A Benchmark Framework for Byzantine Fault Tolerance Testing Algorithms

Louro Neto, João Miguel; Kulahcioglu Ozkan, Burcu

DOI
10.4230/OASIcs.FMBC.2025.13

Publication date
2025

Document Version
Final published version

Published in
6th International Workshop on Formal Methods for Blockchains, FMBC 2025

A Benchmark Framework for Byzantine Fault Tolerance Testing Algorithms

João Miguel Louro Neto
Delft University of Technology, The Netherlands

Burcu Kulahcioglu Ozkan
Delft University of Technology, The Netherlands

Abstract

Recent discoveries of vulnerabilities in the design and implementation of Byzantine fault-tolerant protocols underscore the need for testing and exploration techniques to ensure their correctness. While there has been some recent effort for automated test generation for BFT protocols, there is no benchmark framework available to systematically evaluate their performance.

We present ByzzBench, a benchmark framework designed to evaluate the performance of testing algorithms in detecting Byzantine fault tolerance bugs. ByzzBench is designed for a standardized implementation of BFT protocols and their execution in a controlled testing environment. It controls the nondeterminism in the concurrency, network, and process faults in the protocol execution, enabling the functionality to enforce particular execution scenarios and thereby facilitating the implementation of testing algorithms for BFT protocols.

2012 ACM Subject Classification Security and privacy → Distributed systems security

Keywords and phrases Byzantine Fault Tolerance, BFT Protocols, Automated Testing

Digital Object Identifier 10.4230/OASIcs.FMBC.2025.13

Category Tool Paper

Funding *Burcu Kulahcioglu Ozkan*: This work has been supported by the University Blockchain Research Initiative (UBRI), funded by Ripple.

1 Introduction

Byzantine Fault-Tolerant (BFT) protocols are at the heart of modern consortium-based blockchain systems. These systems use BFT protocols to reach consensus on the total order of transactions to commit, among a cluster of processes in a fault-tolerant manner. BFT protocols promise correctness even if some processes in the cluster behave maliciously or fail to follow the protocol specification. Along with the increasing popularity of blockchain systems in the last decades, numerous BFT protocols have been proposed.

The correct design and implementation of BFT protocols is crucial to ensure the correct functioning of these systems. However, BFT systems are prone to Byzantine fault tolerance bugs, i.e., flaws or errors that break their fault-tolerance and correctness properties in the presence of Byzantine and network attacks. These bugs allow attackers to exploit vulnerabilities in the system, undermining its ability to maintain correct behavior, as any mistake can lead to serious reliability and security problems with potentially catastrophic consequences. Several studies [1, 27, 21, 5, 26, 34] have discovered vulnerabilities in various BFT protocols and their implementations, underscoring the need for robust analysis and testing methodologies.

Recent work [6, 34] proposes new methods for automated testing of Byzantine fault tolerance in large-scale distributed systems. Twins [6], which systematically tests for Byzantine fault-tolerance by simulating Byzantine processes using twin replicas, is implemented in DiemBFT [32] and is available in Diem's production repository. ByzzFuzz [34], a randomized

test generator simulating Byzantine faults using message mutations, is available to test the implementations of Tendermint [8] and the XRP Ledger [29]. Given the discovery of new Byzantine fault tolerance bugs in production BFT systems and their increasing popularity, we can expect the development of more analysis and testing methods to analyze their correctness.

Despite advancements in automated test generation for Byzantine Fault Tolerance (BFT) protocols, the evaluation of these algorithms is often limited to the specific systems for which they were developed. The algorithms are assessed based on different implementations of BFT systems, making it challenging to compare their results across various contexts. To enable a comparative evaluation of their performance, it is necessary to have a collection of benchmark applications that represent a diverse range of bugs. While such benchmark suites exist for concurrency bugs such as RADBench [19] and JaConTeBe [22] or for programs with defects in specific programming languages such as BegBunch [12], Defects4J [20], BEARS [24], and GoBench [35] no such benchmark suite is available for Byzantine fault tolerance bugs. ByzzBench specifically addresses these limitations by enabling Byzantine fault injection capabilities and providing specialized property checkers, making it uniquely tailored to the challenges faced in the design and implementation of BFT protocols.

