Blackbox Fuzzing of Distributed Systems with Multi-Dimensional Inputs and Symmetry-Based Feedback Pruning

Yonghao Zou
Beihang University
Peking University
zouyonghao@live.cn

Jia-Ju Bai*
Beihang University
baijiaju@buaa.edu.cn

Zu-Ming Jiang
ETH Zurich
zuming.jiang@inf.ethz.ch

Ming Zhao
Arizona State University
mingzhao@asu.edu

Diyu Zhou
Peking University
diyu.zhou@pku.edu.cn

***Abstract*—This paper presents DistFuzz, which, to our knowledge, is the first feedback-guided blackbox fuzzing framework for distributed systems. The novelty of DistFuzz comes from two conceptual contributions on key aspects of distributed system fuzzing: the input space and feedback metrics. Specifically, unlike prior work that focuses on systematically mutating faults, exploiting the request-driven and timing-dependence nature of distributed systems, DistFuzz proposes a *multi-dimensional input space* by incorporating regular events and relative timing among events as the other two dimensions. Furthermore, observing that important state changes in distributed systems can be indicated by network messages among nodes, DistFuzz utilizes *the sequences of network messages with symmetry-based pruning* as program feedback, which departs from the conventional wisdom that effective feedback requires code instrumentation/analysis and/or user inputs. DistFuzz finds 52 real bugs in ten popular distributed systems in C/C++, Go, and Java. Among these bugs, 28 have been confirmed by the developers, 20 were unknown before, and 4 have been assigned with CVEs.**

I. Introduction

Various kinds of critical distributed systems, such as distributed databases (*e.g.*, ClickHouse [7] and RethinkDB [47]) and distributed coordination systems (*e.g.*, ZooKeeper [20] and etcd [13]), provide fundamental support for analytical and computational tasks, and form the backbone of modern data center infrastructures [10], [16], [61]. However, they are prone to have subtle bugs, causing large economic losses [3], [4].

Prior work has applied fuzzing, a promising testing technique [2], [14], [19], [22], [70], to distributed systems. A pioneer work is Jepsen [21], a blackbox fuzzer that is well-known for its effectiveness in finding consistency violations. Based on user-provided schedule generators, Jepsen randomly generates: 1) workloads to drive the system, and 2) faults injected into the systems. CrashFuzz [15] and Mallory [37] are greybox fuzzers that advance Jepsen by using different feedback to guide the mutation of injected faults. CrashFuzz uses edge coverage, a popular metric for single-node systems [2], [14]. The feedback in Mallory requires the user to annotate code blocks that are considered to be important. Afterwards, during fuzzing, Mallory collects two types of events: 1) the invocation of user-annotated code blocks, and 2) network messages among nodes. The sequence of these events is used as feedback, and Mallory considers an event sequence is uninteresting, if it is too similar to a previous one.

However, the above prior work still suffers from important limitations on two key aspects of fuzzing effectiveness: 1) fuzzing input, and 2) feedback metric. Regarding fuzzing input, given the huge search space, the random generation approach used by Jepsen is highly ineffective. Furthermore, only mutating and injecting faults based on feedback, as CrashFuzz and Mallory do, can miss many bugs, as we elaborate subsequently and showcase in Figure 6 and Figure 7.

Regarding fuzzing feedback, the edge coverage used by CrashFuzz is ineffective for distributed systems. This is because, unlike single-node systems, distributed systems often execute almost the same code for requests, making edge coverage saturates after exploring only a few states [37], and thus, ineffective for distributed systems. Mallory's feedback requires laborious and error-prone user annotations and more importantly, as we elaborate subsequently, misses interesting states as well as explores redundant states.

This paper presents DistFuzz, which, to the best of our knowledge, is the first *feedback-guided blackbox* fuzzing framework for distributed systems. Table I compares DistFuzz with other fuzzers. The novelty of DistFuzz comes from the two *conceptual* contributions on testing input space and feedback metric, which depart from the conventional wisdom in prior distributed system fuzzers.

Conceptual contribution #1: extending input space with regular events and timing intervals. For the input space of distributed systems, our new insight is that, faults are just *one dimension* of it. In essence, faults are rare *internal* events to trigger state changes in distributed systems. We identify another two important input dimensions: 1) client requests and control commands, which, in contrast to faults, are *external* events to trigger state changes; and 2) relative timing among different events, since, for a distributed system, same events with different timing are likely to result in different states (ex-

* Jia-Ju Bai is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241912
www.ndss-symposium.org

TABLE I: Conceptual comparison of testing frameworks of distributed systems.

Feature	**DistFuzz**	**Jepsen**	**Mallory**	**CrashFuzz**
Fuzzer Type	Blackbox	Blackbox	Greybox	Greybox
Systematic Mutated Events	Regular, fault events and time intervals	Fault events	Fault events	Fault events
Feedback	Message sequences	None	User annotations + message sequences	Code coverage
Pruning Method	Symmetry based	None	Similarity based	None
Bug Reproducibility	Support	Not support	Not support	Not support
Overhead	Low	Low	Intermediate	High

plained in §II). To thoroughly test distributed systems, fuzzing requires systematically *co-mutating all* three dimensions.

With the above observation, DistFuzz uses a sequence of events as input, where each event can be either a fault or a regular event. In addition, DistFuzz associates each event with a timing interval, denoting how long DistFuzz should wait to apply this event after the previous one. As a result, the generated input is inherently fine-grained, potentially covering all possible combinations of the three dimensions. While the high-level idea of fuzzing regular events and/or timing interval has been proposed by fuzzers targeting other domains (*e.g.*, OS [50], [56], [58], library [2], [6], [19], and networking [33], [45], [51], [55], [70]), together with the novel feedback approach presented below, DistFuzz validates its effectiveness for fuzzing distributed systems.

Conceptual contribution #2: proposing network message sequences with symmetry-based pruning as fuzzing feedback. DistFuzz uses a novel and effective feedback approach based on the following observations. First, user-annotated code blocks, as required by Mallory, do not improve the quality of the feedback over network messages, since almost all important state changes in a distributed system are reflected by network messages. As an intuitive explanation, state changes in consensus algorithms either 1) are triggered by receiving certain messages or 2) result in sending out messages. As a result, they are all captured by changes in network message sequences. In fact, we surveyed all the code blocks annotated by Mallory's authors and found that as high as 91% is covered by network messages.

Second, pruning feedback based on similarity, as seen in Mallory, is ad-hoc and ill-suited for distributed systems. Such an approach can mistakenly prune interesting states, since, as we explain in §III-C, two drastically different states in a distributed system often output similar message sequences, which will be pruned by Mallory. Such an approach will also explore redundant states, due to the widely-existed *symmetry* in distributed systems. As an example, one state where A is a leader and B is a follower is symmetric to (and essentially the same as) another state where A is a follower and B is a leader. Without considering symmetry, one would view such states as different states, resulting in redundant exploration and downgrading the performance of the fuzzer.

Leveraging the first observation, DistFuzz *transparently* captures network messages as feedback. With the second observation, DistFuzz departs from prior approaches by *systematically* pruning feedback leveraging two common types of symmetry in distributed systems: *order symmetry* and *role symmetry*. Order symmetry represents the cases where the order of messages for different nodes is not important (*e.g.*, when a node starts an election process, the order of votes it receives does not matter). Similarly, role symmetry describes that, two states only differ in the roles of the nodes (*e.g.*, A is a leader and B is follower vs. A is a follower and B is a leader) should be considered as redundant. In addition, to prevent the feedback from being too sensitive to trivial states (*e.g.*, timestamp) in network messages, observing that messages of different types are likely to have different lengths, DistFuzz abstract the content of the messages with the message length.

We built a prototype of DistFuzz with around 7300 lines of C++ code. We used checkpointing and restoring to maximize fuzzing throughput while leveraging offline deterministic record and replay [42] to reliably reproduce founded bugs to eliminate false positives. We evaluated DistFuzz with 10 mature and widely-used distributed systems. Our evaluation demonstrates that the above fuzzing designs collectively make DistFuzz effective, finding 52 real bugs, with 28 confirmed by the developers, and 20 previously unknown. Among the confirmed bugs, 4 have been assigned with CVEs. By incorporating regular events and timing intervals into the input space, DistFuzz can find 11% more bugs. The feedback mechanism in DistFuzz does not require human annotations, reduces the total bug-finding time by 249.67 hours in average and finds 2 more bugs (§V-D).

In summary, we make the following contributions:

- ***New approaches:*** We make two conceptual contributions on fuzzing input space and feedback methods.
- ***Practical realization:*** We develop DistFuzz, a novel and effective feedback-driven black-box fuzzing framework based on the above ideas.
- ***Promising results:*** DistFuzz in total finds 52 real bugs, 28 of which have been confirmed by the developers, 20 were unknown before, and 4 have been assigned CVEs.

We have released DistFuzz and evaluation results at https://github.com/zouyonghao/DistFuzz.

II. Background and Motivation

A distributed system is a large and complex system whose components run as different processes located on different computers (or nodes) [62]. These processes execute almost the same code, communicate and coordinate each other with network messages, and may have different runtime states. Despite the great diversity of distributed systems (*e.g.*, different functionalities, runtime libraries, or programming languages), techniques used in DistFuzz are based on a few common natures of distributed systems. Subsection II-A presents these common natures. Subsection II-B motivates why we design DistFuzz as a blackbox fuzzer and Subsection II-C presents prior work on fuzzing distributed systems.

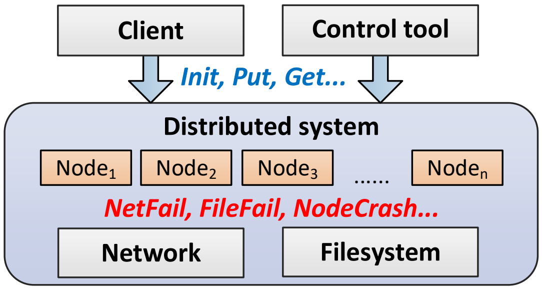


Fig. 1: Multi-dimensional input space of a distributed system.

A. Common Natures in Distributed Systems

Input events. As demonstrated by prior work, testing inputs to distributed system are various kinds of *input events*, as shown in Figure 1. Input events consist of two types. The first type is *regular events*. Examples of them include client requests (*e.g.*, read and write requests in a distributed file system), and management commands from the user (*e.g.*, adding or removing nodes in a distributed system). In addition, since a common design goal of distributed systems is to tolerate various kinds of faults [31] (through techniques such as replication and consensus algorithms [27], [43]), another type of input to the distributed system is various *fault events*, such as node crash and network partitioning.

Network messages capture important state changes. In distributed systems, the history of network messages can effectively capture the overall system state changes. This is because each node's execution state is often affected and/or reflected by the network messages. As a motivating example, consider the protocol of consensus algorithms (*e.g.*, Paxos [27] and Raft [43]), that are widely used in modern distributed systems. All state changes specified in the protocol are either triggered by 1) receiving certain network messages (*e.g.*, in Raft, a node in the candidate state turns into the follower state when the node receives a message from the current leader with a higher term); or 2) timeout, which in turn makes nodes send out network messages (*e.g.*, if a leader node elapses the heartbeat timeout, it sends network messages to maintain its authority and prevent others starting new elections). In both cases, the important state changes can be captured by the network messages among nodes.

Timing-dependence. Distributed systems are highly timing-dependent where two sequences of input events with the same events, but different relative timing are likely to result in different execution states. A key reason behind this is that distributed systems heavily use the *timeout* mechanism, where the system waits for certain events within the time window. The occurrence or non-occurrence of these events within the time window drives the system into different states.

