Understanding and Detecting Software Upgrade Failures in Distributed Systems

Yongle Zhang
yonglezh@purdue.edu
Purdue University

Junwen Yang
junwen@uchicago.edu
University of Chicago

Zhuqi Jin
zhuqi.jin@mail.utoronto.ca
University of Toronto

Utsav Sethi
usethi@uchicago.edu
University of Chicago

Kirk Rodrigues
kirk.rodrigues@mail.utoronto.ca
University of Toronto

Shan Lu
shanlu@uchicago.edu
University of Chicago

Ding Yuan
yuan@ece.utoronto.ca
University of Toronto

Abstract

Upgrade is one of the most disruptive yet unavoidable maintenance tasks that undermine the availability of distributed systems. Any failure during an upgrade is catastrophic, as it further extends the service disruption caused by the upgrade. The increasing adoption of continuous deployment further increases the frequency and burden of the upgrade task. In practice, upgrade failures have caused many of today's high-profile cloud outages. Unfortunately, there has been little understanding of their characteristics.

This paper presents an in-depth study of 123 real-world upgrade failures that were previously reported by users in 8 widely used distributed systems, shedding lights on the severity, root causes, exposing conditions, and fix strategies of upgrade failures. Guided by our study, we have designed a testing framework *DUPTester* that revealed 20 previously unknown upgrade failures in 4 distributed systems, and applied a series of static checkers *DUPChecker* that discovered over 800 cross-version data-format incompatibilities that can lead to upgrade failures. *DUPChecker* has been requested by HBase developers to be integrated into their toolchain.

CCS Concepts: **• Computer systems organization → Availability**; **• Software and its engineering → Software testing and debugging**; *Automated static analysis.*

Keywords: upgrade failure, distributed systems, study, bug detection

ACM Reference Format:
Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–28, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483577

1 Introduction

Internet services today live on distributed systems. Distributed system software upgrade is one of the most disruptive maintenance tasks. No matter with a *full-stop upgrade*, where the whole service goes down and then restarts with every node running a newer version of the software, or a *rolling upgrade*, where nodes take turns to go down and then restart, the internet service would suffer a not-available or partially-available period. Yet, software upgrade is unavoidable as vendors need to add new features, improve performance, and deploy patches. With the rise of continuous deployment [52] in the industry, the frequency of distributed system software upgrade could reach thousands of deployments in a single day [72] in a major Internet company.

Unfortunately, distributed systems could experience failures during software upgrade. In this paper, we define *software-upgrade failures* as those failures that *only* occur during software upgrade. For example, they may be triggered by interaction between two code versions of the same software or between an upgrade operation and a regular software operation, and hence never occur under regular execution scenarios. They are *not* caused by failure-inducing configuration changes [66, 76], which do not involve software code upgrade, or bugs purely about the new version of the software [79], which can be triggered on a fresh installation of the new version and do not require an upgrade scenario.

Software upgrade failures have unique properties that make them particularly problematic:

Large Scale. Upgrade is typically performed on the entire system. Consequently, an upgrade failure can often paralyze the entire cluster. For example, when Windows Azure failed to upgrade to a new version in 2014, the failure brought down all third-party websites hosted on Azure as well as Microsoft's own services including Xbox Live and the Windows Store [25]. Section 3 shows that 28% of the software upgrade failures we studied bring down the entire cluster.

Vulnerable Context. During upgrade, the system has to go through a no-service (full-stop upgrade) or partial-service (rolling upgrade) period. Failures under this context are particularly difficult to mask. They can greatly aggravate the service disruption caused by the upgrade operation itself, and severely affect vendors' reputations. For example, on February 29th, 2012, Azure's service went down after it hit the leap-day bug [28]; in an effort to resolve the issue, developers deployed an upgrade that broke compatibility with a network plugin, causing another three-hour outage.

Persistent Impact. Upgrade often involves the old version and the new version handing over information through persistent storage. Consequently, an upgrade failure can corrupt system state persistently that cannot be easily recovered by rolling back to the old version. For example, in 2014, an upgrade caused severe data loss at Dropbox, which brought the site down and took around two days to fully restore [10].

Difficult to Expose in House. Exposing upgrade failures requires running two versions of software side by side (rolling upgrade) or one right after the other (full-stop upgrade) [1]. This is not supported by traditional testing—the state-of-practice testing framework [18] used by modern distributed systems only runs test cases against a single version of the system. Similarly, it goes beyond most traditional bug detectors that focus on bugs inside one version. Careful canary deployment [3] can potentially expose upgrade failures. However, without testing support, it is often unaffordable in terms of resources and time [2] when an urgent upgrade is requested.

Indeed, a large portion of real-world catastrophic service outages were caused by upgrade failures. For example, out of the five postmortem reports published on the Microsoft Azure Blog [24], which only describes the most severe outages in Azure history, three involve software upgrade failures. Our study also found the portion of severe failures to be much higher in upgrade failures than that in non-upgrade failures (Section 3). The importance of upgrade failures will only increase with the wider adoption of continuous deployment.

In the past, while many have studied system failures [41, 54, 62–64, 66–68, 70, 73, 75, 78, 80], none of them offered detailed analysis on software upgrade failures, partly due to the challenge in collecting them. Despite their importance, software upgrade failures only consist of a small percentage of all system failures and hence rarely show up when one takes a random sample. For example, among the 198 randomly sampled distributed system failures studied by a previous work [80], only 7 are upgrade failures.

To the best of our knowledge, only two prior studies, both of which focus on more severe types of failures, briefly discussed upgrade failures. Gunawi et al. [46] studied 597 publicly available postmortem reports about cloud service outages; they found that 16% of these failures involve hardware or software upgrades. Liu et al. [61] studied 112 high-severity incidents from a Microsoft Azure production cluster and found that 21% were caused by incompatible data-format assumptions, which sometimes are caused by software upgrade. Neither study investigated further to provide a detailed analysis of software upgrade failures.

