CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis

Jie Lu
lujie@ict.ac.cn
SKL Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences, China

Chen Liu
liuchen17z@ict.ac.cn
SKL Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences, China

Lian Li[*][†]
lianli@ict.ac.cn
SKL Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences, China

Xiaobing Feng
fxb@ict.ac.cn
SKL Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences, China

Feng Tan
Jun Yang
Liang You
{tanfeng.tf,muzhuo.yj,youliang.yl}@alibaba.com
Alibaba Group

Abstract

Crash-recovery bugs (bugs in crash-recovery-related mechanisms) are among the most severe bugs in cloud systems and can easily cause system failures. It is notoriously difficult to detect crash-recovery bugs since these bugs can only be exposed when nodes crash under special timing conditions. This paper presents CrashTuner, a novel fault-injection testing approach to combat crash-recovery bugs. The novelty of CrashTuner lies in how we identify fault-injection points (crash points) that are likely to expose errors. We observe that if a node crashes while accessing *meta-info* variables, i.e., variables referencing high-level system state information (e.g., an instance of node or task), it often triggers crash-recovery bugs. Hence, we identify crash points by automatically inferring meta-info variables via a log-based static program analysis. Our approach is automatic and no manual specification is required.

We have applied CrashTuner to five representative distributed systems: Hadoop2/Yarn, HBase, HDFS, ZooKeeper, and Cassandra. CrashTuner can finish testing each system in 17.39 hours, and reports **21** new bugs that have never been found before. All new bugs are confirmed by the original developers and **16** of them have already been fixed (14 with our patches). These new bugs can cause severe damages such as cluster down or start-up failures.

CCS Concepts • **Software and its engineering** → **Software testing and debugging**; **Cloud computing**.

Keywords Crash Recovery Bugs;Fault Tolerance;Distributed Systems;Bug Detection;Fault Injection;Cloud Computing

ACM Reference Format:
Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3341301.3359645

1 Introduction

Distributed systems have become the backbone of computing in the cloud era. More and more applications are built on top of large-scale distributed systems (such as scalable computing frameworks [20, 57] and distributed storage systems [22, 36]), to provide online services to users. High availability of those systems is crucial: failures of the underlying distributed systems can lead to cloud outage, easily costing service providers millions of dollars [2, 9].

High availability of distributed systems largely hinges on how well these systems tolerate node crashes (failures). Large-scale distributed systems are often comprised of thousands of nodes (machines) [55], and it is common that a node may fail due to hardware or software faults [49]. Although various sophisticated crash-recovery mechanisms [4, 13, 16] have been adopted in distributed systems, it is still challenging to handle node crashes correctly. It is very difficult, if not impossible, for developers to anticipate all possible crash scenarios and correctly implement corresponding recovery mechanisms. In this paper, we refer to bugs in crash-recovery-related mechanisms as *crash-recovery bugs*.

[*] corresponding author: lianli@ict.ac.cn
[†] Also with TianQi Soft Inc., China.

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359645

Crash-recovery bugs are among the most severe bugs in distributed systems. Many node crashes can be recovered from by the sophisticated fault-tolerance schemes implemented in distribute systems [25]. However, crash-recovery bugs break such fault-tolerance schemes and hence easily lead to system failures. Moreover, it is notoriously difficult to detect crash-recovery bugs during in-house testing. Node crash events need to be injected under special timing conditions to trigger a bug. As a result, crash-recovery bugs widely exist in deployed distributed systems [21, 25, 26, 39].

The state-of-the-art techniques detect crash-recovery bugs via fault-injection testing [23, 24, 32–34]. However, it is challenging to hit the small bug-triggering windows due to the huge state-space of the system under testing. Random fault injection is ineffective, as is evident in our own experiments and previous work [23]. Systematic approaches (i.e., distributed system model checkers [28, 35, 37, 38, 48, 54, 59]) suffer from the state explosion problem. Researchers apply sophisticated heuristics [59], or resort to manual specifications [38], to effectively restrict the large search space. Although great progress has been made, these approaches still struggle at exploring the huge state-space of distributed systems. Most (>99.9%) injected faults are unnecessary and very few new crash recovery bugs were reported [43].

This paper presents *CrashTuner*, a novel approach to precisely identify bug-triggering points where node crash events can be injected. Hereafter, we refer to such program points as *crash points*. CrashTuner precisely locates crash points by automatically inferring *meta-info variables* (variables referencing high-level system state information), whose access points are fault-injection points likely to expose errors. This is realized via a log-based static program analysis. It is fully automatic and no manual specifications are needed. We have applied CrashTuner to test the **5** representative distributed systems: scale-out computing framework Hadoop2/Yarn [57], distributed key-value storage HBase [22], scalable file system HDFS [19], cluster synchronization service ZooKeeper [31], and decentralized storage system Cassandra [36]. CrashTuner can reproduce **59** out of **66** existing crash recovery bugs , and reports **21** new crash-recovery bugs that have never been found before. These new bugs lead to severe damages such as cluster down or startup failures. To date, **16** new reported bugs have been fixed (14 with our patches).

Observation Distributed systems consist of clusters of nodes. Jobs and large chunks of resources are divided into small pieces, and then assigned to each individual node. The set of nodes, and their associated tasks and resources, together form a high-level view of the system state. Figure 1 depicts a simplified high level view (automatically constructed by our analysis) of the distributed computing framework Hadoop2/Yarn. The system includes a cluster of individual nodes (`Node_0`,...,`Node_n`). Each node manages one or more containers. There are `m` (`m` ≠ `n`) containers in total (`MasterContainer`,`Container_1`,...,`Container_m`) . A JVM process (`JVM_ID`) is spawned on each container, and each JVM process is a particular instance to execute an attempt (`TaskAttempt_ID`) of a given task (`Task_ID`). A user job request (`Job_ID`) is handled by an application instance (`App_ID`). Each job is delegated to a master container . The master node decomposes a job into `m` small tasks then dispatches each task to an available container.

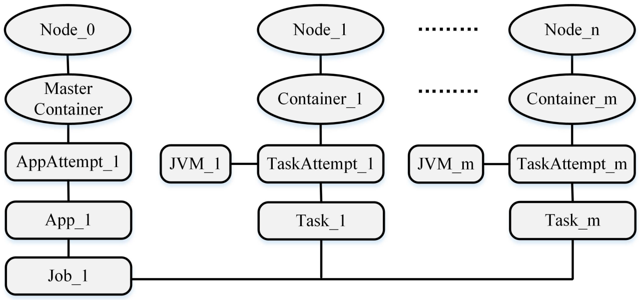


Figure 1. A simplified high level view of Hadoop2/Yarn. Ovals represent resources and squares stand for tasks.

In practice, the above high-level system state information is stored in the heap memory of different nodes and accessed via heap references. For instance, in Hadoop2/Yarn, the instance field `NMContext.nodeId` refers to a particular node. For convenience, we regard those variables referencing high-level state information as *meta-info variables*. Node crash or recovery events will change the system state. It is crucial to update those meta-info variables accordingly. Otherwise, a crash recovery bug may be triggered.

We have examined **66** crash recovery bugs from 4 of the 5 representative distributed systems (Cassandra not included). Our study leads to the following observation:

> The *Crash Points* are the program points accessing meta-info variables (variables referencing high-level system state information). Crash-recovery bugs are triggered when a node crashes at crash points.

14 out of the 66 bugs are not timing sensitive, and they can be trivially triggered with any fault-injection techniques . For the remaining **52** crash recovery bugs, their crash points are observed as above. There are two common scenarios:

- *The pre-read scenario:* Node *N* crashes before its meta-info is read by another node *M*. Node *M* is not aware of the crash and keeps using the stale information of *N*, leading to aborts and job failures.
- *The post-write scenario:* Node *N* crashes after *N* updates the system state (i.e., stores to meta-info variables). In the recovery process, intermediate updates of *N* need to be discarded and rolled back. The recovery

process may mis-handle the corrupted state, leading to failed (or incorrect) recovery attempts.

The CrashTuner Approach The key is to locate crash points. How do we find crash points? In CrashTuner, we apply log analysis, together with a type-based static program analysis to automatically infer meta-info variables. The crash points are those program points before reading a meta-info variable (*pre-read points*), or after writing a meta-info variable (*post-write points*).

An immediate question arises: which variables are meta-info variables? First of all, node referencing variables are meta-info variables. We could easily identify node referencing variables (and their runtime values) from runtime logs. Distributed systems provide a rich set of runtime logs for diagnosis and online monitoring. These logs record messages and events, which contain information such as `nodeId` and `taskId`. For example, the log instance in Hadoop2/Yarn `"NodeManager node1 registered as node1:42349"` indicates that the node `node1:42439` joins the cluster. The meta-info variables are then defined as follows:

> Node-referencing variables and their directly or indirectly *related* variables are meta-info variables. Two variables are *related* if they appear in a same runtime log instance. Object fields holding same values of meta-info variables are meta-info variables.

