Efficient Exposure of Partial Failure Bugs in Distributed Systems with Inferred Abstract States

Haoze Wu and Jia Pan, *Johns Hopkins University;*
Peng Huang, *University of Michigan*

https://www.usenix.org/conference/nsdi24/presentation/wu-haoze

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the
21st USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Efficient Exposure of Partial Failure Bugs in Distributed Systems with Inferred Abstract States

Haoze Wu[†] Jia Pan[†] Peng Huang[‡]

Johns Hopkins University[†] *University of Michigan*[‡]

Abstract

Many distributed system failures, especially the notorious partial service failures, are caused by bugs that are only triggered by subtle faults at rare timing. Existing testing is inefficient in exposing such bugs. This paper presents LEGOLAS, a fault injection testing framework designed to address this gap. To precisely simulate subtle faults, LEGOLAS statically analyzes the system code and instruments hooks within a system. To efficiently explore numerous faults, LEGOLAS introduces a novel notion of *abstract states* and *automatically* infers abstract states from code. During testing, LEGOLAS designs an algorithm that leverages the inferred abstract states to make careful fault injection decisions. We applied LEGOLAS on the latest releases of six popular, extensively tested distributed systems. LEGOLAS found 20 new bugs that result in partial service failures.

1 Introduction

Deployed distributed systems frequently encounter faults in the underlying hardware and dependent software. While these systems are generally fault-tolerant, an unexpected fault can still expose bugs. Indeed, real-world distributed system outages are often triggered by some fault events [6, 14, 16, 34].

Fault injection testing, also known as chaos engineering [48], has gained popularity to find fault-induced bugs early. Various solutions are developed to inject common faults such as crashes [3,15,40], disk faults [13,26], and network partitions [2, 3,27]. Despite the progress, many complex fault-induced bugs remain hidden in existing testing and cause failures after deployment. These bugs share several characteristics.

First, they cause puzzling symptoms where the services seem to work but are partially broken, which are notorious in production distributed systems [10, 12, 22, 23, 36]. Figure 1 shows a real failure from a ZooKeeper deployment. The clients experienced timeouts in `create` requests, but `get` requests still succeeded. Pinging the leader also showed that it was alive. As another example, users reported [5, 9, 25] that their Kafka cluster occasionally experienced partial breakdown, and one broker could not return to an *in-sync* status.

Second, these bugs are triggered under subtle faulty conditions, such as a network error that only affects some operations but not others [1, 36], transient slowness [17, 21], or microburst [28]. In the aforementioned ZooKeeper example,

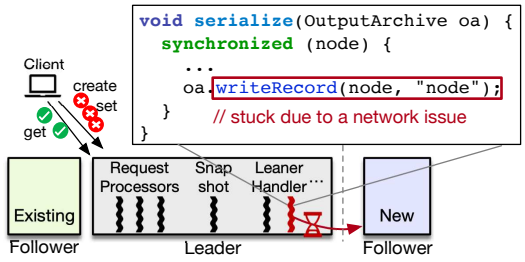


Figure 1: A real ZooKeeper production incident [46] triggered by a partial network fault, which caused the `writeRecord` operation to be stuck while holding a lock.

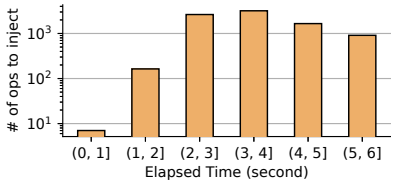


Figure 2: Hundreds to thousands of operations per second are candidates for injecting `IOException` during ZooKeeper's execution.

the buggy code works properly in normal conditions. The failure was only exposed by a partial network fault between a leader and a new follower, and the fault only affects a specific operation (`writeRecord`). The faults can also originate from software and be system-specific, such as a custom exception from an RPC to a remote component (*e.g.*, when the database is overloaded). The Kafka failure example was caused by a custom exception returned from an RPC to the dependent ZooKeeper service. Simulating these fault conditions in testing requires precise control of the fault types and locations.

Third, these bugs require careful choices of when and where the fault occurs. Distributed systems have a large number of fault points (Figure 2), but since these systems are robust, most faults would be tolerated or result in an *expected* failure (*e.g.*, abort on error in reading a file). To expose the ZooKeeper failure, a transient network latency increase must be injected while a new follower is requesting a snapshot from the leader. Injecting the fault at other times or other locations is ineffective. A random injection choice, which is commonly used in existing solutions, will have a high chance of missing the buggy point.

We present LEGOLAS, a fault injection framework designed to efficiently expose the above class of complex fault-induced bugs in large distributed systems. Unlike the practice of

injecting system-agnostic faults externally in the environment or libraries, LEGOLAS uses program analysis to perform fine-grained and system-specific fault injection. It analyzes the fault conditions for each instruction in the code and instruments hooks to precisely simulate subtle faults *within* the system.

With more faults to consider, the problem of a large fault injection space becomes more pronounced.

Our insight is that production bugs occur in unusual conditions—otherwise, existing testing likely has exposed it. Thus, we can *selectively* inject faults by checking if the system reaches an unusual condition. Unfortunately, we do not know beforehand whether a program point is unusual or not. Using *ad-hoc* heuristics, such as only injecting faults when the program is inside a critical section, can miss many failure-inducing conditions. We should still systematically explore the injection choices for generality and completeness.

Based on this insight, LEGOLAS introduces a novel notion of *abstract state* to guide systematic but efficient exploration of the fault injection space. The basic idea is to use system states to *group* injection points. A system's state can be represented by its variables and concrete values. This representation, however, is massive for large systems, and it would make almost all injection points appear in unique groups. For fault injection, we need a more high-level state representation, in which multiple injections likely yield similar effect.

LEGOLAS uses a simple yet novel static analysis that *automatically infers* abstract states from the target system code. The automation is feasible because developers usually leave clear hints in the code about abstract states: the system checks one or more state variables' concrete values in a branch to see if an important condition occurs; if so, it performs some significantly different action. An abstract state thus can indicate that a system enters a unique stage of service, *e.g.*, request parsing, snapshotting, and leader election.

Specifically, LEGOLAS first infers concrete state variables in a system. It then identifies code blocks that are control-dependent on some concrete state variable. It finally creates a mnemonic abstract state variable for each such block, which will be set when the program execution reaches that point.

Leveraging the inferred abstract states, LEGOLAS can efficiently explore the injection space. During testing, the injection hooks LEGOLAS instruments dispatch queries to the LEGOLAS controller. The controller checks the system's current abstract states and decides whether to grant an injection or not. Essentially, LEGOLAS enables *stateful* fault injection.

We design a stateful injection decision algorithm called *budgeted-state-round-robin* (bsrr). Other stateful policies are also feasible, and it is easy to add and switch policies in LEGOLAS. Compared to the straightforward *new-state-only* policy, *bsrr* is robust to tolerate potential inaccuracies in the abstract state analysis. It also reduces biases in injections.

We have built an end-to-end prototype for the LEGOLAS framework, including the static analyzer, fault injection controller, workload driver, and failure checkers.

We apply LEGOLAS to six large distributed systems: Kafka, ZooKeeper, HDFS, HBase, Cassandra, and Flink. LEGOLAS automatically instruments these systems and extracts abstract states without special tuning. We run fault injection experiments on these systems' recent releases. Using the *bsrr* algorithm, LEGOLAS finds 20 new bugs with a median time of 58 minutes. These bugs all cause partial service failure symptoms. We report the bugs to developers. Four reports are marked as critical, fourteen reports are marked as major, and two are marked as normal. Eleven reports have been explicitly confirmed by developers so far. We also compare LEGOLAS (*bsrr*) with state-of-the-art solutions and other policies. The best performing baseline is the *new-state-only* policy with LEGOLAS, which exposes eight bugs. The random injection policy only exposes three bugs in a median of 362 minutes.

In summary, this paper makes the following contributions:

- We propose an approach that uses program analysis to enable customized and fine-grained fault injection.
- We introduce a novel concept of abstract state and a method that automatically infers abstract states from a given system's code. We design a new decision algorithm that leverages the inferred abstracts to guide efficient fault injections for exposing complex bugs that cause partial service failures.
- We build a fault injection framework LEGOLAS and evaluate LEGOLAS on large distributed systems.

2 Overview of LEGOLAS

LEGOLAS is an end-to-end fault injection testing framework for large distributed systems. It aims to efficiently expose fault-induced bugs like the motivating examples.

Scope. Consider a distributed system $\mathcal{S}$ that consists of multiple processes P and provides a range of services R. One definition of a partial failure is that a subset of P are faulty (crash, Byzantine, or gray faults [23]), which may be tolerated and not affect the functionalities of $\mathcal{S}$.

LEGOLAS focuses on exposing partial failures with respect to services, where some $R_f \subset R$ fail to maintain their safety or liveness properties, while other services $R \setminus R_f$ behave as expected. In contrast, in a total failure, all services in R break. An intuitive strategy to uncover partial failures is thus to perturb each service based on their specifications, but this strategy can be difficult to apply with large concrete codebases.