This paper provides the first in-depth analysis on software upgrade failures based on 123 real-world upgrade failures from widely-used distributed systems, including Cassandra, the Hadoop Distributed File System (HDFS), Hadoop MapReduce, Hadoop YARN, HBase, Kafka, Mesos, and ZooKeeper. For each case, we carefully analyzed its report and source code to thoroughly understand the symptom severity, the root cause, and the propagation chain in between.

Symptom-severity study. We find that upgrade failures are significantly more critical than non-upgrade failures. 38% of the software upgrade failures that we study are marked with "Blocker" severity compared to only 10% for non-upgrade failures. We also find that 67% of the software upgrade failures affect all or a majority of users. In comparison, only 24% of all failures have such a catastrophic effect based on a previous study [80]. Finally, we find that only 37% of the software upgrade bugs were caught before corresponding versions were released to public, with the majority (63%) exposed in production. More details are in Section 3.

Root-cause study. About two thirds of software upgrade failures are caused by incompatible interaction between two software versions. The interaction occurs through either persistent data (60%) or network messages (40%), with the latter being a particular concern during rolling upgrades. Incompatible assumption about data syntax or semantics causes one version to fail to parse (about two thirds of the incompatibility) or handle (about one third of the incompatibility) storage files or network messages generated by another version. Our detailed study provides guidance on how to automatically detect incompatibilities and how to avoid cross-version incompatibilities during programming. More details are presented in section 4.

Triggering-condition study. By definition, all of these failures require a full-stop upgrade (57% of the cases) or a rolling upgrade (43% of the cases) to expose, which is not part of today's standard testing framework [18]. Fortunately, our study revealed two unique characteristics of upgrade failures that answer key questions for the design of upgrade testing:

[1]Note that, all the failures that can occur during full-stop upgrades can also occur during rolling upgrades.

1. What software versions should be tested? Although in theory there can be a polynomial number of combinations of software versions to test, we find that more than 80% of the failures can be exposed by running two consecutive major/minor versions, and another 9% by running system instances with just a two-version gap.
2. What workload should be tested? Different from traditional production failures that by definition cannot be triggered by existing in-house test inputs, more than 70% of upgrade failures *can* be triggered by running workloads that already exist in these systems' stress test or unit test cases under an upgrade scenario, given the right version combination. More details are in Section 5.

Tackling upgrade failures. Guided by our triggering-condition study, we built an upgrade testing framework *DUPTester* and applied it to 4 real-world distributed systems. *DUPTester* adapts and utilizes existing stress testing and unit test cases of each distributed system to systematically test the system (full-stop and rolling) upgrade procedure. *DUPTester* reveals 20 previously unknown upgrade failures.

Guided by our root-cause study, we designed a static checker *DUPChecker* to search for incompatibility on data of `enum` types and data defined using serialization libraries. *DUPChecker* identified more than 800 previously unknown incompatibility problems between actively maintained versions of 7 distributed systems, with close to 300 of them confirmed by developers based on our reports. We also got the request from HBase developers to integrate *DUPChecker* into their tool chain. Note that, after developing *DUPChecker*, we noticed that similar incompatibility checkers for serialization libraries already exist in open-source world [26, 29]. The fact that these existing tools have not been used by developers of these systems and that incompatibility problems still widely exist in these popular systems led us to realize the limitations of static incompatibility checkers, including *DUPChecker*, which we discuss in Section 6.2.

DUPTester and *DUPChecker* well complement each other: *DUPTester* can expose upgrade failures with all types of root causes and clearly demonstrate the symptoms of every upgrade failure, as long as the triggering workload is covered; *DUPChecker* does not rely on workload design, but only tackles a specific type of root cause and cannot predict the exact symptom of the potential upgrade failure. Future research can work on combining the strengths of these two tools. All our code and failure-study data will be released.

2 Methodology

Cassandra	HBase	HDFS	Kafka	MapReduce	Mesos	Yarn	ZooKeeper
44	13	38	7	1	8	8	4

Table 1. Numbers of upgrade failures we analyzed.

We choose popular data-intensive distributed systems as our targets, including HDFS distributed file system [47], Hadoop MapReduce data-analytic framework [48], Mesos and Hadoop YARN application resource management frameworks [37, 49], HBase and Cassandra key-value stores [34, 35], ZooKeeper in-memory key-value store [38], and Kafka, a stream-processing service [36].

We selected the set of failures from the issue trackers of the mentioned distributed systems. In the issue tracker, among issues before 2018, we search for *resolved* and *valid* bugs whose titles contain the word "upgrade", and among issues after 2018, we search for *resolved* and *valid* issues whose entire report contains the keyword "upgrade" so that we can get more representatives from more recent systems. We then repeatedly perform rounds of random selection. In each round we random select one case from each system and manually check their reports to exclude non-upgrade failures, such as those purely about the new version. In the end, we get 123 failures as shown in Table 1. During this procedure we exhausted the filtered issues from MapReduce, Mesos, Yarn, and ZooKeeper.

Threats to Validity. As with all characteristics studies, the results of our study should be interpreted with the following limitations in mind.

(1) Representativeness of bug reports from issue trackers. There could be upgrade failures encountered by users that are not reported to the issue tracker, particularly since some upgrade failures could be mitigated by downgrading to an old version, or fully restarting the entire cluster (at the expense of longer service down time). In addition, the issue tracker covers upgrade failures encountered during unit testing, integration testing, staging, canary deployment, etc. Upgrade failures detected in one of these stages could show different characteristics.

