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
Diem Association and Facebook Novi, United States

Abstract

Twins is an effective strategy for generating unit-test scenarios with *Byzantine* attacks in order to find flaws in Byzantine Fault Tolerant (BFT) [10] systems. The main idea of Twins is the following: running twin instances of a node that use correct, unmodified code and share the same network identity and credentials allows to emulate most interesting Byzantine attacks. Because a twin executes normal, unmodified node code, building Twins only requires a thin wrapper over an existing distributed system designed for Byzantine tolerance. To emulate material, interesting attacks by a Byzantine node, it instantiates one or more *twin* copies of the node, giving the twins the same identities and network credentials as the original node. To the rest of the system, the node and all its 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 behavior scenarios that can be filtered at the transport layer, such as producing semantically invalid messages.

Building on configurations with twin nodes, Twins systematically generates Byzantine attacks via enumeration over protocol rounds and 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 the release process of a production setting (DiemBFT [7]) in which it can execute 44M Twins-generated scenarios daily.

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.DISC.2021.41

Category Brief Announcement

Related Version The full version of this work is hosted on ArXiv.
Full Version: `https://arxiv.org/abs/2004.10617`

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

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

1 The Twins Approach

Twins systematically constructs unit scenarios in which some nodes have one or more twins, and the adversary can delay and drop messages between nodes, i.e., the communication is asynchronous. Twins scenarios are constructed with logical protocol rounds. For each round, the scenario indicates which nodes have twins and which nodes can be reached by other nodes. In addition, each round can designate which nodes are acting as *leaders*, which is a common role in BFT protocols. Executing Twins scenarios requires a thin shim layer that emulates message scheduling and delivery and has a handle to designate a protocol leader.

In notation, nodes are represented by capital alphabets letters (e.g., A) and the twin of a node is represented by the same letter with the prime symbol (e.g., A'). Nodes acting in leader roles are underlined, e.g., $\underline{A}$. We denote partitions of nodes by sets P_*, as in $P_1 = \{A, B, C, D\}$, $P_2 = \{E, F, G\}$. For example, a single-round scenario in which a leader equivocates in the first round and partitions the system into two sets of nodes, each getting a different proposal, can be described as follows:

- Set up a system with nodes $\{D, D', E, F, G\}$.
- Initialize D and D' with different inputs v_1 and v_2.
- Execute round 1 with partitions $P_1 = \{\underline{D}, E, G\}$, $P_2 = \{\underline{D'}, F\}$.

Although enumerating round-by-round scenarios is inherently exponential, experience shows that protocols with logical flaws break with a handful of nodes in less than a dozen rounds (see e.g., Ittai et al. [1]). Indeed, the full paper shows several succinct Twins scenarios that expose known BFT protocol flaws, as well as a scenario that surfaces a flaw in a recent protocol that hasn't been exposed before. Of these, we chose to present in Section 3 one Twins scenario. It demonstrates that in Tendermint [4] and Capser [5], a leader must delay the maximal transmission bound; removing this delay would break liveness.

2 Preliminaries

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. Practical Byzantine Fault Tolerance (PBFT) [6] is a hallmark work that was designed to work efficiently in the asynchronous setting. Carrying the classical PBFT solution to the blockchain world, Tendermint [4] and Capser [5] introduced a much simplified *linear* strategy for leader-replacement. However, it has been observed [3, 12] that this strategy forgoes an important property of asynchronous protocols—*Responsiveness*—the ability of a leader to advance as soon as it receives messages from $2f + 1$ nodes.[1]

[1] Tendermint is a precursor to HotStuff [13] and DiemBFT [7] which operates in two-phase views, but has no Responsiveness. HotStuff/DiemBFT solve this by adding a third phase.

3 Example: A Flawed Tendermint Variant

If the leader's delay was removed from Tendermint (equiv Casper), the protocol would lose liveness. In a nutshell, the flawed variant works as follows. A quorum certificate *(QC)* is formed on a leader proposal if it gathers $2f+1$ votes from nodes. A leader proposes to extend the highest QC it knows. Nodes vote on the leader proposal if it extends the highest QC they know. A commit decision on the leader proposal forms if it gathers $2f+1$ votes forming a QC, and then $2f+1$ nodes vote for that QC. Progress is hinged on leaders obtaining the highest QC in the system, otherwise liveness is broken.

We demonstrate through a Twins scenario that liveness is broken. Lack of progress is detected by observing that two consecutive views with honest leaders whose communication with a quorum is timely do not produce a decision.

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 re-proposes 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 itself 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: $P_1 = \{D, E\}$, $P_2 = \{\underline{D'}, F\}$, $P_3 = \{\underline{G}\}$. D broadcasts QC(v_1), which only reaches P_1 i.e., (D, E).

View 2: Drop all proposals from D and D' until View 2 starts.
- Remove all partitions, i.e., $P = \{D, D', E, \underline{F}, G\}$.
- Let F run as leader for one round. F re-proposes v_2 (i.e., D''s proposal in the previous round) to P. (D, E) do not vote as they already have QC(v_1) for that height. F gathers votes from the other nodes and forms QC(v_2).
- Create partitions $P_1 = \{D, E\}$, $P_2 = \{\underline{D'}, F\}$, $P_3 = \{\underline{G}\}$.
- F broadcasts QC(v_2), which only reaches P_2.

View 3: Drop all proposals from F until View 3 starts.
- Create the partitions $P_1 = \{D, \underline{E}, G\}$, $P_2 = \{\underline{D'}, F\}$.
- Let E run as leader for one round. E proposes v_3 which extends the highest QC it knows, QC(v_1). As before, E manages to form Q(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 itself indefinitely, resulting in the loss of liveness.

4 What Else?

The full version of the paper presents a new attack against Fast HotStuff [8] and several known attacks on BFT protocols (Zyzzyva [9], FaB [11], Sync HotStuff [2]) expressed as Twins scenarios. In all cases, exposing vulnerabilities requires only a small number of nodes, partitions, rounds and leader rotations. We implement an automated scenario generator for Twins and show that our implementation covers the described attacks within minutes.

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. Ethan Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. `https://cdn.relayto.com/media/files/LPgoWO18TCeMIggJVakt_tendermint.pdf`, 2016.
4. Ethan Buchman, Jae Kwon, and Zarko Milosevic. The Latest Gossip on BFT Consensus. arXiv preprint arXiv:1807.04938, 2018.
5. Vitalik Buterin and Virgil Griffith. Casper the Friendly Finality Gadget. arXiv preprint arXiv:1710.09437, 2017.
6. Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *USENIX Symposium on Operating Systems Design and Implementation*, 1999.
7. Diem. DiemBFT. `https://github.com/diem/diem`.
8. Mohammad M Jalalzai, Jianyu Niu, and Chen Feng. Fast-hotstuff: A fast and resilient hotstuff protocol. arXiv preprint arXiv:2010.11454, 2020.
9. 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.
10. Leslie Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.
11. J-P Martin and Lorenzo Alvisi. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.
12. Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT Consensus in the Lens of Blockchain. arXiv preprint arXiv:1803.05069, 2018.
13. Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT Consensus with Linearity and Responsiveness. In *ACM Symposium on Principles of Distributed Computing*, 2019.