Owing to the modular designs prevalent in large distributed systems, each service is typically implemented by a specific component made up of threads; each process π encompasses disjoint sets of components that provide different services for $\mathcal{S}$. For example, a leader process in ZooKeeper has dedicated request handlers, snapshot manager, quorum messenger, *etc.* If one thread fails, the recovery mechanisms will try to avoid interruptions to the corresponding service. Thus, LEGOLAS is designed to perturb each component (some thread) within π—instead of crashing π outright—by inducing faults to the instructions executed by the component. This allows for a deeper exploration of the potential partial failures in $\mathcal{S}$.

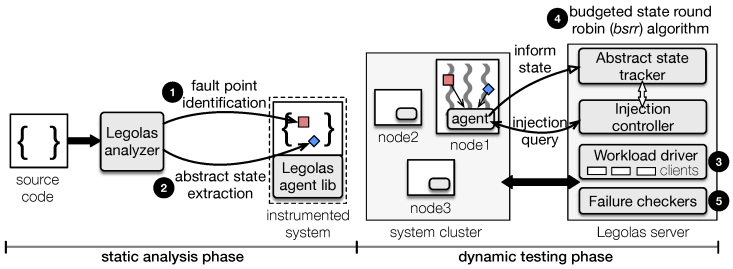


Figure 3: Fault injection workflow with LEGOLAS.

```
InjectionQuery query = new InjectionQuery(serverId,
      threadId, ..., invokedMethodSig, faultIds);
InjectionCommand command = stub.inject(query);
if (command.id == -1) return;  // no injection
/* simulate the decided fault */
...
+ LegolasAgent.inject("org.apache.zookeeper.server.DataTree",
        "serializeNode", 1115,"<org.apache.jute.OutputArchive:
         void writeRecord(...)>", 268, 0, 3);
  outputArchive.writeRecord(node, "node");
```

Figure 4: Injection hook instrumented for code in Figure 1.

Workflow. Figure 3 shows the workflow with LEGOLAS. It takes a system's code as an input, uses static analysis to identify faulty conditions unique to the system, and instruments injection hooks directly within the system (❶) to simulate subtle faults.

In addition, LEGOLAS runs a novel analysis that *automatically* extracts abstract states from the system code (❷). It identifies code locations that may represent an important change in the system's service status, and inserts mnemonic abstract state variables at these locations. LEGOLAS then links a thin *agent* library with the target system.

During testing, LEGOLAS starts a cluster of the instrumented system and runs the workload driver (❸). When a node reaches an injection hook, the embedded agent dispatches an RPC query to the LEGOLAS controller, which decides whether to grant the injection or not. If the agent receives a positive reply, it will simulate the fault inside the node directly.

Importantly, the LEGOLAS server tracks abstract states for each node. When a node enters a new abstract state, the LEGOLAS agent informs the state tracker in the server. The controller leverages the abstract state to make injection decisions. In particular, we design an algorithm called budgeted state round-robin (*bsrr*) (❹). To determine the injection outcome, LEGOLAS runs the failure checkers (❺).

3 Identify and Instrument Injection Points

A widely-used fault injection approach is to introduce node-level faults such as process crashes and network disconnections *externally* in the environment. This approach is suitable for exposing distributed protocol bugs or crash-recovery bugs. However, partial failures are often triggered by subtle faults in the implementations. Existing solutions that inject fine-grained faults focus on the boundaries between an application and libraries or services, and take an *interception* approach. For example, LFI [43] intercepts libc API such as `recv` and returns error codes to applications. Although their injection is more fine-grained (library APIs or service requests) than node-level faults, they miss *internal* errors in a system. Moreover, it is difficult for them to precisely simulate partial faults because they do not directly control the program execution.

To address these issues, LEGOLAS goes deeper—to the program statements inside a system—and takes an *instrumentation* approach. It uses static program analysis to deduce potential faulty conditions for each statement and add injection control points to directly simulate faults *in situ*.

Identify Faulty Conditions. The LEGOLAS analyzer locates each call instruction in the system and examines its invocation target to extract potential fault conditions. A straightforward way is to leverage the method signature. However, two challenges arise. First, a method may internally throw an exception that is not declared in the signature. Some languages also do not enforce or support exception specification in method signature. Second, due to polymorphism and interface, a call site of a method may be *impossible* to encounter an exception declared in the method signature. This is especially problematic with I/O related exceptions. Consider `dump(OutputStream out) throws IOException`, which is declared this way because the argument `out` is an abstract class with `IOException` in its methods' signatures. However, if a call site of `dump` passes a `ByteArrayOutputStream` as an argument, injecting an `IOException` causes an invalid scenario.

To handle the first issue, LEGOLAS inspects the method body and deduces the exceptions. However, it cannot simply collect the exceptions in the throw instructions. This is because the method may have an exception handler that catches the exception. LEGOLAS analyzes the error handlers in the function to determine if an exception may be (i) caught and handled; (ii) caught and re-thrown; (iii) uncaught. Only (ii) and (iii) are treated as the true method-level exceptions.

To address the issue of invalid injection, LEGOLAS designs an inter-procedural, context-sensitive analysis to check call instructions with potential `IOException`. It deduces whether the objects (argument, return, field, class) associated with a call site may come from definition points with known in-memory object types, and ignores the fault if so.

Besides exceptions, the fault condition could also be a delay. All operations could in theory experience some delay. In practice, mild delays are benign and developers need evidence to explain the delay. LEGOLAS by default only considers function calls that involve I/O as delay injection candidates.

Instrument Injection Hooks. For each program point with potential faulty conditions, LEGOLAS instruments an injection hook. LEGOLAS also emits a thin *agent* to link with the target system. At runtime, when an injection hook is reached, the agent creates a query to a controller (Figure 4), which includes the interposed operation, possible fault ids, node id, name and

id of the current thread. If the injection is granted, the agent looks up the fault id, which can be either a delay or some exception. For delay, the agent simply invokes the thread sleep function. We focus on worst-case situations and use 1 minute as the default delay (the `writeRecord` call in Figure 1 can hang for over 15 minutes under default Linux TCP settings). For exception, the agent constructs an exception instance and throws it before the injection hook returns.

Automatically creating an exception instance to throw is a non-trivial task. Some custom exception type includes complex arguments and compositions. LEGOLAS analyzes the constructor and recursively reduces complex arguments to primitive types. It then creates an exception instance by assigning the primitive fields with default values.

Where to add the injection hooks also requires careful considerations. The straightforward way is to instrument each call instruction that LEGOLAS analyzes to possibly encounter a fault. Suppose `foo()` is analyzed to possibly throw `MyError`, but that is only because `foo()` internally calls `bar()`, which can throw that error. If `MyError` is injected at the call sites of `foo()`, we need further explanation of why this exception occurs. For deep call chains, such injections make the exception reasoning difficult and may turn out to be invalid.

To address this issue, LEGOLAS instruments as deep as possible. It identifies faults that *originate* from a method through explicit `throw` statement. It then only instruments calls to either a method that has a non-empty list of such faults, or an external function. If the called method is from an interface or abstract class, LEGOLAS injects at the caller's call sites to handle potential invalid injections.

Benefits. LEGOLAS's approach eases fault simulation without requiring a special environment (e.g., a FUSE-based file system [13, 49]). It also gives precise control to simulate partial faults, *e.g.*, a partial disk failure that only affects a subset of file operations; only some RPCs within certain code region are delayed. It also supports simulating custom errors. While a custom error may be caused by some environment fault, it can be difficult to simulate them with external injection. For example, a method may throw an exception only when all three retries of a connection fail.

4 Abstract State Guided Fault Injection

A key challenge in fault injection for distributed systems is the enormous injection choices (Figure 2). Moreover, only few choices can expose bugs. This is because production distributed systems have extensive fault resilience mechanisms.

Insight. Our insight is that many fault injection attempts are unnecessary because they are testing the same or similar scenarios. Take a ZooKeeper code snippet in Figure 5 as an example. The `SyncRequestProcessor` component is responsible for synchronizing the requests to log files on disks. Suppose we are injecting faults on I/O operations. There are numerous injection points here, including operations inside the called

```
1  public class SyncRequestProcessor extends Thread {
2    public void run() {
3      int logCount = 0;                                   s0
4      while (true) {                                      s1
5        Request si = queuedRequests.take();
6        if (zks.getDB().append(si)) {
7          logCount++;
8          if (logCount > snapCount) {
9            if (snapThd != null && snapThd.isAlive()) {
10             LOG.warn("Too busy to snap, skipping");
11           } else {
12             (snapThd = new Thread(() -> {
13                 zks.takeSnapshot();                     s2
14             })).start();
15           }
16           logCount = 0;
17         }                                               s3
18       }
19       ...
20     }
21   }
22 }
```

Figure 5: The grayed areas are code regions containing I/O operations. The bug in Figure 1 occurs inside a call chain from line 13.

