SandTable: Scalable Distributed System Model Checking with Specification-Level State Exploration

Ruize Tang
SKL for Novel Soft. Tech.,
Nanjing University
Nanjing, China
tangruize@smail.nju.edu.cn

Xudong Sun
University of Illinois
Urbana-Champaign, IL, USA
xudongs3@illinois.edu

Yu Huang*
SKL for Novel Soft. Tech.,
Nanjing University
Nanjing, China
yuhuang@nju.edu.cn

Yuyang Wei
SKL for Novel Soft. Tech.,
Nanjing University
Nanjing, China
yuyang_wei@smail.nju.edu.cn

Lingzhi Ouyang
SKL for Novel Soft. Tech.,
Nanjing University
Nanjing, China
lingzhi.ouyang@smail.nju.edu.cn

Xiaoxing Ma
SKL for Novel Soft. Tech.,
Nanjing University
Nanjing, China
xxm@nju.edu.cn

Abstract

Implementation-level distributed system model checkers (DMCKs) have proven valuable in verifying the correctness of real distributed systems. However, they primarily focus on state space reduction, and often have a bottleneck on another crucial dimension: *exploration speed*. To scale DMCK, we introduce SandTable, a technique for lifting state-space exploration from the implementation level to the specification level, and confirming bugs at the implementation level. We made SandTable practical through a methodology consisting of four essential parts: (1) writing specifications that adhere to the implementation, (2) checking conformance to enhance specification quality and reduce false positives and false negatives, (3) exploring the state space with heuristics for effectiveness and efficiency, and (4) confirming bugs and verifying their fixes in the implementation.

We implemented SandTable with the design of transparently verifying unmodified distributed systems on POSIX systems. SandTable was integrated into eight well-established open-source distributed systems that implement consensus protocols such as Raft and Zab. SandTable identified 23 bugs in total, with 18 new bugs, 17 confirmed, and 13 fixed. SandTable demonstrates exceptional scalability, with one machine-day of specification-level exploration checking up to 10^9 distinct states. Furthermore, specification-level exploration offers a significant speedup, 114×—2989× faster than implementation-level exploration.

CCS Concepts: • **Software and its engineering** → **Model checking**; • **Computer systems organization** → **Reliability**.

Keywords: Distributed systems, model checking, reliability

ACM Reference Format:
Ruize Tang, Xudong Sun, Yu Huang, Yuyang Wei, Lingzhi Ouyang, and Xiaoxing Ma. 2024. SandTable: Scalable Distributed System Model Checking with Specification-Level State Exploration. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3627703.3650077

1 Introduction

Implementing correct and reliable distributed systems is notoriously hard. The developers must reason about system behavior in the face of asynchrony, concurrency and faults. Although protocols such as Paxos [70], Zab [65, 66] and Raft [82] have been proved correct [66, 70, 71, 80, 81], implementing such protocols remains a daunting task [42]. Converting the protocols to production-ready systems requires engineering efforts to implement performance optimizations and handle practical constraints (e.g., limited memory or storage resources). Testing is the most widely-adopted practice to find bugs, but it has no guarantee of the coverage of distributed system state space.

Previous work has made great progress in verifying distributed system implementation by applying model checking techniques [57, 58, 67, 74, 76, 77, 87, 91, 92]; for brevity, we use the term *DMCK* [74] to categorize such systems. Implementation-level model checking pushes the target system into corner-case situations and unearths deep bugs by directly exercising the implementation and systematically

*Corresponding author

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3650077

exploring its state space. Distributed systems have large state spaces, hereby making exhaustive state exploration prohibitively expensive. To address the state-explosion problem, existing work has adopted various state-reduction techniques (e.g., DPOR [50] and symmetry [45]) to avoid exploring redundant states in terms of finding bugs, and optimizations such as virtual clock [92] and state-event caching [76] to improve the efficiency of state exploration.

However, existing implementation-level model checkers for distributed systems still suffer from scalability issues when exploring the state space of complex system implementations. Discovering a bug often requires exploring a large number of states at the implementation level. Such explorations are inefficient due to three primary reasons:

- *Stateless exploration*: repeated initializations of the cluster and explorations of the same states.
- *Slow implementation events*: slow events such as network and disk operations.
- *Model checker overhead*: instrumentation overhead to enforce event interleaving and track system state.

Contributions. We propose SandTable, a technique that improves the scalability of implementation-level model checking for distributed systems. SandTable is based on an observation that the scalability of model checking for distributed systems is plagued by not only the large state space, but also the inefficient state exploration at the implementation level.

SandTable accelerates state exploration by lifting it from the implementation level to the specification level, where the state space is explored efficiently on a formal specification of the system. Specification-level state exploration enjoys stateful exploration, and avoids slow implementation events and instrumentation overhead. For example, a bug [20] we found requires the exploration of over $1,500,000$ distinct states. When explored at the specification level using a single worker, it takes only 3.8 minutes. But finding the same bug through exploration of the corresponding implementation-level traces generated from the state space takes 257 hours.

When SandTable detects a bug in the specification, it confirms the existence of the bug in the implementation by deterministically replaying the bug-triggering event sequence at the implementation level. SandTable achieves deterministic replay by enforcing the runtime interleaving of system events, such as message delivery, timeouts, node crashes, and client requests. SandTable does not require any modification to the system to control the interleaving: it transparently interposes between the target system and the operating system using the shared library preloading technique [26].

SandTable requires a formal specification of the target system to perform model checking. Unfortunately, existing specifications may not always precisely describe the behaviors of the implementations, thus introducing both false negatives and false positives during model checking. For example, discrepancies between the specification and the implementation [89] and bugs [14] in the specification were discovered in the Raft specification developed by the protocol designer [81].

SandTable employs a conformance checking mechanism to help develop specifications for their systems in an iterative manner. During the conformance checking phase, SandTable randomly explores the user-provided specification and uses events from specification traces to execute the implementation. Once any discrepancies between the specification state and the implementation state are detected during the execution, SandTable reports the inconsistent variables and the event sequence that leads to the discrepancy. The user then fixes the specification to make it conform to the implementation and reruns the conformance checking until no discrepancy is reported for a pre-specified period of time.

Key results. We implemented SandTable and applied it to eight popular distributed systems that implement consensus protocols such as Raft and Zab. SandTable found 23 bugs in total, 18 of which were new. After reporting the 18 new bugs, 17 have been confirmed and 13 have been fixed. These bugs are deep semantic bugs: they require complex event sequences to trigger (up to 25 events) and lead to severe consequences, including data loss and data inconsistency.

SandTable scales well and efficiently unearths the deep bugs. All the bugs were triggered under one machine hour. Compared to implementation-level state exploration, SandTable explores the state space 114×—2989× faster. The manual efforts spent on improving the specification using conformance checking are manageable. When applying it to the specifications we developed, it took only half a person day to a few person days to manually rectify all specification modeling errors. When adapting existing specifications, such as those of ZooKeeper [36, 83], it took two person weeks to modify the specification to conform to the implementation.

Summary. The paper makes three main contributions.

- We present a novel and efficient model checking technique for exploring specification-level state space to find bugs in complex distributed system implementations.
- We design and implement SandTable, a model checker that uses our proposed technique for unmodified distributed system implementations.
- We integrate SandTable with eight distributed systems, and present detailed evaluations showcasing the effectiveness and efficiency of SandTable.

SandTable is open sourced at https://github.com/tangruize/SandTable, with instructions to reproduce discovered bugs.

2 Background and Motivation

In this section, we first briefly introduce DMCKs. Then we discuss the speed of state-of-the-art DMCKs and its impact on DMCK scalability. These discussions further motivate the speeding up of DMCKs via specification-level exploration.

2.1 DMCK Overview

We define DMCK [58, 67, 74, 76, 77, 87, 92] as the software model checker which verifies distributed system correctness at the implementation level. It has an interposition layer for each process to deterministically control system execution, and enable system state observation. A DMCK engine schedules system events and checks system properties.

In order to systematically explore the state space, DMCK must implement an approach for state checkpointing and restoration. There exist two approaches: stateful and stateless [53]. The stateful approach stores the previously visited states to avoid redundant exploration. However, it is difficult or prohibitively expensive to compute a canonical representation of program states, making it hard to apply the stateful approach to real-world systems [50].

The stateless approach does not need to store previously visited states. It remembers the trace which leads to a state as a checkpoint, and replays the trace to restore the state. This approach requires little modifications to the target system, making it practical to apply to complex systems. Many recent DMCKs adopt the stateless approach [67, 74, 76, 92]. However, the stateless approach cannot distinguish redundant states, leading to a more severe explosion of state/path space. This problem is alleviated by dynamic partial order reduction (DPOR) [50], a sound state space reduction technique that eliminates equivalent executions. However, DPOR may still be insufficient for large-scale systems in certain scenarios, and more reduction techniques like dynamic interface reduction [58], symmetry rules [76], and semantic-aware rules [74] have been proposed.

2.2 The Need for Specification-Level Exploration

We see much of the related work focuses on state space reduction to scale up DMCK [58, 74, 76, 92]. However, the slow exploration of implementation is also a major obstacle to the efficient detection of bugs. Although previous work has proposed techniques to improve the exploration speed for each trace [76, 92], state-space exploration still remains the performance bottleneck of DMCK.