A major challenge for building a *benchmark suite* of Byzantine fault tolerance bugs in BFT protocols is the lack of a *benchmarking framework* that provides functionality for (i) uniform implementation of BFT protocols and (ii) a controlled protocol execution environment to facilitate the implementation of testing algorithms. Testing algorithms for BFT algorithms generate execution scenarios with particular concurrency, network, and process behavior, e.g., with particular delivery ordering or timing of the protocol messages, network faults to delay or lose some messages, and Byzantine process faults with specific malicious behavior. The enforcement of the generated test scenarios requires controlling the nondeterminism in the executions of the systems under test to exercise the generated specific test scenarios, which demands technical effort and makes it hard to test a broad set of BFT systems.

We present ByzzBench [28], a publicly available and extensible benchmark framework for implementing BFT protocols and evaluating the performance of the testing algorithms in detecting Byzantine fault tolerance bugs. The contribution of ByzzBench is two-fold. First, it allows protocol designers to validate their protocol implementations by testing them using the testing and exploration algorithms in the framework. Second, it allows functionality and interfaces for implementing Byzantine fault tolerance testing protocols in a controlled execution environment, offering a unified evaluation of testing methods on a set of benchmark protocol implementations. In summary, ByzzBench allows the users to:

- *prototype* BFT protocols in a unified framework of benchmark applications,
- *implement* testing and exploration methods for BFT protocols in a unified framework,
- *evaluate* relative performances of different testing and exploration methods,
- *gain insight* about the BFT protocol vulnerability characteristics.

The long-term goal of this work is to build a comprehensive, publicly available benchmark suite of BFT protocol implementations to evaluate the effectiveness of testing and exploration tools for BFT protocols. As an initial step, this paper presents a benchmark framework for BFT protocol implementations.

2 The Benchmark Framework

Figure 1 presents a high-level overview of the architecture of ByzzBench, with the main components of (i) a cluster of processes running a BFT protocol for state machine replication, (ii) client processes that submit requests to the cluster, (iii) controlled transport layer which

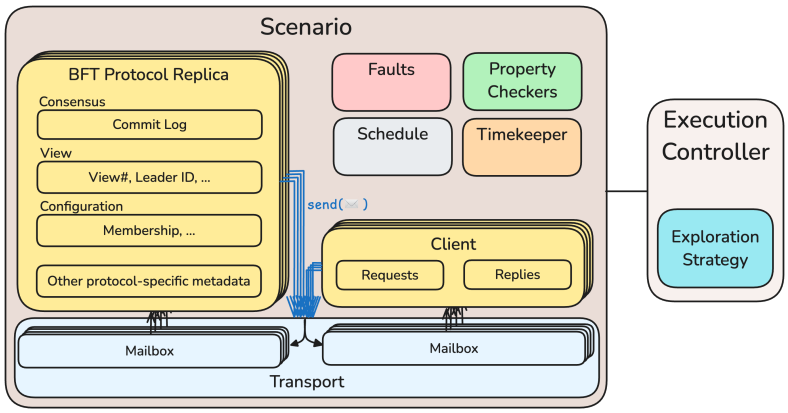

Figure 1 High-level architecture of the ByzzBench framework.

maintains the in-flight messages and delivers them to recipients, and (iv) an execution controller, which dictates a specific execution scenario to the transport layer. The execution controller allows the implementation of different BFT testing algorithms by offering control over the timeouts, network faults, and benign and Byzantine process faults.

ByzzBench allows the implementation of BFT protocols by only focusing on the protocol logic, avoiding the boilerplate code for state machine replication, message communication, and command log operations. Similarly, it provides the controlled replica communication and fault injection features available for test scenario generation algorithms. Hence, the algorithm developers can implement their algorithms on top of the intercepted messages, decoupled from the implementation of the protocol under test. Moreover, it has available correctness checkers that run on the recorded replicated logs, which check for generic consensus properties and can be extended with more checks by developers.

Although our current benchmarks and examples primarily target BFT consensus protocols, ByzzBench is designed to be applicable to any BFT protocol, including those used for state machine replication and broadcast.