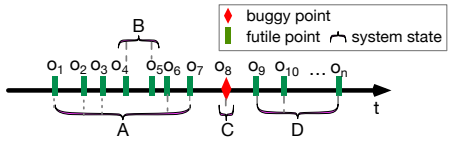


Figure 6: Group the injection points by the state they appear in.

functions. Line 6, which syncs requests to logs, gets executed at each loop iteration, while line 13 only occurs occasionally. With limited testing resources, we may only inject faults on operations inside line because of their frequent occurrences.

Idea. Inspecting the system state for each component can help us make better decisions. For the previous example, we could realize that the system enters a rare state (snapshotting) when it reaches line 13. Operations in this state could be of high interest. Our basic idea is thus to group the injection points based on the underlying system state (Figure 6). Grouping helps avoid being indiscriminate when making injection decisions. The injection points that lie in the same group of state are hypothesized to yield similar outcomes if injected, while the injection points in different groups may yield different outcomes.

However, we do not just focus on rare states, as defining them is subjective. Moreover, the presence of an injection point in a rare state does not imply a bug. Neither does an injection point in a common state guarantee the absence of bugs. For example, a bug may be exposed with a fault occurring inside the `append` call in Figure 5—which is in a common state—when it is executed for the fourth time.

We thus explore the injection space *systematically*. That is, if there were four chances, we try to explore injections in all four states, instead of spending them only in one state.

4.1 State Representation Choices

The next question is *how to define the system state for effective grouping?* Unlike distributed protocols that have specifications, determining the state representation for complex system implementation is not easy. The complete execution states—the program counter, stack traces, and memory snapshot—are

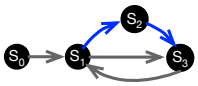


Figure 7: State machine with abstract states for Figure 5. Each state is an abstraction over the concrete state variable `logCount`.

obviously too excessive. A more reasonable representation is to use some key state variables (SV). A value change of these variables then could indicate the system is in a different state. This representation, however, can still be excessive.

Take Figure 5 as an example. If we treat the `logCount` as a state variable (SV), the value is incremented for n times, and each increment is counted as a new state. Using such a representation not only requires frequent state tracking, but also degrades the injection point grouping to be useless.

The key reason is that some values in a concrete state variable do not imply a significant change, at least for fault injection purposes. All the increment-by-one value transitions of `logCount` while `n<=snapCount` indicate the same information about the system, while only the transition of `n>snapCount` indicates something new (starting to snapshot).

Essentially we need a more high-level representation than concrete state variables (SV), which we define as **abstract state variable** (ASV). The intuition behind ASVs is that they represent different *stages of service* in a system. For the example in Figure 5, a natural way to define the abstract state is to divide the execution into four stages—S_0 to S_3. Figure 7 shows the corresponding abstract state machine. This simpler representation can recognize when the system starts to do snapshot (state S_2). In turn, they can effectively group the injection points to make fault injection efficient.

4.2 Infer Abstract State Variables

The LEGOLAS analyzer uses a simple yet novel method to automatically infer ASVs in a system. The feasibility of the automation is based on our insight that developers usually already encode sufficient hints about ASVs. In particular, developers checks one or more state variables (SV) in a branch, and if certain condition occurs, the system performs some action, *i.e.*, `if (func(state_var1, var2, ...)) { do_action1(); }`. From our inspection, in long-lived components, it is a common practice to utilize SV to designate different functionalities at different iterations. For example, the `QuorumPeer` component in ZooKeeper uses a static variable `state` to indicate the node status, which could be `LEADING`, `FOLLOWING`, *etc.* The `QuorumPeer` component then has a `while` loop that does a switch case on this SV to select different functionalities over time.

LEGOLAS first locates all the task-unit classes in the system. These classes are generally threads or workers, such as classes that extend `Thread` or `Runnable` in Java. The analyzer then runs ASV inference on each task-unit class.

Algorithm 1 lists the core algorithm. It starts by inferring the SVs in the code (Line 2). `InferCSV` simply treats all *non-static*, *non-constant* fields defined in a task unit to be SVs.

LEGOLAS then analyzes the main task method of the task unit class, such as the `run()` method of a `Thread`. It finds the

Algorithm 1: Infer abstract state variables

```
1  Function InferASV(task_class):
2      csv_list ← InferCSV(task_class);
3      task_method ← getTaskMethod(task_class);
4      dep_graph ← buildDependence(task_method, csv_list);
5      asv_locations ← [task_method.body().getFirst()];
6      Process(task_method.body(), dep_graph, false);
7  Function Process(instructions, dep_graph, flag):
8      inst ← instructions.begin();
9      hasAction ← false;
10     while inst ≠ instructions.end() do
11         if isBranch(inst) then
12             <cond, blocks, next> ← parseBranch(inst);
13             if dep_graph.contains(cond) then
14                 for block ← blocks do
15                     Process(block.body(), dep_graph, true);
16                 end
17             end
18             inst ← next;
19         else
20             hasAction ← hasAction | isAction(inst);
21             inst ← inst.next();
22         end
23     end
24     if hasAction and flag then
25         asv_locations.add(instructions.begin());
```

basic blocks in the task method that are control dependent on some SV and treats each such basic block as a new ASV.

Specifically, the analyzer iterates through instructions in the task method. Upon a branch instruction, it checks if the branch condition is dependent on some SV (Line 13). This check considers not only direct usage of SV but also indirect data dependence, *i.e.*, a branch condition involving a local variable that gets its value from an SV. Accordingly, the analyzer builds a data dependence graph of the SVs (Line 4). The algorithm then recursively processes the basic blocks control dependent on this branch instruction (Line 15). A system should perform non-trivial actions in an abstract state. Thus, we check if the basic block contains at least one function invocation or an operation that could change a state variable (Line 20).

Once the proper basic blocks are located, the analyzer assigns indexes for them, asv_0, ..., asv_n. The indexes are local to the task class. For each inferred ASV, the analyzer instruments a call to the LEGOLAS agent. At runtime, the agent notifies the LEGOLAS state tracker of the asv_i that is entered, along with the node id, the name and id of the current task.

Note that our ASV is *not* equivalent to conventional control-flow path. We make program paths collapse into more meaningful ones (service stages) that guide fault injection.

Example. Figure 8 shows the ASVs LEGOLAS infers and inserts for the code in Figure 5. The inferred ASV is slightly different from S_2 in the simplified snippet in Figure 7. This is because the `logCount` is a local variable, thus the LEGOLAS analyzer does not treat it as an SV. Another variable `snapThd` is

```
1  LegolasAgent.inform(identityHashCode, ..., 0);          asv0
2  while (true) {
3    Request si = queuedRequests.take();
4    if (request == requestOfDeath) break;                 asv1
5    LegolasAgent.inform(identityHashCode, ..., 1);
6    if (zks.getDB().append(si)) {
7      logCount++;
8      if (logCount > snapCount) {                         asv2
9        if (snapThd != null && snapThd.isAlive()) {
10         LegolasAgent.inform(identityHashCode, ..., 2);
11         LOG.warn("Too busy to snap, skipping");         asv3
12       } else {
13         LegolasAgent.inform(identityHashCode, ..., 3);
14         (snapThd = new Thread(...
15       }
16       logCount = 0;
17 ...
```

Figure 8: The ASVs Legolas infers for Figure 5.

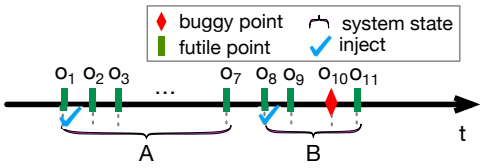


Figure 9: The buggy point may not be the first request in a state.

a non-static field of `SynchRequestProcessor`. Legolas treats it as an SV and infers the asv_3 that represents the snapshot stage. This result is in fact more accurate than using `logCount`, because it infers an additional state (asv_2)—a previous snapshot is ongoing while the snapshot threshold is reached.

Alternative. We also explored other ASV inference methods. For example, we observe that although some function only uses local variables, it can still represent an important system service stage, *e.g.*, handling an event. Defining an ASV at the function entry can be useful. We chose our above inference method for its simplicity. As Section 7 later show, it is general enough to apply on all the popular distributed systems we evaluate and achieve significant performance. It is also feasible to extend Algorithm 1 and analyze the functions called in the task method to extract more thorough stages. However, only analyzing the main task method already provides a good generalization to capture key stages in a component that match the system modularity and design documentations.

4.3 Injection Decision Algorithm

With the inferred abstract states, Legolas enables *stateful* decision policies for efficient fault injection. When the controller receives an injection request from the Legolas agent, the controller checks which abstract state the target system is in at the time of the injection request to make a decision.