For example, previous work has identified that stateless exploration and model checker overhead slow down implementation level model checking. In a 54-event execution example provided by FlyMC [13], the stateless initialization takes 18 seconds, including tasks like cleaning the disk, restarting all nodes and preparing initial disk data. FlyMC reduced this to 2 seconds via snapshotting and restoring the initial state of the memory and disk. The event execution takes 18 seconds, primarily due to the model checker overhead, which requires a non-negligible wait time (e.g., 300ms) before enforcing the next event. FlyMC optimized this to 4 seconds through local ordering enforcement and state caching. Further optimization of the event execution time (4 seconds) is challenging because the time is mainly consumed by implementation execution.

We summarize the orders of magnitude in state space reached by state-of-the-art DMCKs in a single machine day, as reported in their evaluations. MoDist can explore 10^4—10^5 executions, with an average duration of roughly 2 seconds per execution. SAMC explores 10^4 executions, where each execution runs for 40 seconds and involves 20–120 events. FlyMC explores 10^5 executions and, owing to the execution speedup for a single trace, reduces the duration for a 54-event execution from 36 to 6 seconds [13].

Our insights. To further improve the scalability of DMCKs, we need to optimize from a different perspective that can avoid the problems introduced by slow implementation-level execution combined with stateless exploration.

We observed that exploring the state space of the specification for one machine day can cover 10^9 distinct states. If we treat the exploration time for every single state as equal across all states, the duration for exploring 50 events of the ZAB specification would be a mere 40.6 milliseconds. Specification-level exploration is stateful, inherently avoiding redundant states. Moreover, it eliminates issues introduced by DMCK overhead and code execution. These findings motivate us to lift state space exploration to the specification level to further scale DMCKs.

3 SandTable

We propose the SandTable framework to speed up distributed system model checking by lifting state-space exploration from the implementation level to the specification level. SandTable checks the system behavior against correctness properties by exhaustively and efficiently exploring its state space at the specification level. SandTable focuses on finding deep bugs triggered by complex interleavings of node-level events including message handling and failures. To avoid false alarms, for each bug detected at the specification level, SandTable reproduces the bug at the implementation level by deterministically injecting the specification-level events that triggered the bug.

To check a distributed system using SandTable, developers should provide a formal specification of the system (§3.1). The specification is written as a state machine for model checking purposes. The specification does not describe the ideal implementation that satisfies all the correctness properties; instead, it describes the actual (potentially buggy) implementation but in a more concise way.

With the specification, SandTable faces three major challenges: (1) Since specification development is an error-prone process and there could be unexpected discrepancies between the specification and the implementation, how to guarantee the quality of the specification? (2) How to mitigate the state explosion problem during model checking? (3) How to avoid false alarms that cannot be reproduced at

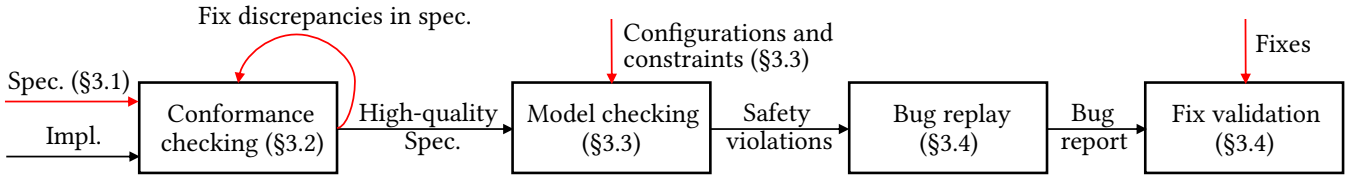

Figure 1. The workflow of SandTable. Red arrows require manual work.

the implementation level due to discrepancies between the specification and the implementation?

To address the first challenge, SandTable employs iterative conformance checking (§3.2) to improve the quality of the specification. It randomly explores the specification-level state space, replays the same trace at the implementation level by enforcing the same event interleaving, and detects discrepancies between the specification and the implementation. The developers need to fix the discrepancies and pass the conformance checking phase until there are no discrepancies found after a timeout (e.g., 30 minutes).

To address the second challenge, SandTable helps developers to decide the constraints (e.g., number of events) of the model checking to bound the state space (§3.3). SandTable randomly searches the state space for each user-provided constraints and then ranks constraints according to different metrics, including branch coverage and exploration depth.

SandTable's conformance checking cannot guarantee the absence of discrepancies between the specification and the implementation, which might lead to false alarms reported by specification-level model checking. For each bug found during model checking, SandTable confirms its existence at the implementation level by replaying the same trace (§3.4).

Once the user fixes the bug in the implementation and updates the specification, SandTable can validate the fix and detect any regression introduced by the fix by running the conformance checking and model checking again.

The workflow of SandTable, presented in Figure 1, is explained in the following sections, addressing the critical challenges mentioned above.

3.1 Formal Specification

We detail the process of writing formal specifications required by SandTable. A specification is written as a state machine, including the initial state, state transitions, and correctness properties. The initial state assigns values to all the variables that construct the system state. Each transition has a precondition (e.g., waiting for an incoming message) and updates variables as the system progresses. The correctness properties describe the intended behavior of the system, which are used as oracles for detecting bugs.

Figure 2 presents an example specification of ZooKeeper, which focuses on describing its core protocol, ZAB [66]. The specification is written in the TLA$^+$ language [73], which is widely used for modeling distributed systems. It defines

```
CONSTANTS Servers, MaxRequests, MaxCrashes, MaxEpoch
CONSTANTS LOOKING, NOTIFICATION_MSG, FOLLOWERINFO_MSG
VARIABLES role, messages, epochLeader, eventCounter

Init ==
  /\ role = [s \in Server |-> LOOKING]
  /\ epochLeader \in [1..MaxEpoch -> SUBSET Server]
  /\ messages = [s \in Servers |-> [d \in Servers \ {s} |-> <<>>]]
  /\ eventCounter = [n_crash |-> 0]

HandleMsg == \E m \in messages:
  CASE m.mType = NOTIFICATION_MSG -> HandleNotMsg(m.dst, m)
    [] m.mType = FOLLOWERINFO_MSG -> HandleFInfoMsg(m.dst, m)
    [] OTHER -> Assert(FALSE, "Error: unknown msg")

Next == \/ HandleMsg
        \/ HandleTimeout
        \/ ClientRequest
        \/ NodeCrash
        \/ NodeStart

StateConstraint == eventCounter.n_crash <= MaxCrashes /\ ...
LeadershipInv ==
  \A epoch \in 1..MaxEpoch: Cardinality(epochLeader[epoch]) <= 1
```

Figure 2. An overview of ZAB specification.

several constants to instantiate the model and bound the exploration depth, such as the number of ZooKeeper servers (`Servers`) and the maximum number of crash (`MaxCrashes`) events. The system state is defined as several variables, such as the role of each server (`role`) and the messages pending delivery (`messages`). There are auxiliary variables defined for other purposes (e.g., `eventCounter` in `StateConstraint` for bounding the state space). The initial state is defined by assigning initial values to every variable (`Init`). There are four types of transitions: message handling (`HandleMsg`), timeouts (`HandleTimeout`), client requests (`ClientRequest`) and failure events (`NodeCrash` and `NodeStart`). The correctness property (`LeadershipInv`) states that there cannot be more than one valid leader at any point.

Specifying system actions. The key is to capture how the relevant system state transitions while abstracting away other implementation details. SandTable mainly focuses on global explorations [74]—interleaving between messages and failures, thus the specification should model node-level events such as message handling, timeouts, client requests, network failures, and node crashes. When writing the specification, we do not include lower-level details such as thread interleaving and message serialization.

```java
public void run() {
  Message response;
  while (!stop) {
    response = manager.pollRecvQueue(3000, TimeUnit.MILLISECONDS);
    ... // observer handling code...
    QuorumPeer.ServerState ackstate = response.buffer.getInt();
    ... // type conversion from int to enum...
    Notification n = new Notification(response, ...); // decoding...
    ... // logging, backward combability...
    if(self.getPeerState() == QuorumPeer.ServerState.LOOKING){
      recvqueue.offer(n);
      if((ackstate == QuorumPeer.ServerState.LOOKING)
        && (n.electionEpoch < logicalclock)){
        ToSend notmsg = new ToSend(ToSend.mType.notification, ...);
        sendqueue.offer(notmsg);
      }
    } else {
      if(ackstate == QuorumPeer.ServerState.LOOKING){
        ToSend notmsg = new ToSend(ToSend.mType.notification, ...);
        sendqueue.offer(notmsg);
      }
    }
  }
} // zookeeper-server/src/.../server/quorum/FastLeaderElection.java
```

(a) The implementation for handling leader election messages

```
HandleNotMsg(dst, response) ==
  LET src == response.src
      notmsg == [mType |-> NOTIFICATION_MSG, ...]
  IN IF role[dst] = LOOKING
     THEN /\ recvqueue' = [recvqueue EXCEPT ![dst] =
                             Append(@, response)]
          /\ IF /\ response.ackstate = LOOKING
                /\ response.round < logicalclock[dst]
             THEN ReplyNotMsg(dst, src, notmsg)
             ELSE DiscardNotMsg(src, dst)
     ELSE /\ UNCHANGED recvqueue
          /\ IF response.ackstate = LOOKING
             THEN ReplyNotMsg(dst, src, notmsg)
             ELSE DiscardNotMsg(src, dst)

ReplyNotMsg(src, dst, m, branch) ==
  messages' = [messages EXCEPT
                ![src][dst] = IF IsConnected(src, dst)
                              THEN Append(@, m) ELSE @,
                ![dst][src] = Tail(@)]

DiscardNotMsg(src, dst) ==
  messages' = [messages EXCEPT ![src][dst] = Tail(@)]
```

(b) The corresponding specification

Figure 3. ZooKeeper's implementation for handling leader election message and the corresponding specification. The tainted variables show how we model important implementation variables in the specification.