(2) Limitations of the filtering criteria. We could have missed real upgrade failures whose issue report do not contain the keyword "upgrade". We did try other keywords like "update", and found that the resulting issue reports contain many more false positives (i.e., non-upgrade failures). We settled down on our filtering methodology because we favor reducing false positives over reducing false negatives—there should be more upgrade failures out there, which we will not have time to fully study any way.

(3) Representativeness of selected distributed systems. We attempt to study a wide variety of popular open source distributed systems: the 8 studied systems cover different types of functionalities, different architectures (master-worker versus peer-to-peer), and different languages (Java and C++). However, without accurate market information it is difficult to conclude whether we have chosen the most widely-used open source distributed systems. In addition, closed-source distributed systems could have different characteristics.

(4) Possible observer errors. There is a risk of observer errors. To minimize the effect, each failure was investigated independently by at least two inspectors. All inspectors used the same detailed written classification methodology, and any disagreement is discussed in the end to reach a consensus.

3 Severity of Upgrade Failures

3.1 Priority Assigned by Developers

To understand the severity of upgrade failures, we examined the priority field of each bug ticket. This is a required field in the issue tracker of every system we studied. It gets an initial assignment from the bug reporter and its final assignment by developers of the target system after the failure is fully diagnosed. In all systems except for Cassandra, there are five priority settings with decreasing severity and urgency: Blocker, Critical, Major, Minor, and Trivial. Cassandra's issue tracker uses three settings: Urgent, Normal, and Low.

For comparison, we also checked the priority assignment of non-upgrade bugs. To obtain them, we first search for all the *resolved* and *valid* bugs from each distributed system. We further remove the set of upgrade failures.

Finding 1. *Upgrade failures have significantly higher priority than regular failures.*

The percentage of high priority bugs among upgrade failures is significantly higher than that in non-upgrade failures. The percentage of Blocker bugs, the most severe and urgent ones, is 3.8X in upgrade failures than that in non-upgrade failures (38% versus 10%). The percentage of high priority bugs, including Blocker and Critical, rises from a small portion of 20% in non-upgrade failures to a majority of 53% in upgrade failures. The comparison in Cassandra shows a similar trend. Among upgrade bug reports in Cassandra, 18% of them have the highest priority, Urgent, and only 7% have the lowest priority, Low. In comparison, among non-upgrade bug reports in Cassandra, only 6% have the Urgent priority, and yet as many as 41% have the Low priority.

3.2 Symptoms of Upgrade Failures

To understand why upgrade failures receive significantly higher priority compared to non-upgrade failures, we further check the symptom of each upgrade failure.

Finding 2. *The majority (67%) of upgrade failures are catastrophic (i.e., affecting all or a majority of users instead of only a few of them). This percentage is much higher than that (24%) among all bugs [80].*

Here, we use the same definition of catastrophic failures as used in a prior study [80]. That prior study analyzed all failures in Cassandra, HBase, HDFS, Hadoop MapReduce, and Redis [71], and found that only 24% of them are catastrophic.

As shown in Table 2, 34 out of 123 (28%) upgrade bugs brought down the entire cluster. In particular, 22 of them brought down all of the worker nodes (or all peer nodes in Cassandra which uses a peer-to-peer design). For example, in MESOS-3834 [23], the old version did not include an ID field in its checkpoint, while the new version always assumed the existence of an ID in every checkpoint. As a result, the upgrade brought down every agent (worker) node when the new version running in the agent node fails to find the ID in the checkpoint saved by the old version. The other 5 whole-cluster failures occurred because the master node crashed. Note that these systems all contain a High Availability (HA) feature that is supposed to tolerate a master node crash by automatically failing over to a secondary master. However, because these failures are deterministic, they immediately crashed the secondary master after the fail over.

Sometimes (18 cases), the symptom only occurs during the period of rolling upgrade[2]. These failures do not corrupt persistent data or bring down the whole cluster, and the service can go back to normal when the rolling upgrade finishes. However, the damage is still huge, as a rolling upgrade can take long time and all failures of this type turned out to be catastrophic— with new nodes unable to work with the old nodes, the partitioned cluster suffers severe quality degradation for a long time. For example, in CASSANDRA-4195, developers changed the UUID version that is part of the gossip networking protocol in a new version of Cassandra. During a rolling upgrade, accepting gossip from nodes running the new version causes old-version nodes to perform schema migration. These old-version nodes then run into a schema ID parsing error. With all the old-version nodes stuck in the process of schema migration, the whole cluster stops working. Note that, although these failures would not have happened during full-stop upgrades, it is impractical to force system administrators always using full-stop upgrades without knowing the existence of a rolling-upgrade bug.

Notably, upgrade bugs can also cause catastrophic data loss. For example, in HDFS-5988 [15], if the cluster was upgraded from a version using an older filesystem checkpoint, not supporting inode, to the new checkpoint format, the entire filesystem would become inaccessible by the upgraded cluster. Specifically, when the new version loads a fsimage, it checks if the version generated the fsimage supports inode. If not, it proceeds to load and parse the fsimage, and create all the files checkpointed in the fsimage, except that it skips populating the inode map. The new version later performs checkpoint in its own fsimage format, skipping the inodes since the inode map is not populated. Eventually, loading this new fsimage fails with all the files lost, because loading fsimage in the new version cannot work without inodes.

Upgrade bugs can also cause catastrophic performance degradation. For example, in CASSANDRA-13441 [6], when a node with Cassandra 3.0 upgrades, it updates the timestamp of system tables in the schema, which causes all other nodes

[2]Failures caused by rolling upgrade do not always belong to this category, as they may cause data corruption or whole cluster down.