2.1 Implementing BFT Protocol Benchmarks

BFT protocols establish a set of rules for message exchanges that enable a group of processes to coordinate in the existence of some arbitrary process behavior. Each message includes specific metadata related to the protocol, such as the current protocol step, which helps the receiving processes understand the protocol's state and respond appropriately. The executions of the protocols are organized as a sequence of lock-step *communication rounds* in which processes exchange messages following the protocol rules. In every round, the processes can send messages to the other processes, receive and process the messages delivered in that round, and update their local states accordingly. This process involves updating the local state of the receiving process and may also include sending, multicasting, or broadcasting new protocol messages to other processes.

ByzzBench models a cluster of *Replica* processes, which run the protocol for state machine replication of processing *Client* requests. The *Client* processes submit operation requests to the cluster and collect replies from the replicas. Given a set of concurrent client requests, the *Replica* processes run the BFT protocol to agree on a total order of client requests to commit in a similar fashion to Paxos or Raft protocols for state machine replication or the commitment of a total order of transactions in a blockchain system.

ByzzBench enables practical implementation of BFT protocol benchmarks by offering essential features such as setting up a distributed cluster, enabling high-level message communication and handling, and managing local replica logs that are typically employed in BFT protocols. The framework implements *Replica* processes using the actor model of programming [18], where each actor runs independently on their local states concurrently to each other and communicates via point-to-point message exchanges over a potentially faulty network. The local state of each *Replica* keeps the data for state machine replication, including the local *commit log*, protocol state, cluster configuration, and other meta-data.

Implementing a BFT protocol benchmark in ByzzBench requires only extending a Replica abstract class and implementing the initialization and message handler methods. To simplify the implementation of message handlers, ByzzBench provides an API that allows Replicas to exchange messages using various methods, including sending, broadcasting, and multicasting. It also includes options for setting timeouts on time-triggered messages and a common API for managing the commit log of a Replica, as provided in Section B.

ByzzBench currently provides two implementations of the PBFT protocol [11], i.e., the buggy implementation in [10, 34] along with its correct implementation. Our ongoing work extends the framework with the implementations of the BFT protocols in Appendix A.

2.2 Modeling Time

BFT protocols often employ time-triggered messages, which resend some protocol messages or propose to advance protocol steps (e.g., for electing a new leader or moving to a new view) in case of suspected failures. The nondeterminism in the time-triggered messages makes it challenging for the BFT exploration algorithms to enumerate or reproduce executions.

ByzzBench controls the advancement of time during the protocol execution to ensure reproducibility and enumeration. This is achieved through the *Timekeeper* component, which manages time progression within each execution. Time-triggered messages can be implemented by timeout-based method calls in the BFT replica interface to enable some events only when a timeout has been reached. The *Timekeeper* also provides methods for getting and advancing time to be used by the protocol implementations.

The test scenario generators can trigger the protocol's timeout messages by advancing the corresponding replica's time value to the designated trigger value. This allows for the controlled execution of time-based messages. The framework does not synchronize the clocks of the distributed replicas and allows their local clocks to drift apart in certain execution scenarios, as in the real-world execution of the protocols.

2.3 Modeling and Injecting Network and Process Faults

BFT protocols promise correctness even in the existence of network and Byzantine process faults. However, recent studies [1, 27, 21, 5, 26, 34] have shown that several BFT protocols fail to uphold their assumptions under these conditions. Therefore, it is crucial for a testing algorithm to generate scenarios that incorporate network faults, such as isolating specific processes or partitioning the network, as well as process faults, including benign crashes or Byzantine behaviors.

Testing algorithms for consensus protocols [14, 6, 34] inject network and process faults based on the *communication rounds* of the consensus protocols. The communication round provides an abstraction of the protocol state and the step of the execution that is useful for effectively specifying the period of faulty behaviors.

ByzzBench supports round-based fault injection using an extended model of predicate-based fault injection. The model describes a fault as a pair consisting of a boolean predicate and a side effect, represented as (*predicate*, *effect*). The side effect *effect* on a message is applied only if the message satisfies the *predicate*. This model facilitates round-based fault injection by allowing the specification of the predicate for a particular protocol round, receiver, and sender. Moreover, it enables specifying more general conditions (e.g., not only checking the protocol round but any other information of the in-flight message or distributed cluster state) for injecting faults for certain messages.