A straightforward stateful policy is to grant an injection request only if the system is in a new state, which we call a *new-state-only* policy. While this policy matches the intuition that complex bugs are often only triggered when the system enters an unusual condition, it has several drawbacks. As Figure 9 shows, there can be multiple injection requests from one state, and a buggy point may not be the first request. Indeed, for the ZooKeeper example, even though the bug only appears in the snapshot state, the snapshot function performs several write operations before it reaches the buggy point. This policy also

Algorithm 2: Budgeted state round robin (bsrr) policy

```
   Global Vars: Queue<State> rrl, Map<State,Info> visit
   /* invoked at start of a fault injection trial                    */
1  Function setupNewTrial():
2    resetIfAllUsed(rrl, visit);
3    while !rrl.empty() do
4      s ← rrl.pop();
5      info ← vist.get(s);
6      if info = nil or info.budget > 0 then
7        rrl.append(s);
8        break;
9      end
10   end
11   while !rrl.empty() and visit.get(rrl.front()).budget = 0 do
12     rrl.pop();
13   end
14   updateProbabilities(visit);
   /* invoked for each injection request                             */
15 Function shouldInject(request):
16   curr ← getCurrentState(request);
17   if not visit.contains(curr) then
18     visit.put(curr, new Info());
19     rrl.append(curr);
20   end
21   info ← visit.get(curr);
22   info.occur ← info.occur + 1;
23   if rrl.front() ≠ curr then return false;
24   if info.budget > 0 and rand() < info.prob then
25     info.budget ← info.budget - 1;
26     return true;
27   end
28   return false;
```

relies on the abstract state analysis to be precise. If the static analysis misses instrumenting an ASV close to the buggy point, the buggy point will likely be treated as in a seen state. In addition, the system can take a long time to enter a new state. If we only wait for new states, we may not inject anything when the workload finishes and waste an experiment trial.

To address these drawbacks, we design a **budgeted-state-round-robin** (*bsrr*) policy. Algorithm 2 lists its algorithm.

The algorithm allocates a budget (default 5) for each state to be potentially granted injection more than once. This relaxes the stringent new state requirement. After *all* states use up their budgets, the budgets are reset (Line 2).

It keeps a round-robin list of the abstract state tuples (`rrl` in Algorithm 2). Suppose the list has $s_1, s_2, \ldots, s_n$. The algorithm intends to grant injection requests from state s_1 for the first trial, grant requests from s_2 for the second trial, and so on. In other words, it **focuses on** one state in one trial.

Specifically, before each trial, *bsrr* rotates the state focused in the last trial to the end of the round-robin list (Line 7). If a state's budget is used up, it is removed from the list.

The round-robin design addresses the imbalanced injections problem illustrated by Figure 10: in all three experiment trials,

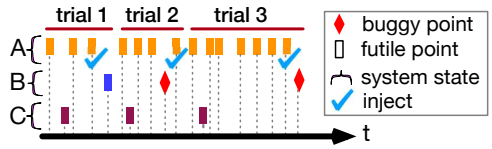


Figure 10: All injection chances are given to operations in state A.

we would inject operations in the frequent state A (within its budget), while no operation in state B or C is injected.

The algorithm also applies randomization to allow exploring different choices when a state has multiple injection requests. The probability p should be set properly. If it is too large, we would always grant the first (few) requests in a state. If it is too small, we may waste the injection trial.

We calculate p for each state tuple based on c—the times this state appears in injection requests. We set $p = 1 - e^{ln(0.01)/(c+1)}$. This formula's rationale is that we want to (i) grant at least one injection among the c requests to avoid wasting the trial; (ii) let the injection occur neither too early nor too late among the c requests. The probability that all c injections are *not* granted is $(1-p)^c$. Because of (i), this probability should be close to 0. Suppose $(1-p)^c = \epsilon$. With (ii), ϵ should not be too small; otherwise p is too close to 1 and the injection would be too early. We set ϵ to 0.01, and solve this equation, which gives $p = 1 - e^{ln(0.01)/c}$. We use $1 - e^{ln(0.01)/(c+1)}$ instead to handle corner cases of $c = 2$ or 3.

The *bsrr* policy is adaptive to leverage information from prior trials. Upon each injection request, the algorithm dynamically updates the parameter c (Line 22). Before a trial starts, it uses the occurrences from previous trials to re-calculate the probabilities for the visited states (Line 14). Similarly, *bsrr* updates the round-robin list dynamically (initially empty). If a state in an injection request is not visited before, it is added to the round-robin list for later exploration (Line 19).

5 Testing Experiment

LEGOLAS starts fault injection testing after the analyzer finishes instrumentations (§3, § 4.2) on the target system. LEGOLAS uses a client-server architecture to manage the testing (Figure 3), where the LEGOLAS agents embedded in the system send RPC requests to a LEGOLAS server that is composed of an abstract state tracker, injection controller, workload driver, and failure checkers. The testing proceeds in continuous trials.

5.1 Injection Trial

In each trial, LEGOLAS starts a cluster of the target system and then invokes the workload driver (Section 5.2). The trial ends when the workload finishes (successfully or not).

To support stateful injection decision algorithm (Section 4.3), while the target system is restarted in each trial, the LEGOLAS server will live throughout the experiment. Thus, it carries information such as the round-robin list across trials.

When a node enters a new abstract state, the LEGOLAS agent notifies the state tracker, which maintains one Abstract State Machine (ASM) per task-unit (usually a thread) for each system node. Each state update event is a tuple of node id, ASM-name (class name), ASM-instance (class instance), and ASV. The tracker records the current ASV and transitions for each ASM.

When a node reaches an injection hook, the LEGOLAS agent sends an injection query to the controller, which is a tuple of node id, ASM-instance, operation, and fault ids. The controller runs the *bsrr* algorithm to decide whether to grant the injection or not. Notice, however, that the injection query does not carry the ASV information. The controller obtains the associated ASV by indexing the node id and ASM-instance from the injection query to the ASM map in the state tracker.

LEGOLAS by default grants at most one injection in one trial. Allowing multiple injections in a trial only requires a simple change. While it seems more attractive to keep injecting faults in a trial, that choice has several disadvantages. Although distributed systems are designed to be fault-tolerant, each system has a limited tolerance level. If we keep injecting in a single run, the system may likely break as expected. Moreover, each injected fault can alter the system state and leave side effects. With continuously injected faults, it becomes very difficult to judge the system behavior and tell which fault is responsible for the symptom. Also importantly, if injections are performed non-stop, we may go deeper in an execution path, but we will not inject earlier, skipped operations or explore other paths, sacrificing completeness.

5.2 Workload Driver

LEGOLAS uses a workload driver to exercise the target system. For each system, we select several existing, representative test cases to create the workload driver.

To better suit our objectives, we make a few adaptations to the test cases. First, the driver creates multiple clients and each client is typically dedicated to interacting with one node. In this way, LEGOLAS can observe the status of every system node without mixing signals. Second, the driver divides workloads into phases, *e.g.*, create, read and write. Only when the current workload phase finishes successfully will the next phase starts. This is to localize the failed system functionalities and avoid unnecessary errors that mislead the results. Third, in one workload phase, each client is expected to send a series of requests and will report its progress to LEGOLAS server after one request completes. The LEGOLAS server also tracks when a client timeouts or encounters exceptions. This allows LEGOLAS to more accurately assess the failure impact. Lastly, we use a small workload scale, such that a trial does not take long and LEGOLAS can explore more trials.

5.3 Failure Checkers

To determine the testing results, LEGOLAS currently provides three failure checkers:

- *Crash checker*: it monitors the OS signals to check if a system node crashes, aborts, or exits with a non-zero status.
- *Client checker*: it approximates Panorama [22], a state-of-the-art gray failure detector, to identify whether differential observability exists. In particular, it marks a trial suspicious

System	Release	SLOC	Type
ZooKeeper	3.6.2	95K	Coordination service
HDFS	3.2.2	689K	Distributed file system
Kafka	2.8.0	322K	Event streaming system
HBase	2.4.2	728K	Distributed database
Cassandra	3.11.10	210K	Distributed database
Flink	1.14.0	78K	Stateful streaming system

Table 1: Evaluated distributed systems in latest releases.

if (1) a fault is injected in one node, but *only* another node's clients report errors; (2) the system's own detector indicates a node is active, but the node's clients report errors; (3) only a subset of clients fail to complete their workloads.

- *Log checker*: it scans the logs of each system node to identify whether there are log entries at warning or error level.

As a testing framework, LEGOLAS is extensible to add more checkers. For example, users can add checkers about inconsistency [38], semantic failures [37], or transaction isolation [30].

When a fault is injected, LEGOLAS records the stack trace of the originating operation. With the stack traces, LEGOLAS further clusters the trials by stack trace similarities so that similar symptoms are investigated together.

6 Implementation

We implement LEGOLAS with around 7,500 SLOC for the core components, and 100–300 SLOC for the workload driver for each evaluated target system. The LEGOLAS static analyzer is built on top of the Soot [52] framework, so it supports systems in JVM bytecode, including Java and Scala. Its core analysis algorithms are based on universal programming language constructs such as thread classes, member variables, branches, and conditionals. Thus, they are language-independent. The controller and orchestrator are designed in a client-server architecture using Java RMI for local RPCs.