The meta-info variables precisely reflect the high-level system state information, i.e., a cluster of nodes and resources/tasks associated to each node. We apply log analysis to discover the set of meta-info variables logged at runtime. A type-based static analysis is then applied to infer all other meta-info variables in the program. Our static-analysis examines the types of existing meta-info variables, to derive other meta-info variables with equivalent types. Finally, after identifying crash points from meta-info variables, CrashTuner will apply fault-injection testing at each crash point individually . The detailed approach is illustrated in Section 3.

Contributions We make the following contributions.

- We propose a novel approach to crash-recovery bug detection. Our approach differs from existing fault-injection testing techniques in that we locate fault-injection points via *meta-info analysis.* Meta-info analysis automatically infers meta-info variables (variables referencing high-level state information), whose accessing points are fault-injection points likely to expose bugs. The approach is fully automatic and no manual specification is needed.
- We develop CrashTuner, a simple yet effective tool to detect crash-recovery bugs. In CrashTuner, meta-info analysis is realized via a log-based static program analysis. In a separate testing phase, CrashTuner performs minimal instrumentation for fault-injection testing. The analysis is non-intrusive and suitable for online monitoring. It can be easily adopted and is very effective in detecting crash recovery bugs.
- We extensively evaluate CrashTuner using five representative distributed systems: Hadoop2/Yarn, HBase, HDFS, ZooKeeper, and Cassandra. CrashTuner can finish testing each system in 17.39 hours, and reports **21** new crash recovery bugs that have never been reported before, including **8** critical bugs (classified by the original developers). We have provided patches to **20** of the **21** bugs and 14 patches have been accepted.

The rest of the paper is organized as follows. Section 2 motivates our approach with an empirical study of 66 crash recovery bugs. We present the design and implementation of CrashTuner in Section 3 and evaluate its efficiency and effectiveness in Section 4. Section 5 reviews related work and Section 6 concludes the paper.

2 Motivation

To better understand crash recovery bugs, we thoroughly examine the crash-recovery bugs from existing bug study databases [21, 25]. We focus our study on the following four distributed systems: Hadoop2/Yarn, HDFS, HBase, and ZooKeeper. The two databases in [21, 25] provide a total of 116 crash recovery bugs in the above 4 systems. In this work, we focus on bugs triggered by one crash event only. Hence, 50 bugs are omitted since they involve multiple crash events (34 bugs) or require IO operations (16 bugs). Previous detection techniques [23, 33] for crash-recovery bugs involving multiple events could help extend our approach to tackle these bugs. For the 66 remaining bugs, we have an in-depth look at each of them .

14 out of the 66 crash-recovery bugs are not timing-sensitive. They are due to implementation or logic errors in the recovery process, and can be triggered at anytime when a node crashes. For example, in MR-3463, the master node uses the format *"host:port"* to represent the host name of a node. If the master node crashes, the recovery process tries to get a node with the wrong format *"host"*, which always fails. In ZK-131, there is a data race in the recovery process. When a node crashes, the recovery process sends out two event messages: one to reset the corrupted state and the other to read the reseted state. The bug is triggered if the read event is handled before the reset event. These bugs can be detected by existing techniques for distributed concurrency bugs [42, 47, 65]. Therefore, we only discuss the remaining 52 timing-sensitive crash recovery bugs in our study.

Table 1 characterizes each bug according to its crash point. All 52 bugs are triggered at program points accessing meta-info variables. Column 2 summarizes the meta-info being accessed for each bug. In practice, a particular type of meta-info can be referenced by variables with various types. For instance, in HBase, the meta-info `HRegionServer` represents a

Table 1. The studied timing-sensitive bugs. Column 2 gives the meta-info being accessed at each crash point.

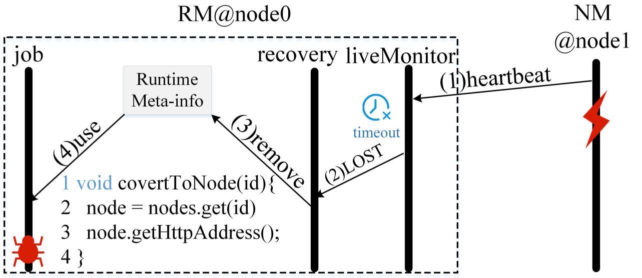
System	Meta-info	Bugs
Hadoop2	AppAttemptId	YARN-8664
	NodeId	YARN-2273 YARN-4227 YARN-5195 YARN-8233 YARN-5918
	ApplicationId	YARN-7007 YARN-7591 YARN-8222 YARN-4355
	AppState	YARN-4502
	ContainerId	MR-3596 YARN-4152 MR-4833 MR-3031
	File	MR-4099
	TaskAttemptId	MR-3858
HDFS	DatanodeInfo	HDFS-6231 HDFS-3701
	File	HDFS-4596
	BPOfferService	HDFS-8240 HDFS-5014
	NameNode	HDFS-4404 HDFS-3031
HBase	RegionTransition	HBASE-4539 HBASE-6070 HBASE-10090 HBASE-19335
	HRegion	HBASE-4540 HBASE-3365 HBASE-5927 HBASE-5155
	HRegionServer	HBASE-3617 HBASE-3874 HBASE-3023 HBASE-3283 HBASE-3362 HBASE-3024 HBASE-18014 HBASE-14536 HBASE-14621 HBASE-13546 HBASE-10272 HBASE-2525 HBASE-5063 HBASE-8519 HBASE-2797
	ZNode	HBASE-7111 HBASE-5722 HBASE-5635
	File	HBASE-3722
ZooKeeper	ZNode	ZK-569

node in the system, which can also be referenced by variables of types HServerInfo and HServerAddress. These variables can be converted to variables of types byte[], string and Integer, all referring to the same type of meta-info, i.e., HRegionServer. Section 3 illustrates how we infer meta-info variables in detail.

The crash points are further classified into two scenarios: 1) The pre-read scenario, i.e., before reading a meta-info variable, and 2) The post-write scenario, i.e., after writing a meta-info variable.

2.1 The Pre-read Scenario

37 bugs belong to this scenario: Node M tries to read meta-info of node N without knowing its availability, leading to aborts and job failures. Figure 2 depicts a typical crash-recovery bug [8] in Hadoop2/Yarn. The bug is triggered when the job thread tries to read resources of the crashed node NM@node1 from the shared data structure nodes. Since node1 is removed from nodes, a null value is returned and a null pointer exception is raised at line 3.



```
1 void covertToNode(id){
2   node = nodes.get(id)
3   node.getHttpAddress();
4 }
```

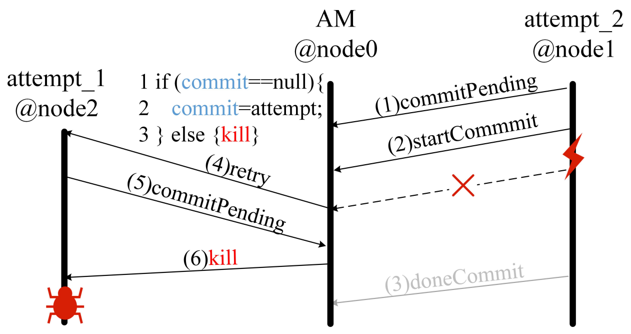
Figure 2. YARN-5918: a real-world crash-recovery bug in Hadoop2/Yarn. Two nodes are involved in this bug, the ResourceManager (RM for short) node0 and the NodeManager (NM for short) node1. 1) NM@node1 sends heartbeat message to RM@node0 when it is alive. After node1 crashes, no heartbeat message will be sent. 2) The liveMonitor thread in node0 detects the crash of node1 after a timeout period. A LOST event is dispatched to the recovery thread. 3) The recovery thread removes node1 from nodes, a shared data structure to record all available nodes. 4) Another running thread job tries to get resources of NM@node1.

How to detect these bugs? The crash point is the program point before reading a meta-info variable (in Figure 2, before reading node1 in line 2). The node corresponding to the meta-info variable being read from needs to be crashed to trigger such bugs, e.g., node1 in Figure 2. How can we find which node to crash? We develop an online log analysis to relate the runtime values of meta-info variables to a particular node. Hence, before reading a meta-info variable, we could query its runtime value to find the corresponding node.

The node crash event can only be detected after a timeout period. To trigger the bug in Figure 2, the job thread needs to wait until the liveMonitor thread detects the node crash event and sends out the LOST message. To speedup the testing process, we could set the default timeout period to a small interval to quickly unveil this bug. Alternatively, in our implementation, we leverage the shutdown script (most distributed systems provide such script to gracefully shutdown a node) provided by the system to let node1 leaves the cluster pro-actively, without waiting.