Operationally, ByzzBench injects specific faults into the execution of a BFT protocol operating at the Transport layer. The Transport layer shown in Figure 1 maintains the in-flight messages sent to the Replicas in their mailboxes. Depending on the prescribed network faults in the scenario, it reorders the delivery of messages or drops some messages, e.g., to isolate a process from the network or partition the network by dropping the messages exchanged between partitions. Similarly, it can inject process faults by omitting, duplicating, or modifying the in-flight messages sent from a Byzantine Replica.

2.4 Implementing Testing Algorithms for BFT Protocols

ByzzBench enables implementing algorithms for testing and exploration of BFT protocol implementations using an *Exploration Strategy* interface. Simply, the developers implement a testing algorithm by providing fault injection predicates as explained in Section 2.3 and implement the functionality for selecting and processing the next in-flight message from the transport layer. The transport layer enforces the delivery of or fault injection into the selected in-flight message, as dictated by the exploration strategy. During the execution, the processed messages and injected faults are recorded in an execution schedule and made available for inspection after the execution.

ByzzBench currently provides the implementations of the following testing algorithms for BFT protocols: a naive random fault injection algorithm, which decides to deliver, drop or mutate messages at each execution step; ByzzFuzz randomized testing algorithm [34], which models Byzantine faults using round-based small-scope message mutations, and a preliminary implementation of the Twins systematic test scenario generator using twin replicas of the processes. Our ongoing work explores the relative performances of these testing algorithms in the exploration of bugs in the BFT protocol implementations.

2.5 Correctness Specification of BFT Protocol Executions

ByzzBench's *Property Checker* checks the correctness of BFT protocol test executions by checking the following correctness properties of BFT consensus [9]: (i) Agreement: No two processes decide differently, (ii) No correct process decides twice, (iii) Validity: A correct process may only decide a value that was proposed by a correct process, and (iv) Termination: Every correct process eventually decides on some value.

Typically, BFT protocols promise the safety properties of agreement, integrity, and validity under both synchronous and asynchronous network conditions. For the liveness property of *termination*, most protocols rely on a partially synchronous network [15] to overcome the FLP impossibility result [16] for asynchronous networks.

ByzzBench simulates a variant of partial synchrony, called eventual synchrony, by employing a graceful period of execution without any faults after the execution of a test scenario with some network and process faults. While eventual synchrony satisfies the network assumptions of some BFT protocols, some protocols (e.g., Tendermint [8], Sync

HotStuff [3]) rely on stronger synchronization and delivery requirements. ByzzBench leaves the enforcement of the network assumptions of protocols to the developers, which depends on the BFT protocol under test and the test generation algorithm.

ByzzBench checks the termination property by checking the two conditions of *deadlock* and *violation of bounded termination*. The deadlock condition occurs when all the replicas' mailboxes are empty, i.e., there are no available protocol messages to process, and the cluster has not yet reached consensus. An example of such a deadlock condition occurs in the violating execution of Fast Byzantine Consensus [25] uncovered in recent work [2]. In that execution, the protocol rules lead to a stuck network state, where the processes do not exchange any more messages and cannot make a decision. The violation of bounded termination occurs when the processes continue exchanging messages, but the communication does not lead to consensus in a predefined, bounded length of execution. The violations to bounded termination do not guarantee that the detected execution is a violation of (unbounded) termination, as it can return false positives. However, it is useful in detecting termination violations. An example violation detected by this condition is the termination violation in a previous version of the XRP Ledger [34], where some process faults lead to corrupted states of some processes, preventing them from achieving consensus despite ongoing message communication. ByzzBench checks bounded termination by checking agreement for the client requests after an execution with a bounded duration of test execution.

ByzzBench checks the safety properties of agreement, integrity, and validity using the replicas' commit logs and the exchanged messages between the processes, which keep the proposed values and commitment decisions.

In addition to these properties, the *Property Checker* supports specifying more expressive predicates, allowing ByzzBench to be extended with progress-related checks that are crucial for uncovering subtle liveness and fairness issues not captured by deadlock or bounded termination alone.

3 Preliminary Evaluation on PBFT

We demonstrate how ByzzBench can be used to assess the effectiveness of different testing algorithms through a preliminary evaluation on the seminal PBFT [11] protocol.