Taking Raft as an example, where the two key timeouts are the heartbeat timeout and the election timeout. If a follower node does not receive a heartbeat message from the leader within the heartbeat timeout, it becomes a candidate and initiates a new election. Otherwise, it remains a follower. Therefore, the timing of the heartbeat message arrival relative to the timeout affects the system state. Similarly, during the election timeout, if a candidate node receives a majority of votes, it becomes the new leader; otherwise, it remains a candidate and starts a new election round. Thus, the timing of vote messages relative to the election timeout also impacts the system state.

B. Blackbox Fuzzing for Distributed Systems

Blackbox fuzzing is widely used and has shown great success in detecting bugs in non-distributed systems [11], [30], [38], [59], [68]. A key advantage of blackbox fuzzing is the minimal manual effort for deployment since it requires no user annotations nor code analysis and instrumentation.

Furthermore, compared to single-node systems, a blackbox fuzzer is more critical for distributed systems because of the following three reasons. First, distributed systems often involve complex logic, which may even overwhelm their developers. Thus, manual code annotations are likely to be error-prone and may lead to false positives. Second, distributed systems are highly non-deterministic [17], [44], and techniques like code instrumentation, which can cause high overhead (*e.g.*, 307% [15]) for distributed systems, inevitably perturb their execution, potentially masking bugs that would occur in production environments where such instrumentation is absent [23]. Third, there are several popular programming languages used for developing distributed systems, such as C/C++, Java, Go, and Rust. Testing frameworks that depend on source code analysis or code instrumentation incur significant manual effort in porting their functionality across these diverse systems, limiting their deployment in industrial settings.

C. Prior Work on Fuzzing Distributed System

This subsection discusses prior work on fuzzing distributed systems (*i.e.*, Jepsen [21], Mallory [37], and CrashFuzz [15]) focusing on two fundamental aspects: 1) generating input events; and 2) fuzzing feedback.

Generating input events. In summary, prior work generates input events by using novel approaches to effectively mutate *fault events* (§II-A). In terms of regular events (§II-A), prior fuzzers either make them fixed (as in CrashFuzz) or randomly generated (as in Jepsen and Mallory) them. These approaches follow the conventional wisdom that fault events, by their natures, are rare and thus are more likely to trigger bugs [65]. Below we detail how the prior fuzzers generate testing input.

Before the fuzzing campaign starts, CrashFuzz takes a sequence of concrete regular events as input, and they are used to drive the SUT during the whole fuzzing process. Jepsen and Mallory takes as input a set of regular event types. Afterward, during fuzzing, concurrent with the fault injection process discussed below, Jepsen and Mallory randomly choose certain events from this set, and concretize them to drive the SUT.

In each iteration of the fuzzing, the fuzzer decides 1) which fault event to inject and 2) where (*i.e.*, after which event) should the fault event be injected. CrashFuzz and Mallory make these decisions based on their fuzzing feedback, while Jepsen makes random decisions. The fuzzer stops injecting fault events until a pre-defined condition (*e.g.*, the number of fault events reaches a threshold) is met.

Fuzzing feedback. CrashFuzz [15] uses the popular edge coverage as its feedback metric. Despite its great success

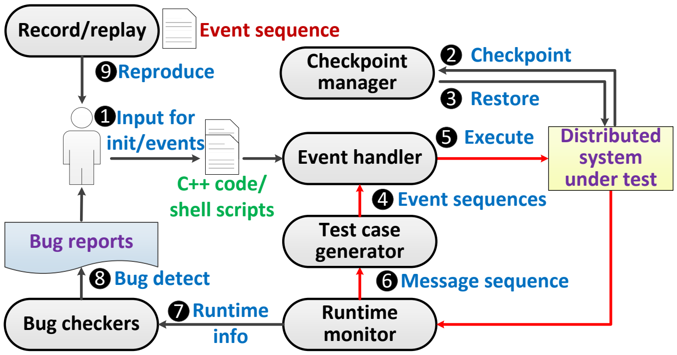


Fig. 2: The workflow and architecture of DistFuzz. User inputs are marked in green, DistFuzz's components are marked in black, and the fuzzing loop steps are marked in red.

in fuzzing single-node systems [2], [14], [19], Mallory [37] reported that code coverage is unsuitable for distributed system fuzzing. This is because distributed systems are request-driven and execute almost the exact code for every request. As a result, code coverage saturates very quickly, limiting the fuzzer to explore only a small number of different states.

Mallory proposes a feedback metric based on two types of events: (1) invocation of user-annotated interesting code blocks; and (2) network messages among nodes. The default mechanism [1] in Mallory is to collect these events with relevant information (*e.g.*, the ID of the node that invokes the code blocks; the sender, receiver, and the content of the network messages.) from each node, sort them by their occurrence time, and use the sorted events as fuzzing feedback. To avoid exploring redundant states, Mallory prunes an event sequence if it is too similar to a previous one (*e.g.*, an event sequence only differs from the previous one in the receiver of one message). Mallory measures similarities using a hash function that maps similar input values to similar hash values.

III. The DistFuzz Fuzzing Framework

This section presents DistFuzz, a novel and effective blackbox fuzzing framework for distributed systems. This section with an overview of DistFuzz (§III-A), followed by the contributions DistFuzz makes to advance state of the art. We present the remaining parts of DistFuzz in Section IV.

A. DistFuzz Overview

Figure 2 shows the workflow and architecture of DistFuzz. ❶ To use DistFuzz, a user provides certain application-specific information including (i) commands to initialize the system under test (SUT), which also specify the heartbeat timeout and the election timeout (§II-A) of the SUT; (ii) a status check command with the expected return value, which DistFuzz uses to detect if the SUT has been successfully initialized; (iii) the formats for regular events (§IV-B) for the SUT; and (iv) three tunable parameters, namely, the maximal length of the event sequence, the maximum value of the timing intervals and the fuzzing granularity of the timing intervals (§III-B).

To improve fuzzing throughput, DistFuzz uses checkpointing and restore to skip the initialization process of the SUT (§III-D). ❷ Therefore, before the fuzzing campaign starts, DistFuzz first brings up the SUT and waits for the initialization process (*i.e.*, gossip) to complete. Afterwards, the checkpoint manager takes a checkpoint of each node of the SUT. ❸ When the actual fuzzing campaign starts, at the beginning of each fuzzing iteration, the checkpoint manager restores the state of each node from its checkpoint, and pauses the SUT. ❹ Next, the test case generator produces a sequence of events, along with the timing interval between each event, and sends each event to the event handler in the corresponding node, based on where the event should occur. ❺ DistFuzz then starts the execution of the SUT and the event handler in each node applies events one by one, applying an event only when the time since the last one was applied exceeds the specified timing interval (§III-B). ❻ During execution, the runtime monitor collects the network messages among nodes. DistFuzz uses the collected network messages as fuzzing feedback to guide the mutation of events and their timing intervals (§III-C). ❼ The runtime monitor also collects other runtime information and sends it to the checkers to detect bugs (§IV). ❽ If a bug is detected, DistFuzz generates a bug report that users can utilize to confirm the bug. ❾ Finally, users can use a deterministic recorder and replayer provided by DistFuzz to effectively reproduce and validate the bug DistFuzz detects (§IV).

B. Mutating Regular Events and Timing Interval

Input events are an effective form of testing inputs for distributed systems (§II). Thus, distributed systems fuzzing should focus on generating rare and interesting sequences of input events. Unfortunately, we found that existing fuzzers are prone to miss many bugs, due to the two factors below.

Finding #1. **Effective distributed system fuzzing requires frequent and guided mutations on regular events.**

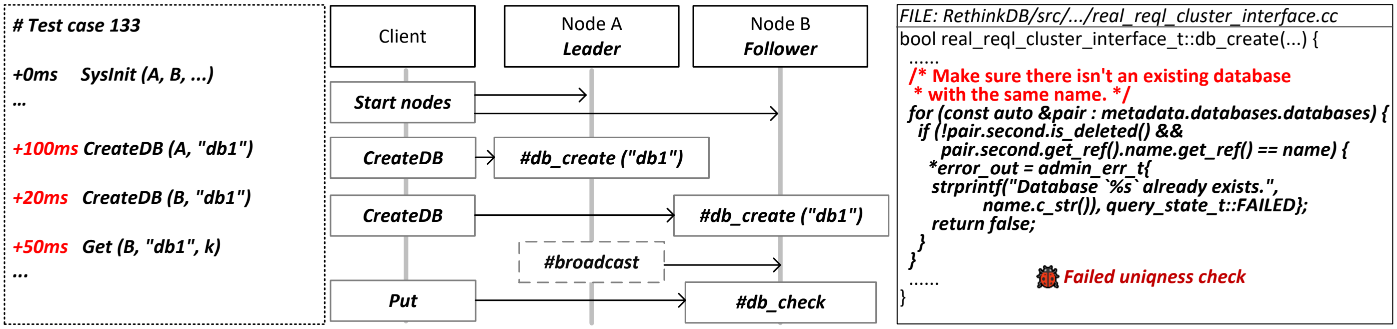
First, due to the focus on injecting fault events (§II-C), existing fuzzers do not systematically mutate regular events, thereby missing bugs caused by changes in regular events.

As a motivating example, Figure 3 shows a bug that is founded by DistFuzz but is missed by existing fuzzers. This bug corrupts the internal data structures of RethinkDB, rendering a newly created database unusable. Triggering this bug does not require even a single fault event, motivating the need to generate interesting regular events to expose bugs.

Furthermore, we note that the search spaces for regular events are huge, making it nearly impossible for user- or randomly-generated testing workloads to effectively expose certain bugs. As an example, RethinkDB supports tens of user commands and even if we limit the commands to a few common ones (*e.g.*, creating/dropping tables, and reading/writing entries), the problem still exists. This is because, each command, other than its type, also involves several other parameters (*e.g.*, at which node to execute the command, which table should be the target of the command). All these parameters must be correct to trigger a deep bug.