Symptom	All	Catastrophic	Catastrophic in Production
Whole cluster down (all nodes crash, master node crash)	34	34	18
Severe service quality degradation during rolling upgrade	16	16	10
Data loss and data corruption	20	15	12
Performance degradation (increased latency, wasted computation, etc.)	10	4	4
Part of cluster down (part of worker nodes down, secondary master down)	12	7	3
Incorrect service result (failed read/write requests, UI error, etc.)	24	6	4
Unknown*	7	–	–
Total	123	82	51

Table 2. Symptoms of failures observed by end-users or operators. The last column shows the number of catastrophic failures caught after software release. *: Developers filed these two reports without explaining the failure symptom.

in the cluster to perform schema migration. This process repeatedly occurs whenever a node performs upgrade, and eventually leads to millions of migration tasks, each resulting in a flood of network traffic bouncing across the cluster.

There are 7 upgrade failures that bring down part of the cluster. These failures require some special conditions to be triggered on each node. They can still be catastrophic when the condition is met in a large number of the nodes. For example, in HDFS-8676 [16], each DataNode deletes its trash directory synchronously at the same time after the upgrade operation finalizes. When the trash directory is large, the deletion operation takes a long time and thus delays the DataNode's heartbeat with NameNode, which is the master, causing the NameNode to mark the affected DataNodes as dead. In the issue report, the reporter experienced NameNode losing hundreds of DataNodes.

In addition to the detailed symptom breakdowns above, we also checked how many upgrade failures have easily observable symptoms like node crashes or fatal exceptions, instead of subtle symptoms, like periodic performance degradation or random packet loss [50, 51, 57].

Finding 3. *Most (70%) upgrade failures have easy-to-observe symptoms like node crashes or fatal exceptions.*

This finding indicates that many of these upgrade failures have the potential to be identified through automated testing without sophisticated testing oracles.

3.3 When were Upgrade Failures Caught?

To understand how many upgrade bugs leaked into released code, we compared the creation date of each bug report against the release date of the affected new version (i.e., the destination of the upgrade). For example, in MESOS-3834 [23], the failure happened during an upgrade from version 0.22 to 0.24. It is categorized as a caught-after-release bug, because MESOS-3834 was filed after the release date of Mesos 0.24. We excluded 11 issues whose reports do not contain user-reported version information. Among the remaining 112 upgrade failures, only 42 were caught before release, while the remaining 70 escaped into production code.

Finding 4. *The majority (63%) of upgrade bugs were not caught before code release.*

Syntax	data type	data defined using serialization lib.	7
		enum	2
		system-specific data	41
Semantics		mishandling of serialization lib.	6
		incomplete version handling	16
		other semantics issue	5
		total	77

Table 3. Incompatible cross-version interaction categories

4 Root Causes of Upgrade Failures

We categorize the root causes of software upgrade failures into four types: incompatible cross-version interaction (63%), broken upgrade operation (33%) misconfiguration (3%), and broken library dependency (2%). Our analysis of root cause patterns guides the design of our static checker - *DUPChecker* - described in Section 6, and suggest good practices that could avoid many failures.

Finding 5. *About two thirds of upgrade failures are caused by interaction between two software versions that hold incompatible data syntax or semantics assumption.*

4.1 Incompatible cross-version interaction.

We further categorize incompatible cross-version interaction along two dimensions: (1) what type of data is the target of the failure-causing incompatibility—persistent data or transient network data. (2) what type of incompatible assumption the two versions hold: data syntax incompatibility, which leads to data parsing errors, or data semantics incompatibility, which leads to data processing errors.

Along the first dimension, we found that 60% of incompatibility is centered on persistent storage data, while 40% is centered on network messages. The former could lead to failures during both full-stop upgrades and rolling upgrades. For example, the MESOS-3834 bug and the HDFS-5988 bug discussed in Section 3.2 are both caused by storage data incompatibility. The latter only manifests during rolling upgrades. For example, the failures in CASSANDRA-4195 and KAFKA-7403 both belong to this type.

Along the second dimension, we found that close to two thirds of incompatibility is about syntax difference of the same data in different versions, and the remaining one third is about semantic difference. Since the different categories along this dimension provide direct guidance to future work in detecting and fixing upgrade bugs, we discuss each category in details below.

4.1.1 Data syntax incompatibility.

At the first glance, syntax incompatibility should be easy to catch than semantics incompatibility. In practice, this depends on whether the corresponding message/file data syntax is clearly defined.

Finding 6. *Close to 20% of syntax incompatibilities are about data syntax defined by serialization libraries or enum data types. Given their clear syntax definition interface, automated incompatibility detection is feasible.*

Serialization library data. To avoid the hassle of writing error-prone serialization/deserialization functions, developers often adopt declarative serialization libraries such as Google Protocol Buffers [26]. Using a serialization library, developers could declare a data format like declaring data members in a class and the (de)serialization functions will be generated automatically.

One common problem is that the new version adds a *required* data member for a class. Then, during a rolling upgrade, the serialization library in a new node cannot find this data member in the data generated by an old node and triggers an upgrade failure. For instance, in HDFS-14726 [12], developers added a new *required* data member - `committedTxnId` - which causes exceptions during rolling upgrade. The fix for this type of problems is to change the new data member from *required* to *optional*, as suggested by almost all serialization libraries [26, 30].

Enum. There were also syntax incompatibilities on data of enum types. In HDFS-15624 [13], developers used index to serialize and deserialize values in the `StorageType` enum. When they added a new storage type NVDIMM in the middle of the enum class, later members in `StorageType` all have their indices incremented by one causing incompatibility if any of them is used in communicating with the old version. A good practice adopted by developers is to pad the enum class with enough placeholder values so that inserting a new member does not change other existing members' indices.

As we can see, the above problems have clear patterns and are feasible to automatically detect and fix, which we will practice in Section 6.2.