2.2 The Post-write Scenario

15 bugs belong to this scenario: Node N crashes after updating the system state and the recovery process fails to recover (or incorrectly recovers) from the corrupted state. Figure 3 gives a bug of such scenario [3]. If node1 crashes after doneCommit, attempt_1 is committed to the global state and no recovery is needed. If the crash happens before commitPending, the recovery process will fork another



```
1 if (commit==null){
2   commit=attempt;
3 } else {kill}
```

Figure 3. MR-3858: a real-world crash-recovery bug in MapReduce. Three nodes are involved in this bug, `node0` (the Application Master, AM for short), `node1` and `node2`. 1) `node1` sends message to AM@`node0` via an RPC `commitPending`, to get commit permission and get its attempt-ID `attempt_1` recorded. 2) `node1` starts to commit the result via an RPC `startCommit`. 3) `node1` crashes. The task is not committed, hence, the RPC `doneCommit` is missing. 4) AM detects the crash and starts a new node (i.e., `node2`), to re-commit the results of the same task in another attempt (`attempt_2`). 5) `node2` contacts AM@`node2` via the same RPC `commitPending`. 6) `node2` fails the `commit` checking and is then killed by AM@`node0`.

attempt instance, which will correctly redo the task. However, if `node3` crashes in the small time window between the two RPCs, the commit status `commit` is contaminated. The recovery process always fails, and the job will never finish.

How to detect these bugs? The crash point is the program point after writing a meta-info variable (in Figure 3, after writing the commit status `attempt_1` in line 2). To trigger such bugs, we need to crash the corresponding node of the stored meta-info variable, e.g., `node1` in Figure 3. Similarly, with online log analysis, we query values of stored meta-info variables to locate their corresponding nodes. In Figure 3, by querying the runtime value of stored meta-info variable in line 2 (`attempt_1`), we get the target node `node1`.

3 The CrashTuner Approach

In a nutshell, CrashTuner first identifies all crash points then performs fault-injection testing at each individual crash point. Figure 4 overviews our approach. It consists of two phases. The first phase (top half of Figure 4) locates crash points via log-based program analysis and profiling. The second phase (bottom half of Figure 4) exercises each individual crash point one by one. A light-weight online log analysis is employed to relate runtime meta-info values to a particular node. Thus, at a crash point, we can crash the right target node by querying the runtime value being accessed.

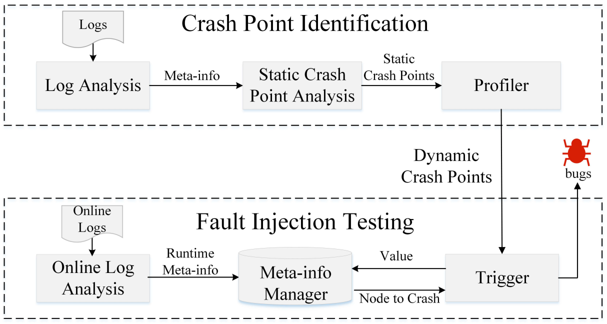


Figure 4. Overview of the CrashTuner approach.

3.1 Identify Crash Points

We apply a series of analyses to identify the *dynamic crash points* of a system, which is defined below:

Definition 1. *A dynamic crash point is represented as a tuple of two elements* $< P, Context >$, *where* P *is the program point and* $Context$ *is the call stack.*

We use the runtime call stack to differentiate the contexts in executing a same program point. Hence, when the same point is being executed with different calling contexts, they are considered as distinct crash points.

The process of identifying dynamic crash points is depicted in Figure 4 and summarized below:

- Log Analysis analyzes the runtime logs to discover meta-info variables, i.e., node referencing variables and their related variables. Those meta-info variables printed in logs are identified.
- Static Crash Point Analysis finds out all other meta-info variables in the system by examining the types of existing meta-info variables. Object fields with equivalent types of existing meta-info variables are regarded as meta-info variables. This type-based approach enables us to derive meta-info variables in the system, without precisely tracking program dependences. The program points before reading (after writing) meta-info variables are then identified as *static crash points*.
- Profiler runs the given workload to record the dynamic crash points (an executed static crash point with a distinct call stack) at runtime. Those static crash points not executed are discarded.

In the end, we obtain a set of dynamic crash points. Fault injection testing will exercise each individual crash point to trigger an error. Next, we illustrate the analyses in detail using the example in Figure 3.

3.1.1 Log Analysis

Log analysis mines existing runtime logs (obtained from online system or via profiling) to discover meta-info variables and record their runtime values.

```
1 LOG.info("NodeManager from " + host + " registered as " + nodeId);
2 LOG.info("Assigned container " + containerId + " on host " + nodeId);
3 LOG.info("Assigned container " + containerId + " to " + tId);
4 LOG.info("JVM with ID: " + jvmId + " given task: " + task.getTaskID());
```

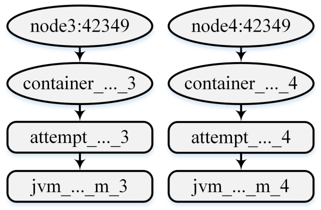
(a) Logging statements

```
1 NodeManager from (.*) registered as (.*)
2 Assigned container (.*) on host (.*)
3 Assigned container (.*) to (.*)
4 Jvm with ID: (.*) given task: (.*)
```

(b) Log patterns

```
1 NodeManager from node3 registered as node3:42349
2 NodeManager from node4 registered as node4:42349
3 Assigned container container_..._3 on host node3:42349
4 Assigned container container_..._3 to attempt_..._3
5 Assigned container container_..._4 on host node4:42349
6 Assigned container container_..._4 to attempt_..._4
7 JVM with ID: jvm_..._m_4 given task: attempt_..._4
8 JVM with ID: jvm_..._m_4 given task: attempt_..._4
```

(c) Log instances



(d) Meta-info

Figure 5. Simplified logging statements (a), log patterns (b), and runtime log instances (c) of the example in Figure 3. The runtime meta-info is given in (d).

We examine all logging statements in the system under testing. All the 4 distributed systems we studied use common logging libraries such as Log4j [10] and SLF4J [14]. Those libraries provide common logging interfaces with the following names: `fatal`, `error`, `warn`, `info`, debug, and `trace`. Hence, we find logging statements by simply matching the invoked method name at a call site with the name of a logging interface. Figure 5(a) gives a set of simplified logging statements for our illustration example. Their corresponding log patterns are extracted in Figure 5(b) (as in previous work [47, 58, 64]), where runtime values of logged variables are represented with regular expression (.*).

Figure 5(c) shows the simplified runtime log instances of our illustration example (Figure 3). Each log instance is processed separately, to match it with a particular log pattern. We adopt the approach in [58] to efficiently match a run time log instance with a log pattern. The runtime values of logged variables can then be derived as highlighted in red.

To infer meta-info variables, we examine the runtime values of logged variables. Those variables whose runtime values contain host names or IP addresses (as specified in the configuration file) are regarded as node-referencing variables, e.g., `node3:42349` and `node4:42349` in Figure 5. We relate other runtime values to nodes by checking whether they appear in a same runtime log instance or not, e.g., `container_..._3` (log instance 3) and `container_..._4` (log instance 5). The figure in Figure 5(d) depicts the runtime meta-info derived from log analysis, where related runtime values are connected together. The figure precisely reflects the high-level system state in Figure 1. Those variables holding meta-info values at runtime are meta-info variables.

3.1.2 Static Crash Point Analysis

Meta-info values are stored in the heap memory of a node, and referenced via object fields. To precisely identify object fields holding meta-info values, we need to compute and track the precise dependence information between variables using pointer analysis [40, 41, 56]. This is a daunting task given the complexity of distributed systems and precise pointer analysis for distributed systems remains to be an open research topic. Hence, we develop a simple type-based analysis to deduce meta-info fields instead.

Definition 2. *T is a meta-info type if there exists a meta-info variable of type T. The subtypes and collection types of T are meta-info types. Class C is a meta-info type if it contains an instance field $C.f$ of meta-info type T, and $C.f$ is only set in the constructors of C.*

Definition 2 defines our type-based analysis. Intuitively, meta-info is typed and variables with equivalent meta-info type T refer to the same type of meta-info. We also consider T's containing class C if there exists a field $C.f$ of type T which is only set in the constructors of C. This is to handle the common case where an instance of C is uniquely indexed by its field $C.f$, e.g., objects of class `RMContainerImpl` are uniquely indexed with a field of type `ContainerId`. In practice, the two classes refer to the same type of meta-info interchangeably. (Theoretically, class C may contain such fields of different meta-info types. In that case, C could be classified as either a field type or both. It does not affect our analysis to deduce meta-info types and meta-info variables. We did not observe such a case in the 4 systems we studied.)

To avoid introducing too many irrelevant variables, we do not apply the above generalization rules to the following base types: `Integer`, `String`, `Enum`, `byte[]`, and `File`. Instead, we identify meta-info fields of base types via log analysis and regard their containing classes as meta-info types.