Finding #2. **Due to the timing-dependent nature of distributed systems, timing among events should be mutated.**

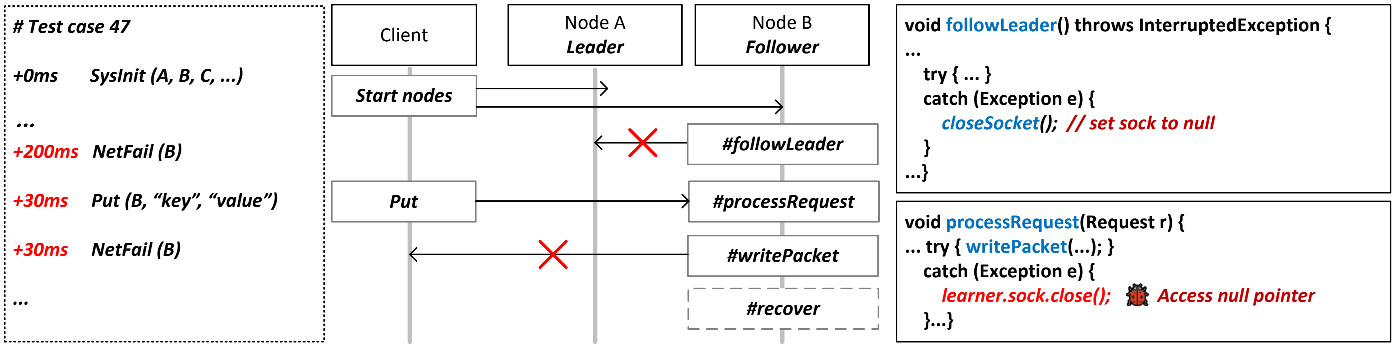
[1] Mallory also allows users to specify custom mechanisms for the feedback. We do not discuss this aspect in this paper because (1) Mallory does not present nor evaluate any custom policy, and (2) DistFuzz is a blackbox fuzzer.



```
# Test case 133

+0ms    SysInit (A, B, ...)
...

+100ms CreateDB (A, "db1")

+20ms   CreateDB (B, "db1")

+50ms   Get (B, "db1", k)
...
```

```
FILE: RethinkDB/src/.../real_reql_cluster_interface.cc
bool real_reql_cluster_interface_t::db_create(...) {
  ......
  /* Make sure there isn't an existing database
   * with the same name. */
  for (const auto &pair : metadata.databases.databases) {
    if (!pair.second.is_deleted() &&
        pair.second.get_ref().name.get_ref() == name) {
      *error_out = admin_err_t{
      strprintf("Database `%s` already exists.",
              name.c_str()), query_state_t::FAILED};
      return false;
    }
  }
  ......                    🐞 Failed uniqness check
}
```

Fig. 3: A bug in RethinkDB [47] that makes a newly created databases unusable. This bug is triggered by the following sequences of regular events. First, the client sends CreateDB("db1") requests to Node A and B at almost the same time. Node A creates the database with the name "db1" and sends Node B a message to announce the creation. However, before receiving the message, Node B has already created the database with the name "db1". As a result, any request involving access "db1" will fail, since the servers found that there are two databases with the same name "db1". So, the Put request to "db1" will fail and causes DistFuzz detecting the bug. The fix is to synchronize the database creation among nodes with consensus protocols.



```
# Test case 47

+0ms      SysInit (A, B, C, ...)

...
+200ms   NetFail (B)

+30ms    Put (B, “key”, “value”)

+30ms    NetFail (B)

...
```

```
void followLeader() throws InterruptedException {
...
   try { ... }
   catch (Exception e) {
      closeSocket(); // set sock to null
   }
...}
```

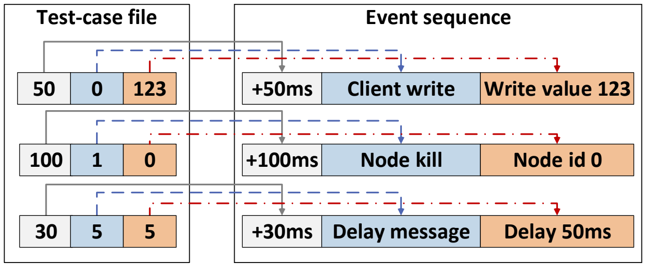
```
void processRequest(Request r) {
... try { writePacket(...); }
   catch (Exception e) {
      learner.sock.close();  🐞 Access null pointer
   }...}
```

Fig. 4: A bug that crashes a node in ZooKeeper [69]. This bug can only be triggered with specific relative timing among events. After a leader election, when Node B calls the *followLeader* method, DistFuzz injects a network access fault (*NetFail*) on Node B. This fault causes Node B to throw an exception, sets the *sock* variable to null, cleans up stale states, and then initiates the recovery process. However, before the recovery starts, Node B receives a *Put* request and processes it by invoking the *processRequest* method. During the execution of *processRequest*, DistFuzz injects another *NetFail* to cause *writePacket* to fail. This failure makes Node B access *sock*, a null pointer, and crashes.

Second, all prior fuzzers do not consider the timing-dependent nature of distributed systems (§II); they feed each input event to SUT either immediately after the previous one or with a fixed/random timing interval. Such an ad-hoc approach is likely to miss many subtle bugs that require distributed systems to execute timing-dependent logic.

To motivate the above statement, Figure 4 shows another bug that DistFuzz found but others miss. This bug causes ZooKeeper, a popular centralized co-ordination service for distributed systems, to crash, and thus brings down the whole distributed system. This bug can only be triggered with very specific timing. Specifically, the *Put* request must be sent to Node B before it starts the recovery process, and the two network failures must be precisely injected during the execution of *followLeader* and *processRequest*, respectively.

DistFuzz’s approach. Based on the above two findings, DistFuzz must *systematically* (1) mutate both the regular events and the fault events, and (2) mutate and enforce the relative timing between events.



Test-case file		
50	0	123
100	1	0
30	5	5

Event sequence		
+50ms	Client write	Write value 123
+100ms	Node kill	Node id 0
+30ms	Delay message	Delay 50ms

Fig. 5: Process of generating event sequence.

To co-mutate all three dimensions, DistFuzz equally encodes regular and fault events in input event sequences as a 3-tuple *<timing_interval, event_type, parameters>*. The *event_type* tuple denotes the type of the event. For example, an event can be a *Nodekill* event that shuts down a computing node in the system by force. The specific meaning of the *parameters* tuple depends on the type of the events (*e.g.*, for *SysInit*, it has one parameter that specifies how many nodes need to be initialized before testing). Finally, to enable timing

TABLE II: Events supported by DistFuzz.

Event	Description	Regular	Fault
SysInit	Initialize the whole system	✓	
Get	Send a "get" request to read a value given a specified key	✓	
Put	Send a "put" request to write to the specified key with the specified value	✓	
Cas	Send a "cas" request to compare and swap the value of the specified key	✓	
CreateDB	Create a database, namespace or file	✓	
DeleteDB	Delete a database, namespace or file	✓	
StatCheck	Check the status of the system	✓	
CustomREQ	Send a request specified by the user to the system under test	✓	
NetDup	Duplicate a message		✓
NetFail	Fail a send/receive of a message		✓
NetDelay	Delay a message		✓
NetPart	Start a network partition by dropping all messages to/from certain nodes		✓
NetHeal	Stop the network partition		✓
FileFail	Fail a read/write on a regular file		✓
FileDelay	Delay a read/write on a regular file		✓
NodeStart	Bring up a node	✓	
NodeStop	Stop a node gracefully	✓	
NodeKill	Kill a node by force		✓
NodeRestart	Kill a node by force and restart it		✓
NodeChange	Add/remove a node to/from the system	✓	
Total		11	9

mutation, the *timing_interval* represents the time difference (in milliseconds) between applying the previous event and the current event.

As discussed in Section III-A, the maximum value and the fuzzing granularity of *timing_interval* are user configurable parameters. These two values affect the effectiveness of fuzzing and should be set according to the characteristics of SUT. First, the maximum value cannot be too large. Otherwise, the SUT will always complete the leader re-election or failure recovery before DistFuzz applies the next event, thereby preventing DistFuzz from exposing bugs during the re-election or recovery process. Thus, we set maximum value to be 200ms, since we set the election timeouts of the SUT to 150ms and the network latency in our experimental environment is around 50ms. Second, the fuzzing granularity cannot be too short. A short granularity not only makes DistFuzz prone to explore redundant states but also makes DistFuzz inject many faults in a short time, thereby often overwhelming the fault tolerance mechanisms in SUT, causing the whole system to fail, and thus, prevents DistFuzz from exposing bugs. As a result, we set the default value to be 10ms. We further evaluate the impact of these values in §V-E.

Table II shows all the events DistFuzz support, which are generic enough for a wide range of distributed systems; all the systems we evaluate (§V-A) support these events.

To enforce relative timing, DistFuzz only applies an event when the time since it applies the previous one is larger than the *timing_interval*. DistFuzz applies these events transparently, by either intercepting system calls (*e.g.*, *send and receive*) or invoking the interfaces exposed by the system under test (*e.g.*, *Put* for writing values into the SUT). For example, DistFuzz injects a network partitioning fault (*NetPart*) by dropping all messages to/from certain nodes. We discuss how DistFuzz applies other events in §IV-B. A non-trivial portion of distributed systems bugs are caused by timeouts. DistFuzz triggers timeouts indirectly by injecting fault events (*e.g.*, by injecting *NetPart*).

TABLE III: Mutation options of an event sequence.

Mutation type	Description
Deletion	Delete this event.
Addition	Add a new event.
Replacement	Replace this event with a new event.
Parameter change	Change the parameters of this event.
Interval change	Change the timing interval between events.

For mutation, DistFuzz follows the popular fuzzers [2], [19], [22], [45], [70] to use a test-case file to generate the input event sequence. Figure 5 shows how a test-case file is translated to DistFuzz's input sequences. Mutation on the test-case file leads to the mutation of event sequences. Table III shows the corresponding mutation options.

The input space of DistFuzz is larger than prior fuzzers, due to the additional mutation on regular events and timing intervals. This is not an issue for DistFuzz, since as we discuss in the next subsection, the effective feedback it uses prunes the non-important mutations.

C. Network Messages with Symmetry-Based Pruning

Another conceptual contribution DistFuzz makes is a novel fuzzing feedback approach for distributed systems. Unlike prior approaches, DistFuzz's feedback is highly effective, while does not require source code access nor user inputs, thereby aligning with DistFuzz's blackbox approach (§II-B).

As discussed in Subsection II-C, Mallory is the state-of-art approach to generate fuzzing feedback. However, Mallory suffers from two limitations: (i) requiring users to annotate interesting code blocks, which is laborious and error-prone; and (ii) the similarity-based pruning is ad-hoc and ill-suited for distributed systems, resulting in both missing possible interesting states and exploring redundant states, as we elaborate next in our findings.

***Finding #1.* Information solely in network messages is sufficient as an effective fuzzing feedback.**

Our first finding is that user annotation on interesting code blocks, as required by Mallory, may not effectively improve fuzzing performance in distributed systems. Specifically, the observation that network messages capture important state changes (§II) can also be generalized to interesting events in the code. In other words, we found that network messages also cover most of the interesting code blocks, since these code blocks either 1) are invoked due to receiving certain network messages, or, 2) will, in turn, send out network messages.

As a motivating example, Mallory reports a bug (detailed in [36] and shown in Figure 1 in [37]), that is triggered, in part, by the *snapshot* operation. Mallory believes that the bug can only be efficiently detected if the user annotates the *snapshot* operation. However, we found that the *snapshot* operation will result in sending a network message, which informs other nodes that this node has taken a snapshot. Thus, DistFuzz can also find this bug by just using network messages as feedback. In fact, we survey all the annotations made by Mallory's

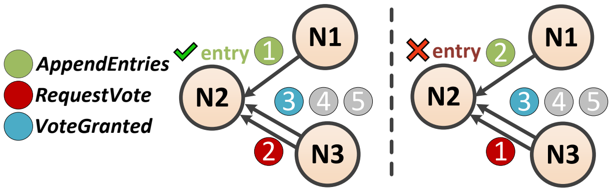


Fig. 6: Different states output similar network messages.

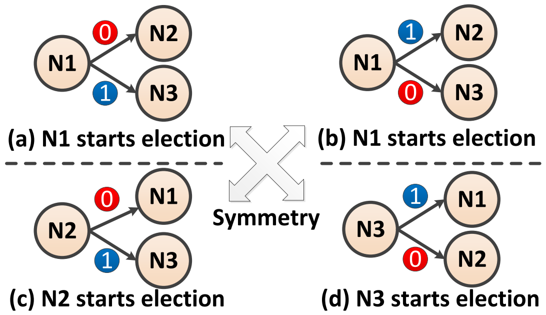


Fig. 7: Symmetry of the states in a distributed system

authors for their fuzzing campaigns, and confirms that 91% of them are covered by network messages.