Figure 3 presents an example of translating an event handler implementation code into one action in the specification. The `run` method implements how the ZooKeeper server handles leader election messages from other servers: The server waits for a response from the other server and sends back a notification if both servers are looking (`LOOKING`) for a new leader and the server's logical clock is higher, or if only the other server is `LOOKING`. The notification is pushed into a queue (`recvqueue`) if the server is `LOOKING`, and is further processed by another thread.

In the specification, we model three important variables: (1) the input response message (`response`), (2) the notification message (`notmsg`) to send to other servers, and (3) the message buffer (`recvqueue`) that stores the response messages for later processing. The `HandleNotMsg` action is triggered by an incoming response message, updates the message buffer and sends back the notification message according to the response content and the server's local state. For example, if the server is looking for a leader when receiving the response message, it assigns the next state of the message buffer (`recvqueue'`) with a new message buffer that appends the response message to the old buffer (`recvqueue`).

The specification abstracts away low-level implementation details, such as message decoding, type conversion, logging and support for backward compatibility. We also choose to not model the handling logic for an observer server in the specification as we want to focus on checking the interaction between ZooKeeper leaders and followers.

Specifying environment actions. To check how the target system tolerates unexpected failures, we model node and network failures in our specifications.

For node failures, we model node crash and rejoin behaviors. A node crash breaks all its network connections and clears all the volatile data. When the node restarts and rejoins the system, it returns to its initial state with the persistent data, and reconnects to other nodes.

For network failures, we model different behaviors for TCP and UDP semantics. The TCP connection ensures that there is no message loss, duplication or reordering. Thus, we introduce a network partition action to simulate scenarios where all connections crossing a partition are broken until the system recovers from the partition. In the case of UDP, we further model message loss, duplication, and out-of-order delivery.

Specifying correctness properties. We focus on safety properties as oracles for finding bugs. Safety properties state that something bad never happens (e.g., "there is never more than one valid leader") and are specified as invariants in the specification. We also approximate liveness property (something good eventually happens) checking based on the checking of safety properties, as in [67, 92].

We choose safety properties from (1) the original protocol design (e.g., Raft [82] and Zab [65, 66]), and (2) the system-specific guarantees and regressions (e.g., source code assertions, documentation and historical issues).

```
1 CheckLeader(self, votes, leader, round) ==
2   IF leader = self
3-  THEN (IF round = logicalClock[self] THEN TRUE ELSE FALSE)
3+  THEN TRUE
4   ELSE (IF votes[leader].vote.proposedLeader = NULL THEN FALSE
5         ELSE (IF votes[leader].state = LEADING THEN TRUE ELSE FALSE))
```

Figure 4. Fixing discrepancies found by conformance checking. The red line is the cause. The green line is the fix.

3.2 Conformance Checking

Conformance checking ensures specification quality by detecting discrepancies between the specification and the implementation. After SandTable detects a discrepancy, developers need to revise the specification and start a new round of conformance checking for the revised specification. As a by-product, general system correctness bugs, such as unhandled exceptions, or memory leaks, can be also detected during the conformance checking process.

SandTable detects discrepancies by randomly exploring the specification-level state space to generate traces, replaying them in the implementation, and comparing the specification trace with the corresponding implementation trace.

To replay a trace, SandTable converts the specification trace events to replay commands that are executed by the implementation-level deterministic execution engine, which will be detailed in §4.1. Message delivery events and failure events are automatically converted. The user needs to provide timeout values for timeout events and shell commands for client request events.

To compare the two traces, SandTable compares the variables in the specification and their counterparts in the implementation. The values of system variables are acquired by querying the systems' APIs or by parsing logs. The network and node environment (e.g., message counts and node status) is managed by SandTable, and can be compared directly. For example, in Figure 3, the specification variable `role` is compared with the system's peer state (`self.getPeerState()`, which is acquired by parsing logs).

Figure 4 shows a discrepancy and its fix. The code checks a condition for becoming a leader upon receiving a notification message. The discrepancy arose when applying the specification to an older version of ZooKeeper. It was found by comparing the `role` variable, which in the specification trace is `LOOKING`, but in the implementation trace is `LEADING` after executing a message delivery event. The root cause was debugged by analyzing the message delivery context and comparing the message handling code with the specification.

Once a discrepancy is resolved, we initiate a new round of conformance checking. After multiple rounds, we gain sufficient confidence in the quality of the specification. For example, we can use a 30-minute timeout (based on the experience that discrepancies are seldom found after 30 minutes) as the stopping condition when no further discrepancies are detected.

Algorithm 1 Ranking constraints for each configuration

Input: Configurations set $Configs$
Input: Constraints set $Constraints$
for x in $Configs$ **do**
  Initialize an empty map M
  for y in $Constraints$ **do**
    Random walk with $\langle x, y \rangle$ and collect $data$
    Add $\langle y, data \rangle$ to M
  end for
  Sort y in M based on its $data$
end for

3.3 Specification-Level Model Checking

SandTable uses BFS to explore the specification-level state space. The BFS algorithm is stateful and avoids exploring the same state repeatedly. To further reduce the state space, SandTable leverages symmetry [45] in the system: permuting the nodes and workload values does not change whether an action satisfies an invariant.

A challenge in running model checking for distributed systems is to decide how to bound the state space, otherwise model checking has to explore an infinitely large space. SandTable helps developers to bound the space. Developers need to provide a collection of system configurations $Configs$ and budget constraints $Constraints$. Each system configuration specifies the number of nodes and workload values to instantiate the model. Each budget constraint specifies the maximum number of timeouts, failures and workloads that can happen. SandTable ranks budget constraints for each configuration based on heuristics.

Algorithm 1 presents the algorithm for ranking the constraints. For each configuration, SandTable combines it with a constraint to instantiate and bound the model, and performs a random walk in the state space. The random walk starts from the initial state and randomly chooses the next step until there are no more steps to take or a predefined timeout happens. For each random walk, SandTable collects data including branch coverage, event counter and exploration depth. SandTable then sorts all the budget constraints according to the collected data. SandTable relies on a heuristic that if the state exploration covers more branches and involves more diverse events, then it is more likely to trigger a bug. By default, SandTable favors random walk with a smaller depth as it indicates a smaller state space that model checking can exhaustively explore. Thus, the built-in sorting function of SandTable first sorts by the branch coverage in decreasing order, then by the diversity of the events in decreasing order, and finally by the depth in increasing order. Developers can extend SandTable to install different sorting functions.

3.4 Avoid False Alarms

To prevent false alarms of bugs in the specification-level exploration, SandTable reproduces the bugs at the implementation level by replaying the event interleaving, similar to the conformance checking phase. If this process does not uncover any discrepancies, SandTable confirms the existence of the bug in the implementation. Otherwise, developers should fix the discrepancies and restart the workflow of conformance checking and model checking.

After fixing the bug, developers also need to update the specification accordingly. Developers can use SandTable to run conformance checking to ensure no new discrepancies are introduced during the fix, and run model checking again to ensure that the bug has been correctly fixed and that no new bugs have been introduced by the fix.

4 Implementation and Integration

We choose TLA$^+$ [73] as the formal specification language and implement specification-level exploration using TLC [32], a model checker for TLA$^+$. We employ TLC's simulation mode for random walk and the BFS mode for model checking. We implement an implementation-level deterministic execution engine from scratch, as existing DMCKs are either proprietary [92] or restricted to specific languages [67, 74, 76].

We will describe the SandTable implementation of deterministic execution (§4.1), followed by the integration of SandTable into eight well-established open-source distributed systems in industry and community (§4.2).

4.1 SandTable Implementation

SandTable is implemented in 5,700 lines of C/C++ code (for automated interposition, network proxy, failure simulation, and conformance checking) and 1,300 lines of bash and python code (for network configuration, cluster initialization and remote control). SandTable utilizes containers and virtual machines, supporting both Docker [24] and LXD [27], to run a cluster on a single machine. It can also operate on multiple machines. We release SandTable as an open-source project [37].

We have the goal to implement a portable and transparent deterministic execution mechanism capable of running a wide range of distributed systems without modifications to either the target system or the underlying operating system. This goal guides multiple design choices in our implementation.

The architecture of the implementation-level deterministic execution engine is presented in Figure 5. The engine has control and observation over the target system, handling things like node status (e.g., start, pause, and crash), network tasks (e.g., message delivery and network failures) and nondeterminisms (e.g., timeout) by executing node, state and network commands. It includes a state checker for conformance checking, and a transparent network proxy (TPROXY)

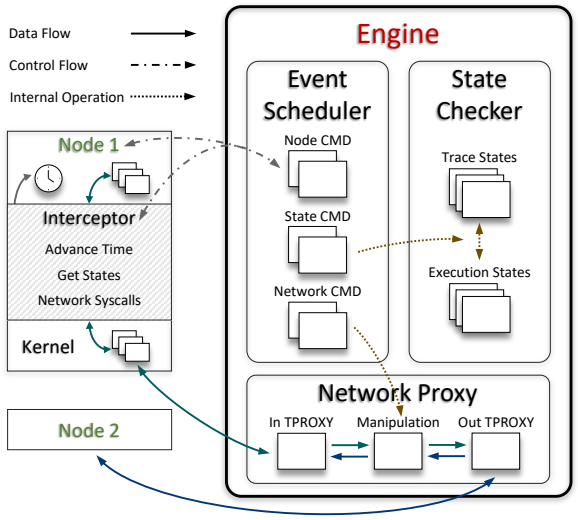

Figure 5. The architecture of implementation-level deterministic execution engine. The control flow and internal operation arrows illustrate how engine commands interact with each component. The data flow arrows depict how network traffic is proxied.

mechanism [10] for controlling cluster network traffic. The interceptor runs inside the target system's address space and controls nondeterminisms by preloading shared libraries [26] to execute code and override functions, particularly system call wrapper functions in the C standard library (libc). More details of the implementation are provided in Appendix §A.