Table 2 presents the meta-info types for our example. Not all meta-info types are given. The types annotated with * are types obtained from log analysis (e.g., types of logged meta-info variables), and all other types are deduced by static analysis. Types referring to the same type of meta-info are grouped together.

Table 2. Meta-info types for the example in Figure 3. Not all meta-info types are given. The types identified in log analysis are annotated with *. Other types are derived by static analysis.

—	Meta-info Types
Node	`yarn.api.records.NodeId`* `java.net.InetSocketAddress`* `yarn.api.records...NodeIdPBImpl`

App Attempt	`yarn.api.records.ApplicationAttemptId`*
	`yarn.server...SchedulerApplicationAttempt`
	`yarn.server...RMAppAttemptImpl`
	`yarn.api...ApplicationAttemptIdPBImpl`

Application	`yarn.api.records.ApplicationId`*
	`yarn.server...RMAppImpl`
	`yarn.server.resourcemanager.Application`
	`yarn.server.nodemanager...ApplicationImpl`
	`yarn.api.records...ApplicationIdPBImpl`

Container	`yarn.api.records.ContainerId`*
	`yarn.api.records.Container`*
	`yarn.server.nodemanager...ContainerImpl`
	`yarn.server...RMContainerImpl`
	`yarn.api.records...ContainerPBImpl`
	`yarn.api.records...ContainerIdPBImpl`

Task Attempt	`mapreduce.v2.api.records.TaskAttemptId`*
	`mapreduce.MapTaskAttemptImpl`
	`mapreduce.ReduceTaskAttemptImpl`
	`mapreduce.v2.app...TaskAttemptImpl`
	`mapreduce.v2.api....TaskAttemptIdPBImpl`

Static crash points are access points to fields of meta-info types. The `putField` and `getField` instructions to fields of non-collection types are identified in a straight-forward manner. Fields of collection types are read/written via generic APIs. Hence, for collection types, we check the invoked method name with the APIs in Table 3, to find a matching read/write operation.

Optimizations We discard references to field `C.f` if it is only set in constructors of its containing class `C`. By definition, type `C` is also a meta-info type (Definition 2) and references to objects of type `C` are already regarded as crash points. Thus, it becomes redundant to perform fault-injection testing at later references to `C.f`. Read references with no usages (or only used in logging statements and call statements to the 3 Java APIs `toString`, `hashCode`, `equals`) are omitted. In addition, we filter out read references whose values are sanity-checked (i.e., used as conditions of `if` statements) before being used. The checks suggest fault-tolerance schemes in the implementation (in fact, 14 of our studied bugs and 7 new bugs are fixed by introducing a sanity check).

All above optimizations do not guarantee soundness, which means we may miss true crash points. However, we randomly selected 3,000 optimized-out crash points for fault-injection testing , no new bugs can be detected.

Table 3. Keywords of read and write operations for collection types.

read	`get`, `peek`,`poll`, `clone`, `at`, `element`, `index`, `toArray`, `sub`, `contain`, `isEmpty`, `exist`, `values`
write	`add`, `clear`, `remove`, `retain`, `put`,`insert`, `set`, `replace`, `offer`, `push`, `pop`, `copyInto`

Finally, those program points before reading (after writing) a meta-info field are static crash points. For convenience, hereafter, we refer to them as *pre-read/post-write* points, respectively. If a read reference is only used in the return statements of a method, we promote the corresponding static crash point to the call-sites of the method. Such promotion helps to simplify the call stacks of corresponding dynamic pre-read points.

3.1.3 Profiler

The profiler generates dynamic crash points, i.e., executed static crash points with distinct calling stacks. Those static crash points not exercised are discarded.

We profile the system under testing with workloads of different sizes until a fixed point. Starting from the default size, we keep doubling the size until no new dynamic crash points can be generated. The process quickly converges in 2 or 3 iterations. We instrument the system at each static crash point. During profiling, the instrumented code will record each hit static crash point together with its corresponding call stack. The Java API `Thread.currentThread().getStackTrace()` is invoked to get the runtime call stack. The call stack is represented using call strings and bounded to a depth of 5 (starting from the method of the crash point to its callers).

3.2 Fault-injection Testing

We perform fault-injection testing at each dynamic crash point. As discussed in Section 2, to inject the right faulty event, we need to figure out which node to crash (or shutdown) when hitting a dynamic crash point. How to find out the right node to crash? We apply a light-weight online log analysis (Figure 4) to record the values of meta-info variables and relate them to a particular node. The recorded meta-info is a simplified implementation of the graph in Figure 5(d). Trigger can then efficiently query the recorded meta-info with the read (written) value of meta-info variables to locate the target node .

3.2.1 Online Log Analysis

We collect runtime logs on each node of the cluster with a Logstash [11] agent. Logstash is a popular log collection tool which can perceive log file changes, then sends the change-sets to a *custom stash* (a nominated destination node) in time. To avoid sending unnecessary data, only the runtime

HashSet	[node3:42439,node4:42439]	
HashMap	Key	Value
	container_..._3	node3:42439
	attempt_..._3	node3:42439
	jvm_..._m_3	node3:42439
	container_..._4	node4:42439
	attempt_..._4	node4:42439
	jvm_..._m_4	node4:42439

Figure 6. Recorded runtime meta-info.

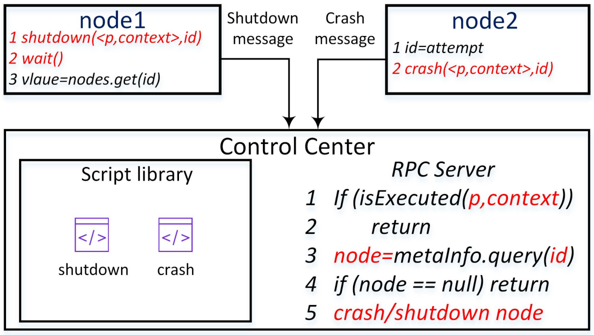
values of meta-info variables are sent out (i.e., those values highlighted in red in Figure 5(c)).

The custom stash processes received runtime values of meta-info variables in FIFO order. For efficiency, instead of constructing the graph as in Figure 5(d), we record the runtime values of nodes in a HashSet, and associate other meta-info values to a particular node via a HashMap. Figure 6 gives the recorded meta-info for our example. The received values node3:42439 and node4:42439 are inserted to the node set since they match host names. Log instance 3 sends out two related meta-info values: container_..._3 and node3:42439. Hence, the HashMap is updated with the key value pair <container_..._3, node3:42439>. When processing the two values in log instance 4: attempt_..._3 and container_..._3. We query each value in the HashMap to get its associated node (i.e., node3:42439 for the value container_..._3), then update the HashMap accordingly (<attempt_..._3, node3:42439>). We discard values unassociated to any node.

3.2.2 Trigger

Trigger instruments the system to inject crash events at a dynamic crash point, as shown in Figure 7. The pre-read point (node1) and post-write point (node2) are instrumented differently. Note that for illustration, Figure 7 presents two types of instrumentation together. In our implementation, we only instrument one dynamic crash point at a time.

For pre-read points, we instrument a shutdown RPC followed by a wait. The wait works as a timeout period (10 seconds by default) for the shutdown event to be handled. For post-write points, we instrument a crash RPC. Both RPCs are invoked with arguments <p, context> (the dynamic crash point) and id (the accessed runtime meta-info value). The Control Center is a separate node to handle the instrumented RPCs. If the dynamic crash point has not been exercised (line 1), we query the input runtime meta-info value to get its associated node (line 3). The procedure simply returns if no such node exists (line 4). Alternatively, we could randomly select a node to crash. However, this alternative approach has no impact on our experimental results. At line 5, we invoke the script library to crash/shutdown a node accordingly.



```
1 shutdown(<p,context>,id)
2 wait()
3 vlaue=nodes.get(id)
```

```
1 id=attempt
2 crash(<p,context>,id)
```

```
1 If (isExecuted(p,context))
2     return
3 node=metaInfo.query(id)
4 if (node == null) return
5 crash/shutdown node
```

Figure 7. Trigger.

Finally, we report a bug in any of the following 3 cases: 1) job failures; 2) system hangs; and 3) there exists uncommon exceptions in the logs. Currently, we do not report silent errors which lead to unexpected behaviors, e.g., silent data corruptions. How to develop test oracles for silent errors (e.g., gray failures) is an important topic worth separate investigation [27, 30, 44, 45, 52, 60].

3.3 Implementation Details

We implement our static analyses in WALA [5] and perform instrumentation with Javassist [1]. The popular log collection framework Logstash [11] is used for runtime log collection. The implementation consists of 9,933 lines of Java code and 550 lines of Shell code.