Finding #2. **Similarity-based feedback pruning misses interesting states in distributed systems.**

Our second finding is that pruning similar network message sequences, as in Mallory, is prone to miss interesting states, since, for a distributed system, significantly different states may output similar network message sequences.

Figure 6 shows a motivating example with Raft. The current leader N1 sends an *AppendEntries* message to follower N2. Concurrently, due to a network partition, N3 starts the election and sends a *RequestVote* message to N2. Subsequently, N3 sends multiple messages to N2. If N2 receives *AppendEntries* before *RequestVote* (as shown in the left part of the figure), N2 will append the entry. Otherwise, it won't (as shown in the right part of the figure). Since the only difference in these two long sequences of network messages is the order between the first two messages, Mallory considers the two sequences similar, and thus may only explore one of these different states.

Finding #3. **Similarity-based feedback pruning explores redundant states due to symmetry.**

Our third finding is that Mallory's similarity-based pruning might lead to exploring redundant states. Figure 7 illustrates a motivating example using the election process of the consensus algorithm. To start an election, node $N1$ sends a message to nodes $N2$ and $N3$. In this example, the overall system state in all four parts of the figure is essentially the same as (or symmetric to) each other. However, in modulo symmetry, the sequences of network messages are quite different. As an example, in the (a) and (c) parts of the figure, all corresponding messages have different senders and receivers. Furthermore, two out of three nodes (*i.e.*, N1 and N2) have different numbers of sending and receiving messages. Thus, the similarity-based pruning approach in Mallory will consider these symmetric states differently.

DistFuzz's approach. For illustration purposes, we first present the initial design of DistFuzz's feedback, followed by our optimizations for effectiveness.

Based on Observation #1, DistFuzz only uses sequences of network messages (or message sequences) as the feedback. Specifically, DistFuzz encodes message sequences as a 5-tuple, where each item represents a network message, consisting of 1) a global sequence number recording the order of the events; 2) the type of the event (*i.e.*, send or receive); 3) the sender 4) and the receiver ID of the message; and 5) the message content. Note that to capture the state difference caused by unreliable networking (e.g., delays, message reordering), DistFuzz uses two message events (a send event and a receive event) to describe a message. The tables in Figure 8 show how DistFuzz encodes the message sequence for Figure 7.

Based on Observations #2 and #3, DistFuzz proposes a *systematic* symmetry-based pruning technique. DistFuzz departs from the *ad-hoc* approach that prunes network messages based on their similarities, and thus will not miss interesting states (*e.g.*, the one in Figure 6).

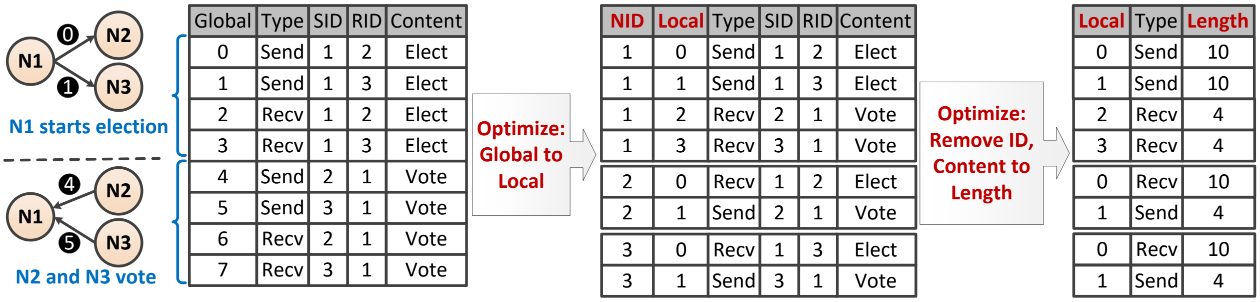
Motivated by the example in Figure 7, the insight behind our pruning technique is that the *symmetry in the system state* is reflected by *the symmetry* in the network messages. Furthermore, we leverage two kinds of symmetry in network messages: (1) *order symmetry*, where, as shown in parts (a) and (b), the difference in the order in which N2 or N3 receives the message does not lead to new states. and (2) *role symmetry*, where, as shown in parts (a), (c), (d), whether node N1, N2, or N3, starts the election does not lead to new states.

To prune *order symmetry*, as shown in the middle part of Figure 8, DistFuzz replaces the global sequence number with a local sequence number plus node ID. Thus, for two different nodes, the order they send or receive messages does not result in different encodings. Lastly, to prune *role symmetry*, as shown in the right part of the figure, DistFuzz removes all the node IDs in its encoding. As a result, the encoding stays the same regardless of which node starts the election.

A potential drawback of discarding similarity-based pruning is that the feedback is sensitive to an unimportant state in the network message content (*e.g.*, timestamps). We find that, essentially, the key information in the message content is the type of the message (*e.g.*, if a message is an "elect" or "vote" message). Thus, to avoid being too sensitive, DistFuzz should only encode the message type in the feedback.

To achieve this, a possible approach is to ask the user to provide a function that classifies a message based on its content, but this is laborious and error-prone. DistFuzz works around this problem by using the length of messages to approximate the message type (as shown in the right part of Figure 8). The intuition behind this is that different types of messages often have different lengths, especially when the length of various names in the system is fixed. Specifically, DistFuzz sets the name of databases, file, and the keys to a fixed length (1 and 3 in our evaluated systems) and does not fuzz these names.

To evaluate the accuracy of the approximation, we analyzed all the evaluated systems (§V-B) and found that, for each system, on average, 81% of message types (ranging from


⓿
❶
N1
N2
N3
N1 starts election
❹
❺
N2 and N3 vote
Optimize: Global to Local
Optimize: Remove ID, Content to Length

Global	Type	SID	RID	Content
0	Send	1	2	Elect
1	Send	1	3	Elect
2	Recv	1	2	Elect
3	Recv	1	3	Elect
4	Send	2	1	Vote
5	Send	3	1	Vote
6	Recv	2	1	Vote
7	Recv	3	1	Vote

NID	Local	Type	SID	RID	Content
1	0	Send	1	2	Elect
1	1	Send	1	3	Elect
1	2	Recv	2	1	Vote
1	3	Recv	3	1	Vote
2	0	Recv	1	2	Elect
2	1	Send	2	1	Vote
3	0	Recv	1	3	Elect
3	1	Send	3	1	Vote

Local	Type	Length
0	Send	10
1	Send	10
2	Recv	4
3	Recv	4
0	Recv	10
1	Send	4
0	Recv	10
1	Send	4

Fig. 8: Examples of DistFuzz's basic feedback encoding and various optimizations.

63% to 100%) are of different lengths, and messages with different lengths are of different types. In addition, for the message types of the same length, we found that most are used to confirm that a node (Node A) receives a request from another node (Node B). However, the request messages from Node B are of different lengths, and thus, the accuracy of our approximation is not compromised. In summary, our analysis shows that the approximation is accurate enough. Section V-D further evaluates the effectiveness of the feedback mechanism.

D. Minimizing Testing Overhead

A fuzzer must minimize the runtime testing overhead it incurs since this can significantly improve fuzzing effectiveness, as reported by prior work [49], [51]. Unfortunately, prior distributed system fuzzers suffer from high runtime testing overhead. For example, CrashFuzz reports that their techniques on average increase the execution time of a test case by 307%.

Fuzzing acceleration with checkpointing. To speed up fuzzing, DistFuzz employs checkpointing and restore techniques to minimize the initialization time. Specifically, before the fuzzing starts, DistFuzz brings up the SUT and takes a checkpoint of it by checkpointing the states of each node. Subsequently, during fuzzing, in each iteration of the fuzzing loop, DistFuzz restores from the checkpoint the state of the SUT. The checkpoint of DistFuzz consists of two kinds of state: (1) the state of processes in each node, including their registers and memory states, as well as in-kernel state (*e.g.*, file descriptors, sockets); (2) the file system state of the node, such as configuration files and files that store data of a distributed database. To further minimize the overhead, DistFuzz stores the checkpoint in memory. This approach incurs modest overhead (at a maximum of 700MB of memory space in our evaluation).

The initialization process of a distributed system may have bugs, and thus should be fuzzed. Hence, DistFuzz allows a user to provide the probability for DistFuzz to bring up the system under test directly without using the checkpoint, thereby fuzzing the initialization process.

Moving operations outside of the testing. The other mechanisms DistFuzz uses to minimize the testing overhead are based on the general idea of moving as many operations as possible from the performance-critical testing stage, which is already reflected in Figure 2. Specifically, DistFuzz 1) decides the input events before the actual testing (❹ in the figure) and 2) invokes the checker for bug detection after the testing (❼). Therefore, during testing, (❺ in Figure 2), DistFuzz only needs to apply input events and logs the relevant information for feedback and bug detection. Note that the reported evaluation time in §V (*i.e.*, 48 hours) includes the time of the two operations DistFuzz moves out of the actual testing.

E. Discussion

This subsection discusses a few more characteristics of DistFuzz. First, DistFuzz scales to large distributed systems with many nodes. This is because generating input sequences is fast as it is simply mutating and distributing a small seed file, and the event handler and runtime monitor is per-node, thereby avoiding a central bottleneck. Second, unlike prior distributed system fuzzers, DistFuzz does not need to directly deal with asynchronous behaviors and race conditions since it does not control the execution or scheduling of each node; it applies events by intercepting system calls (§IV-B) and uses the record/replay technique (RR) to reproduce bugs offline (§IV-C). Finally, DistFuzz does not suffer from false positives, thanks to its use of RR to deterministically reproduce the found bugs (§IV-C).

IV. Implementation

We implemented DistFuzz in C++ with a total of about 7300 lines of code. DistFuzz leverages strace [57] to intercept system calls, CRIU [9] for checkpointing and restoring, and RR [42] for deterministic record and replay.

To simplify deployment for other users, for all the systems that we have already deployed DistFuzz on, we provide a docker image. The users can thus easily bootstrap their testing process. DistFuzz also provides a template generator that automatically generates the docker image on the system it deploys, to ease testing in practice.

A. Checkers

DistFuzz uses five types of checkers at present:

Memory bug checkers for low-level languages. A memory bug is caused by programming mistakes involving pointer operations and/or dynamic memory management which lead to out-of-bounds memory access and dangling pointer dereference. Only distributed systems written in legacy languages like C/C++, require memory bug checkers; DistFuzz uses ASan [53] in our evaluation due to its high performance. We note that this does not break DistFuzz's design goal of being black-box. Memory checker is not a part of DistFuzz's design and DistFuzz can use Valgrind [60] for this purpose as well.

Linearizability violation checker for distributed DB. Distributed databases support linearizability as one option for a consistency model. To detect bugs that violate linearizability guarantees in distributed databases, DistFuzz uses an existing linearizability checker Knossos [26]. Knossos checks whether this sequence of requests and responses is possible under the linearizability consistency model. DistFuzz generates the sequence automatically based on the functions the user implements or *Get*, *Put* and *Cas* (§III-B).

Node crash checker. The node crash checker, by its name, detects whether a node has crashed. DistFuzz implements a simple one by checking whether the corresponding process of the distributed system still exists. DistFuzz runs the node crash checker after each event in the sequence.

Whole system availability checker. The availability checker validates whether the distributed system is still functional and can respond to client requests. DistFuzz invokes the checker at the end of the testing process, where all the events in an event sequence have been applied. The availability checker works by sending a status check command (§III-A) and checks if it returns the expected value within a timeout (2 seconds).