7 Evaluation

Our evaluation aims to answer several key questions: (1) does LEGOLAS work on large distributed systems? (2) can LEGOLAS expose new complex fault-triggered bugs? (3) does the abstract states LEGOLAS infers significantly help the fault injection efficacy? (4) how does LEGOLAS using the *bsrr* policy compare to other policies and state-of-the-art solutions?

Evaluated Systems. We evaluate LEGOLAS on the recent stable releases of six popular, large-scale distributed systems (Table 7). These systems have different functionalities, written with various programming paradigms. Our appendix lists the workloads we use in the testing.

Measure. For a testing tool, its ability to exposes new bugs is a key measure. Our evaluation thus centers around this aspect (Section 7.2). Since our target systems are widely deployed in production and have been extensively tested for years, finding new bugs in their latest releases is not an easy task.

Additionally, we apply LEGOLAS on a number of randomly sampled known bugs in old releases of the systems (Section 7.5), including the running example in Figure 1.

System	Class	ASM	ASV				Static. Injected	
			Total	Mean	Min	Max	Methods	Points
ZK	708	36	226	6	1	31	484	1947
HDFS	4636	104	390	4	1	16	2127	3913
Kafka	5829	51	220	4	1	15	343	754
HBase	10462	96	312	3	1	17	5874	11051
CSD	4636	104	390	4	1	18	2127	3913
Flink	4852	48	110	2	1	6	997	2299

Table 2: Statistics of applying LEGOLAS static analyzer. *Class*: analyzed Java classes; *ASM*: classes analyzed as abstract state machines; Mean, min, and max of *ASV* are abstract state variables in each ASM.

Setup. We run experiments on servers with a 20-core 2.20GHz CPU and 64 GB memory running Ubuntu 18.04.

Each system's fault injection experiment consists of 2000 trials. A trial's time is dominated by the system startup and workload execution. The trials' durations vary depending on how the system reacts to the injected faults and whether it fails early or not. The experiment time for the six systems is 2.7 hrs, 10.7 hrs, 23.5 hrs, 8.4 hrs, 54.6 hrs, and 26.5 hrs respectively.

We use the *bsrr* policy (Section 4.3), and set the state budget to the default value of 5 *for all systems*.

Due to the large scale of experiments and time constraints, our testing focuses on the following faults: (1) I/O related exceptions, e.g., `IOException`, `ClosedChannelException`; (2) custom exception types that inherit from `IOException`; (3) delays to function calls that involve disk or network I/O. We run two separate experiments (exception and delay) for each system. We observe that `IOException` is widely used to represent more than hardware issues. For example, developers add `throw new IOException` statements for situations such as "unreasonable length", "missing signature", "current epoch is less than accepted epoch", and "snapshot already exists", which are difficult to simulate by external fault injection tools.

7.1 Injection Points and Abstract States

LEGOLAS successfully applies on the six systems. Besides scaffolding information (*e.g.*, class paths, task class types), the analyzer does not require additional input for a new system. The injection policies are also not specially tuned.

As Table 2 shows, the number of task classes (ASMs) LEGOLAS extracts is much smaller compared to the number of classes in the system. It also varies across different systems due to their design choices. For example, ZooKeeper has a relatively small number, while Cassandra has over 100; yet, ZooKeeper has the largest ASVs per ASM. This is because ZooKeeper uses long-running threads, while Cassandra adopts an event-driven architecture that uses many short-lived runnables. The mean ASVs per ASM is moderate, because currently LEGOLAS only analyzes the direct ASM classes and task entry methods.

We further manually inspect the 36 ASMs and 226 ASVs LEGOLAS generates for ZooKeeper. We find that they can represent the state transitions in ZooKeeper at different granularities. For example, in the `QuorumPeer` ASM, the ASVs exactly match the states of a node in the quorum: for the states such as

Looking, Observing, Leading, and Following, there exists exactly one ASV for each. In the `SyncRequestProcessor` ASM, the ASVs capture local state transitions: there is one ASV in which the transaction log is written, one when flushing in-memory log, and one when the snapshot is taken.

Dynamically, 24 ASMs and 76 ASVs are traversed during our testing. For the 150 non-traversed ASVs, 46 are from the 12 unutilized ASMs. We check the rest 104 ASVs to see if they encode meaningful execution states. In 20 of them, the code blocks have at least one I/O operation. In another 58 ASVs, they are in the exception handlers or shutdown blocks. In 8 ASVs, they only print a log. The remaining 18 ASVs do not contain significant operations and are introduced due to code optimizations by Soot. For the 20 unvisited ASVs that contain I/O operations, we tried to enlarge our workload (more reads/writes, reconfiguration), which did not help. However, our small workload achieves decent utilization of the ASVs.

7.2 Finding New Bugs

Our overall experience in the fault injection experiments is that the evaluated systems are robust to tolerate or at least cleanly abort various faults in most places. Take ZooKeeper as an example. If a thread is doing a socket write and a network delay is injected, this thread will get stuck. In general, ZooKeeper can handle the fault correctly even though this thread hangs. For example, if the `LearnerHandler` thread hangs in this way, the `QuorumPeer` is able to confirm the stale `PING` state and abandon the problematic `QuorumPeer`.

Despite the robustness, LEGOLAS finds new bugs in *all* tested systems. It finds 20 unique bugs (Table 8 in Appendix). All bugs are non-trivial and require domain knowledge to understand, such as mishandling of errors, design flaws, and synchronization issues. They all trigger partial failure symptoms, such as some requests get stuck while others succeed.

We reported the bugs to developers. Four reports are marked as critical, fourteen as major, and two as normal. Eleven reports have been explicitly confirmed by developers so far. Our bug reports generate substantial discussions with developers, with a median of 21 comments and a maximum of 42 comments. Our reports to ZooKeeper inspired the developers to adopt fault injection testing practice.

Case Studies. *HDFS-15925* In one trial, LEGOLAS injects an `IOException` in the `BlockReceiver` module while one datanode is forwarding the data blocks to a mirror (another datanode). One client gets stuck without any error log, and its workload progress is partial (1/5), while other clients finish the workload (5/5). After investigation, we find that normally the datanode in such a situation will inform the client of this error state *immediately*. Then the client will resend the blocks. This process would be fast. Through code analysis, we find the root cause is a complex timing bug. In particular, when the datanode encounters the `IOException` it sets the `mirrorError` flag (Figure 11). However, a concurrency condition exists in which the `mirrorError` flag set could be shortly

```
class BlockReceiver {
  private int receivePacket() {
    if (mirrorOut != null && !mirrorError) {
      try {                          IOException injected inside
        ...                          ↘
        packetReceiver.mirrorPacketTo(mirrorOut);
        ...
      } catch (IOException e) {      → set flag mirrorError
        handleMirrorOutError(e);
      }
    }
    return lastPacketInBlock?-1:len;
  }
}
class PacketResponder {
  public void run() {
    while (isRunning() && !lastPacketInBlock) {
      PipelineAck ack = new PipelineAck();
      try {
        if (type != LAST_IN_PL && !mirrorError) {
          ack.readFields(downstreamIn);
        }                            ← gets stuck
      } catch (IOException ioe) {
        ...
      }
    }
  }
}
```

Figure 11: A timing bug that causes the packet responder to get blocked when the datanode encounters an IOException.

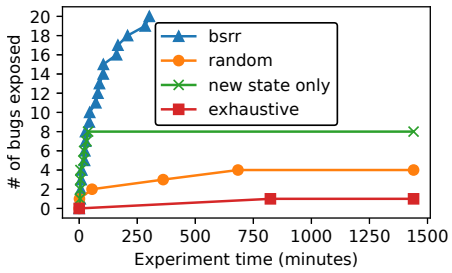


Figure 12: Efficacy of decision policies in LEGOLAS on detecting new bugs. *bsrr*: our budgeted-state-round-robin algorithm.

after the `PacketResponder` thread checks this flag, causing `PacketResponder` to not notice this status and get blocked, and the `ACK` packet will not be sent by the mirror datanode.

LEGOLAS exposes the bug five times in the experiment, with the first time in trial 124 at around 43 minutes.

HDFS-15869 The HDFS namenode uses the `EditLog` to maintain a transaction log of the namespace modifications. In one trial, LEGOLAS injects a delay to a remote write in the `FSEditLogAsync` thread. The injection occurs when the thread sends a response to the client and other servers, after it commits a transaction. This causes the whole `FSEditLogAsync` to be unable to proceed. The critical `logSync` function cannot be executed for incoming transactions. This is undesirable because `FSEditLogAsync`'s key feature is asynchronous edit logging that is supposed to tolerate slow I/O.

7.3 Impact of Abstract States and BSRR

This paper's thesis is that our inferred abstract states can enable efficient fault injection. Section 7.2 shows that LEGOLAS finds complex new bugs with our *bsrr* algorithm. To further validate our thesis, we compare the *bsrr* algorithm with alternative decision policies on the 20 new bugs.

