 **Round 7:** Set node  $C$  as leader, i.e.,  $P_1 = \{A, \underline{C}, D\}$  and  $P_2 = \{B\}$ .

- 1051 ■  $C$  gathers votes from the previous round, forms a QC, and includes the QC in a new  
1052 block proposal.
- 1053 ■ Create partitions:  $P_1 = \{A, B, D\}$  and  $P_2 = \{\underline{C}\}$ .
- 1054 ■ These partitions prevent  $C$  from broadcasting the new block (and the newly formed  
1055 QC it embeds).  $C$  is thus the only node knowing the QC certifying the block of round  
1056 6.
- 1057 ■ Nodes of  $P_1$  time out and send a NEW-VIEW message to the leader of the next round  
1058 (node  $B$ ) containing their highest known QC.

1059 **Round 8:** Set node  $B$  as leader, i.e.,  $P_1 = \{A, \underline{B}, D\}$  and  $P_2 = \{C\}$ .

- 1060 ■  $B$  selects the highest QC from the NEW-VIEW messages (i.e., the QC certifying the  
1061 block of round 4, presented by  $B$ ), and embeds it in a new block proposal. All nodes  
1062 vote on this proposal and send their votes to the leader of the next round (node  $B$ ).

1063 **Round 9:**

- 1064 ■  $B$  gathers all votes from the previous round, forms a QC, and includes the QC in a  
1065 new block proposal.
- 1066 ■ Create partitions  $P_1 = \{A, C, D\}$  and  $P_2 = \{\underline{B}\}$ .
- 1067 ■ The partitions prevent  $B$  from broadcasting its newly block (and the newly formed QC  
1068 it embeds).  $B$  is thus the only node knowing the QC certifying the block of round 8  
1069 and committing the block at round 4.
- 1070 ■ Nodes of  $P_1$  time out and send a NEW-VIEW message to the leader of the next round  
1071 (node  $C$ ) containing their highest known QC.

1072 **Round 10:** Set node  $C$  as leader, i.e.,  $P_1 = \{A, \underline{C}, D\}$  and  $P_2 = \{B\}$ .

- 1073 ■  $C$  selects the highest QC from the NEW-VIEW messages from the previous round  
1074 (the QC certifying the block of round 6, presented by  $C$ ), and embeds it in its new  
1075 block proposal. The highest QC in the NEW-VIEW messages.

- 1076     ■ All nodes of  $P_1$  vote on this proposal and send their votes to the leader of the next  
1077       round (node  $C$ ).

1078 **Round 11:** Set node  $C$  as leader, i.e.,  $P_1 = \{A, \underline{C}, D\}$  and  $P_2 = \{B\}$ .

- 1079     ■  $C$  assembles votes from the previous round into a QC certifying the block of round 10,  
1080       thus committing the block of round 6.

1081 The safety violation appears at round 11 when node  $C$  commits the block of round 6 while  
1082 node  $B$  previously committed the block of round 4: both blocks have the same height and  
1083 fork from the block of round 3.

### 1084 F.3 Implementation of the Attack

1085 We implemented a Python simulator of Fast-HotStuff using the discrete event simulator  
1086 *simpy*. We demonstrate the safety violation by running a manually-crafted scenario in the  
1087 simulator. We are open sourcing our Fast-HotStuff simulator as well as our Twins scenario  
1088 used for the attack<sup>9</sup>.

---

<sup>9</sup> <https://github.com/asonnino/twins-simulator>