Log analysis We adopt the approach in [58] to efficiently match a run time log instance with a log pattern. A reverse index is built as a hash for each log pattern, which can be used to quickly calculate a matching score for each runtime log instance. The higher the matching score, the more likely the log pattern matches the log instance. For a given log instance, we select 10 logging patterns with the highest scores. Then we parse the log instance according to the 10 logging patterns, to find an exact match.

Code instrumentation We represent crash points as program points before/after WALA instructions. However, Javassist performs instrumentation at the source code level. A statement can spread across multiple source lines and a WALA instruction may refer to a source line in the middle of a statement. In that case, the instrumented class will not compile. Hence, for pre-read points, we try to instrument the source line and its preceding source lines (succeeding lines for post-write points) until the instrumented class compiles.

Online log analysis We leverage the results from offline log analysis and implement a filter to extract runtime values of meta-info variables from log instances efficiently.

The filter consists of regular expressions which are derived by offline analysis for the toString method of meta-info variables. For our illustration example, the filter for variables of type `nodeId` is "(.*):(.*)". Hence, when processing log instance 1 and log instance 2 in Figure 5(c), values matching the filter are extracted and sent to the custom stash (i.e., `node3:42439` and `node4:42439`).

Runtime meta-info values For fields of non-collection types, the meta-info values are obtained by calling their corresponding `toString` methods. For fields of collection types, the meta-info values are derived differently, according to given read/write operations. For instance, given `m.add(e)`, the result of `e.toString()` is considered as the stored meta-info value. For read operations such as `v = m.get(key)`, we get two related values: `v.toString()`, and `key.toString()`.

3.4 Limitations

CrashTuner is not sound and does not guarantee the absence of crash-recovery bugs. The effectiveness of CrashTuner relies on the log qualities of the system under testing. For instance, CrashTuner cannot reproduce the 3 bugs HBASE-13546, HBASE-14621, and YARN-4502 because the accessed variables are not printed in logs. Hence CrashTuner fails to identify them as meta-info variables. Nevertheless, CrashTuner can reproduce 59 out of 66 existing bugs and reports 21 new crash-recovery bugs.

The log-based implementation makes CrashTuner easily deployable to cloud systems. However, it also impacts its effectiveness to systems with limited logging information. For instance, CrashTuner did not detect any new bug in ZooKeeper, where only 290 runtime log instances are generated and node is simply represented with `Integer` type. There are alternative approaches to analyze meta-info, e.g., static analysis and instrumentation-based approaches. These approaches do not depend on logging. However, static analysis approaches suffer from high false positives and instrumentation-based approaches are difficult for deployment. We may need to pick the right trade-off between precision, effectiveness, and generality.

This paper does not target crash recovery bugs involving multiple crash events or IO operations. However, CrashTuner can be extended with existing techniques to tackle these bugs. For example, [50, 53] target crash-consistency bugs, and [23, 33] target bugs involving multiple crash events. These works will be covered in our future work.

4 Evaluation

We evaluate CrashTuner using the five widely-used open-source distributed systems in Table 4: Hadoop2/Yarn (distributed computing framework), HDFS (distributed file system), HBase (distributed key-value stores), ZooKeeper (distributed synchronization service), and Cassandra (distributed storage system). Note that Cassandra is not included in our empirical study. All systems are tested with their latest versions in the trunk when the experiments were conducted (Column 2). We apply the default configurations (including the default logging configurations) to all systems, except for Hadoop2/Yarn. To reproduce existing bugs in Hadoop2/Yarn (Table 1), the configuration needs to be set to *"enable opportunistic"* [7]. We evaluate the systems with common workloads (Column 3). WordCount, TestDFSIO, PE (performance evaluation) and Stress are built-in workloads in their corresponding systems and SmokeTest [17] is a popular workload for testing ZooKeeper. In addition, we append each workload with a "curl" command to test user queries via web interfaces.

Table 4. Systems under test.

System	Latest Version	Workload
Hadoop2/Yarn	3.3.0-SNAPSHOT	WordCount+curl
HDFS	3.3.0-SNAPSHOT	TestDFSIO+curl
HBase	3.0.0-SNAPSHOT	PE+curl
ZooKeeper	3.5.4-beta	SmokeTest+curl
Cassandra	3.11.4	Stress

All the experiments are conducted on a cluster with three identical nodes. Each node has a CentOS 6.5 system on an Intel(R) Xeon(R) E7-4809 processor with 32 GB of memory. The evaluation will answer the following research questions:

- **RQ1.** How effective is CrashTuner in detecting bugs, especially new bugs?
- **RQ2.** How does CrashTuner compare with other fault-injection testing approaches?
- **RQ3.** How efficient is CrashTuner?

4.1 RQ1:Effectiveness

We evaluate how effective CrashTuner is in reproducing existing bugs, as well as its ability in detecting new bugs.

4.1.1 Reproducing Existing Bugs

We try to reproduce all bugs in Table 1 (14 non timing-sensitive bugs are trivially reproduced hence not further discussed). For each bug, we first check whether CrashTuner can correctly locate its corresponding crash point or not. If this is successful, we then inject a crash event at the crash point to trigger the bug.

CrashTuner can successfully trigger **45** bugs out of the **52** bugs in Table 1. There are 7 bugs not reproduced. For the 3 bugs HBASE-13546, HBASE-14621, and YARN-4502, the accessed variables are sub-fields of node instances which are not directly printed in logs (e.g., `Master.infoPort_`). Hence CrashTuner fails to locate their crash points. To expose these bugs, manual annotation may be needed to identify such variables as meta-info variables. For the other 3 bugs , and HBASE-7111,HBASE-5722 and HBASE-5635, CrashTuner fails to associate the accessing meta-info to the right target node, which is in the lower layer system ZooKeeper. In HDFS-4596, the bug is triggered when accessing a MD5 file whose name is not associated to any node instance. To trigger these

Table 5. New bugs detected. All bugs are confirmed by the original developers, and 16 of them are already fixed. In YARN-9164, YARN-8650, HDFS-14216, two bugs are grouped under one issue since they share the same root cause and can be fixed with identical patches.

Bug ID	Priority	Scenario	Status	Symptom	Meta-info
YARN-9238	Critical	pre-read	Fixed	Allocating containers to removed ApplicationAttempt	ApplicationAttemptId
YARN-9165	Critical	pre-read	Fixed	Scheduling the removed container	ContainerId
YARN-9193	Critical	pre-read	Fixed	Allocating container to removed node	NodeId
YARN-9164(2)	Critical	pre-read	Fixed	Cluster down due to using the removed node	NodeId
YARN-9201	Major	pre-read	Fixed	Invalid event for current state of ApplicationAttempt	ContainerId
HDFS-14216(2)	Major	pre-read	Fixed	Request fails due to removed node	DataNodeInfo
YARN-9194	Critical	pre-read	Fixed	Invalid event for current state of ApplicationAttempt	ApplicationId
HBASE-22041	Critical	post-write	Unresolved	Master startup node hang	ServerName
HBASE-22017	Critical	pre-read	Fixed	Master fails to become active due to removed node	ServerName
YARN-8650(2)	Major	pre-read	Fixed	Invalid event for current state of Container	ContainerId
YARN-9248	Major	pre-read	Fixed	Invalid event for current state of Container	ApplicationAttemptId
YARN-8649	Major	pre-read	Fixed	Resource Leak due to removed container	ApplicationId
HBASE-21740	Major	post-write	Fixed	Shutdown during initialization causing abort	MetricsRegionServer
HBASE-22050	Major	pre-read	Unresolved	Atomic violation causing shutdown aborts	RegionInfo
HDFS-14372	Major	pre-read	fixed	Shutdown before register causing abort	BPOfferService
MR-7178	Major	post-write	Unresolved	Shutdown during initialization causing abort	TaskAttemptId
HBASE-22023	Trivial	post-write	Unresolved	Shutdown during initialization causing abort	MetricsRegionServer
CA-15131	Normal	pre-read	Unresolved	Request fails due to using removed node	InetAddressAndPort

```
OpportunisticContainerAllocatorAMService.OpportunisticAMSProcessor
1  public void allocate(ApplicationAttemptId appAttemptId) {
2      if (!appCache.exist(appAttemptId)) return;
3      .  .  .
4      SchedulerApplicationAttempt appAttempt=rmContext.getAppAttempt(appAttemptId);
5      + if (!appAttempt.getApplicationAttemptId().equals(appAttemptId)) {
6      +    LOG.error("Calling allocate on removed application attempt " + appAttemptId);
7      +    return;
6      + }
8      //allocate container for appAttempt
9  }
```

Figure 8. The simplified code snippet and patch for YARN-9238.

4 bugs, we need to introduce extra logs to associate the meta-info variable to the right target node.