PBFT provides distributed agreement in a cluster of $3f + 1$ processes, tolerating up to f faulty processes, which can deviate arbitrarily from the protocol specification. An execution of the PBFT protocol is decomposed into views, in each of which one of the processes acts as a leader. In each view, the leader executes a sequence of client operations. For each request, the leader broadcasts a proposal, followed by two rounds of message exchanges in which the participants vote for the proposal. If a quorum agrees on the proposal, they execute the operation and reply with the result to the client. Processes store the total order of executed client operations in their message logs, and the protocol ensures that the correct processes agree on their contents.

The ByzzBench framework provides a buggy implementation of the PBFT protocol as a benchmark, seeding the same implementation errors found in the publicly available implementation, PBFT-Java [10]. The benchmark is vulnerable to agreement violations due to (i) incorrect assignment of sequence numbers in the protocol messages, (ii) incorrect processing of prepared certificates, and (iii) missing implementation of message digests.

We compare the performance of detecting the agreement violations in the benchmark using a recent testing algorithm, ByzzFuzz, compared to the naive random testing algorithm. Table 1 lists the percentage of test executions out of 1000 runs that detect violations using

Table 1 Percentage of test executions (out of 1000 runs) in which random and ByzzFuzz testing strategies detected violations in PBFT-Java benchmark. ByzzFuzz results (denoted BF(c, d)) represent runs with c process and d network partition faults injected over 10 rounds ($r = 10$).

Random	BF(0,1)	BF(0,2)	BF(1,0)	BF(1,1)	BF(1,2)	BF(2,0)	BF(2,1)	BF(2,2)
0.0%	0.0%	0.0%	7.1%	5.4%	5.0%	11.6%	9.2%	9.5%

random testing and the ByzzFuzz algorithm. We evaluate ByzzFuzz using different algorithm parameters, i.e., $c = [0, 2]$ process faults and $d = [0, 2]$ network partition faults distributed into $r = 10$ protocol rounds of the execution. Our results show that ByzzFuzz can detect violations more frequently than the naive random testing, in alignment with the findings in the previous work [34]. Moreover, we observe that the increasing number of faults injected into the execution increases the likelihood of exposing violations.

4 Related Work

Several benchmark frameworks have been designed for evaluating the testing approaches to detect concurrency bugs [19, 22], data storage bugs [23], or benchmark programs in specific programming languages [12, 20, 24, 35]. However, none of these frameworks target Byzantine fault tolerance benchmarks and Byzantine fault tolerance bugs.

Previous work on BFT protocol benchmarks targets performance benchmarking under various workloads, computation power, and network configurations. BFTSim [31] offers a network simulator to evaluate protocol performance based on a range of network conditions. BFT-Bench [17] evaluates and compares the performance of BFT protocols under faulty network behaviors and workloads. BLOCKBENCH [13] targets private blockchains, allowing fine-grained testing of the blockchain system to evaluate their performance under smart contract workloads. BFTDiagnosis [33] is an automated fault injection framework for the security evaluation of BFT protocols. Bedrock [4] explores the trade-offs between different design space dimensions to determine the protocol that best meets application needs.

Unlike the existing work, which focuses on the performance of BFT protocols, ByzzBench targets the correctness of BFT protocols and provides a framework to evaluate the performances of exploration algorithms for detecting bugs in BFT protocol implementations.

5 Conclusions

We presented ByzzBench, an extensible benchmark framework for evaluating the testing algorithms for Byzantine fault tolerance bugs. The framework offers practical implementation of BFT protocols and provides functionality for developing testing algorithms to control the time, network, and process fault nondeterminism in the executions of test scenarios.

ByzzBench lays the groundwork for benchmarking in this space, although it currently has a limited set of built-in benchmark protocols and testing algorithms. At present, the available benchmark applications include both correct and faulty implementations of PBFT, and the built-in testing algorithms consist of a basic random tester, ByzzFuzz, and an initial version of Twins. Our ongoing efforts aim to expand the range of implemented BFT protocols, particularly those with known bugs, and to integrate state abstraction mechanisms.

The long-term objective of this work is to develop a comprehensive, publicly accessible benchmark suite of BFT protocol implementations to assess the effectiveness of testing and exploration algorithms for BFT protocols.

References

1 Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting Fast Practical Byzantine Fault Tolerance, December 2017. arXiv:1712.01367.

2 Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting Fast Practical Byzantine Fault Tolerance: Thelma, Velma, and Zelma, January 2018. Comment: arXiv admin note: text overlap with arXiv:1712.01367. arXiv:1801.10022.