To reproduce a specification trace in the implementation, the trace events and states must be converted into corresponding SandTable deterministic execution commands. The environment specification defines an auxiliary variable to record the event to be converted and assigns common event commands (e.g., message delivery and failure events). SandTable provides scripts to parse the trace. Users need to assign the auxiliary variable and extend the scripts for system-specific action events (e.g., timeout duration for timeout event and client request shell commands).

4.2 Integration

In our work, the target systems are PySyncObj [7], WRaft [3], RedisRaft [9], DaosRaft [5], RaftOS [8], Xraft [11], Xraft-KV [12] and ZooKeeper [1]. They implement distributed consensus protocols, including Raft [80, 82] and Zab [65, 66], to provide state machine replication. Specifically, PySyncObj is a full-featured Raft library designed for replicating Python objects in a cluster. It has gained broad adoption in the open-source community as well as in industry. WRaft ("W" is the first letter of the author's name) is a Raft library written in C. It has been subsequently adopted by Redis (RedisRaft) and the DAOS Storage Stack (DaosRaft). RedisRaft and DaosRaft

Systems	Impl.		Spec.				Est. Effort	
	#Stars	#LOC	#LOC	#Var.	#Act.	#Inv.	Spec.	Conf.
PySyncObj	658	4.6K	490	12	9	13	14	15
WRaft	1.0K	3.4K	879	14	15	13	14	3
RedisRaft	766	5.3K	600	14	9	15	7	5
DaosRaft	596	3.5K	584	13	9	14	3	3
RaftOS	339	1.3K	610	12	9	13	17	3
Xraft	219	6.7K	605	14	11	15	2	1
Xraft-KV	219	7.9K	618	18	10	18	2	1
ZooKeeper	11.6K	11.8K	2037	39	20	15	7	7

Table 1. Integrated distributed systems and formal specification effort.

extend WRaft with system-specific enhancements and optimizations. RaftOS is another Raft library to replicate Python Objects, featuring an asynchronous replication framework. ZooKeeper is an industrial-strength distributed coordination service in Java, which serves as the backbone for a number of distributed systems. Xraft is a Raft implementation in Java for educational uses. Xraft-KV is a distributed key-value store based on Xraft. Table 1 provides a summary of the target systems and the formal specification efforts. The two **#LOC** columns represent modeled lines of code and their corresponding specification lines. The columns **#Var.**, **#Act.**, and **#Inv.** represent variables, actions and safety properties in the specification respectively. The **Est. Effort** column represents the estimated specification and conformance checking effort in person days, calculated from the git history. It is notable that the conformance checking process is quick and can find numerous discrepancies (e.g., 34 for WRaft and 20 for RaftOS).

The SandTable framework does not require manual instrumentation of the target system and has minimal restrictions on the underlying language. SandTable adheres to the assumptions adopted by the underlying target systems, in order to prevent false positives. These assumptions influence the system's failure model. For example, in the case of a TCP connection, failures like message duplication, loss, or out-of-order delivery will not occur. Developers often rely on these guarantees to optimize the implementations. We have formally specified reusable network modules for both TCP and UDP semantics (476 lines of code in total). We apply failure models of UDP semantics to WRaft and RaftOS, which make no assumptions about the network or explicitly use UDP. For other systems, we apply failure models of TCP semantics.

We formally specified the latest versions (i.e., the main branch or the latest release) of each system. For Raft-based systems, we developed specifications from scratch, as existing TLA$^+$ specifications for Raft are abstract and do not match the target system implementations. We modeled the basic Raft protocol modules, including leader election and log replication, for every system. We modeled the PreVote extension for RedisRaft, DaosRaft and Xraft, and modeled the log compaction module for WRaft. We modeled the Put and Get operations of the key-value store in Xraft-KV, which is based on Xraft but does not include PreVote. Most safety properties in Raft systems are common, such as having only one valid Leader, log consistency in the cluster, log durability, commitment requirements, and the monotonicity of specific variables. Some safety properties are specific to individual systems, such as the requirement that retrying requests should not contain an empty log for replication in WRaft, and linearizability for Xraft-KV.

ZooKeeper offers official formal specifications contributed by the community [36, 83], which come in two versions: a *protocol specification* modeling the Zab protocol, as described in [66], and a *system specification* that adheres to the system code of version v3.7.0. We leveraged the system specification with modifications, replacing message channels with our network modules and eliminating local thread explorations by immediately scheduling enabled thread actions. For example, the specification models both the receiver thread (as illustrated in Figure 3) and the worker thread. The receiver places messages in a queue, enabling the worker to retrieve and process them from the queue. Since the processing result will not change as observed by other nodes, we removed any interleavings between these two actions.

To replay a specification-level exploration trace at the implementation level, two key requirements must be met: the system should be runnable, and the engine should properly interpret trace events and states. To meet the first requirement, we developed a portable driver for WRaft, RedisRaft, and DaosRaft to handle network messages and client requests, and to tick the system. We developed a simple Python script for PySyncObj and RaftOS running on Python 3.7 to import, initialize, and execute their core protocols. Xraft key-value store and ZooKeeper are already runnable, and we simply use their release binaries to run on OpenJDK 19. To meet the second requirement, the client request commands, timeout durations and variables to compare should be specifically converted to different commands for SandTable. Failures and network delivery can be readily converted into commands since they are common to all systems.

5 Evaluation

In this section, we evaluate the performance of SandTable with respect to the following questions: (1) How effective is SandTable in finding bugs? (§5.1) (2) How efficient is SandTable? (§5.2) (3) How much speedup does specification-level exploration provide when compared with implementation-level exploration? (§5.3)

5.1 Effectiveness of Finding Bugs

As shown in Table 2, SandTable finds a total of 23 bugs. The **Stage** column categorizes these bugs into three types based on the stage at which they are discovered: model checking (16), conformance checking (6), and modeling (1). There are

ID	Stage	Status	Bug Consequence	Time	#Depth	#States
PySyncObj#1 [18]	Conformance	New	Unhandled exception during disconnection		-	
PySyncObj#2 [19]	Verification	New	Commit index is not monotonic	6s	13	93713
PySyncObj#3 [20]	Verification	New	Next index ≤ match index	7s	18	189725
PySyncObj#4 [20]	Verification	New	Match index is not monotonic	35s	25	1512679
PySyncObj#5 [21]	Verification	New	Leader commits log entries of older terms	2min	14	2364779
WRaft#1 [17]	Verification	New	Incorrectly appending log entries	9min	22	5954049
WRaft#2 [17]	Verification	Old	Inconsistent committed log	22min	20	20955790
WRaft#3 [17]	Conformance	New	Follower lagging behind until next snapshot		-	
WRaft#4 [17]	Verification	Old	Current term is not monotonic	39min	23	48338241
WRaft#5 [17]	Verification	New	Retry messages include empty logs	11min	24	10576917
WRaft#6 [17]	Conformance	Old	Memory leak		-	
WRaft#7 [17]	Verification	New	Next index ≤ match index	8min	23	7401586
WRaft#8 [17]	Conformance	New	Prematurely stopping sending heartbeats		-	
WRaft#9 [17]	Modeling	Old	Cannot elect leaders due to incorrectly getting term		-	
DaosRaft#1 [22]	Verification	New	Leader votes for others	5s	8	476
RaftOS#1 [28]	Verification	New	Match index is not monotonic	5s	10	60101
RaftOS#2 [29]	Verification	New	Incorrectly erasing log entries	4s	9	19455
RaftOS#3 [30]	Conformance	New	Unhandled exception during receiving messages		-	
RaftOS#4 [31]	Verification	New	Prematurely stopping checking commitment	4min	14	16938773
Xraft#1 [34]	Verification	New	More than one valid leader in the same term	3s	8	3534
Xraft#2 [35]	Conformance	New	Unhandled concurrent modification exception		-	
Xraft-KV#1 [33]	Verification	New	Read operations do not satisfy linearizability	15s	10	124409
ZooKeeper#1 [2]	Verification	Old	Votes are not total ordered	4min	41	7625160

Table 2. Effectiveness and efficiency in detecting bugs. The "Stage" column categorizes bugs by the stages in which they were found: model checking (i.e., "Verification"), modeling, and conformance checking. The "Status" represents whether the bugs were first found by us ("New") or not ("Old"). "Time", "#Depth" and "#States" indicate elapsed time, the number of events and the number of explored distinct states to hit the bug. Only the bugs found in the model checking stage have the "Time", "#Depth", and "#States" metrics.

18 new bugs and 5 old bugs. So far, 17 (out of 18) new bugs have been confirmed and 13 have been fixed. SandTable independently detects the 4 old bugs.

SandTable can consistently reproduce bugs detected at the specification level. The bug-triggering trace is deterministically replayed at the implementation level. The bug traces in both levels are invaluable for finding the root causes and fixing the bugs.

Most bugs SandTable finds are unlikely to be detected by manual testing or random testing, because these bugs are deep, and the bug-triggering event interleavings are intricate. Take PySyncObj#4 as an example, which has a triggering prefix of 8 intricate events interleaved with a timeout event inserted between two consecutive failed response messages due to a network partition.