Finding 7. *Most (about 80%) data syntax incompatibilities are caused by missing or incomplete deserialization functions for system-specific data.*

System-specific data. There is still much message/file data that is not defined by serialization libraries due to backward compatibility concerns or due to its complexity. They require developers to provide (de)serialization functions.

17 failures occurred because developers did not anticipate receiving data defined in a slightly different syntax and the software aborted without corresponding deserializer. For example, in CASSANDRA-4195 the old node receives an `ApplicationState` message with a UUID field that it does not know how to deserialize. These problems are fixed by either checking the version and rejecting the received data (as in CASSANDRA-4195), or implementing a correct deserialization function (often for deserializing legacy data).

In 24 cases, developers actually were aware of and did implement deserialization functions for different syntax. Unfortunately, the deserialization function is buggy and cannot parse the target data.

To detect missing-deserializer problems, the main challenge is to figure out what data is to be serialized and how its syntax is defined, which is feasible but complicated without the standard interface provided by serialization libraries. A good defensive practice is to insert version IDs in any data that is written to storage or sent over network and always check the version ID in the deserialization function.

To detect incomplete-deserializer problems, one can potentially generate unit tests that target developers' deserialization functions.

4.1.2 Data semantics incompatibility.

The remaining one third of cross-version incompatibility is about data semantics (27 cases).

A number of them are about data using serialization library—although the library correctly serializes and deserializes data, the exact meaning of the data is interpreted differently between the two versions. An example from KAFKA-7403 [21] where a client runs Kafka version 0.11 and a broker runs Kafka version 2.1.0. When the client sends an `OffsetCommitRequest` to the broker, it sets the `retentionTime` field to be `DEFAULT`. This message is correctly parsed by the broker, but the `DEFAULT` setting triggers a semantics error in the broker's message handling. Broker version 2.1.0 assumes a `DEFAULT` setting of the `retentionTime` field to mean that `expireTimestamp` will not be used and hence it sets `expireTimestamp` to be `None`. Unfortunately, this semantics assumption is not held by older versions of the software. Eventually `expireTimestamp` is used with an invalid `None` setting, and the system fails.

Finding 8. *Close to two thirds of of data semantics incompatibilities are caused by incomplete version checking and handling.*

Many semantics incompatibilities are caused by version checking and handling problems. Incorporating a version identifier, typically an integer number, in the data being exchanged and checking it during deserialization is a common strategy to avoid cross-version incompatibility. However, it is difficult to conduct proper version handling.

We summarize the following lessons and good-practice suggestions about how to properly conduct version checking and version handling by studying developers' patches and discussions. Adopting these practices would prevent most of these incomplete version handling cases.

(1) If version checking is used to manage cross-version interaction, every version of source code should have their own distinct version identifier. For example, KAFKA-10173 [19] happened because developers changed a data format but did not change corresponding version identifier.

(2) Enough room should be left between consecutive version identifiers in case a new software version is released in between existing versions. In CASSANDRA-5102 [7], a developer complained that "I think it highlights a deeper problem, which is that if we ever do need to do another protocol bump in a minor, stable branch, we're out of luck because there's no space between VERSION_117 and VERSION_12". A possible approach is to map a version number's major, minor, and bug-fix digits to the first, second, and fourth byte respectively in its version identifier integer.

(3) Each version identifier should be mapped to a set of features that are supported in the parsing logic. For example, HDFS stores a snapshot as an image file called fsimage with a version identifier called LayoutVersion. In HDFS-1936 [14], because developers bumped up HDFS 0.20's LayoutVersion from 19 to 31, HDFS 0.20 starts generating fsimage with LayoutVersion 31. However, HDFS 0.20 does not perform fsimage compression which is expected for any LayoutVersion larger than 24. Thus, when a later version tries to deserialize the fsimage generated by HDFS 0.20, it checks the LayoutVersion (31) and expects the fsimage to be compressed, but run into deserialization error. This example shows that it is hard to keep track of the semantic meaning of each version identifier and handle each version identifier differently in the parsing logic. Instead, developers propose to decouple supported features in the deserialization function using feature sets and each version identifier can be mapped to a set of such features. For instance, HDFS 0.20 can be mapped to `{edits_checksum, image_checksum, ...}`, and the deserialization function invokes parsing logic corresponding to a feature only if it is contained in HDFS 0.20's feature set.

(4) If version checking is used to manage cross-version interaction, the version ID should be used in all messages that may be exchanged cross versions. For example, in CASSANDRA-6678 [8], during a rolling upgrade from Cassandra 1.2 to 2.0, an upgraded node re-joins the Cassandra cluster and sends a gossip message to other nodes running Cassandra 1.2. Upon receiving such a gossip message, a Cassandra 1.2 node N1 checks the version identifier of the upgraded node N2 and sends a pull schema request if the check passes. This check should fail because schema migration is forbidden during rolling upgrade to avoid schema conflict. However, the version identifier of N2 is only obtained when Cassandra's MessagingService establishes a connection between N1 and N2. If a race condition happens and N1 receives the gossip message before the MessagingService connection is established, it will treat N2 as a Cassandra 1.2 node and the check will pass. The fix to this problem is to simply add the version ID into the gossip message, so that version checking can be easily conducted.

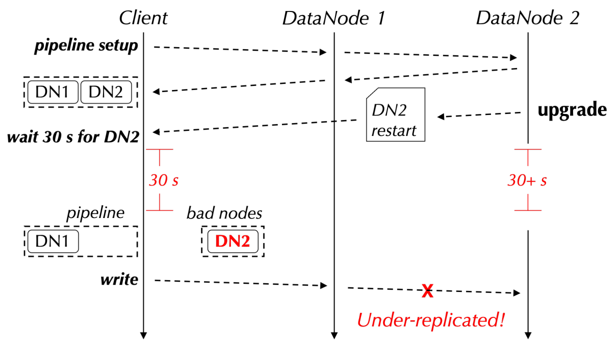

Figure 1. HDFS-11856 [11]: Blocks are under-replicated as upgraded nodes are incorrectly marked as bad permanently.