3 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118, San Francisco, CA, USA, May 2020. IEEE. doi:10.1109/SP40000.2020.00044.

4 Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 371–400, Santa Clara, CA, April 2024. USENIX Association. URL: https://www.usenix.org/conference/nsdi24/presentation/amiri.

5 Ignacio Amores-Sesar, Christian Cachin, and Jovana Micic. Security Analysis of Ripple Consensus. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, volume 184 of *LIPIcs*, pages 10:1–10:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.OPODIS.2020.10.

6 Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. Twins: BFT Systems Made Robust. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *25th International Conference on Principles of Distributed Systems, OPODIS 2021, December 13-15, 2021, Strasbourg, France*, volume 217 of *LIPIcs*, pages 7:1–7:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.OPODIS.2021.7.

7 Christian Berger, Hans P. Reiser, and Alysson Bessani. Making reads in BFT state machine replication fast, linearizable, and live. In *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021*, pages 1–12. IEEE, 2021. Comment: 12 pages, to appear in the 40th IEEE Symposium on Reliable Distributed Systems (SRDS). doi:10.1109/SRDS53918.2021.00010.

8 Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, University of Guelph, 2016.

9 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. Ed.)*. Springer, 2011. doi:10.1007/978-3-642-15260-3.

10 Johnny Cao. A Practical Byzantine Fault Tolerance (PBFT) emulator written in Java, 2020. http://github.com/caojohnny/pbft-java.

11 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999. URL: https://dl.acm.org/citation.cfm?id=296824.

12 Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. BegBunch: Benchmarking for C bug detection tools. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 16–20, Chicago Illinois, June 2009. ACM. doi:10.1145/1555860.1555866.

13 Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In Semih Salihoglu,

Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1085–1100. ACM, 2017. Comment: 16 pages. doi:10.1145/3035918.3064033.

14 Cezara Dragoi, Constantin Enea, Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. Testing consensus implementations using communication closure. *Proc. ACM Program. Lang.*, 4(OOPSLA):210:1–210:29, 2020. doi:10.1145/3428278.

15 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.

16 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

17 Divya Gupta, Lucas Perronne, and Sara Bouchenak. BFT-Bench: Towards a Practical Evaluation of Robustness and Effectiveness of BFT Protocols. In Márk Jelasity and Evangelia Kalyvianaki, editors, *Distributed Applications and Interoperable Systems - 16th IFIP WG 6.1 International Conference, DAIS 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9687 of *Lecture Notes in Computer Science*, pages 115–128. Springer, 2016. doi:10.1007/978-3-319-39577-7_10.

18 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, August 1973. Morgan Kaufmann Publishers Inc.

19 Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. Radbench: a concurrency bug benchmark suite. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, page 2, USA, 2011. USENIX Association. URL: https://www.usenix.org/conference/hotpar-11/radbench-concurrency-bug-benchmark-suite.

20 René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, San Jose CA USA, July 2014. ACM. doi:10.1145/2610384.2628055.

21 Minjeong Kim, Yujin Kwon, and Yongdae Kim. Is Stellar As Secure As You Think? In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 377–385, Stockholm, Sweden, June 2019. IEEE. doi:10.1109/EuroSPW.2019.00048.

22 Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. JaConTeBe: A benchmark suite of real-world java concurrency bugs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–189, November 2015. doi:10.1109/ASE.2015.87.

23 Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

24 Fernanda Madeiral, Simon Urli, Marcelo de Almeida Maia, and Martin Monperrus. BEARS: an extensible java bug benchmark for automatic program repair studies. In Xinyu Wang, David Lo, and Emad Shihab, editors, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 468–478. IEEE, 2019. doi:10.1109/SANER.2019.8667991.

25 J.-P. Martin and L. Alvisi. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006. doi:10.1109/TDSC.2006.35.

26 Lara Mauri, Stelvio Cimato, and Ernesto Damiani. A formal approach for the analysis of the XRP ledger consensus protocol. In Steven Furnell, Paolo Mori, Edgar R. Weippl, and Olivier Camp, editors, *Proceedings of the 6th International Conference on Information Systems*

Security and Privacy, ICISSP 2020, Valletta, Malta, February 25-27, 2020, pages 52–63. SCITEPRESS, 2020. doi:10.5220/0008954200520063.