5.1.1 How SandTable Detects Bugs. SandTable aims to find safety violation bugs during model checking stage. Only traces that violate the safety properties are executed in the implementation to confirm the bug. As these safety properties are necessary to ensure system correctness, bugs confirmed in the implementation are true bugs. Moreover, since these bugs are identified through BFS, their triggering traces have minimal depth. There are also by-product bugs found during the modeling and conformance checking stages (e.g., unchecked simple conditions and unhandled exceptions).

5.1.2 Bugs Detected by SandTable. To demonstrate the effectiveness of SandTable, we discuss bugs classified by systems. Due to space limitations, we only provide two detailed bug explanations. In bug detection, we use a 2- or 3-node configuration, with two workload values and constraints of 3–6 timeouts, 3–4 client requests, 1–4 failures, and 4–10 message buffer sizes. Constraints are determined using Algorithm 1, taking into account branch coverage, event diversity, and maximum depth. For example, the top three constraints are selected based on branch coverage, and further selections can be made based on a smaller estimated state space (maximum depth) to make BFS explore deeper within a limited time frame.

PySyncObj. We found 5 bugs in PySyncObj: 4 are safety violations, and 1 crashes the system. Bug #1 causes unexpected node crashes due to the incorrect handling of disconnections. The remaining 4 bugs involve violations of safety properties required by the Raft protocol. Bugs #2, #3, and #4 violate

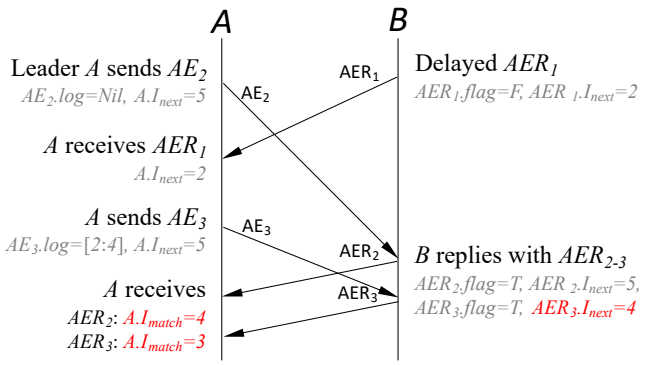

Figure 6. PySyncObj#4: Timing diagram of non-monotonic match index. The variables and messages are in gray. The bug points causing the consequence are in red.

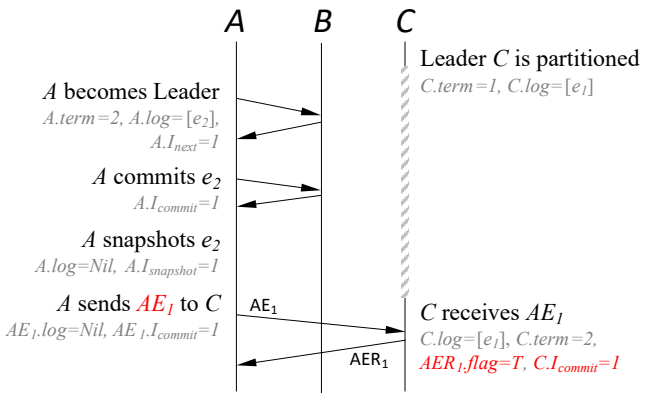

Figure 7. Timing diagram of data inconsistencies in WRaft. The network partition is shaded in gray diagonal lines. The variables and messages are in gray. The bug points causing the consequence are in red.

the requirements related to protocol variables, including monotonicity and size relationships. Bug #5 is related to the current Leader incorrectly advancing the commit index to a log entry created by a previous Leader, which is prohibited by the Raft protocol.

We provide a detailed description of PySyncObj#4, which results in the match index variable not being monotonic. As the match index directly influences the commitment, there exist potential risks of data inconsistency and data loss. Figure 6 presents a simplified space-time diagram illustrating the root cause of the bug. This diagram condenses several rounds of elections, synchronizations and a network partition failure at the beginning. In brief, Leader A aggressively advances its next index to send only the latest logs to B. If log synchronization fails, A resets the next index to an older value provided by B. However, when B receives already synchronized logs, it provides a wrong next index value, leading A to set a non-monotonic match index based on the next index. Details of the explanations are provided below. The network partition leads to a log inconsistency between Leader A and Follower B. Consequently, B rejects A's synchronization in AppendEntriesResponse message AER_1 ($AER_1.flag = F$ indicating rejection and a requirement to reset the next index to $AER_1.I_{next} = 2$). This AER_1 message is delayed in reaching A. Subsequently, A synchronizes its logs, and by the time A sends AE_2, their logs have already been synchronized (not depicted in the diagram). The AE_2 message contains no log entries because they are already in sync. When the delayed AER_1 reaches A, A resets the next index for resynchronization with B through AE_3, which includes log entries from index 2. B receives both AE_2 and AE_3 and responds with successful handling ($flag = T$) since they are in sync. However, a bug causes B to incorrectly set $AER_3.I_{next} = 4$ when the AppendEntries message contains logs, which should be the same as $AER_2.I_{next} = 5$. Upon receiving AER_2, A sets the match index to $AER_2.I_{next} - 1 = 4$. However, when A receives AER_3, it incorrectly sets the match index to 3 without verifying the monotonicity. It is worth noting that changing I_{next} after sending AE and $flag$ in AER messages are unverified optimizations for Raft.

WRaft, DaosRaft and RedisRaft. DaosRaft and RedisRaft are downstream systems of WRaft, and we will discuss them together. In WRaft, we uncovered a total of 9 bugs, including 4 independently found old bugs (#2, #4, #6, and #9) that had been resolved in DaosRaft and/or RedisRaft but remained unresolved in WRaft. The remaining 5 new bugs comprise 3 safety violations (#1, #5, and #7) and 2 liveness issues (#3 and #8), with 4 of them confirmed by RedisRaft [16] and DaosRaft [15] developers. Most bugs are introduced during incremental development, such as when introducing new modules (e.g., log compaction) and extensions (e.g., PreVote).

WRaft#1 ignores a specific condition, leading to the conflicting log entries being appended and committed, resulting in data inconsistency. WRaft#5 incorrectly handles a special case, resulting in unnecessary synchronizations. WRaft#7 causes the next index to be no larger than the match index. WRaft#3 can cause Follower nodes to lag behind due to the incorrect rejection of the Leader's snapshot when log inconsistency exists. WRaft#8 can prematurely disrupt the broadcast of heartbeats in the event of a sending failure.

In DaosRaft, we found one additional bug introduced by an extension, namely PreVote, which aims at enhancing cluster stability when occasional failures occur. The bug causes the Leader to vote for other nodes. In RedisRaft, no additional bugs were found.

We present a detailed description of the data inconsistencies that result from the combination of WRaft#1 and WRaft#2. A simplified timing diagram in Figure 7 illustrates the root cause of this bug. This diagram removes some starting message interactions. In short, Leader A should synchronize snapshots when log entries are compacted. However, due to a bug, A incorrectly sends empty log entries instead

of the snapshot to C. Upon receiving the message, C fails to remove unmatched log entries and incorrectly advances the commit index, leading to data inconsistencies. Details of the explanations are provided below. Node C is elected as the Leader and processes a client request, appending log entry e_1. However, a network partition prevents the synchronization of e_1 with Nodes A and B. Subsequently, Node A becomes Leader after receiving B's vote and processes a client request, appending log entry e_2. This entry is synchronized with B and committed. Node A then compacts its committed logs as snapshots. Afterwards, the network partition is resolved, and A synchronizes with C. However, due to WRaft#2, a Snapshot message should be sent in this case, but an incorrect AppendEntries message AE_1 is sent instead. Since log entry e_2 has been compacted, AE_1 contains no log entries, but the commit index (denoted as I_{commit} in the figure) is set to 1. When C receives AE_1, it incorrectly accepts the synchronization because it ignores checks for a special case when the log entry to append is the first entry due to WRaft#1. Consequently, log entry e_1 is not deleted, but the committed index is set to 1, resulting in inconsistent committed log entries across the cluster.

RaftOS. We found 4 bugs in RaftOS: 3 bugs are safety violations, and one bug is an implementation bug. RaftOS#1 violates the monotonicity property of the match index due to an assignment without appropriate checks. RaftOS#2 leads to data loss as a result of an incorrect condition check that erases already matched log entries. RaftOS#3 is an implementation bug that crashes the node during message handling. RaftOS#4 incorrectly breaks the commitment checking loop in the presence of log inconsistency, which intends to prevent a similar issue as in PySyncObj#5, but causes the cluster to fail to make progress.

Xraft and Xraft-KV. We found two bugs in Xraft, and one bug in Xraft-KV. Xraft#1 leads to multiple valid Leaders in the cluster due to unconditional acceptance of stale vote messages. Xraft#2 results in Leader crashing due to an unhandled concurrent modification exception, arising from a subtle thread race condition. Xraft-KV#1 reveals that the KV store implementation does not provide a guarantee of linearizability. All bugs were promptly confirmed and fixed by the developer.

ZooKeeper. We conducted verification on the latest version (v3.9.0) of ZooKeeper, but we did not find new bugs. To show the effectiveness of our method, we reproduced one known bug in version v3.4.3. The bug violates the total order property of the votes, resulting in either multiple valid Leaders or the inability to elect a Leader. Notably, while the bug is located in the leader election module, the optimal triggering trace involves the leader election, discovery and synchronization modules of Zab, showing the complexity of the bug.