Log checker. Distributed systems often log runtime errors in specific files [66], [67]. To detect such errors, we implement a simple log checker that searches for keywords like *"FATAL ERROR"* and *"BUG"* in log files.

B. Events

As discussed in §III-A, DistFuzz uses events to interact with the distributed system. DistFuzz implements an event registry to help users add new events. DistFuzz automatically generates event sequences by mutating events in the registry.

Regular events. Regular events, as described in §III-A, need user inputs. DistFuzz allows users to define regular events by implementing C++ classes or shell scripts. *SysInit* is usually a shell script that initializes required environments and services like making directories, setting environments variables and starting services. Users need to write them according to target system's official documents. *Get*, *Put*, *Cas*, *CreateDB*, *DeleteDB*, *StatCheck* events are similar. They are just simply calling the corresponding APIs or client binaries with the given parameters. DistFuzz also provides a template to help users implement them. *NodeStart*, *NodeStop* and *NodeChange* need user to specify how to start, stop and change a node given the index of the node.

Fault events. *NetDup*, *NetFail*, *NetDelay*, *FileFail* and *FileDelay* are implemented by modifying strace to intercept the system calls. *NetDup* is calling the same system call with same parameters twice. *NetPart* and *NetFail* are forcing the system call to return error codes. *NetDelay* is waiting a while before the network system call. *FileFail* and *FileDelay* are implemented similarly for file system calls. *NetHeal* stops *NetPart*. *NodeKill* is implemented by killing the corresponding process. *NodeRestart* is implemented by killing the process and starting a new one.

C. Bug Reproduction

Prior work on distributed system fuzzing (*i.e.*, Jepsen [21], CrashFuzz [15] and Mallory [37]) do not provide mechanisms for users to reproduce the reported bugs. As a result, several bugs reported by prior work turned out to be false positives [34], [35], caused by bugs in the fuzzer. DistFuzz enables low-overhead reliable bug reproduction through an existing record and replay tool: RR [42]. To minimize testing overhead, DistFuzz reproduces bugs offline (*i.e.*, only after fuzzing campaigns finish). To reproduce a bug, DistFuzz repetitively feeds the SUT with the same bug-triggering input events and timing intervals. If one execution triggers the bug, all the non-deterministic events are recorded by RR, thereby bugs can be reproduced reliably. We have reliably reproduced all the bugs founded by DistFuzz.

TABLE IV: Information about the tested distributed systems.

System	Description	Lang	Version
Braft [5]	Raft implementation by Baidu	C++	commit 0c5a59
NuRaft [41]	Raft implementation by eBay	C++	commit 5a7a40
Dqlite [12]	Embeddable distributed DBMS	C	commit 37af7c
Redis [46]	Distributed key-value Store	C	commit e18c38
RethinkDB [47]	Distributed NoSQL DBMS	C++	v2.4.1
AerospikeDB [1]	Distributed NoSQL DBMS	C	v5.6.0.4
ClickHouse [7]	Distributed DBMS	C++	v21.9.2.17
etcd [13]	Distributed key-value store	Go	v2.2.0
ZooKeeper [20]	Distributed coordination system	Java	v3.5.1
HDFS [54]	Distributed file system	Java	v3.2.4

V. Evaluation

A. Experimental Setup

As shown in Table IV, we evaluate DistFuzz with 10 open-source and popular distributed systems, written in three languages: C/C++, Go and Java. Two of them are two industrial Raft implementations (Braft, NuRaft) and the other eight are production-level distributed systems. We chose the same evaluated systems as prior work (*i.e.*, Mallory and CrashFuzz) to enable a direction comparison. The design of DistFuzz is based on the common characteristics of distributed systems (§II-A) and we believe DistFuzz applies to other distributed systems (such as machine learning systems and network protocol/systems) as well.

Our evaluation machines are equipped with 18-core Intel i9-10980XE processors and 60GB memory. We disable hyperthreading and turboboost in BIOS and use *cpupower* to set the CPU frequency to the base one (*i.e.*, 3.00GHz). We follow the official documents of the evaluated systems for initialization and configure the length of the event sequence to 10. We only enable memory checkers for C/C++ systems and enable other checkers for all systems. Following [21], [37], [43], we test each distributed system on five nodes. Following [25], we repeat each fuzzing experiments for five times with a time limit of 48 hours for each fuzzing.

B. Runtime Testing

Table V shows the fuzzing experiment results about message sequences and bug detection for the ten tested distributed systems.

Testing coverage. DistFuzz exploits *message sequences* as fuzzing feedback to guide the fuzzing process. To understand the feedback improvement of DistFuzz, we also run each

TABLE V: Results of fuzzing distributed systems.

System	Message Sequences		Bug detection	
	Fuzzing	*Non-fuzzing*	*Found*	*Confirmed*
Braft	12.1K	1.1K (rand data)	10	5
NuRaft	11.5K	1.5K (rand data)	2	1
Dqlite	5.3K	1.7K (test suite)	9	5
Redis	13.1K	2.1K (test suite)	4	2
RethinkDB	11.3K	4.5K (test suite)	9	4
AerospikeDB	98.5K	55.3K (rand data)	3	3
ClickHouse	3.6K	1.5K (test suite)	2	1
etcd	14.4K	2.2K (test suite)	2	1
ZooKeeper	17.6K	7.2K (test suite)	8	5
HDFS	1.5K	1.3K (rand data)	3	1
Total	188.9K	78.4K	52	28

TABLE VI: Bugs found by checkers.

System	Checkers				
	ASan	*Linear*	*Crash*	*Avail*	*Log*
Braft	2	0	2	3	3
NuRaft	0	0	1	1	0
Dqlite	4	0	5	0	0
Redis	1	0	3	0	0
RethinkDB	4	0	2	1	2
AerospikeDB	1	1	1	0	0
ClickHouse	0	0	2	0	0
etcd	-	0	1	1	0
ZooKeeper	-	0	5	3	0
HDFS	-	0	2	0	1
Total	12	1	24	9	6

distributed system for 48 hours by executing its official test suites; if it has no test suite, we generate random client requests for 48 hours without feedback. Compared to non-fuzzing testing, DistFuzz in total finds 2.11x more message sequences in the SUTs, due to the novel techniques used by DistFuzz.

Bug detection. DistFuzz in total finds 52 real bugs in the 10 distributed systems. We have reported these bugs to the related developers, and 28 of them have been confirmed. We are still waiting for responses for the remaining bugs. According to the developers' feedback, 20 of the 28 confirmed bugs are previously unknown. Among the 28 confirmed bugs, 13 have been fixed by related developers, and 4 new CVEs have been assigned. The remaining 15 bugs are not fixed, as the developers have not found proper ways to fix them correctly in a short time, indicating the difficulty of bug fixing in distributed systems.

Bug-finding process. We analyze how DistFuzz finds these 52 bugs, and show the results in Table VI. 12 memory bugs are found by ASan, and 40 semantic bugs are found by our non-memory checkers. Specifically, only 1 linearizability violation is found by our linearizability checker, indicating distributed systems pay significant attention to data consistency; 24 node-crash bugs are found by our crash checker, and these bugs can cause partial failures in distributed systems; 9 availability bugs and 6 error-log bugs are found by our availability checker and log checker, respectively. These 52 bugs are found, as DistFuzz can cover many infrequent message sequences using our techniques. To clearly understand this reason, we run each distributed system for 48 hours by running its test suites or providing random client requests without fuzzing, and the five kinds of checkers do not find any bugs during testing.

TABLE VII: Bugs characteristics.

System	Events		Minimal events			
	Regular	*Fault*	*1*	*2*	*3*	*>3*
Braft	7	9	1	6	2	1
NuRaft	2	1	0	2	0	0
Dqlite	9	9	0	2	4	3
Redis	4	3	0	2	2	0
RethinkDB	8	8	1	7	1	0
AerospikeDB	1	3	0	3	0	0
ClickHouse	1	1	0	1	1	0
etcd	2	2	0	1	0	1
ZooKeeper	5	8	0	0	3	5
HDFS	3	3	0	0	1	2
%	81%	88%	4%	46%	27%	23%
Total	42	47	2	24	14	12

Root causes of the found bugs. We check the reports and source code of the 44 found bugs to analyze their root causes:

Memory bugs. Among the 12 memory bugs found by ASan, 3 are memory leaks, 6 are buffer-overflow issues (one example shown in Figure 9(a)), 1 is a null-pointer dereferences, 1 is a use-after-free, and 1 is a bad-free issue.

Linearizability violation. This violation is caused by a stale read [8] in AerospikeDB. The developers admitted that this violation is real in the community version that we tested, and it can be eliminated in the enterprise version.

Node-crash bugs. Among the 24 node-crash bugs, 13 are caused by assertion failures. As for these failed assertions, the developers mistakenly assume the related failures never happen. However, the assumptions about the 13 bugs are wrong. 10 node-crash bugs are caused by unhandled exceptions, which are neglected by the developers. 1 node-crash bug is caused by the wrong calculation of the snapshot index when a snapshot is loaded at the initialization phase in Dqlite (as shown in Figure 9(b)).

Availability bugs. Among the 9 availability bugs, 6 are caused by incorrect or missing recovery processes of node exceptions. For example, an availability bug found in ZooKeeper is incorrect handling of network failures (as shown in Figure 9(c)). 3 are caused by incorrect handling of special client requests, causing the distributed system to malfunction after a test of providing these client requests.

Error-log bugs. The 6 error-log bugs are caused by missing exception handling. Indeed, the developers know that related exceptions can occur but do not know how to handle them correctly, so only error logs are printed when these exceptions occur. Although these error-log bugs do not cause obvious problems in our tests, we believe that they may cause practical issues when the distributed system runs for a longer time.

Event types of bugs . To understand the event types responsible for bugs, we analyzed events needed for triggering each bug as shown in the columns *"Regular"* and *"Fault"* in Table VII. *SysInit* is removed during analysis as it is needed for all bugs. Many bugs need multiple events, so the total number is larger than the number of found bugs. The result shows that most of the bugs are triggered by both regular and fault events. In the table, 81% of the bugs need regular events, and 88% need fault events. The result indicates that both regular and fault events are important for triggering bugs in distributed systems.

Minimal event sequence for triggering bugs We manually analyzed the event sequences of triggering the bugs, referring to [52]. The results are shown in the column *"Minimal events"* in Table VII. 34 of the 42 bugs can be triggered by two or three events, and most of these need to be triggered by both regular and fault events. This finding coheres with bug studies [65] that find most of the reported distributed-system bugs can be triggered by two or three events.

Security impact of the found bugs Certain memory bugs DistFuzz finds may cause data leakage and code execution that can be exploited by attackers. Linearizability violations can lead to data inconsistency and data loss. Node-crash bugs cause partial failures which, although tolerable by distributed systems, degrade overall system performance. Finally, availability bugs, once being exploited, renders the system unable to respond to client requests for certain amount of time (2 seconds). Therefore, both node-crash bugs and availability bugs are vulnerable to DoS attacks.

The found bugs currently get assigned 4 CVEs (CVE-2023-50575 and CVE-2023-50576 for Braft, CVE-2023-50577 for AerospikeDB and CVE-2024-22937 for RethinkDB) that may have catastrophic security impacts. These 4 CVEs are caused by memory bugs, and they can be triggered easily by attackers. These CVEs are applied and assigned via MITRE [39] and have been reported to the developers.