4.2 Broken upgrade operation.

About one third of the failures are caused by unexpected interaction between the upgrade operation and specific regular operations of the system. HDFS-11856 [11] is such an example, illustrated in Figure 1. when the system is performing a rolling upgrade, a HDFS client sets up a write pipeline with DataNode 1 and DataNode 2, so that any newly written data is first sent to DataNode 1 and then replicated to DataNode 2. Later on, DataNode 2 starts its upgrade, as part of a rolling upgrade procedure. It notifies the client about its up-coming restart. The client handles any unreachable data node by waiting for 30 seconds, which is enough for a non-upgrade restart or network glitches. However, an upgrade operation typically takes more than 30 seconds. This longer-than-expected period of unavailability caused the client to mark DataNode 2 as a bad DataNode. If there are no other DataNodes in the cluster, any newly written data will be under-replicated, which could result in data loss.

4.3 Others

4 cases happen when a configuration worked in the old version but no longer works in the new version. All of these 4 cases were fixed by adjusting the configuration setting and developers improved documentation for the upgrade procedure accordingly. Note that, they are different from failure-inducing configuration changes [66, 76], as it was the lack of change that led to failures.

Broken dependency occurs when the distributed system stops working with a library after an upgrade of itself or the library. They are rare in our study.

5 Triggers of Upgrade Failures

This section discusses the findings on the opportunities for automated testing to detect upgrade failures. It guides the design of our testing framework - *DUPTester* - described in Section 6.

5.1 Triggering Version Combinations

A unique challenge faced by testing upgrade failure is that it requires two versions of the software. Theoretically this could be an expensive exercise as one needs to enumerate N^2 combinations where N is the number of different versions of the software. Therefore we study the minimum gap between versions that is required to trigger each failure.

Major Gap	2	1	0	0	0	0	any
Minor Gap	any	any	>2	2	1	<1	any
# of Upgrade Failures	3	37	3	8	31	6	32
percentage	2.5%	31%	3%	7%	26%	5%	27%

Table 4. Gaps between software versions required to expose the upgrade failures. "< 1": the two versions are different bug-fix versions within the same minor version. The last column includes failures that can be exposed by upgrading from any old version to a particular new version.

All studied systems adopt a three-digit release version numbering scheme in the form of `<major>.<minor>.<bug-fix>`. Therefore, we measured the gaps using major-version difference or minor-version difference, if the major version is the same. For example, the first row in Table 4 means that given a version X.Y.Z, upgrading to it from any release with major version (X-2) can trigger the upgrade failure. The total number of failures in the table is fewer than the total number of upgrade failures studied, because the reporter did not report versions in one case.

Finding 9. *All but 14 upgrade failures can be triggered by consecutive major or minor versions.*

This suggests that instead of enumerating $O(N^2)$ version combinations, one only needs to test the $O(N)$ combinations of *consecutive versions* to expose a vast majority (over 80%) of the upgrade failures. In addition, another 9% of upgrade failures can be exposed by testing versions with gap 2 (either major version gap or minor version gap).

An interesting fact is that logically one would expect that there would be more upgrade failures between major versions since a major version implies a significant change. However, most bugs are between minor versions.

5.2 Triggering Workload

Number of nodes. A unique question for distributed systems is how many nodes are needed to expose failures. We checked all the 123 failure reports that have this information. In theory, the number of node types needed to expose failures is also important. However, since most of the studied systems only have 2 to 3 types of nodes, we consider the number of nodes as the main complexity contributor.

Finding 10. *All of the upgrade failures require no more than 3 nodes to trigger.*

More than half (57%) of upgrade failures only require a single (worker) node to trigger (when the system has a master-worker architecture, we only count the number of worker nodes). They are mostly caused by incompatibility on persistent data between versions. To expose these failures, one only needs to first start the node with the old version to generate persistent data, and then upgrade the node to a new version to load the data. Another 43% of upgrade failures require 2 nodes. Some of them are caused by incompatibility on network messages, and hence require at least two nodes to trigger, one running an old version and the other running the new version. Others involve communication between the master and the secondary master, or between worker nodes that serve different roles like the HDFS failure that requires multiple nodes to form a write pipeline. There is only one case that requires 3 nodes to trigger: ZOOKEEPER-1805[31]. Here, a rolling upgrade operation interferes with ZooKeeper's leader election, and the failure can only be triggered when two nodes send different `peerEpoch` to a third node that is in the middle of an upgrade operation.

Timing. Distributed systems are inherently concurrent and non-deterministic. Since non-deterministic bugs are notoriously difficult to expose [63], a natural question is whether upgrade failures are deterministic. Fortunately, we find that:

Finding 11. *Close to 90% of the upgrade failures are deterministic, not requiring any special timing to trigger.*

In other words, as long as the required failure-triggering workloads are performed with the right combination of software versions, these failures are guaranteed to manifest—a testing tool does not need to explore different timing.

The remaining 11% failures are non-deterministic. The previous HDFS-11856 [11] shown in Figure 1 is such an example. It requires the upgrade to occur on a DataNode after the write pipeline is formed.

Operations and configurations. We next analyze what operations, in addition to the upgrade itself, and configurations are needed to expose the failures. Since every distributed satem that we study comes with (1) a default workload generator that generates most common system operations like read and write for stress testing purpose; (2) a large set of unit tests, we particularly check whether the failure-triggering operations are part of the stress-testing workload or unit tests, or neither—if neither, much manual effort is needed to design testing operations to trigger upgrade failures.