System	Experiment #1			Experiment #2	
	Time	#Depth	#States	#Depth	#States
PySyncObj	57min	41	63185747	24	1880642320
WRaft	2.1h	48	94475424	19	1064901869
RedisRaft	2.9h	45	161245842	19	1379707906
DaosRaft	59min	53	80684948	22	1720868573
RaftOS	23min	34	31569538	14	3347361061
Xraft	42min	47	67862168	21	1646089192
Xraft-KV	30min	39	34192341	20	1601906684
ZooKeeper	1.7h	106	167834292	50	2125891595

Table 3. Efficiency of state exploration. Both experiments use a 3-node configuration. Experiment #1 employs more restrictive constraints that make the state space exhaustible, while experiment #2 doubles the constraints and utilizes a 1-day time budget to stop.

5.2 Efficiency of SandTable

To evaluate the efficiency in finding bugs, we measured both the wall-clock time and the number of distinct states to hit bugs. To evaluate the efficiency of specification-level state exploration, we conducted two experiments on bug-fixed specifications. First, we measured the time required to explore a small state space exhaustively. Second, we measured the number of distinct states in a large state space within a one-day time frame. All experiments were carried out at the specification level in BFS mode on an Ubuntu 22.04 server equipped with a 3.5GHz CPU with 10 cores (20 hyperthreads) and 64 GB memory.

Efficiency in finding bugs. We employed configurations and constraints identical to §5.1. These settings are reasonable since they are targeted for finding new bugs. In Table 2, the **Time** and **#States** columns are metrics. Only safety violation bugs have the required metrics, while others are denoted with "-". The **Time** column shows the wall-clock time to hit each bug. The time to detect a bug ranges from 3 seconds to 39 minutes. It shows the efficiency in finding bugs, as just one machine hour of model checking can reveal numerous bugs. The **#States** column shows the number of distinct states explored to trigger each bug. The value reaches up to 10^7, showing the fast exploration speed and the complexity of these bugs.

Efficiency of state exploration. We evaluated the efficiency of state exploration through two experiments, both of which utilized a 3-node configuration and were conducted on the specification with bug fixes.

In experiment #1, we slightly reduced the timeout events and network buffers to 3–4 based on constraints of §5.1, which resulted in a smaller state space that could be exhaustively explored in hours. Notably, these constraints can still find 87% safety violations when evaluating the specifications without bug fixes. The results in Table 3 show that the explo-

ration time to reach full coverage ranged from 23 minutes to 2.9 hours.

In experiment #2, we doubled each constraint value, resulting in a significantly larger state space, and we adopted a one-day time budget to bound the exploration. The results in Table 3 show that the exploration covered up to 10^9 distinct states.

In both experiments, the number of distinct states steadily increased over time, with average speeds ranging from 739, 515 to 2, 324, 556 distinct states per minute.

In conclusion, the two experiments illustrate that the speed of specification-level state exploration is fast, enabling the checking of a large scale of distinct states while covering deep depth within hours and days of model checking. We believe that the scale provides confidence in the correctness of the system since the most complex bug we found requires only 10^7 states to trigger.

5.3 Specification-Level Speedup

In our specification-level speedup measurements, we focused on two key metrics: (1) the time it took to explore one trace at the specification level; (2) the time it took to explore one trace at the implementation level.

Our experiment setup was as follows. We explored the specification state space in random walk mode and replayed the corresponding traces at the implementation level. We adopted a 3-node configuration with constraints identical to those used for bug detection. Each specification exploration involved running 10, 000 traces with only one worker. Subsequently, we randomly selected 1, 000 traces for deterministic replay at the implementation level. We calculated the average time for both metrics. The experiments were conducted on the same machine as discussed in §5.2, with the implementation running in LXD containers consisting of 1 engine node and 3 worker nodes.

We chose to run specification-level exploration in random walk mode because there is no straightforward way to control the implementation run in a stateful manner, such as BFS, which would require checkpoints and system restoration. Additionally, counting traces generated from the state space graph is also not straightforward. To reduce the bias in the comparison, we adopted the random walk mode.

The evaluation results are presented in Table 4. The trace depth ranges from 7 to 78, with an average of 41. The speedup (Column **Speedup**, computed as **Impl./Spec.**) ranges from 114× to 2989×. Notably, specification-level exploration proved to be fast, with elapsed times varying between 5.83 and 20.7 milliseconds to explore one single trace. The variation in speed is primarily due to the different complexities of each specification.

The speed of implementation-level exploration exhibits large variance due to the sleep for cluster initialization and synchronization between two actions. For PySyncObj, RedisRaft, DaosRaft, and WRaft, our test driver has no sleep, resulting in an average trace exploration speed of approximately 2 seconds. RaftOS, although also using our test driver, relies on sleep for certain asynchronous actions before executing the next action, resulting in an average trace exploration time of 4.8 seconds. In contrast, Xraft and ZooKeeper rely on sleep for both initialization and synchronization, introducing notable delays. Xraft required an average of 24 seconds, while ZooKeeper required 28 seconds to explore a single trace. It is worth noting that the sleep mechanism was not intentionally introduced, and we set the durations reasonably short. Similar mechanisms were applied in related work, such as SAMC [74], where exploring a ZooKeeper trace within a depth range of 20–120 took 40 seconds.

System	Trace Depth	Average Depth	Spec. (ms)	Impl. (ms)	Speedup
PySyncObj	9–54	40	14.18	1798.53	127
WRaft	13–60	47	20.70	2496.53	121
RedisRaft	10–78	45	15.87	1802.40	114
DaosRaft	11–64	48	11.96	2115.82	177
RaftOS	10–44	31	5.83	4813.74	825
Xraft	21–49	38	8.14	24338.57	2989
Xraft-KV	7–51	35	8.64	24032.17	2781
ZooKeeper	16–59	46	17.14	28441.65	1660

Table 4. Comparison of speed between specification-level and implementation-level exploration. The "Speedup" is computed as the ratio of "Spec."/"Impl."

In summary, the specification-level exploration in SandTable can achieve speedups of up to 2989 times, compared to the implementation-level exploration.

6 Discussion

6.1 Generalizability

SandTable can be applied to distributed systems with well-defined protocols and correctness properties, and is not specific to any formal languages or model checkers.

Our evaluation primarily focuses on evaluating consensus systems because they are representative, critical and hard to get right. However, the targeted distributed systems are not limited to consensus, as the event-driven structure is common in distributed systems, and SandTable can replay these events deterministically. The Xraft-KV system illustrates how SandTable can be applied to other systems by specifying the KV actions and properties.

We choose TLA$^+$ and TLC for their widespread use in modeling and checking distributed systems and their mature toolchains. It is also possible to use alternative state-machine based specification languages such as PlusCal [72] and the P language [48, 49]. The choice of the specification language and its model checker affects the modeling/conformance

difficulty and the state exploration speed. For example, using PlusCal may simplify writing imperative specifications but complicates conformance checking as the TLC model checker operates at the TLA$^+$ level.

6.2 Soundness and Completeness

SandTable can miss bugs due to discrepancies between the specification and the implementation that conformance checking fails to detect. SandTable also cannot detect bugs in the components that are not modeled by the specification. Besides, SandTable's model checking is bounded and does not detect bugs outside the explored state space. We do not observe any false alarm reported by SandTable in our evaluation because SandTable always replays the buggy trace at the implementation level to confirm the existence of the bug.

6.3 Lessons Learned

Trusting the specification for the implementation can be risky, as bugs and gaps in the specification can compromise its ability to prevent critical bugs. We use conformance checking and run systems on real-world environments to avoid false positives and boost confidence in system robustness.

Human-written unit test expectations is brittle. For example, we encountered a failed unit test after fixing a critical bug in WRaft because it asserts the buggy behavior as the expected behavior [4]. It indicates that even when developers meticulously analyze the corner conditions, errors can still occur because the oracle can be mistaken. We employ safety properties derived from the system design, documentation and source codes to ensure that bad events never occur, while excluding the unit test oracles.

The manual effort (as shown in Table 1) of specification and conformance checking varies significantly. When we are unfamiliar with the protocol (e.g., Raft), writing a specification (e.g., PySyncObj and WRaft) can be time-consuming. However, as our familiarity with the Raft systems grew, the time needed for these tasks significantly decreased (e.g., integrating Xraft took only 3 person days from scratch). When the SandTable framework is in development, the time required for conformance is also significant due to additional enhancement (e.g., PySyncObj). Integrating an existing ZAB specification into SandTable required a considerable amount of time due to the need to address different action granularity and gaps with real-world failure models. RaftOS was checked by a novice user who was unfamiliar with TLA$^+$ and Raft (with assistance from an experienced user). The time spent suggests that for non-expert users, the manual effort required to use SandTable is manageable.

7 Related Work

We compare SandTable to previous work that hardens distributed systems using implementation-level model checking, testing and formal verification.

Implementation-level model checking. The concept of model checking, which involves exhaustive exploration of the state space of abstract specifications, was originally proposed by Clarke and Emerson [44]. Leading corporations like Amazon [79] and Microsoft [86] have been using model checking to verify their distributed system protocols before implementation. In order to leverage model checking to verify the correctness of real system implementations, instead of the abstract model, two broad approaches are proposed.

In the first approach, a model is extracted from the source code through automated static analysis or manual efforts [39, 43, 46, 62, 63]. The abstract model is then checked by the classical model checking approach. If counter-examples are identified, they are subsequently mapped back to the source code. SandTable relies on manual efforts to write the model (i.e., the specification), and uses conformance checking to improve its quality. There are also domain-specific languages designed for model checking distributed systems and generating code automatically from specifications (e.g., Mace [68], P [48, 49] and PGo [59]). However, they cannot directly find bugs in existing distributed systems written in other languages.