	Detected Bugs	Median Detection Time
FATE	1	1057.9 minutes
CrashTuner	4	20.4 minutes
CORDS	0	N/A

Table 3: Effectiveness of existing work on the 20 new bugs.

For each policy, we run a 2000-trial experiment and measure the number of bugs it exposes, as well as the time it takes to expose the bugs. The latter is an important metric. If a solution cannot expose a bug within a reasonable time, developers in practice likely will not use it even if the solution in theory may expose the bug after a long time.

Figure 12 shows the result. The exhaustive policy only exposes one bug. The random policy only exposes three bugs. It is also inefficient. It takes a median of 208 minutes and a max of 994 minutes to find the three bugs. After finding the third bug, it fails to find more bugs in 24 hours.

The *new-state-only* policy (§ 4.3) exposes eight bugs in a median of 11.4 minutes. It is the best among the baseline policies, showing the advantages of our inferred abstract states.

The *bsrr* significantly outperforms all alternatives. It exposes 20 bugs in a median of 58.2 minutes (min 4.0 minutes, max 302.0 minutes). Compared to *new-state-only*, it is more robust in leveraging the imperfectly inferred abstract states, exposing much more bugs while achieving good efficiency.

The inferred ASVs help improve the fault injection efficacy in two ways. First, they can capture the unusual system service stages, allowing LEGOLAS to inject in places that are not well tested and buggy. For example, in HDFS-15957, one of the ASVs LEGOLAS infers for `FSEditLogAsync` represents the state of sending a response to client while `FSEditLogAsync` is committing transactions. The ASVs for the running ZooKeeper example also belong to this category. Second, the inferred ASVs can help make progress in skipping uninteresting injection points. For example, in HDFS-15925, the injection (Figure 11) occurs inside the `DataXceiver` thread in the datanode. The relevant ASV that LEGOLAS infers corresponds to the `processOp` stage. Although this ASV is just the main stage of the thread, the other ASVs LEGOLAS infers in other threads help avoid wasting too much time in injecting in other places.

7.4 Comparing with Other Solutions

Research Baselines. We compare LEGOLAS with three state-of-the-art fault injection research projects, FATE [15] CrashTuner [40] and CORDS [13]. FATE tests *multiple failures* by using a concept of *failure IDs* to efficiently enumerate the combinations of failures. CrashTuner uses *meta-info variable accesses* to decide the timing of injecting node crashes for exposing crash recovery bugs. CORDS uses a FUSE file system to inject a single corruption or read/write error to one file-system block at a time, and enumerate all possible faults.

The first two works focus on node-level faults, making them not directly comparable to LEGOLAS. We apply their key ideas to attempt meaningful comparisons. We define the *failure IDs* as described in the FATE paper and implement a policy in LEGOLAS to grant an injection request when its associated failure ID has not been visited. For CrashTuner, because its analyzer component is not available, we re-implement its static analysis to identify all meta-info variable accesses and assign each access point a global ID. We instrument each access point to record the accesses at runtime. Then we grant an injection request when some meta-info variable access occurs within the past 5 ms and the access ID has not been seen. The latter is needed because a system may access the meta-info variable in a deterministic order, leading to only one injection being always granted if the access ID is not checked.

For CORDS, we utilize similar procedures as described in the paper to enumerate file system level errors on the requests to FUSE. For the experiment, we use the same workloads as in LEGOLAS but use the injection algorithm in CORDS.

Table 3 shows the result. FATE only detects one of the 20 new bugs in 1057.9 minutes. CrashTuner only detects four bugs. LEGOLAS significantly outperforms both solutions. CORDS does not detect any of the 20 new bugs despite enumerating all of its injection choices during the experiment. Although CORDS is a fine-grained fault injection tool, its fault scope is limited. It only injects corruption or error of a file block. Only 2 of the 20 bugs' root causes are related with that. For the two cases, they require a transient error and special timing, while CORDS injects persistent corruption or error that more likely leads to a total failure (node crash).

Popular Tool Baselines. We further compare LEGOLAS with three fault injection tools that are popular among developers: `CharybdeFS` [49] (a fault-injection filesystem), `tcconfig` [19] (a network fault injection tool based on Linux Traffic Control), and `byte-monkey` [53]. Byte-monkey is closer to LEGOLAS in that it also performs bytecode-level fault injection.

These tools rely on user-provided parameters to configure the injection, such as the packet loss rate and probability of returning an error code. Settings that are too large or too small produce meaningless results. We choose one moderate setting and one mild setting for each tool. We exercise the target systems using the same workloads from LEGOLAS.

Most injections lead to either a high percentage of successful trials or a high percentage of early exits (shown in appendix). For the small percentage of *partial_progress* injection trials, the failed client requests either happen directly because of the injected fault (*e.g.*, the server logs that it is unable to read data from client) or the system is in the middle of fault handling and successfully recovers. We verify that none of these trials expose any of the 20 new bugs LEGOLAS finds. We also vary the parameters, but the conclusions remain the same.

In the LEGOLAS decision policy comparison experiments (§ 7.3), its random policy exposes three bugs. In comparison, the evaluated popular tools do not expose any bugs with their random strategies. The discrepancies are due to the probability factor and the fact that LEGOLAS's in-situ injection mechanism has more precise control—it instruments operations that are possible to throw `IOException` (or its subtype) errors, which

System	Bug Id (Exposure Time)
ZooKeeper	ZK-2029 (15.4 min), ZK-2201 (30.6 min), ZK-2247 (52.1 min), ZK-2325 (2.6 min), ZK-2982 (18.5 min)
Cassandra	CA-6364 (10.0 min), CA-6415 (330.6 min), CA-8485 (25.3 min), CA-13833 (86.7 min)
HDFS	HDFS-11608 (29.2 min), HDFS-12157 (39.9 min)

Table 4: LEGOLAS exposes *known* bugs in old releases.

ZooKeeper	HDFS	Kafka	Cassandra	HBase	Flink
8.9 s	31.6 s	36.9 s	20.9 s	77.6 s	63.9 s

Table 5: Time of static analysis and instrumentation.

may be caused by complex environment errors. For instance, while `tcconfig` injects packet loss, it fails to trigger errors for function calls that internally throw `IOException` only upon a sequence of network errors. Also, its injection has a low chance of affecting only a special subset of operations.

7.5 Exposing Known Bugs

Besides finding new bugs, we further evaluate LEGOLAS's capabilities on exposing known partial failure bugs. We sample 11 *real-world* partial-failure issues from older system releases. In particular, we first use keywords matching on Jira to collect all cases whose root cause is related to I/O exception or delay. Then we randomly sample 50 of them, and select all the cases that satisfy our definition of partial failures and require one fault, which result in 11 cases. As Table 4 shows, LEGOLAS can relatively quickly reproduce these known bugs.

7.6 Performance

We measure the fault injection trial duration for LEGOLAS. Figure 13 shows the results. Although LEGOLAS tracks abstract states and dispatches the fault injection queries through RPCs, the injection duration is still acceptable, with a maximum of around 70 seconds. LEGOLAS uses local RPCs based on Java RMIs. Our microbenchmark shows the RMI latency is between 10 μs–50 μs. The *bsrr* policy function for one injection request has a median latency of 3 ms.

Table 5 shows the performance of the LEGOLAS static analysis and instrumentation. The analysis is fast, with the longest time being (73 s) in analyzing HBase.

7.7 Effort and False Positive

The static analysis and instrumentation steps in LEGOLAS are fully automated. The fault injection experiment does not require manual tuning. Our workload drivers are adapted from existing test cases and re-usable across versions. The main effort in using LEGOLAS is to confirm a bug after testing. However, this is common for testing tools, not a unique requirement of LEGOLAS. For the new bug finding exercise in Section 7.2, it roughly takes a graduate student author one to two weeks per system to examine the testing results, read and understand the relevant source code, and confirm where the bug is.

A false positive occurs if the fault being injected turns out to be impossible in reality. We observe such false positives

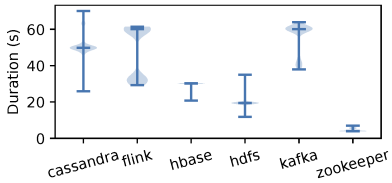


Figure 13: Distribution of one fault injection trial duration.

	ZooKeeper	HDFS	Kafka	Cassandra	HBase	Flink
w/ i.i.a	0	0	0	0	0	0
w/o i.i.a	45 (6)	20 (9)	0	894 (10)	86 (10)	0

Table 6: Number of trials that have invalid injections, with and without the invalid injection analysis in Section 3. The numbers in parentheses are the unique locations of these invalid injections.

in initial development of LEGOLAS and they are caused by the same problem: it injects an `IOException` to a function that declares `IOException` in its signature but cannot possibly have I/O errors. For example, we inject an `IOException` to the `writeBoolean` call inside the Cassandra `serialize` method. This injection triggers a buggy symptom. However, this method uses a memory buffer for the `writeBoolean` call.