4.1.2 Detecting New Bugs

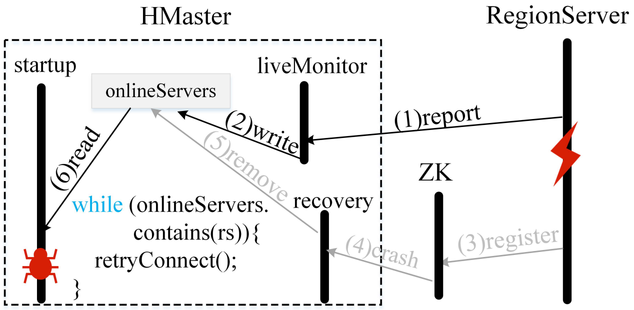
CrashTuner detects **21**[1] new bugs that have never been reported before (Table 5), including **8** critical bugs (classified by the original developers). All reported bugs are confirmed by the original developers and 16 of them have already been fixed (14 patches provided by us).

When submitting a bug issue, we are often required to provide a unit test exposing the bug (7 unit tests). It is tricky since we are not allowed to modify the source code to inject faults at the crash point. The default method in writing unit tests only supports fault injection before the invocation to a public method. Very often, a bug cannot be exposed if its crash point lies in the middle of a method.

For instance, in YARN-9238 (Figure 8), variable appAttemptId refers to an attempt instance to execute a given application. If the current attempt fails, the recovery process will try another attempt. The field currentAttempt (not shown in the code) refers to the attempt instance for each application. Hence, if the current attempt node (node associated to appAttemptId) crashes, field currentAttempt is reset to the new attempt node by the recovery process. In Figure 8, line 4 get the field value of currentAttempt, which is the new attempt node whose state is uninitialized. However, in line 8, the buggy code is not aware of the crash and uses it as the old attempt node (node associated to appAttemptId), leading to aborts .

CrashTuner successfully exposes this bug by crashing the node appAttempId associated to, before reading currentAttempt in line 4. After the crash, field currentAttempt is reset and the new attempt node is returned, exposing the error. However, the bug cannot be exposed by unit tests created via

[1] The website https://github.com/lujiefsi/CrashTuner shows how to reproduce all new bugs in detail.



```
while (onlineServers.
    contains(rs)){
  retryConnect();
}
```

Figure 9. HBASE-22041 . (1) `RegionServer` (RS) reports to `HMaster` when it is alive. (2) The `liveMonitor` thread handles the request and add RS to the list of online servers. (3) RS crashes before it is registered in ZooKeeper (ZK). (4) ZK cannot detect the crash. The recovery process did not get a notification of the crash event. (5) RS remains in the list of online servers, without being removed. (6) The `startup` thread fails to read from RS.

the default method. If the crash event is handled at the entry of the method `allocate`, the sanity-check at line 2 will detect the node crash and prevent the error. Hence, in our unit tests, we manually reset the field `currentAttempt` before invoking the method `allocate`, to emulate the system state at the crash point.

Lines 5-6 present our patch, which validates `currentAttempt` before it is used (line 8). In general, it is easy to figure out the root cause of a bug and provide a corresponding fix, by examining its crash point and logs. We submitted 20 patches in total, 15 patches were accepted, 5 patches were under review and 1 patch was rejected. Among the 14 accepted patches, 8 patches introduce sanity checks and another 7 add handlers for unexpected exceptions or events. HBASE-22017 is a data race bug caused by crash-recovery procedure in the server process `HRegionServer`. The original developers rejected our patch, which fixes the race condition in the server process. Instead, they created a new issue (HBASE-22047) and fixed the problem on the client side. We did not patch HBASE-22041, which is discussed below.

HBASE-22041 The bug is triggered after `RegionServer` (RS) reports to `HMaster` (1), and before it is registered in ZooKeeper (ZK) (3). If RS crashes after (3), ZK will detect the crash and start the recovery process. The list of online server can then be correctly updated (5). However, if the crash happens between the two messages (1) and (3), the `startup` thread fails to read from RS and will keep retrying forever, causing system hang.

When trying to fix this bug, we find the following comment: `//TODO: How many times should we retry`. The original developers were already aware of the problem. But for some reason, they did not fix it. We could not figure out the right retry threshold. Hence, we did not to patch this bug.

```
1 AbstractYarnScheduler:HashMap<NodeId, N> nodes;
2 ScheNode getScheNode(NodeId id){return nodes.get(id);}
3 public void completeContainer(Container container) {
4   ScheNode node = getScheNode(container.getNodeId());
5   node.releaseContainer(container.getContainerId());
6 }
```

Figure 10. Simplified code snippet for YARN-9164 .

YARN-9164 In YARN-9164, when a job finishes or fails, the method `completeContainer` releases containers of the job on each node (Figure 10). At line 3, method `getScheNode` is invoked to get the node that a container belongs to. However, if the target node crashes just before line 4, a `null` value is returned which will raises a `NullPointerException` at line 5. The master node cannot handle the exception and aborts immediately, bringing down the entire cluster.

CrashTuner successful identifies `NodeId` as meta-info types and `nodes` as meta-info variables. By definition, the crash point is the read access to `nodes` at line 2. Since at line 2, the read reference is directly returned, CrashTuner promotes the crash point to the callsites of method `getSchedNode` (e.g., line 4). There are 43 call-sites in total, corresponding to 43 potential static crash points, and 30 of them are optimized out since their return values are either not used (5) or sanity-checked (25). In the end, Profiler generates two dynamic crash points from the 13 static crash points, including the bug-triggering point before line 4. The other dynamic crash point does not expose errors.

4.1.3 Timeouts

CrashTuner reports four timeout issues (default timeout threshold is 4 times of 1 run). It is debatable whether these issues are true bugs or false positives. Although the tasks eventually finish (> 10 mins), they significantly slowdown the system.

Three timeout issues exist in Hadoop2/Yarn. In the first issue, when the map attempt task finishes, the task records the attempt instance in its field `successAttempt` and changes its state to `success`. CrashTuner injects a node crash event after setting the field `successAttempt`. The reduce task fails to read output from the map task due to the crash and it will retry for a long time (≃10 mins), which exceeds our default timeout threshold. In the other two issues, the application attempt is initialized to set the field `container` to its corresponding node. If the node crashes after setting the field, the application attempt will stuck in the `running` state. Eventually, after 10 minutes, the stuck application attempt will be killed by `AbstractLivelinessMonitor`.

There is one similar timeout issue in HBase, which will make the `Region` stuck in the `OPENING` state, before it is killed in 10 minutes.

Table 6. The complexity of fixing newly detected bugs/existing bugs in CREB [21]. This table is discussed in Section 4.1.4.

	LOC of patch	# patches	# days to fix	# comments
CREB bugs	117	4	92	26
New bugs	114.8	3.8	16.8	8.6

4.1.4 Complexity of Fixes

Table 6 compares the patches of newly detected bugs with existing bugs in CREB [21] (an existing bug study database). The number of lines of code (Column 2) per patch and the number of patches (Column 3) per bug are almost the same, suggesting similar complexity between new bugs and existing bugs. However, for newly detected bugs, the average time in fixing a bug (Column 4) and the number of comments per bug issue (Column 5) are significantly less. This is because most comments are discussing on how to reproduce the bug [62]. Once the bug is reproduced, it is generally easy to figure out the root cause and fix the bug. In our submitted bug issues, we illustrate in detail on how to reproduce each bug (some with unit tests) and provide patches for them, which significantly speeds up the fixing processes.

Discussion CrashTuner detected new bugs for all systems, except for ZooKeeper. Unlike the other four systems where global system states spread across multiple nodes, ZooKeeper keeps a copy of the entire global states on each node. As a result, even CrashTuner found 40 dynamic crash points (Table 10), they can only trigger 4 different types of IO exceptions, which are all handled by the system.

4.2 RQ2: Comparison with Alternative Approaches

We compare CrashTuner with two alternative fault-injection testing approaches: the random fault-injection approach and the OpenStack approach [34] to inject faults around IO events.

4.2.1 Random Crash Injection

In this experiment, each system is profiled with the normal workload to get its run time T. We perform random fault-injection testing by running each system 3000 times, and for each time injecting a node (randomly chosen) crash (or shutdown) event at a random time between range [0, T]. Table 7 gives the result.

Random fault-injection testing can successfully trigger 3 bugs:YARN-9194, HBASE-21470, and MR-7178, with each bug being triggered 2, 12, and 2 times, respectively. All bugs are also detected by CrashTuner. The three bugs are triggered when a node crashes during the process of starting a new node. Since it is time-consuming to start a new node, the random approach has a good chance to hit the relatively-large bug-triggering window. In summary, the random approach can trigger one bug in every 17.03 hours with averagely 937.5 runs (90.83 hours with averagely 5000 runs if repeated bugs not considered). On the other hand, CrashTuner can find one bug in every 1.70 hours with averagely 50.29 runs, which is much more efficient and effective.

Table 7. Results of random crash injection. This table is discussed in Section 4.2.1.