The second approach involves the direct exploration of the state space in the implementation code by controlling execution nondeterminisms. Initially starting with Verisoft [52] and CMC [77], there have been many model checkers that directly check implementation code [41, 57, 58, 67, 74, 76, 78, 87, 91, 92]. Our focus here is primarily on model checkers for distributed systems. CMC [77] is a stateful model checker that directly checks C code, and it has successfully detected numerous bugs in network protocol implementations. However, it comes with the requirement for invasive modifications to execute the target system inside CMC's address space. MaceMC [67] emphasizes checking liveness and employs bounded depth-first search alongside random walks to identify safety and liveness bugs. Yet, it necessitates that the programs to be checked are written in the Mace language. CrystalBall [91] and LMC [57] are based on MaceMC, consequently limited to the Mace language. In contrast to these checkers, SandTable does not rely on the programming language used in the implementation code and can run the implementation in its native execution environment.

MoDist [92] is the first transparent DMCK, controlling nondeterminisms at the system call level of the WinAPI. Our interposition of the POSIX API takes inspiration from this work. MoDist is stateless, employs DPOR [50] to reduce the state space, and can discover both safety and liveness bugs. DeMeter [58] primarily focuses on the decoupling of local and global explorations to reduce the state space. SAMC [74] and FlyMC [76] reduce the state space in the exploration of complex distributed systems using semantic-aware, symmetry-based, and event independence-based algorithms. These model checkers primarily emphasize state space reduction. SandTable is complementary to these state

space reduction techniques, and we believe that these techniques can also be integrated into SandTable [56].

Testing. Previous work has found various types of distributed systems bugs using systematic testing [6, 38, 40, 41, 47, 51, 55, 69, 75, 88, 89]. The most closely related work is Mocket [89]. Mocket requires users to provide a high-quality specification that describes the desired system behavior, uses TLC to generate test cases and treats any discrepancies between the specification and the implementation as bugs. However, as reported by Mocket and other work [14], existing specifications also contain bugs that introduce false alarms. Mocket also suffers from the scalability issues of implementation-level state exploration, as it executes each test case generated by TLC at the implementation level. In contrast, SandTable employs conformance checking to guide the users to write specifications that precisely describe the actual system behavior, and addresses the scalability issues by exploring the state space at the specification level. Besides, model-based test-case generation is used for MongoDB's operational transformation functions [47], and property-based testing is applied in Amazon S3 ShardStore node [41]. As a model checking tool, SandTable exhaustively explores a bounded space to detect deep bugs in distributed system implementations.

Formal verification. Formal verification is an effective approach to ensure the absence of bugs by proving the implementation satisfies the specification. Recent work has made great progress on building formally verified distributed systems [54, 60, 61, 64, 84, 85, 90]. However, formal verification requires heavy proof efforts and cannot directly find bugs in existing distributed systems.

8 Conclusion

In this work, we present SandTable, a technique and methodology that significantly scales distributed system model checking by exploring the state space at the specification level, and confirming bugs at the implementation level. We find that the SandTable methodology is effective, efficient and practical for finding true bugs and provides coverage guarantees for real distributed systems. SandTable's ability to transparently control unmodified distributed systems to run on unmodified POSIX systems plays a critical role in debugging and fixing deep bugs. We believe the integration efforts required by SandTable are cost-efficient in practice, especially for critical distributed systems where developers aim to achieve formal verification. In our future work, we plan to implement continuous integration of conformance checking to keep the specification and implementation in sync, and to further reduce the state space by applying existing state space reduction techniques. Our ultimate goal is to establish SandTable as a common practice in developing correct mission-critical distributed systems. We have made SandTable publicly available at https://github.com/tangruize/SandTable.

Acknowledgments

We thank the anonymous reviewers and our shepherd, James Bornholt, for their insightful comments. We thank Tianyin Xu for the valuable feedback that helped improve our work. Ruize Tang, Yu Huang, Yuyang Wei, Lingzhi Ouyang, and Xiaoxing Ma were supported by the National Natural Science Foundation of China (62025202, 62372222), the CCF-Huawei Populus Grove Fund (CCF-HuaweiFM202304), the Cooperation Fund of Huawei-Nanjing University Next Generation Programming Innovation Lab (YBN2019105178SW38), and the Fundamental Research Funds for the Central Universities (020214912222). Xudong Sun was supported by NSF CNS 2145295 and CNS 2130560.

References

[1] 2008. *Apache ZooKeeper.* https://zookeeper.apache.org/
[2] 2012. *ZooKeeper#1: Leader election never settles for a 5-node cluster.* https://issues.apache.org/jira/browse/ZOOKEEPER-1419
[3] 2013. *WRaft.* https://github.com/willemt/raft
[4] 2016. *An incorrect unit test in WRaft.* https://github.com/willemt/raft/blob/e428eeb921a014192d1d703dd317f3f29f5916c5/tests/test_snapshotting.c#L498
[5] 2016. *DaosRaft.* https://github.com/daos-stack/raft
[6] 2016. *Jepsen.* https://jepsen.io/
[7] 2016. *PySyncObj.* https://github.com/bakwc/PySyncObj
[8] 2016. *RaftOS.* https://github.com/zhebrak/raftos
[9] 2018. *RedisRaft.* https://github.com/RedisLabs/raft
[10] 2018. *Transparent proxy support - The Linux Kernel documentation.* https://docs.kernel.org/networking/tproxy.html
[11] 2018. *Xraft.* https://github.com/xnnyygn/xraft/tree/master/xraft-core
[12] 2018. *Xraft-KVStore.* https://github.com/xnnyygn/xraft/tree/master/xraft-kvstore
[13] 2019. *FlyMC Technical Report.* https://tinyurl.com/flymc-technical-report
[14] 2020. *A bug in Raft TLA+: bugfix: HandleAppendEntriesRequest.* https://github.com/ongardie/raft.tla/pull/6
[15] 2021. Bugs confirmed by DaosRaft. Personal communication.
[16] 2021. *Bugs confirmed by RedisRaft.* https://github.com/RedisLabs/raft/issues/49
[17] 2021. *WRaft bugs.* https://github.com/willemt/raft/pull/118
[18] 2022. *PySyncObj#1: Fix disconnection when connecting.* https://github.com/bakwc/PySyncObj/pull/161
[19] 2022. *PySyncObj#2: Raft commit index is not monotonic.* https://github.com/bakwc/PySyncObj/issues/166
[20] 2022. *PySyncObj#3 and #4: Raft match index is not monotonic.* https://github.com/bakwc/PySyncObj/issues/167
[21] 2022. *PySyncObj#5: Leader commits log entries of older terms.* https://github.com/bakwc/PySyncObj/issues/169
[22] 2023. *DaosRaft#1: Reject request vote if self is leader.* https://github.com/daos-stack/raft/pull/69
[23] 2023. *dlfcn(3) manual pages.* https://man.openbsd.org/dlopen.3#dlsym
[24] 2023. *Docker.* https://www.docker.com/
[25] 2023. *ld.bfd(1) manual pages.* https://man.openbsd.org/ld.bfd.1#version-script=
[26] 2023. *ld.so(1) manual pages.* https://man.openbsd.org/ld.so.1#LD_PRELOAD
[27] 2023. *LXD.* https://ubuntu.com/lxd
[28] 2023. *RaftOS#1: Raft match index is not monotonic.* https://github.com/zhebrak/raftos/issues/25
[29] 2023. *RaftOS#2: Log may be erased incorrectly.* https://github.com/zhebrak/raftos/issues/26

[30] 2023. *RaftOS#3: KeyError in handling append_entries_response message.* https://github.com/zhebrak/raftos/issues/27

[31] 2023. *RaftOS#4: Change in update commit index.* https://github.com/zhebrak/raftos/pull/30

[32] 2023. *TLC and TLA+ Toolbox.* https://github.com/tlaplus/tlaplus

[33] 2023. *Xraft-KV#1: Read operations do not satisfy linearizability.* https://github.com/xnnyygn/xraft/issues/40

[34] 2023. *Xraft#1: Two leaders with the same term.* https://github.com/xnnyygn/xraft/issues/33

[35] 2023. *Xraft#2: Xraft-kvstore does not satisfy linearizability.* https://github.com/xnnyygn/xraft/issues/39

[36] 2023. *ZooKeeper TLA+ specification.* https://github.com/apache/zookeeper/tree/master/zookeeper-specifications/system-spec

[37] 2024. *SandTable.* https://github.com/tangruize/SandTable

[38] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15).*

[39] Thomas Ball and Sriram K. Rajamani. 2002. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02).*

[40] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos Engineering. *IEEE Software* 33, 3 (2016).

[41] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21).*

[42] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC'07).*

[43] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003).

[44] Edmund M. Clarke and E. Allen Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop.*

[45] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. 1998. Symmetry reductions in model checking. In *Computer Aided Verification (CAV'98).*

[46] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. 2000. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00).*

[47] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. 2020. Extreme modelling in practice. *Proc. VLDB Endow.* 13, 9 (2020).

[48] Ankush Desai. 2013. *The P programming language.* https://github.com/p-org/P

[49] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13).*

[50] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05).*

[51] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. 2023. Coverage Guided Fault Injection for Cloud Systems. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23).*

[52] Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97).*

[53] Patrice Godefroid and Koushik Sen. 2018. *Combining Model Checking and Testing.* Springer International Publishing.

[54] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed causal memory: modular specification and verification in higher-order distributed separation logic. *Proc. ACM Program. Lang.* 5, POPL (2021).