As described in Section 3, we introduce an inter-procedural, context-sensitive analysis to eliminate such false injections. Table 6 shows the number of false injection trials for each system with and without this analysis. The result shows that the analysis successfully eliminates all of these invalid injections.

Another potential source of false injection is the automatic exception instance creation. Our method may create some invalid exception instances. However, we did not observe such false positives in our experiments.

Our failure checkers (Section 5.3) can make mistakes. When they mark a trial as being suspicious, it may turn out not to be a bug. For example, in ZooKeeper, the write requests to followers are forwarded to the leader. When the client checker flags a trial where a fault is injected in node 1 and the clients connected to node 2 experience write timeouts, it is possible that the node 1 is the leader and it is temporarily unavailable because of the delay injection. Developers may consider this to be a false positive, but the false positive is from the checker rather than the fault injection—the injected fault is legitimate. Our current checkers are basic. From our experience, to enhance checkers with more comprehensive bug rules, the knowledge of system-specific protocols and modularity practices must be exploited. After the enhancement, developers may still disagree on the definition of a bug. However, LEGOLAS's core contribution is on fault injection methodology.

Our bug confirmation process is as follows. We first utilize the basic checkers to highlight suspicious trials. For each suspicious trial, we analyze how the fault propagates its effect from one component to another and leads to the suspicious symptom. We then diagnose whether that is a bug and check if the design documentation describes the expected behavior. Finally, we will report it to developers to discuss with them.

8 Discussion and Limitations

Our ASV inference currently only analyzes states inside the task classes and the task entry functions. Thus, the inferred ASVs for each task are relatively coarse-grained. The analysis can be extended to other task functions and states that are passed to other classes through function calls, which will extract more detailed ASVs. However, it is not always the more detailed the better. A large system typically has tens to hundreds of task classes, so too fine-grained ASVs can lose the benefits of effectively grouping injection requests.

Our workload drivers use workloads on a small scale to exercise the target system. More workloads can be added to the LEGOLAS workload drivers, which is not a difficult task and would allow LEGOLAS to expose fault-induced bugs that require large workloads (*e.g.*, performance bugs).

LEGOLAS injects a single fault in one fault injection trial. It would miss bugs triggered by multiple faults. Supporting injection of multiple faults in LEGOLAS only requires a simple change. However, the decision algorithm and failure checkers would likely require significant changes. Indeed, we tried enabling multi-fault injection for the ZooKeeper experiment, but it only improved the efficiency for one bug.

LEGOLAS does not explicitly control non-determinism in the target system, such as thread schedules, which is the focus of concurrency testing tools. Thus, while LEGOLAS can expose a concurrency bug, it may not expose it reliably or efficiently. LEGOLAS can be combined with concurrency testing tools.

9 Related Work

Fault Injection. Early fault injection work targets standalone software. Faults are injected into hardware, simulated environment [20], or libraries (LFI [43]). Fault injection testing becomes popular in distributed systems with much research [2–4, 7, 13, 15, 24, 26, 27, 40, 45]. Many inject coarse-grained faults externally such as node crashes to expose protocol or crash recovery bugs. The injection is done randomly, or exhaustively, or based on user specifications. Several solutions proposed more advanced techniques. For example, LDFI [3] leverages data lineage to inject crashes or network faults if the faults could prevent correct outcomes; CrashTuner [40] injects crashes when meta-info variables are accessed.

LEGOLAS focuses on complex partial failure bugs. It uses an instrumentation approach to inject system-specific, instruction-level faults within a target system. It designs a novel static analysis method that automatically infers abstract states from distributed system code. Its decision algorithm leverages the abstract states to efficiently explore the fault space.

Recent fault injection research addresses other applications, such as multi-threaded programs [32], cluster-management controllers [51], microservices [44, 57], and REST applications [8]. LEGOLAS is orthogonal to these efforts. It targets large-scale distributed systems and aims to expose partial failure bugs triggered by exceptions or delays in the operations of a component within a distributed system node.

Model Checking. Model checking enumerates the possible interleaving of non-deterministic events such as messages. It has been applied to distributed systems [18, 29, 31, 50, 54]. Distributed system model checkers (dmcks) including MODIST [54], SAMC [31], and FlyMC [42] also explore the interleaving of crash/reboot failure events. LEGOLAS shares high-level similarity with these solutions in that it systematically explores the fault injection space. However, LEGOLAS is a complementary effort. Existing dmcks target protocol bugs caused by complex interleaving of node-level events, while LEGOLAS targets implementation-level bugs triggered by diverse faults in fine-grained program instructions. LEGOLAS can leverage a dmck to drive the target system into unexplored states, allowing LEGOLAS to try more injections.

Distributed Concurrency Bug Detection. Several projects [33, 35, 39, 56] aim to detect concurrency bugs in distributed systems. FCatch [35] applies happens-before analysis on correct execution traces to identify unprotected conflicting operations. LEGOLAS is a general fault injection framework aiming to expose diverse bugs.

Partial Failure Detection. Failure detectors are part of a running production distributed system to determine whether the system is faulty or not. Recent works [22, 36, 41, 47] explore advanced detectors for the notorious partial failures. LEGOLAS is an offline testing tool. It can leverage these advanced techniques in its checkers to find more bugs in testing.

Error Handling Bug Detection. Error handling code is known to be buggy. Studies [36, 55] have shown that this is also true for distributed systems. Aspirator [55] uses rules to statically find simple error handling bugs such as empty handlers. LEGOLAS focuses on fault injection to systematically test distributed systems and uncover diverse types of bugs.

10 Conclusion

This paper presents LEGOLAS, a fault injection testing framework that aims to catch complex partial failure bugs in large distributed systems. LEGOLAS uses static analysis to enable fine-grained, system-specific fault injection. It designs a novel method to extract abstract states from system code and uses them to efficiently explore the fault injection space. We apply LEGOLAS on six distributed systems and find 20 new bugs. LEGOLAS is available at https://github.com/OrderLab/Legolas.

Acknowledgments

We thank our shepherd, Peter Alvaro, and the anonymous reviewers for their valuable and detailed feedback that improved our work. We appreciate the help from the developers of the open-source distributed systems we evaluated. We thank CloudLab [11] for providing the resources to run our experiments. This work was supported in part by NSF grants CNS-2317698, CNS-2317751, and CCF-2318937.

References

[1] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 351–368. USENIX Association, November 2020.

[2] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 51–68, Berkeley, CA, USA, 2018. USENIX Association.

[3] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 331–346, New York, NY, USA, 2015. ACM.

[4] Cory Bennett and Ariel Tseitlin. Chaos monkey released into the wild. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html, 2009.

[5] Dmitry Bugaychenko. Kafka production failure because of BadVersionException. https://issues.apache.org/jira/browse/KAFKA-1407, 2014.

[6] Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. Understanding exception-related bugs in large-scale cloud systems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 339–351. IEEE Press, 2020.

[7] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. CoFI: Consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 536–547, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. Push-Button reliability testing for Cloud-Backed applications with rainmaker. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1701–1716, Boston, MA, April 2023. USENIX Association.

[9] Kim Christensen. Kafka partial cluster breakdown. https://issues.apache.org/jira/browse/KAFKA-3577, 2016.

[10] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 14:1–14:14, New York, NY, USA, 2013. ACM.

[11] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, jul 2019. USENIX Association.

[12] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *Proceedings of the 2022 USENIX Annual Technical Conference*, USENIX ATC '22, pages 1037–1048, Carlsbad, CA, July 2022. USENIX Association.

[13] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST '17, page 149–165, USA, 2017. USENIX Association.

[14] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 126–141, New York, NY, USA, 2022. Association for Computing Machinery.

[15] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 238–252, Berkeley, CA, USA, 2011. USENIX Association.

[16] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, pages 1–16, October 2016.

[17] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 1–14, Berkeley, CA, USA, 2018. USENIX Association.

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

[19] Tsuyoshi Hombashi. tcconfig: A tc command wrapper. https://github.com/thombashi/tcconfig, 2022.

[20] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

[21] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[22] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16. USENIX Association, October 2018.

[23] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI, British Columbia, Canada, May 2017. ACM.

[24] LLC. Jepsen. Jepsen: a framework for distributed systems verification, with fault injection. https://github.com/jepsen-io/jepsen, 2023.

[25] Jiahongchao. updateisr should stop after failed several times due to zkVersion issue. https://issues.apache.org/jira/browse/KAFKA-3042, 2015.

[26] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A programmable tool for multiple-failure injection. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 171–188, New York, NY, USA, 2011. ACM.

[27] Xiaoen Ju, Livio Soares, Kang G. Shin, Kyung Dong Ryu, and Dilma Da Silva. On fault resilience of OpenStack. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13, pages 2:1–2:16, New York, NY, USA, 2013. ACM.

[28] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. Debugging transient faults in data centers using synchronized network-wide packet histories. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '21, pages 253–268. USENIX Association, April 2021.

[29] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI 07)*, Cambridge, MA, April 2007. USENIX Association.