In the following, we provide three case studies to illustrate bugs that were exposed by DistFuzz. Figure 9 shows these bugs, and they have been confirmed by related developers. The figure also shows the event sequences for the bugs.

Case study: Memory bug in Redis. In Figure 9(a), when the function `parseMovedReply` parses a client request that contains an empty space, the pointer variable `p` points to the location of this space, but the function passes the pointer `p+1` with the length of `p` to the function `NodeAddrParse`, which causes a buffer-overflow issue. To fix this bug, we have submitted a patch to move the pointer past the space, which matches the pointer and its length.

Case study: Node-crash bug in Dqlite. In Figure 9(b), during the start process of a node, it attempts to load an existing snapshot and also restores the variable `start_index` in the function `uvLoadSnapshotAndEntries`. When it is started for the first time and there is no snapshot, the node start is still successful. However, after a client request is provided and a snapshot is created, if the node is stopped forcibly, it cannot be restarted. Indeed, at that time, the `start_index` is mistakenly assigned to 1, which makes the following variable `last_index` less than `(*snapshot)->index`, causing the snapshot loading to fail. To fix this bug, the developers have submitted a patch to correctly calculate `start_index`.

Case study: Availability bug in Zookeeper. As shown in Figure 9(c), when two *NetFail*s happened on a node, the `zkServer` variable is set to a non-running state in the function `receiveMessage`, that will cause the `zk.isRunning` to return false. In such a case, the node cannot process the client request *Stats check* that should be processed successfully to report the node status. To fix this bug, the developers have submitted a patch to add a specific check for the non-running state of `zkServer` variable before processing *Stats check*.

C. Comparison to Existing Frameworks

We experimentally compare DistFuzz to four state-of-the-art distributed-system testing frameworks Jepsen [21], Mallory [37], CrashFuzz [15] and Namazu [40].

We evaluate each framework on all the target systems it can support. We ran DistFuzz and Jepsen on all target systems. Mallory only supports C/C++ and Rust program, so we evaluate it with Braft, NuRaft, Dqlite, Redis, RethinkDB, AerospikeDB ClickHouse. The annotations of Braft, Dqlite, and Redis are from the authors of Mallory; we annotate other target systems with our best efforts following the style of the Mallory author. CrashFuzz only supports Java program and we evaluate it on ZooKeeper and HDFS. While Namazu supports Go and Java program, it has not been kept up-to-date; its last commit is six years ago, and as a result, despite our best efforts, we can only evaluate it on etcd and Zookeeper. We repeat each fuzzing experiment for five times.

Message sequences. Figure 10 plots communication sequences covered by DistFuzz and the 4 frameworks. In the figure, DistFuzz covers more message sequences than all other frameworks. Jepsen and Namazu only generate inputs randomly. Mallory generates random regular events and injects fault events based on its feedback. In contrast, DistFuzz generates test cases from multiple dimensions with more effective program feedback of message sequence, so DistFuzz can generate more effective test cases to improve message sequence coverage.

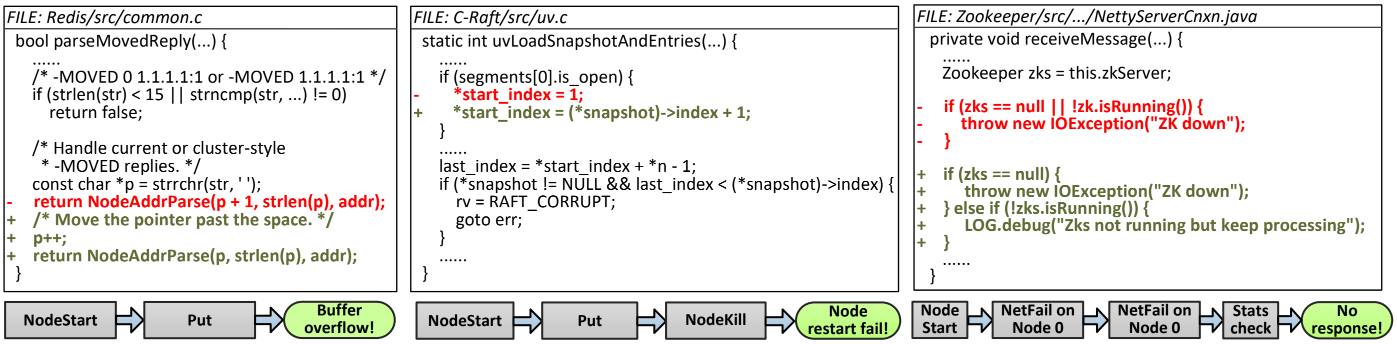
Found bugs. Figure 11 shows the numbers of bugs found by DistFuzz and the four frameworks. Jepsen, Mallory, CrashFuzz and Namazu find 12, 16, 2 and 2 bugs, respectively. Specifically, Jepsen and Mallory, Jepsen and Namazu find common bugs. Thus, these testing frameworks in total found 19 unique bugs. Among these 19 bugs, there are two Dqlite bugs found by Mallory that are missed by our framework. However, after our investigation, we found that these two bugs are false positives, as they are caused by the bugs in the testing framework itself [34], [35]. All remaining bugs are found by DistFuzz, and it also finds 35 bugs missed by the four frameworks, due to covering more message sequences.

D. The Effectiveness of Two Conceptual Contributions

To understand the benefits of the two conceptual contributions, we run DistFuzz without 1) mutating regular events and timing interval (DistFuzz-T), and 2) pruning message sequences (DistFuzz-S), to understand the benefits of these techniques. We also run DistFuzz without both techniques (DistFuzz-WF) as a baseline.

We ran a 48-hour experiment, and repeat each experiment for five times. We analyzed the time of finding bugs in the 2 distributed systems as listed in Table VIII. We find that the overall bug-finding time is increased for DistFuzz-T and DistFuzz-S by 338.39 and 249.67 hours, and the average bug-finding time is increased by 18.25 and 13.87 hours, respectively. 2 and 3 bugs are missed by DistFuzz-T and DistFuzz-S, indicating the importance of these techniques in bug detection.

Compared to the baseline, we found that by using our two conceptual contributions, the bug-finding time is reduced by



```
FILE: Redis/src/common.c
 bool parseMovedReply(...) {
    ......
    /* -MOVED 0 1.1.1.1:1 or -MOVED 1.1.1.1:1 */
    if (strlen(str) < 15 || strncmp(str, ...) != 0)
       return false;

    /* Handle current or cluster-style
      * -MOVED replies. */
    const char *p = strrchr(str, ' ');
-   return NodeAddrParse(p + 1, strlen(p), addr);
+   /* Move the pointer past the space. */
+   p++;
+   return NodeAddrParse(p, strlen(p), addr);
 }
```

```
FILE: C-Raft/src/uv.c
 static int uvLoadSnapshotAndEntries(...) {
    ......
    if (segments[0].is_open) {
-      *start_index = 1;
+      *start_index = (*snapshot)->index + 1;
    }
    ......
    last_index = *start_index + *n - 1;
    if (*snapshot != NULL && last_index < (*snapshot)->index) {
       rv = RAFT_CORRUPT;
       goto err;
    }
    ......
 }
```

```
FILE: Zookeeper/src/.../NettyServerCnxn.java
 private void receiveMessage(...) {
    ......
    Zookeeper zks = this.zkServer;

-   if (zks == null || !zk.isRunning()) {
-      throw new IOException("ZK down");
-   }

+   if (zks == null) {
+      throw new IOException("ZK down");
+   } else if (!zks.isRunning()) {
+      LOG.debug("Zks not running but keep processing");
+   }
    ......
 }
```

(a) Memory bug in Redis (b) Node-crash bug in Dqlite (c) Availability bug in Zookeeper

Fig. 9: Example bugs found by DistFuzz in Redis, Dqlite and Zookeeper.

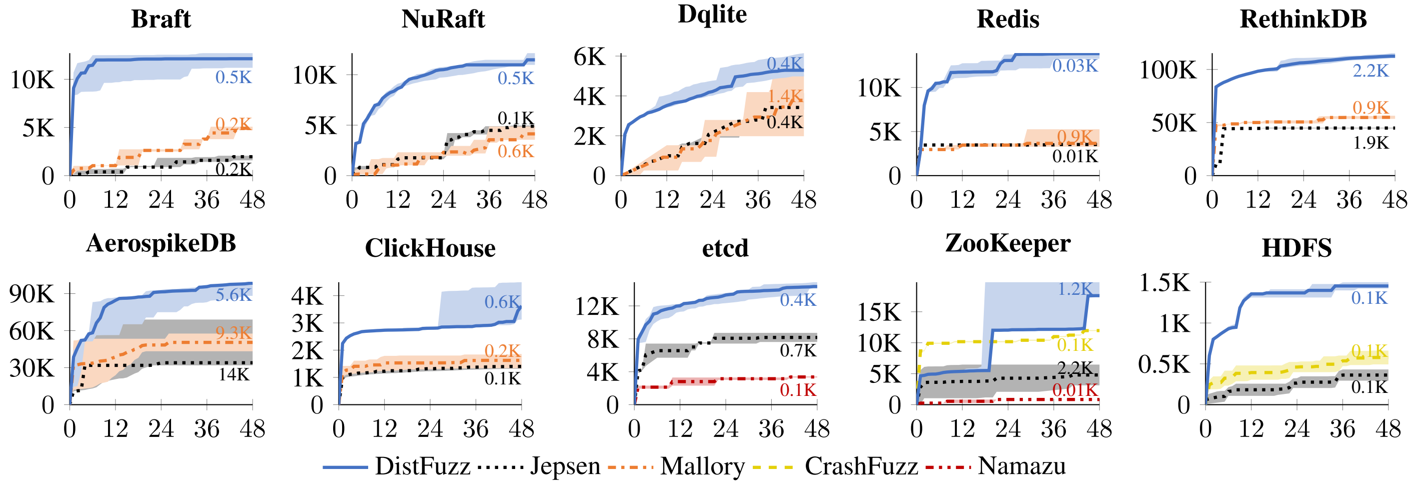


Fig. 10: Comparison of the covered message sequences. The line represents the median value, while the shaded area shows the minimum and maximum ranges observed across five experiments. The number at the end of each line is the margin of error at the 95% confidence level.

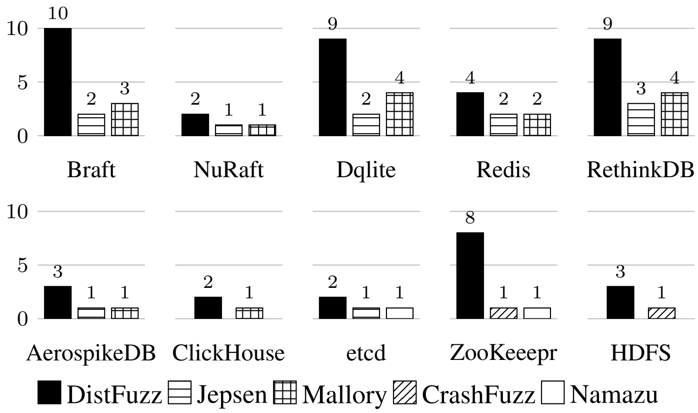


Fig. 11: Comparison of the found bugs.