[55] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 2023. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23).*

[56] Xiaosong Gu, Wei Cao, Yicong Zhu, Xuan Song, Yu Huang, and Xiaoxing Ma. 2022. Compositional Model Checking of Consensus Protocols via Interaction-Preserving Abstraction. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS'22).*

[57] Rachid Guerraoui and Maysam Yabandeh. 2011. Model Checking a Networked System Without the Network. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11).*

[58] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11).*

[59] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. 2023. Compiling Distributed System Models with PGo. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS'23).*

[60] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. 2023. Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23).*

[61] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15).*

[62] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02).*

[63] Gerard J. Holzmann. 2001. From Code to Models. In *Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01).*

[64] Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2022. Adore: atomic distributed objects with certified reconfiguration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'22).*

[65] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (ATC'10).*

[66] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN'11).*

[67] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07).*

[68] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. Mace: language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI'07).
[69] Beom Heyn Kim, Taesoo Kim, and David Lie. 2022. Modulo: Finding Convergence Failure Bugs in Distributed Systems with Divergence Resync Models. In *2022 USENIX Annual Technical Conference (ATC'22).*
[70] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998).
[71] Leslie Lamport. 2019. *TLA+ specification for Paxos.* https://github.com/tlaplus/Examples/blob/master/specifications/PaxosHowToWinATuringAward/Paxos.tla
[72] Leslie Lamport. 2021. *PlusCal Introduction.* https://lamport.azurewebsites.net/tla/tutorial/intro.html
[73] Leslie Lamport. 2022. *TLA+ Home Page.* https://lamport.azurewebsites.net/tla/tla.html
[74] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14).*
[75] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19).*
[76] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys'19).*
[77] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. 2002. CMC: A Pragmatic Approach to Model Checking Real Code. In *5th Symposium on Operating Systems Design and Implementation (OSDI'02).*
[78] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08).*
[79] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (2015).
[80] Diego Ongaro. 2014. *Consensus: bridging theory and practice.* Ph. D. Dissertation. Stanford University, USA.
[81] Diego Ongaro. 2014. *TLA+ specification for Raft.* https://github.com/ongardie/raft.tla
[82] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (ATC'14).*
[83] Lingzhi Ouyang, Yu Huang, Binyu Huang, and Xiaoxing Ma. 2023. Leveraging TLA+ Specifications to Improve the Reliability of the ZooKeeper Coordination Service. In *Dependable Software Engineering. Theories, Tools, and Applications: 9th International Symposium, SETTA 2023, Nanjing, China, November 27–29, 2023, Proceedings (SETTA'23).*
[84] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.* 2, POPL (2017).
[85] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: a Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23).*
[86] Dharma Shukla. 2018. *Azure Cosmos DB: Pushing the frontier of globally distributed databases.* https://azure.microsoft.com/en-us/blog/azure-cosmos-db-pushing-the-frontier-of-globally-distributed-databases/
[87] Jiri Simsa, Randy Bryant, and Garth Gibson. 2010. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV'10).*
[88] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing For Cluster Management Controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22).*
[89] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. 2023. Model Checking Guided Testing for Distributed Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys'23).*
[90] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15).*
[91] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. 2009. CrystalBall: predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09).*
[92] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09).*

A Implementation Details

A.1 Interposition

Interposition is the enabling technique to achieve deterministic control and observation of the target system running on each worker node. We implement the interposition layer, which we refer to as the "interceptor", to run in the address space of the target system. The interceptor intercepts POSIX API functions and interacts with the engine node. We will describe how the interceptor operates and what it intercepts to achieve deterministic control and observation.

The `LD_PRELOAD` trick. In POSIX systems that use ELF, the dynamic linker provides a feature to execute code and to override functions in other shared libraries. This feature can be easily employed by setting the `LD_PRELOAD` environment variable to a user-defined shared library [26]. The preloaded library can contain functions to execute before `main()` and to override same-name functions.

We leverage this technique to override POSIX APIs, particularly system call wrapper functions defined in the C standard library (libc), as they are major sources of nondeterminisms and provide access to the target system's resources. Additionally, we use this technique to execute commands from the engine.

To override POSIX APIs, we simply define functions with the same name as the functions we wish to override. For instance, to override the `write()` syscall, we just need to define the function in our interceptor. After executing our custom code logic, we can invoke the original `write()` syscall by utilizing `dlsym()` [23] to find the original function address. To disable the overriding of specific functions, we use a version script [25] to change the visibility of these functions to "local". In total, we have intercepted 20 POSIX APIs.

For executing engine commands, we start a thread that establishes a connection with the engine, and receives commands from it. These commands serve various purposes, such as advancing the virtual clock, retrieving system states and exiting the target system.

Virtual clock. The virtual clock controls the code's perception of time. It allows us to advance time arbitrarily, triggering timeout events without waiting for the real wall clock. We achieve this by intercepting time-related system calls, including functions like `clock_gettime()` and `gettimeofday()`.

While it's true that many system calls have timeout parameters (e.g., `select()`), we find that our intercepted system calls are sufficient for a wide range of systems. This observation is based on how programs typically handle timeouts. They begin by retrieving the current time, establish a deadline by adding a timeout value to it, and then periodically compare the current time to this deadline. When the current time exceeds the deadline, the corresponding event is triggered. As a result, syscalls for retrieving the current time are the primary determinant for triggering timeout events.

Virtual clock increases under two conditions: when executing a time advancement command from the engine, and when the target system requests the current time, causing a small predefined increment (e.g., 1 nanosecond) to maintain time monotonicity.

Network interception. We intercept network syscalls in the interceptor for tracking messages and the network status of the target system. To minimize modifications to syscall requests, we do not change the destination of these syscalls to the engine node. Instead, the network traffic control is implemented in the engine and described in §A.2.

When sending a message, if the destination matches a configured address of concern, the interceptor adds a header including message boundary information to instruct the engine to enqueue the message in the network buffer. Before receiving a message, the interceptor checks if the receiving queue is empty. An empty receiving queue can lead to blocking or receiving no messages.

States observation. If there is no public API for observing the states of the target system, the interceptor provides a mechanism to intercept logging file descriptors. It parses the debug-level and trace-level logs using user-defined regular expressions to extract corresponding states. While this method may not capture all variables for local states, it is often sufficient for critical variables of interest used in conformance checking. Industrial systems typically feature detailed logging, especially for core protocols.

A.2 Network Proxy

To proxy network messages to the engine, we implement a transparent network proxy mechanism in the engine based on the Linux transparent proxy (TPROXY) [10] feature (only affecting the engine's compatibility with Linux; the interceptor is not restricted). As the name suggests, the target systems are unaware that their messages are being proxied. The engine buffers these messages, and exercises manipulation over them.

This technique allows the capture and delivery of network messages. To enable TPROXY for a specific subnet, we configure the route forwarding rules on the router (i.e., the host running cluster containers) to route the subnet traffic to the engine node. We configure the firewall on the engine node to redirect the subnet's traffic to the TPROXY socket (i.e., a socket with the `IP_TRANSPARENT` flag) port listened by the engine. To masquerade the true sender when delivering messages to the receiver, the engine binds the TPROXY socket to the true sender's address before connecting or sending to the receiver. This setup ensures that senders believe they are connected to their intended receivers, and receivers perceive messages as coming from the original senders.

We implement a message manipulation mechanism that provides control over messages. For TCP connections, messages are stored in a queue, with only the head message available for sending. The failure model permits only network partition, as described in §A.3. For UDP messages, the messages are stored in a list, allowing for selective dropping, duplication, or out-of-order delivery as needed.

A.3 Failure Simulation

The engine injects two types of failures that can happen in real distributed systems: node crashes and network failures.

For node crashes, we simulate them by sending a `SIGQUIT` signal, which aborts the target system without cleanups before termination. We implement it by sending keyboard bindings of the signal, such as "Ctrl+\" in the virtual console allocated by SSH. To rejoin a crashed node, the engine restarts the target system by sending the shell commands.

For network failures, we simulate network partition failures for TCP connections, which disconnect the connections, clear network buffers between the sender and the receiver, and prevent further connections until network recovery. For UDP packets, we simulate message drops, duplications, and out-of-order deliveries. These failure injections are implemented in the engine by manipulating the network buffers and connections.

A.4 State Checking

The state checker is used for conformance checking, which compares specification trace states with implementation execution states.

State comparison occurs after each action, and if discrepancies emerge, the engine prompts the discrepancy's location and aborts. This checking process is automated. It provides valuable information for developers to manually address specification discrepancies.

There are some challenges to obtaining execution states. While network states can be retrieved from the network proxy component, obtaining the target system's state is more complex due to its reliance on the target system's semantics.

We implement two methods for retrieving target system states without making modifications to the source code. The first method is to access states through the APIs provided by the target system, which are commonly used for debugging purposes. In cases where the first method is not applicable, we utilize instrumentation of logging file descriptors, as described in §A.1, to parse debugging logs and extract critical variable states of interest. However, if both of these methods are not applicable, developers have the option to modify the source code to provide an interface for obtaining states, although, in practice, we did not utilize it.

A.5 Event Execution

The engine utilizes three types of commands to execute events. Network commands manipulate network-related actions, such as message delivery and network failure injection. Node commands are designed for controlling target system status (e.g., start, pause, and crash), managing interposition of nondeterminisms, and retrieving states of the target system. Lastly, state commands are executed for conformance checking.

There is an issue of when to execute the next event. Executing an event that is not enabled in the target system can lead to false positives during conformance checking. Although some commands in the interposition are used for tracking enabling conditions, in situations where enabling conditions are challenging to fully track, we implement sleeps for a brief delay before scheduling the next event.