Finding 12. *Half of upgrade bugs can be triggered by stress-testing operations with default configurations.*

This is a great news: simply running the stress testing in the context of software upgrades with default configuration can expose these failures. For example, CASSANDRA-4195 can be triggered by running rolling upgrade on a 2-node cluster with default configuration.

For the remaining failures that require special configurations or operations or both, there is still an opportunity to expose them without extra manual effort in test-case design.

Finding 13. *7% of upgrade failures need non-default configuration to be triggered. Fortunately, most of these non-default configurations (78%) are covered in existing unit tests.*

About one third of upgrade failures need special operations to trigger. Fortunately, about 60% of them are already covered by existing unit tests or can be covered by new combinations of existing tests.

For example, in KAFKA-6238 [20], a configuration file from the old version is used to start a new version, and the user did not update a conflict value in the field `message.version`, which crashed all the upgraded brokers. This field and value exist in one of Kafka's unit tests and could have been used to expose the failure. As another example, in CASSANDRA-10652 [4], upgrade failure happened because a table `system_traces` generated by Cassandra's tracing tool is not properly upgraded. Although the tracing functionality is not run by default in Cassandra's stress testing, it is tested in Cassandra's unit test `test_cqlsh_completion`.

5.3 State of the Art of Upgrade Testing

Our findings in this section suggest that there exist opportunities to expose software upgrade failures through automated testing, which system developers may not be aware of.

Among the 8 systems we studied, Cassandra, Mesos, Kafka, and HDFS contain some limited testing scripts for upgrade operations. Cassandra and Kafka's upgrade testing uses multiple Java Virtual Machines (JVM) [17] to mimic multiple nodes in a cluster that run different software versions and communicate with each other. Mesos uses multiple processes to mimic multiple nodes in a similar manner. HDFS uses JUnit testing framework for a more limited upgrade testing: instead of running two versions, it prepares a limited set of pre-defined filesystem images generated by older versions and tests whether the new version can work with these old images. Some of these images come *after* an upgrade failure is caught and fixed, and serve for regression testing purpose.

The percentage of upgrade failures that are caught-after-release in these four systems are actually *not* smaller than other systems. In fact, Cassandra has close to 80% of upgrade failures caught-after-release, the highest among all systems that we studied.

Based on our study, we see at least two key limitations of the upgrade testing schemes used by these 4 systems. First, they do not solve the problem of testing-workload generation. Particularly, their upgrade testing scripts are separate from their stress testing or unit testing framework, with testing workload designed from scratch, instead of leveraging the mature and much larger amount of workload already designed for stress testing and unit testing. In fact, in CASSANDRA-10822 [5], one developer said "Our coverage of upgrade scenario is really bad (as exemplified by this) and we need to fix that ASAP." Second, there are also no mechanisms for their upgrade testing to systematically explore different version combinations, different configurations, different upgrade scenarios (e.g., full-stop, rolling, adding new nodes during upgrade, etc.), etc. In KAFKA-10173 [19], one developer said "I've also just discovered that our system test that covers application upgrades had suffered an oversight that made it skip these versions."

6 Testing and Detecting Upgrade Failures

Guided by our study, we have designed *DUPTester*, short for Distributed system UPgrade Tester, to expose upgrade failures through in-house testing (Section 6.1, and *DUPChecker*, short for Distributed system UPgrade Checker, to detect upgrade failures caused by data-syntax incompatibility through static program analysis (Section 6.2). *DUPTester* has exposed 20 previously unknown upgrade failures in 4 distributed systems; *DUPChecker* has detected 878 potential incompatibilities in 7 distributed systems and is requested by HBase developers to integrate into their toolchain.

6.1 *DUPTester* Upgrade Testing Framework

6.1.1 Testing environment. To support efficient and systematic testing of a large number of workloads, version combinations, and upgrade scenarios, *DUPTester* packages what the target system (e.g., Cassandra) is supposed to run on each physical node, with all its dependencies, in a container [22]. *DUPTester* pre-loads these containers with many different versions to save the installation time. *DUPTester* simulates a 3-node cluster of the target distributed system on one single physical machine using a container orchestration tool [9]—based on our **Finding 10**, most upgrade failures can be triggered using 3 nodes or fewer. *DUPTester* preserves any persistent data generated by a container in a shared directory between each container instance and the host machine, so that the persistent data is accessible by other containers even after the container who generated the data shuts down.

For every pair of versions to test, *old* and *new*, *DUPTester* systematically tests three upgrade scenarios:

1. Full-stop upgrade. The cluster with *old* software executes a testing workload; after the workload is processed, the cluster gracefully shuts down; the cluster then restarts with every node using the *new* software.
2. Rolling upgrade. A rolling upgrade from *old* software to *new* software starts on a cluster; immediately next,

before the rolling upgrade ends, the cluster starts executing a testing workload.

3. New node joining. Several nodes running a *new* version joining a cluster of nodes running the *old* version, while the cluster executes a testing workload.

In all cases, upgrading a node is done by replacing its corresponding container with another container instance that (1) runs the newer version of the target system and (2) inherits the old instance' persistent data by sharing the same directories on the host machine. Following **Finding 2**, *DUPTester* treats error log message, exceptions, and crashes as indication for upgrade failures.

6.1.2 Testing workload. As discussed in Section 5.3, a main challenge facing all existing systems is to come up with workload for upgrade testing. Guided by our **Finding 12 & 13**, *DUPTester* will make the best use of existing *stress testing* and *unit testing*.

Leveraging *stress testing* is straightforward. Test cases in stress testing consist of client-side commands to be issued to a cluster. *DUPTester* directly uses them as testing workload in the three upgrade scenarios described above.

Leveraging *unit testing* is much harder. On one hand, widely-adopted distributed systems all have rich suites of unit tests, offering a precious source of testing workload. For example, the latest stable version of Cassandra (3.11.9) has 209,447 lines of source code and 122,418 lines of unit tests. On the other hand, a unit test invokes internal functions of a system-under-test from a specially designed test harness and hence cannot be directly used as testing workload during system upgrade.