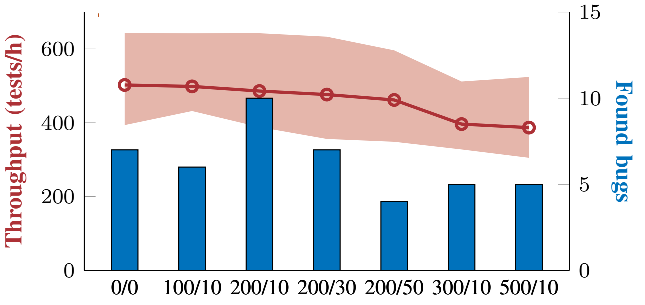


Fig. 12: Fuzzing throughput (line) and the number of founded bugs (bar) in Braft with different maximum timing intervals and granularity. The shaded area shows the minimum and maximum ranges across five experiments.

299.64 hours in total and 16.6 hours on average than running without them (DistFuzz-WF). 5 bugs are missed by DistFuzz-WF, indicating the importance of these techniques in bug detection. The min time for bug Zookeeper-3 is a bit longer with DistFuzz as it is a concurrent bug that has much non-determinism to trigger during early testing.

E. Impact of Timing Intervals

Figure 12 shows the impact of different timing values with Braft. We find that the throughput only decreases from 502 to 486 (tests per hour) when the maximum timing interval is less than 200ms, but decrease from 486 to 386 when the timing interval is larger. The granularity does not affect the throughput significantly, but it affects the found bugs. We find that 200/10 is the best choice, as it can find more bugs than others.

TABLE VIII: The median bug finding time in hours for Braft and Zookeeper. TO means the bug is not detected in 48 hours. The minimum, maximum and margin of error of bug-finding time across five runs are shown in parentheses. Total is the total of average bug-finding time for all bugs.

Bug	DistFuzz	DistFuzz-T	DistFuzz-S	DistFuzz-WF
B-1	**0.16**(.16,.16,0)	0.16(.15,.18,.02)	0.18(.17,.18,.01)	0.16(.16,.16,0)
B-2	**8**(5.8,13,4.5)	28(27,TO,13)	TO(TO,TO,0)	TO(TO,TO,0)
B-3	**1.1**(.5,1.5,.6)	TO(23,TO,16)	9(4.5,12,4.7)	TO(TO,TO,0)
B-4	**1.7**(1.7,2,.21)	29(12,TO,21)	32(16,TO,18)	32(4.3,TO,25)
B-5	**2.1**(2,4.1,1.4)	TO(TO,TO,0)	TO(TO,TO,0)	TO(TO,TO,0)
B-6	**3.3**(2.5,4.1,.95)	4.3(3.1,4.6,.94)	22(17,26,4.7)	25(3.3,27,15)
B-7	**1**(.8,1.3,.3)	1.4(.8,2.1,.77)	23(16,TO,19)	3.6(2,5.5,2.1)
B-8	**2.7**(2.1,29,18)	TO(TO,TO,0)	TO(TO,TO,0)	TO(TO,TO,0)
B-9	**0.06**(.06,.06,0)	0.06(.06,.08,.01)	0.11(.1,.14,.02)	3.1(.4,4.1,2.2)
B-10	**0.02**(.02,.02,0)	0.02(.02,.03,.01)	0.02(.02,.02,0)	0.02(.02,.05,.02)
ZK-1	**0.52**(.42,.83,.25)	2.6(2.1,3.5,.83)	1.1(.9,2.3,.9)	0.7(.53,1.1,.35)
ZK-2	**0.53**(.44,.6,.1)	0.9(.86,1.4,.38)	1.3(.9,6.1,3.4)	11(2.7,28,15)
ZK-3	**8.8**(7.4,8.9,.98)	8.8(8.3,1.1,1.2)	9.3(4.5,31,17)	25(6.7,26,13)
ZK-4	**0.5**(.43,1.2,.5)	TO(42,TO,3.9)	13.1(3.5,14,6.9)	TO(TO,TO,0)
ZK-5	**0.02**(.02,.02,0)	0.02(.02,.02,0)	0.07(.03,.1,.04)	0.02(.02,.02,0)
ZK-6	**0.02**(.02,.03,.01)	0.02(.02,.02,0)	0.03(.03,.03,0)	0.09(.02,.13,.07)
ZK-7	**0.22**(.22,.47,.17)	TO(37,TO,7.5)	22(1.2,41,24)	4.1(.33,7.3,4.2)
ZK-8	**4.3**(4.1,5.77,1.1)	TO(31,TO,12)	6.8(5.7,31,17)	14(5.1,42,23)
Total	**35.05**	**363.44**	**284.72**	**359.69**
Bugs	18	16	15	13

VI. Limitations

Compared to other distributed system fuzzers, DistFuzz has increased the testing input space by including regular events and timing intervals among events. Our evaluation shows that, for all systems we evaluate, the increase in the testing input space significantly improves rather than reduces DistFuzz's effectiveness (§V-B). This is largely due to DistFuzz's effective feedback approach. However, if certain bugs can be triggered by simple regular events and/or are not timing-dependent, it may take DistFuzz a longer time to expose them.

As demonstrated in §V-D, our feedback approach significantly improves the fuzzing performance in general. However, we note that, in some rare cases, our feedback approach may identify two potentially different message sequences as the same. Specifically, suppose a distributed system's message lengths are always the same, then discarding the message contents will negatively affect fuzzing performance. Similarly, if all nodes in a system have different roles, the node IDs are thus important information and should not be discarded. However, we believe that few real-world distributed systems confirm the above two characteristics.

We note that the limitations in DistFuzz are fundamentally due to the constraints in blackbox fuzzing. Due to this constraint, to increase effectiveness, DistFuzz makes assumptions that apply to most distributed systems. Considering the huge improvement in general efficiency, such limitations do not undermine the value of techniques.

VII. Related Work

The most relevant state-of-the-art work (*i.e.*, Jepsen [21], Mallory [37], and CrashFuzz [15]) has been qualitatively (§I and §III) and quantitatively compared (§V-C) across the paper. This section discusses other related work.

Model checking. Compared to DistFuzz, distributed systems model checkers [29], [32], [64] are for different purposes and thus are orthogonal to DistFuzz. Fundamentally, model checkers aim to validate that, given a set of *fixed* testing workloads, all the states that the SUT can reach are valid. As a result, model checkers *exhaustively* enumerate all the reachable states with the given testing workloads. In contrast, DistFuzz aims to explore *different* workloads and faults to expose hidden bugs. DistFuzz does not aim to systematically explore all possible states but instead relies on feedback mechanisms to prioritize exploring interesting states.

In addition, due to the need to (i) *exhaustively* enumerate all reachable states, model checkers further differ from DistFuzz in the following aspects. First, model checkers require access to source code [64] and/or user inputs [32] while DistFuzz does not. This is because, for exhaustive exploration, model checkers need extra insights on the SUT such as the code semantics and/or users' domain knowledge. For example, to ensure that the timeout event leads to different states, Modist [64] employs specific code static analysis to identify the timeout handlers. Second, model checkers have a much higher testing overhead compared to DistFuzz. This is because, for exhaustive exploration, model checkers need to ensure the SUT executes deterministically, by, *e.g.*, controlling the thread scheduling to explore all possible interleavings [64] caused by multithreading. Such mechanisms often significantly increase the testing overhead. DistFuzz is free of this kind of overhead and still maintains the advantages of deterministically reproducing reported bugs (§IV-C).

DistFuzz's approach on leveraging symmetry to prune the essentially same message (§III-C) is related to FlyMC [32] model checker. Both of them perform reduction based on partial orders. The difference is that FlyMC applies symmetry to prune redundant *states* and FlyMC relies on user inputs to define and identify which part of the states are significant. DistFuzz applies symmetry to prune redundant *message sequences* without requiring user inputs.

Fuzzing. Fuzzing is a promising testing technique and has been widely used in testing system software. Prior works apply fuzzing to complex system software, including operating systems [50], [56], [58], fundamental libraries [2], [6], [19], hypervisors [18], [48], filesystems [24], [63] and network programs [33], [45], [51], [55], [70]. These approaches have produced good results in bug detection. Some of them also use techniques like multi-dimension inputs [6], [58], [70] and checkpointing [51], [56] to improve the fuzzing efficiency. DistFuzz, referring to existing model checking and testing methods [21], [28], [37], [52], apply fuzzing to distributed systems and overcome the shortcomings of existing fuzzing and testing tools for distributed systems by mutating regular events and timing interval, and leveraging symmetry to prune redundant message sequences.

VIII. Conclusion

This paper presents DistFuzz, the first feedback-driven black-box fuzzing framework for distributed systems based on our findings on the nature of real-world distributed systems and defects of prior fuzzers. DistFuzz proposes a multi-dimensional input space by incorporating regular events and relative timing among events, and utilizes sequences of network messages with symmetry-based pruning as feedback. We deploy DistFuzz to test 10 popular distributed systems. DistFuzz finds 52 real bugs, and 28 have been confirmed by the developers. 20 of the 28 confirmed bugs are previously unknown, and 4 have been assigned with CVEs.

IX. Acknowledgement

We want to thank the anonymous reviewers for their suggestions to improve the final version of this paper. This work was supported by the Fundamental Research Funds for the Central Universities. Jia-Ju Bai is the corresponding author.

Appendix A
Artifact Appendix

This Appendix contains a complete description on the artifacts presented in our paper, and detailed instructions on how to access, install, and evaluate them.

A. Description & Requirements

This section provides all the information necessary to recreate the experimental setup to run our artifacts. A recent X86 desktop or server machine should be sufficient to run the experiments.

1) How to access: The artifact is available on Github [2]. We recommend using the latest Docker image `zouyonghao/distfuzz:artifact` as described in the `README.md` file. A Docker image for offline use is also available on Zenodo [3].

2) Hardware dependencies: We have tested DistFuzz on Intel i9-10980XE and Intel(R) Xeon(R) Gold 6248R with 60GB memory. No special settings are needed. A recent commodity PC build should be able to run DistFuzz and its reproduction feature. We recommend using a similar Intel CPU series with our tested CPUs. DistFuzz does not require too much memory, but to ensure a good performance, a machine with more than 16GB of memory is recommended.

3) Software dependencies: A recent Linux operating system is required. Our artifacts have been tested on Ubuntu 20.04 LTS (Focal). Docker is required to run the artifact. We recommend using the latest Docker version (27.0.3).

4) Benchmarks: None.

B. Artifact Installation & Configuration

We suggest to run the experiments using the Docker image we provide. Many capabilities are required to run the artifact, so we suggest running the Docker image with the following command.

```
docker run -it --network none \
    --cap-add=NET_ADMIN \
    --device=/dev/net/tun \
    --cap-add=SYS_PTRACE \
    --cap-add=SYS_NICE \
    --cap-add=IPC_LOCK \
    --security-opt \
    seccomp=unconfined \
    zouyonghao/distfuzz:artifact
```

Alternatively, you can build the Docker image from the source code. This method requires first cloning the repository via `git clone` and then building the Docker image via `docker build` in the cloned directory. Running without Docker is also possible, but it requires installing all the dependencies manually which is described in the `INSTALL.md` file in the repository.

C. Experiment Workflow

The artifacts contain three experiments. The first is using DistFuzz to fuzz the evaluated systems described in the paper. The second is reproducing the bugs found by DistFuzz for etcd and ZooKeeper. The third is testing the checkpoint time of different systems. The proposed workflow runs the three experiments sequentially. The repository contains scripts that can be used to automate all experiments. This Appendix only illustrates the commands to run at each step, while extensive documentation is provided in the repository.

D. Major Claims

- (C1): DistFuzz can fuzz and find bugs in distributed systems with multi-dimensional inputs and symmetry-based feedback prune.
- (C2): DistFuzz's `rr` can reproduce bugs in distributed systems.
- (C3): DistFuzz's checkpointing mechanism can reduce the boot time of distributed systems.