System	Times(h)	Known bugs	New bugs
Hadoop2/Yarn	71.03	2(4)	0
HBase	61.37	1(12)	0
HDFS	72.55	0(0)	0
ZooKeeper	19.62	0(0)	0
Cassandra	47.92	0(0)	0

Table 8. Number of IO classes, methods and IO points. This table is discussed in Sectino 4.2.2.

Systems	# IO classes	# IO methods	# Static IO points	# Dynamic IO points
YARN	539	823	1342	1312
HBase	341	667	456	196
HDFS	432	940	1658	1252
Zookeeper	145	580	619	492
Cassandra	203	687	1063	1248
Total	1660	3697	5138	4500

Table 9. Results of IO fault injection. This table is discussed in Section 4.2.2.

System	Times(h)	Known bugs	New bugs
Hadoop2/Yarn	57.66	1(6)	0
HBase	20.91	0	0
HDFS	44.05	0	0
ZooKeeper	3.74	0	0
Cassandra	30.51	0	0

4.2.2 IO Fault Injection

Table 8 summarizes the number of IO points for each benchmark. IO classes are those classes implementing the interface `java.io.Closeable` (Column 2), e.g., network and file stream classes. IO methods are public methods of IO classes starting with one of the following keyword: `read, write, flush,` and `close` (Column 3). Static IO points are call-sites to IO methods (Column 4). We use the same profiling strategy to obtain dynamic IO points, i.e., static IO points with calling contexts (Column 5). In this experiment, we inject crash event before and after each dynamic IO point. Table 9 gives the results.

IO fault injection can trigger only 1 bug (for 6 times): YARN-9201. This bug is also reported by CrashTuner. In summary, IO fault injection can trigger one bug in every

Table 10. Number of Types, fields, access points, and crash points. This table is discussed in Section 4.3.

System	# Total			# Meta-info			# Crash Points	
	Types	Fields	Access Points	Types	Fields	Access Points	Static	Dynamic
Hadoop2/Yarn	6,265	43,223	206,087	107	1,251	5,109	1,524	453
HBase	3,257	26,802	130,969	34	733	4,032	920	257
HDFS	3,924	30,645	138,269	43	315	1,924	495	237
ZooKeeper	532	4,787	28,072	3	13	90	41	40
Cassandra	3,175	17,295	95,259	1	122	666	197	69
Total	17,153	122,752	598,656	188(1.10%)	2,434(1.98%)	11,821(1.97%)	3,177(0.53%)	1,056(0.18%)

Table 11. Analysis and testing times. This table is discussed in Section 4.3.

System	Analysis(s)	Profile(s)	Test(h)	Total(h)
Hadoop2/Yarn	265.67	365.34	17.22	17.39
HBase	276.34	812.72	7.97	8.27
HDFS	218.61	417.06	8.46	8.65
ZooKeeper	37.83	26.44	0.25	0.27
Cassandra	213.36	71.23	1.02	1.10

Table 12. Crash points pruned by different optimizations. This table is discussed in Section 4.3.1.

System	Constructor	Unused	Sanity check
Hadoop2/Yarn	1,140	1,778	608
HBase	849	876	1,387
HDFS	355	373	701
Zookeeper	27	14	7
Cassandra	248	116	105

24.15 hours with averagely 750 runs (156.88 hours with averagely 4500 runs if repeated bugs not considered). Most bugs detected by CrashTuner cannot be triggered with IO fault injection because the real crash points are far away from any IO points.

Surprised by the results, we thoroughly checked the logs generated during IO fault injection testing. Many tests lead to exceptions. For instance, in HDFS, after crashing the name node when it is writing a log file, the recovery node throws a `LogHeaderCorruptException` due to the corrupted file. However, the exception is well handled by the system. Frequently, developers introduce exception handlers for IO operations. As a result, IO faults are often tolerated and IO fault injection is not as effective in triggering new bugs.

4.3 RQ3: Efficiency

Table 10 compares the number of meta-info types, fields, and access points (Columns 5-7) to the total number of types, fields, and access points for each system (Columns 2-4). According to log analysis and type-based static analysis, 2.20% access points are accessing meta-info variables (Column 7). Static optimizations and profiling further reduce the number of static and dynamic crash points to 0.58% (Column 8) and 0.19% (Column 9) of the total access points in the program, respectively. In the end, there are 453 dynamic crash points for Hadoop2/Yarn, 257 dynamic crash points for HBase, 237 dynamic crash points for HDFS, 40 dynamic crash points for ZooKeeper and 69 for Cassandra.

Table 11 shows the times of CrashTuner in testing each system. The analysis times (Column 2) include the time in running the system (we run each system once with the given workload in Table 4) to generate logs, the log analysis time, as well as the static analysis time. The profiling time is given in Column 3, where at most 3 runs are sufficient to find out all dynamic crash points. Column 4 is the total time to test all dynamic crash points (Column 9 in Table 10) one by one.

CrashTuner is very efficient. It finishes its analyses in 5 minutes for all benchmarks (Column 2). In the testing phase (Column 4), CrashTuner tests the 453 dynamic crash points (Column 9 in Table 10) for Hadoop2/Yarn in 17.22 hours. Our instrumentation and online log analysis do not introduce any noticeable performance degradation in testing each individual crash point.

4.3.1 Optimizations

As stated in Section 3.1.2, we discard field references in any of the three cases: 1) fields only set in the constructors of their containing classes (Constructor); 2) read references unused or only used in logging statements (Unused); 3) read references checked before being used (Sanity check). Table 12 shows the number of pruned crash points by each optimization. The 3 optimizations together reduce the number of crash points by 3.76X, significantly improving efficiency.

To evaluate the soundness of CrashTuner, we randomly selected 3000 crash points pruned by optimizations and 3000 non meta-info access points for fault-injection testing. However, no new bugs were triggered.

4.4 Discussions

We implement CrashTuner in Java and evaluate it using 5 Java-based distributed systems in the Hadoop eco-system. However, the approach is applicable to a wide range of different distributed systems. We studied the 14 scheduling-related critical crash-recovery bugs in Kubernetes [15], a popular distributed resource management system written in Golang [12]. Table 13 shows that the 14 bugs are also

Table 13. The studied bugs in Kubernetes [15].

Kubernetes	Node	#53647 #68984 #55262 #56622 #69758 #71063 #73097 #78782
	Pod	#72895 #68173 #68892 #70898 #71488 #72259

triggered when nodes crash at program points accessing meta-info. Kubernetes dynamically allocates and deallocates nodes (often from a cloud provider, e.g., google cloud), leading to frequent meta-info updates. When a node crashes, its meta-info still scatters around in the system. It is very difficult for developers to find all meta-info of the crashed node and update them accordingly, which often leads to bugs.

We believe that meta-info is a well-suited abstraction in analyzing distributed systems. The root causes of crash-recovery bugs are either 1) failing to handle corrupted meta-info (the post-write scenario), or 2) using stale meta-info (the pre-read scenario). This paper demonstrates the effectiveness of meta-info analysis to fault-injection testing. It can also help static detectors [61] and distributed system model checkers [28, 35, 37, 38, 48, 54, 59] to more effectively detect such bugs.

5 Related Work

Crash recovery bug study and detection There are many empirical studies on crash recovery bugs [21, 29, 39, 46]. Mesbahi et al. [49] pointed out that 2.4% nodes can crash in Google cluster per day, which may lead to many bugs [25]. TaxDC [39] shows that 63% of distributed concurrency bugs suffice in the presences of node crash and other faults. In [21], researchers conducted extensive empirical studies on crash-recovery related bugs. The two papers [29, 46] study node change bugs and exception-related bugs, respectively. These studies motivate our approach.

Recent research focuses on crash recovery bug detection via fault-injection testing, where faults are injected either randomly [6] or systematically [18, 23, 32, 34]. The systematic approaches rely on user specifications to guide fault-injection. They provide domain specific languages that allow users to specify fault-injection sequences, scenarios, and so on. Distributed model checkers [28, 35, 37, 38, 48, 54, 59] intercept messages and events (e.g., crash) in the system at runtime, then permute their orderings exhaustively. Although powerful, they still suffer from the state space explosion problem. FCatch [43] models time-of-fault bugs as a special type of concurrency bugs. It traces system execution via instrumentation and can predicate crash recovery bugs from the correct execution trace. Aspirator [61] is a static detector for exception handler bugs, some of which may also be triggered by node crash events. However, many crash-recovery bugs manifest themselves without involving any exception handlers. It will be difficult for static detectors to report these bugs with good precision. This paper introduces a novel fault-injection testing approach for crash recovery bug detection by automatically inferring meta-info variables, whose access points are likely to be crash points.

DCatch [42] extends the classic happen-before relations to distributed systems and adopts dynamic analysis to detect distributed concurrency bugs. CloudRaid [47] detects concurrency bugs in distributed systems by flipping the order of a pair of messages that always happen in a fixed order. The two works can be combined with our approach to uncover more concurrency bugs in the crash recovery process.