[30] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, 14(3):268–280, November 2020.

[31] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, page 399–414, USA, 2014. USENIX Association.

[32] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 162–180, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 677–691. ACM, April 2017.

[34] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating*

Systems, HotOS '19, page 155–162, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. FCatch: Automatically detecting time-of-fault bugs in cloud systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 419–431. ACM, 2018.

[36] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, Santa Clara, CA, February 2020. USENIX Association.

[37] Chang Lou, Yuzhuo Jing, and Peng Huang. Demystifying and checking silent semantic violations in large distributed systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 91–107, Carlsbad, CA, USA, July 2022. USENIX Association.

[38] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 295–310, New York, NY, USA, 2015. Association for Computing Machinery.

[39] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. CloudRaid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, FSE '18, pages 3–14. ACM, November 2018.

[40] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. CrashTuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 114–130, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, and Jiesheng Wu. PERSEUS: A fail-slow detection framework for cloud storage systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, FAST '23, USA, 2023. USENIX Association.

[42] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Paul D. Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, DSN '09, pages 379–388. IEEE, June 2009.

[44] Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. Service-level fault injection testing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 388–402, New York, NY, USA, 2021. Association for Computing Machinery.

[45] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 33–50, USA, 2018. USENIX Association.

[46] Donny Nadolny. Debugging distributed systems. In *SREcon 2016*, Santa Clara, CA, April 7-8 2016.

[47] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. IASO: A fail-slow detection and mitigation framework for distributed storage services. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 47–61, USA, 2019. USENIX Association.

[48] Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones, and Ali Basiri. *Chaos Engineering*. O'Reilly Media, Inc., 2017.

[49] ScyllaDB. CharybdeFS: A fuse based fault injection filesystem. https://github.com/scylladb/charybdefs, 2021.

[50] Jiri Simsa, Randy Bryant, and Garth Gibson. dbug: Systematic evaluation of distributed systems. In *5th International Workshop on Systems Software Verification (SSV 10)*, Vancouver, BC, October 2010. USENIX Association.

[51] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Automatic reliability testing for cluster management controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 143–159, Carlsbad, CA, July 2022. USENIX Association.

[52] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–, Mississauga, Ontario, Canada, 1999. IBM Press.

[53] Alex Wilson. Bytecode-level fault injection for the JVM. https://github.com/mrwilson/byte-monkey, 2019.

[54] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, page 213–228, USA, 2009. USENIX Association.

[55] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265, Berkeley, CA, USA, 2014. USENIX Association.

[56] XinHao Yuan and Junfeng Yang. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1141–1156. ACM, March 2020.

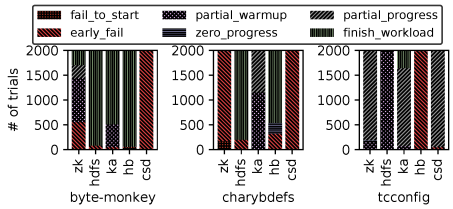
[57] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 3MileBeach: A tracer with teeth. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 458–472, New York, NY, USA, 2021. Association for Computing Machinery.

	CharybdeFS	tcconfig	Byte-monkey
	FS syscalls error, delay	packet loss, delay	exceptions
moderate	10%, 8 s	20%, 300 ms	10%
mild	1%, 1 s	10%, 80 ms	1%

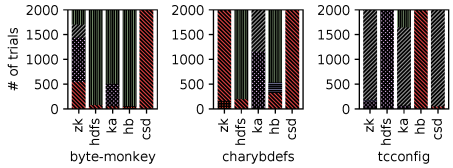
Table 9: Configurations of the baseline tools.

System	Release	SLOC	Workload
ZK	3.6.2	95K	3 clients create 10 entries in total, each writes and reads a random entry 40 times.
HDFS	3.2.2	689K	5 clients each writes a 10KB file 5 times, 5 clients each reads 5 files 5 times.
Kafka	2.8.0	322K	3 clients creates 15 topics, 15 clients each produces a topic, 30 clients consume a topic.
HBase	2.4.2	728K	creates 5 tables, each with 5 columns, each with 5 rows, 5 clients r/w a random row 100 times.
Cassandra	3.11.10	210K	creates 1 table, inserts 5 rows, 3 clients select and update a random table field 100 times
Flink	1.14.0	78K	1 client submit batch workload, 1 client submit streaming workload, 1 Kafka consumer to receive 170 messages from the batch workload, 1 Kafka producer to send 5 messages to the streaming workload, 1 Kafka consumer to receive 10 messages from the streaming workload

Table 7: Evaluated distributed systems in latest releases, and the workload LEGOLAS uses in the fault injection experiments.



(a) w/ moderate fault injection config.



(b) w/ mild fault injection config.

Figure 14: 2000 fault injection trials with existing tools: Byte-monkey, CharybdeFS, tcconfig.

Appendix A Evaluation Details

Table 8 lists the injected fault, symptom, and root cause for each of the 20 new bugs that LEGOLAS finds.

Table 9 shows the configuration settings we use for the baseline fault injection tools (Byte-monkey, CharybdeFS, and tcconfig) in the evaluation.

Figure 14 shows the aggregated fault injection results using the baseline tools.

Bug Id	Injected Fault	Symptom	Root Cause
ZK-4074	delay when `Learner` is executing `writePacket`	requests to one follower get stuck and the follower cannot rejoin the quorum for a long time	write in a critical section and prevents `QuorumPeer` from entering the receiving stage
ZK-4203	an exception during accepting a connection from the second follower	one follower keeps trying to join the quorum but keeps failing, even though the other 2 nodes get the request	the `ERROR` state set by learner is not discovered by the leader in some condition
ZK-4419	an exception when a learner creates a socket to connect to leader	one follower takes a long time to join the quorum and causes temporary service unavailability	the server state is prematurely changed, which triggers unnecessary re-election
ZK-4424	an exception when leader is configuring socket options for a follower	causes re-election and partial service unavailability	error is unnecessarily re-thrown, which causes handler exit and costly re-creations
KA-13457	an exception when a broker is accepting a connection	some client requests to create topics experience InvalidReplicationFactorException errors for a long time	error is swalloed without closing the socket channel
KA-13468	an exception when the log manager initalize a log	a broker proceeds but consumers hang for more than 3 minutes without any error log	the log manager should handle the error but instead let it propagate to the request handler
KA-13538	an exception when a broker is accessing checkpoint file	some clients get unexpected TopicExistsException even though they have never created the topic before	a design flaw in the client library for handling broker change
KA-14882	a delay when a broker sends request to ZooKeeper	some retry of topic creation requests gives the client TopicExistsException	broker controllers do not roll back the metadata in ZooKeeper when topic creation fails
KA-14886	a delay when handling a request from consumer and storing data to disk	a critical thread pool in broker gets full soon after the delay of a single request from client	the delayed thread blocks multiple threads and causes the thread pool to be used up
HD-15925	IOException when a datanode is forwarding a packet to the mirror	normally a client is immediately notified of the error, but now the client hangs for 1min	race condition causes PacketResponder to be blocked without notifying the client
HD-15957	exception when namenode finishes sync edit log and notifies journalnode	some client hangs forever without any log and the expected file does not exist in HDFS	namenode dismisses one client RPC, adding retry of the notification resolves the issue
HD-15869	delay when namenode sends the edit log notifications	namenode hangs even with the async edit logging	the notification sending is performed synchronously and blocks queued edit logs
HA-17552	error after the namenode accepts a socket before creating a reader	some client hangs instead of timing out after ping interval	read method does not re-throw the socket timeout exception
HA-18024	an exception when namenode configures socket options	some client hangs for a long time	socket connection is not closed when error happens
HB-26256	a delay when the region server tries to open a region using HDFS RPC	table creation command hangs for a long time without any error but list command shows the table exists	region sequence id file write operation is blocking without any timeout
HB-26955	an exception when the master tries to do an update operation	some table create requests experience a long delay	retry code misses the case when a server is quickly reinitialized
CS-16603	a delay in serializing a mutation to commit log in node 2	clients to node 1 experience sporadic CQL operation timeout due to unconfigured table	the add method is not protected with a timeout
CS-17564	exception when deleting file during a compaction task	node continues after erroreous startup state and later causes client failures	missing sync. to wait for compaction completion before setting node startup flag
FL-30032	an IOException when sending a watermark to Kafka	synchronous batch processing request from client finishes without errors while the job is actually not finished	the exception is not handled properly and then a few messages do not get sent to Kafka
FL-31746	an IOException when task managers finish a job and commit some data	when the job is finished, the client throws confusing errors due to a fault in commit phase	The commit phase does not affect the correctness of output but its fault is propogated to the client with confusing messages

Table 8: New bugs found by LEGOLAS. All issues cause partial failure symptoms. The root causes are diverse. ZK: ZooKeeper; KA: Kafka; HD: HDFS; HA: Hadoop; HB: HBase; CS: Cassandra; FL: Flink.