E. Evaluation

1) Experiment (E1): [Fuzzing distributed systems] [30 human-minutes + 24 compute-hour]: fuzzing the systems evaluated in the paper using DistFuzz. For each system, there is a folder called `${SYSTEM}_test`, e.g., `braft_test`. The experiment should run in the folder.

[Preparation] In a new Docker container started with the provided command, go to the `${SYSTEM}_test/bin` folder.

[Execution] Execute the script `fuzz.sh` in the folder.

[Results] The script will start the fuzzing process. The output will show that DistFuzz is starting the target system nodes and sending events to them. After fuzzing for a while, you can check test cases that have errors under `test_cases`. Fuzzing status is in the file `output/fuzzer1/plot-curve` and the column starting with `bc` is the coverage.

2) Experiment (E2): [Reproducing bugs using DistFuzz's `rr`] [30 human-minutes + 2 compute-hour]: reproducing the bugs found by DistFuzz in etcd and ZooKeeper using DistFuzz's `rr`. We provide example bug reproduction Docker images we used during our communication with developers. Please note that `rr` has requirements for CPU, so it is better to use a machine described in the hardware dependencies section.

[Preparation] Pull all three Docker images for the bug reproduction. The images are `zouyonghao/etcd-13493`

[2] https://github.com/zouyonghao/DistFuzz

[3] https://doi.org/10.5281/zenodo.14098389

and zouyonghao/etcd-10166 for 2 etcd bugs in the paper, and zouyonghao/distfuzz:zookeeper-rr for 8 ZooKeeper bugs. rr requires hardware and software settings to run, including precise hardware performance counters and kernel setting perf_event_paranoid = 1. Please check these settings before running the experiments.

[Execution] Run the following commands to reproduce the bugs.

```
# For the first etcd bug
docker run -it --rm \
    --cap-add=SYS_PTRACE \
    --security-opt \
    seccomp=unconfined \
    zouyonghao/etcd-13493 \
    rr replay -a \
    /root/test_cases/3169/rr_rec_1_0

# For the second etcd bug
docker run -it --rm \
    --cap-add=SYS_PTRACE \
    --security-opt \
    seccomp=unconfined \
    zouyonghao/etcd-10166 \
    rr replay -a \
    /root/10166/rr_rec_2_0

# For ZooKeeper bugs (8 bugs)
docker run -it --rm \
    --cap-add=SYS_PTRACE \
    --security-opt \
    seccomp=unconfined \
    zouyonghao/distfuzz:zookeeper-rr \
    rr replay -a \
    /home/zyh/zookeeper-[1-8]
```

[Results] The commands will start a process running a system's node. The output will show that the system is starting and replaying the recorded events. After replaying, the system will crash or output error logs. The bug is the same as the one found by DistFuzz.

3) Experiment (E3): [Checkpointing] [30 human-minutes + 2 compute-hour]: comparing the boot time of systems with no checkpointing, checkpointing and checkpointing + tmpfs. For each system's ${SYSTEM}_test/bin, there is a folder called experiments/boot_time_test. The experiment should run in the folder.

[Preparation] In a new Docker container started with the provided command, go to the ${SYSTEM}_test/bin/experiments/boot_time_test folder.

[Execution] Execute the script test_boot_time.sh in the folder.

[Results] The script will start booting the system with different configurations. The boot time for each configuration will be recorded to a file called test_boot_time_result.

References

[1] "Aerospike database server: flash-optimized, in-memory, nosql database." 2022, https://github.com/aerospike/aerospike-server.

[2] "AFL: American Fuzzy Lop," 2020, https://github.com/google/AFL.

[3] "Amazon Web Services outage causes issues at Disney, Netflix, and Coinbase," 2021, https://www.cnbc.com/2021/12/07/amazon-web-services-outage-causes-issues-at-disney-netflix-coinbase.html.

[4] "Summary of the AWS service event in the US East Region," 2024, https://aws.amazon.com/message/67457/.

[5] "An industrial-grade C++ implementation of Raft consensus algorithm," 2021, https://github.com/baidu/braft.

[6] P. Chen, Y. Xie, Y. Lyu, Y. Wang, and H. Chen, "Hopper: Interpretative fuzzing for libraries," in *Proceedings of the 30th International Conference on Computer and Communications Security (CCS)*, 2023, pp. 1600–1614.

[7] "ClickHouse: an open-source, high performance columnar OLAP database management system for real-time analytics using SQL." 2022, https://clickhouse.com/.

[8] "Consistency model used in distributed systems," 2024, https://en.wikipedia.org/wiki/Consistency_model.

[9] "CRIU, a project of checkpoint/restore functionality for Linux," 2022, https://criu.org.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[11] Z. Y. Ding and C. Le Goues, "An empirical study of OSS-Fuzz bugs," in *Proceedings of the 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 131–142.

[12] "Dqlite: an embeddable, replicated and fault-tolerant sql engine." dqlite.io.

[13] "etcd: a distributed, reliable key-value store for the most critical data of a distributed system," 2022, https://etcd.io/.

[14] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[15] Y. Gao, W. Dou, D. Wang, W. Feng, J. Wei, H. Zhong, and T. Huang, "Coverage guided fault injection for cloud systems," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2211–2223.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP 03)*, 2003, pp. 29–43.

[17] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. A. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 International Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015, pp. 139–152.

[18] A. Henderson, H. Yin, G. Jin, H. Han, and H. Deng, "VDF: targeted evolutionary fuzz testing of virtual devices," in *Proceedings of the 20th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2017, p. 23.

[19] "Honggfuzz: a security oriented, feedback-driven, evolutionary, easy-to-use fuzzer with interesting analysis options." 2023, https://github.com/google/honggfuzz.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: wait-free coordination for Internet-scale systems," in *Proceedings of the 2010 USENIX Annual Technical Conference*, 2010, p. 14.

[21] "Jepsen: a framework for distributed systems verification, with fault injection." 2024, https://github.com/jepsen-io/jepsen.

[22] Z.-M. Jiang, J.-J. Bai, and Z. Su, "Dynsql: stateful fuzzing for database management systems with complex and valid sql query generation," in *Proceedings of the 32nd USENIX Security Symposium*, 2023, pp. 4949–4965.

[23] M. E. Joorabchi, M. MirzaAghaei, and A. Mesbah, "Works for me! characterizing non-reproducible bug reports," in *Proceedings of the 2014 International Conference on Mining Software Repositories (MSR)*, P. T. Devanbu, S. Kim, and M. Pinzger, Eds., 2014, pp. 62–71.

[24] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *Proceedings of the 27th International Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 147–161.

[25] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 25th International Conference on Computer and Communications Security (CCS)*, 2018, pp. 2123–2138.

[26] "Knossos: verifies the linearizability of experimentally accessible histories." 2021, https://github.com/jepsen-io/knossos.

[27] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[28] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru, "Turret: A platform for automated attack finding in unmodified distributed system implementations," in *Proceedings of the 2014 IEEE International Conference on Distributed Computing Systems*, 2014, pp. 660–669.

[29] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "TaxDC: a taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 517–530.

[30] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, p. 6, Jun. 2018.

[31] J. Lu, L. Chen, L. Li, and X. Feng, "Understanding node change bugs for distributed systems," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 399–410.

[32] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa, A. Gupta, S. Lu, and H. S. Gunawi, "FlyMC: highly scalable testing of complex interleavings in distributed systems," in *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, 2019, pp. 1–16.

[33] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, "Polar: function code aware fuzz testing of ICS protocol," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5s, pp. 93:1–93:22, 2019.

[34] "Addresssanitizer: stack-buffer-overflow in server.c," 2023, https://github.com/redis/redis/issues/12005.

[35] "For jepsen: extra online spare," https://github.com/canonical/dqlite/issues/585.

[36] "Raft: membership rollback issue," https://github.com/canonical/raft/issues/250.

[37] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, "Greybox fuzzing of distributed systems," in *Proceedings of the 30th International Conference on Computer and Communications Security (CCS)*, 2023, pp. 1615–1629.

[38] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[39] "CVE - MITRE," 2024, https://cve.mitre.org/.

[40] "Namazu: programmable fuzzy scheduler for testing distributed systems," 2016, https://github.com/osrg/namazu.

[41] "NuRaft: C++ implementation of Raft core logic as a replication library." 2022, https://github.com/eBay/NuRaft.

[42] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017, pp. 377–389.

[43] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014, pp. 305–320.

[44] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the 2017 Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 69–84.

[45] V. Pham, M. Böhme, and A. Roychoudhury, "AFLNet: a greybox fuzzer for network protocols," in *Proceedings of the 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 460–465.

[46] "RedisRaft: a Redis module that make it possible to create a consistent Raft cluster from multiple Redis instances." 2022, https://github.com/RedisLabs/redisraft.

[47] "RethinkDB: the open-source database for the realtime web," 2022, https://github.com/rethinkdb/rethinkdb.

[48] S. Schumilo, C. Aschermann, A. Abbasi, S. Worner, and T. Holz, "HYPER-CUBE: high-dimensional hypervisor fuzzing," in *Proceedings of 27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[49] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: greybox hypervisor fuzzing using fast snapshots and affine types," in *Proceeding of the 30th USENIX Security Symposium*, 2021, pp. 2597–2614.

[50] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 167–182.

[51] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," in *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*, 2022, pp. 166–180.

[52] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker, "Minimizing faulty executions of distributed systems," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016, pp. 291–309.

[53] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, Jun. 2012, pp. 309–318.

[54] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[55] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *Proceedings of the 23rd International Conference on Computer and Communications Security (CCS)*, 2016, pp. 1492–1504.

[56] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *Proceedings of the 29th USENIX Security Symposium*. USENIX Association, Aug. 2020, pp. 2541–2557.

[57] "Linux strace is a diagnostic, debugging and instructional userspace utility for Linux." 2023, https://strace.io/.

[58] "Syzkaller: an unsupervised coverage-guided kernel fuzzer," 2021, https://github.com/google/syzkaller.

[59] "Trinity: a linux system call fuzz tester," 2019, https://github.com/kernelslacker/trinity.

[60] "Valgrind: an instrumentation framework for building dynamic analysis tools," 2023, https://valgrind.org/.

[61] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: yet another resource negotiator," in *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013, pp. 1–16.

[62] "Distributed computing," https://en.wikipedia.org/wiki/Distributed_computing.

[63] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019, pp. 818–834.

[64] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "MODIST: transparent model checking of unmodified distributed systems," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009, p. 16.

[65] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: an analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th International Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 249–265.

[66] W. Yuan, S. Lu, H. Sun, and X. Liu, "How are distributed bugs diagnosed and fixed through system logs?" *Information and Software Technology (IST)*, vol. 119, pp. 1–18, 2020.

[67] X. Zhao, Y. Zhang, D. Lion, M. FaizanUllah, Y. Luo, D. Yuan, and M. Stumm, "lprof: a non-intrusive request flow profiler for distributed systems," in *Proceedings of the 11th International Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, p. 16.

[68] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys*, Jan. 2022.

[69] "Zookeeper-4409 nullpointerexception in sendackrequestprocessor," https://github.com/apache/zookeeper/pull/1774.

[70] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "TCP-Fuzz: detecting memory and semantic bugs in TCP stacks with fuzzing," in *Proceedings of the 2021 USENIX Annual Technical Conference*, 2021, pp. 489–502.