Log analysis for distribute systems Log analysis has been widely adopted in analyzing, monitoring, and diagnosing distributed systems. Xu et al. [58] detect anomaly executions by applying machine learning techniques to console logs from a system. DISTALYZER [51] studies performance of system components by comparing logs from abnormal execution and normal execution. Iprof [64] extracts request IDs and timing information from logs to profile request latency. Stitch [63] organizes log instances into tasks and sub-tasks, to profile different components in the entire distributed software stack. CloudRaid [47] employs log analysis for detection of distributed concurrency bugs. We mine logs to discover meta-info in distributed systems, for effectively detecting crash recovery bugs.

6 Conclusions

We present CrashTuner, a novel fault-injection testing approach to crash recovery bug detection. CrashTuner precisely identifies fault-injection points via meta-info analysis, which automatically infers meta-info variables (variables referencing high-level system state) whose accessing points are fault-injection points likely to expose errors. We evaluate CrashTuner against five representative distributed systems. CrashTuner can successfully reproduce **59** out of **66** existing bugs, and can detect **21** new bugs that have never been reported before. These bugs can cause severe damages such as cluster down or start-up failures.

In our future work, we plan to further extend CrashTuner to tackle crash-consistency bugs and deep bugs involving multiple crash events.

Acknowledgement

We thank our shepherd Haryadi Gunawi, and other anonymous reviewers for their valuable inputs. We thank Ting Yuan for his study on Kubernetes. This paper is supported by the National Key R&D program of China (No. 2016YFB1000201), the Innovation Research Group of National Natural Science Foundation of China (No.61521092), and the National Natural Science Foundation of China (U1736028 and 61872043).

References

[1] 1999. Java bytecode engineering toolkit since 1999. https://www.javassist.org/.

[2] 2012. Downtime costs per Hour. http://iwgcr.org/?p=404.

[3] 2012. MapReduce bug 3858. https://jira.apache.org/jira/browse/MAPREDUCE-3858.

[4] 2015. Understanding HDFS Recovery Processes. https://blog.cloudera.com/blog/2015/02/understanding-hdfs-recovery-processes-part-1/.

[5] 2015. WALA Home page. http://wala.sourceforge.net/wiki/index.php/Main_Page/.

[6] 2016. Fault Injection Framework and Development Guide. https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/FaultInjectFramework.html.

[7] 2016. Scheduling of opportunistic containers. https://issues.apache.org/jira/browse/YARN-5542.

[8] 2016. YARN bug 5918. https://jira.apache.org/jira/browse/YARN-5918.

[9] 2018. Lloyd's Estimates the Impact of a U.S. Cloud Outage at $19 Billion. https://www.eweek.com/cloud/lloyd-s-estimates-the-impact-of-a-u.s.-cloud-outage-at-19-billion.

[10] 2019. Apache log4j, a logging library for Java. http://logging.apache.org/log4j/2.x/.

[11] 2019. Centralize, Transform & Stash Your Data. https://www.elastic.co/products/logstash

[12] 2019. Go is an open source programming language that makes it easy to build simple, reliable, and efficient software. https://golang.org/

[13] 2019. HintedHandoff. https://wiki.apache.org/cassandra/HintedHandof.

[14] 2019. Simple logging facade for Java (SLF4J). http://www.slf4j.org/.

[15] 2019. What is Kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[16] 2019. Write Ahead Log (WAL). http://hbase.apache.org/book.html#wal.

[17] 2019. ZooKeeper Smoketest. https://github.com/phunt/zk-smoketest.

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

[19] Dhruba Borthakur et al. 2008. HDFS architecture guide. *Hadoop Apache Project* 53 (2008).

[20] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[21] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-scale Distributed Systems. In *Proceedings of the 2018 26th ACM SigSoft International Symposium on the Foundations of Software Engineering (FSE'18)*. ACM, New York, NY, USA, 539–550.

[22] Lars George. 2011. *HBase: the definitive guide: random access to your planet-size data.* " O'Reilly Media, Inc.".

[23] 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. 2011. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*. USENIX Association, Berkeley, CA, USA, 238–252.

[24] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Koushik Sen. 2010. Towards Automatically Checking Thousands of Failures with Micro-specifications. In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability (HotDep '10)*. USENIX Association, Berkeley, CA, USA, 1–8.

[25] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '14)*. ACM, New York, NY, USA, Article 7, 14 pages.

[26] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing?: Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 1–16.

[27] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. 2018. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)* 14, 3 (2018), 23.

[28] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 265–278.

[29] Chen Haicheng, Dou Wensheng, Jiang Yanyan, and Qin Feng. 2019. Understanding Exception-Related Bugs in Large-Scale Cloud Systems. In *Proceedings of the 34rd ACM/IEEE International Conference on Automated Software Engineering (ASE '19)*. ACM.

[30] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing in Situ System Observability for Failure Detection. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 1–16.

[31] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC '10)*. USENIX Association, Berkeley, CA, USA, 11–11.

[32] Pallavi Joshi, Malay Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. 2013. SETSUDŌ: Perturbation-based Testing Framework for Scalable Distributed Systems. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS '13)*. ACM, New York, NY, USA, 1–14.

[33] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. 2011. 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)*. ACM, New York, NY, USA, 171–188.

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

[35] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation (NSDI'07)*. USENIX Association, Berkeley, CA, USA, 18–18.

[36] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40.

[37] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi. 2015. SAMC: A Fast Model Checker for Finding Heisenbugs in Distributed Systems (Demo). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*. ACM, New York, NY, USA, 423–427.

[38] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Berkeley, CA, USA, 399–414.

[39] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 517–530.

[40] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-sensitive Points-to Analysis Using Value Flow. In *Proceedings of the 19th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '11)*. ACM, New York, NY, USA, 343–353.

[41] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and Scalable Context-sensitive Pointer Analysis via Value Flow Graph. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 85–96.

[42] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. 2017. 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)*. ACM, New York, NY, USA, 677–691.

[43] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. 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)*. ACM, New York, NY, USA, 419–431.

[44] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. D3S: Debugging Deployed Distributed Systems. In *5th USENIX Symposium on Networked Systems Design & Implementation (NSDI '08)*. USENIX Association, 423–437.

[45] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. 2007. WiDS Checker: Combating Bugs in Distributed Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation (NSDI '07)*. USENIX Association, Berkeley, CA, USA, 19–19.

[46] Jie Lu, Liu Chen, Lian Li, and Xiaobing Feng. 2019. Understanding Node Change Bugs for Distributed Systems. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 399–410.

[47] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. CloudRaid: Hunting Concurrency Bugs in the Cloud via Log-mining. In *Proceedings of the 2018 26th ACM International Symposium on the Foundations of Software Engineering (FSE'18)*. ACM, New York, NY, USA, 3–14.

[48] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, 1–16.

[49] Mohammad Reza Mesbahi, Amir Masoud Rahmani, and Mehdi Hosseinzadeh. 2017. Cloud dependability analysis: Characterizing Google cluster infrastructure reliability. In *2017 3th International Conference on Web Research (ICWR '17)*. IEEE, 56–61.

[50] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-consistency Bugs with Bounded Black-box Crash Testing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 33–50.

[51] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 26–26.

[52] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S Gunawi. 2019. {IASO}: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 47–62.

[53] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, 433–448.

[54] Jirí Simsa, Randy Bryant, and Garth A. Gibson. 2010. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV '10)*. USENIX Association, 1–8.

[55] Cesar A. Stuardo, Tanakorn Leesatapornwongsa, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, Wei-Chiu Chuang, Shan Lu, and Haryadi S. Gunawi. 2019. Scalecheck: A Single-machine Approach for Discovering Scalability Bugs in Large Distributed Systems. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Berkeley, CA, USA, 359–373.

[56] Yulei Sui and Jingling Xue. 2016. On-demand Strong Update Analysis via Value-flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. ACM, New York, NY, USA, 460–473.

[57] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 5, 16 pages.

[58] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting Large-scale System Problems by Mining Console Logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 117–132.

[59] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*. USENIX Association, Berkeley, CA, USA, 213–228.

[60] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 1–10.

[61] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. 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)*. USENIX Association, Berkeley, CA, USA, 249–265.

[62] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 19–33.

[63] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Berkeley, CA, USA, 603–618.

[64] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. Lprof: A Non-intrusive Request

Flow Profiler for Distributed Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Berkeley, CA, USA, 629–644.

[65] Long Zheng, Xiaofei Liao, Hai Jin, Bingsheng He, Jingling Xue, and Haikun Liu. 2018. Towards Concurrency Race Debugging: An Integrated Approach for Constraint Solving and Dynamic Slicing. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. ACM, New York, NY, USA, Article 26, 13 pages.