27 Atsuki Momose. Force-Locking Attack on Sync Hotstuff. *IACR Cryptol. ePrint Arch.*, page 1484, 2019. URL: https://eprint.iacr.org/2019/1484.

28 João Miguel Louro Neto. ByzzBench: A benchmark framework for byzantine fault tolerance testing algorithms, 2024. https://github.com/joaomlneto/byzzbench. URL: https://github.com/joaomlneto/byzzbench.

29 David Schwartz, Noah Youngs, and Arthur Britto. The Ripple Protocol Consensus Algorithm. *Ripple Labs Inc White Paper*, 2014.

30 Nibesh Shrestha, Mohan Kumar, and SiSi Duan. Revisiting hBFT: Speculative Byzantine Fault Tolerance with Minimum Cost, April 2019. Comment: 4 pages. arXiv:1902.08505.

31 Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT protocols under fire. In Jon Crowcroft and Michael Dahlin, editors, *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 189–204. USENIX Association, 2008. URL: http://www.usenix.org/events/nsdi08/tech/full_papers/singh/singh.pdf.

32 The Diem Team. DiemBFT v4: State Machine Replication in the Diem Blockchain, 2021.

33 Jitao Wang, Bo Zhang, Kai Wang, Yuzhou Wang, and Weili Han. BFTDiagnosis: An automated security testing framework with malicious behavior injection for BFT protocols. *Computer Networks*, 249:110404, July 2024. doi:10.1016/j.comnet.2024.110404.

34 Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. Randomized Testing of Byzantine Fault Tolerant Algorithms. *Proc. ACM Program. Lang.*, 7(OOPSLA1):757–788, 2023. doi:10.1145/3586053.

35 Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. Gobench: A benchmark suite of real-world go concurrency bugs. In Jae W. Lee, Mary Lou Soffa, and Ayal Zaks, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 187–199. IEEE, 2021. doi:10.1109/CGO51591.2021.9370317.

A Benchmark BFT Protocols

Our ongoing work extends the currently available BFT protocol benchmarks in ByzzBench with implementations of BFT protocols with known protocol and implementation bugs from the literature, as summarized in Table 2. The table shows the discovery year of the bug, the violation type, and the source of the bug (i.e., whether it is a protocol, implementation, or configuration bug). It also lists the number of processes, protocol views (or ledger blocks), and protocol rounds with process and network faults in the violating execution, along with the reference to the study reporting the violation.

B API for Implementing BFT Protocols

Listings 1 and 2 show the *Replica Interface* for implementing BFT protocols and the API available for the *Replica*s for exchanging messages, setting timeout-based messages, and committing operations to the replicated log.

Table 2 BFT protocol benchmarks with known bugs in their design or implementation.

Year	Protocol	Violation	Bug Source	Execution Parameters				Reference
				#processes	#views (or blocks)	#process faults	#network faults	
2017	FaB	liveness	protocol	4	2	1	2	[1]
2017	Zyzzyva	safety	protocol	4	3	1	4	[1]
2019	hBFT	safety	protocol	4	2	2	2	[30]
2020	Sync HotStuff	safety	protocol	5	3	2	7	[27]
2020	XRPL	liveness	trust config	7	2	1	0	[5, 34]
2020	XRPL	safety	trust config	7	2	2	0	[5, 34]
2021	PBFT	liveness	protocol	4	1	2	0	[7]
2022	Fast-HotStuff	safety	protocol	4	11	0	3	[6]
2023	PBFT	safety	pbft-java	4	2	1	0	[34]
2023	XRPL	liveness	rippled v1.7.2	7	3	1	0	[34]

Listing 1 Replica Interface in ByzzBench, for protocol implementations.

```
// Initialize this Replica
void initialize();

// Handle a message received by this Replica
void handleMessage(String sender, MessagePayload message);
```

Listing 2 API available for Replicas in ByzzBench.

```
// Messaging API
void sendMessage(MessagePayload message, String recipient);
public void broadcastMessage(MessagePayload message);
public void multicastMessage(MessagePayload message, SortedSet<String> recipients);

// Timeouts
long setTimeout(String name, Runnable r, Duration timeout);
void clearTimeout(long eventId);
void clearAllTimeouts();

// Commit Log
void commitOperation(LogEntry operation);

// Time
Instant getCurrentTime();
```