To make use of unit tests, *DUPTester* designs two schemes.

One scheme aims to *translate* some unit tests into client-side scripts that can be used directly as testing workload for all three upgrade scenarios discussed above. We describe the *DUPTester* test translator in details in Section 6.1.3.

The other scheme executes each unit test *u* as it is in the *old* versioned system, and then checks if a cluster with the *new* versioned software can successfully start from the persistent state left by *u*. Since not all unit tests produce valid system-wide persistent states, *DUPTester* first restarts a cluster with the same version *old*. If the restart fails, *DUPTester* concludes that unit test *u* is invalid for full-system testing and moves on to the next unit test. This scheme is easy to carry out, but can only test full-stop upgrade scenarios.

6.1.3 *DUPTester* Unit test translator. To better leverage unit tests, *DUPTester* develops a test *translator* that transforms some unit tests into client-side Python programs.

At high level, *DUPTester* translator first constructs an abstract syntax tree (AST) for each unit test, and then synthesizes a Python program based on this AST.

During this process, *DUPTester* needs to handle situations where direct translation does not work. For example,

	Failure	From	To	C.?	Cause
Cassandra	15794	3.11.4	4.0	✓	Data-syntax Incomp.
	16258	3.11.6	4.0		Data-syntax Incomp.
	16301	3.11.9	4.0	✓	Code Incompatibility
	16292	3.0.0	3.2.0		Data-syntax Incomp.
	16257	2.1.0	2.2.0		Data-syntax Incomp.
	16264	2.0.0	2.1.0		Data-semantics Incomp.
	16265	2.0.0	2.1.0		Data-syntax Incomp.
	16266	2.0.0	2.1.0	✓	Data-syntax Incomp.
	16267	1.1.0	1.2.0	✓	Data-semantics Incomp.
	16268	1.1.0	1.2.0		Data-syntax Incomp.
	16269	1.1.0	1.2.0		Data-syntax Incomp.
HBase	25239	2.3.3	3.0		Broken Upgrade Op.
	24430	2.2	2.4		Broken Dependency
	24556	2.2	2.3	✓	Broken Dependency
	25238	2.2.0	2.3.3	✓	Data-syntax Incomp.
	25259	2.1.1	2.2.0		Broken Upgrade Op.
	25260	2.0.6	2.1.1		Broken Upgrade Op.
Kafka	10041	1.1	2.4	✓	Broken Dependency
Hive	24440	2.3.7	3.0.0		Data Syntax Incomp.
	24493	2.1.1	2.3.7		Upgrade Operation

Table 5. *DUPTester*'s result on real-world systems. Failure number is the report ticket number on JIRA. *C.?*: whether the bug is already confirmed by developers.

DUPTester replaces some Java library functions and classes with corresponding Python functions and classes, like replacing Java HashMap objects with Python dictionary objects, etc. In addition, *DUPTester* replaces some software-specific test methods or software internal functions with corresponding Python functions that can be issued from the client side. For example, for Cassandra, *DUPTester* replaces every invocation to a unit test method `CQLTester.execute(q)` with an invocation to `Cluster.Session.execute(q)` in the target Python program. The current prototype of *DUPTester* does not guarantee to translate all statements. When *DUPTester* cannot translate a statement *s*, it omits *s* and all the statements that depend on *s* from the resulting Python program.

We have developed such a unit test translator for Cassandra. To adapt the translator for other systems, developers mainly need to specify the mapping between those unit test methods or software internal functions, which are invoked inside unit tests, and external commands, which can be issued by the clients. We expect that few updates are needed to such a specification across code versions, as the interface of these functions and commands is typically stable, even though their internal implementation may change frequently.

6.1.4 Evaluation results. We used *DUPTester* to test the upgrade procedure among the recent release versions of 4 distributed systems: Cassandra, HBase, Kafka, and Hive. In addition to Cassandra, HBase, and Kafka, which are part of our upgrade failure study in earlier sections, we added Hive here to evaluate how well *DUPTester* performs in systems that we have not studied.

We picked the version gap to be either 1–2 minor versions or 1 major version guided by **Finding 9**. We carefully studied detected failures and filtered out those caused by operator errors. As shown in Table 5, *DUPTester* is able to find 20 previously unknown (to our best knowledge) upgrade failures in four distributed systems. 7 of them are already confirmed by developers. A majority of them (14 out of 20) are severe failures that crash the upgraded node. 10 out of 20 are caused by data-syntax incompatibility; 2 are caused by data-semantics incompatibility; 4 are caused by broken upgrade operation; 3 are caused by broken dependency. 2 cases (CASSANDRA-16301, CASSANDRA-16292) require special configurations or special operations that are not covered by the stress testing. Fortunately, *DUPTester* covered them by by utilizing *unit tests*. For Cassandra, we also compared the 11 upgrade failures found by *DUPTester* against its existing upgrade testing scripts. Based on our checking, they were missed by existing testing scripts mainly because the triggering workloads and configurations are not covered by existing testing scripts. We explain some upgrade failures discovered by *DUPTester* below.

CASSANDRA-15794 is a failure that not only prevents the system from successful upgrade but also successful downgrade. In this case, when Cassandra 3.11.4 is upgraded to 4.0, if the data generated by Cassandra 3.11.4 has a special form called COMPACT STORAGE, Cassandra 4.0 fails to start because it does not support COMPACT STORAGE. However, before it fails, Cassandra 4.0 already generated commit logs (Cassandra's Write Ahead Log). This effectively stops the user from working around the problem by downgrading the system to Cassandra 3.11.4, because Cassandra 3.11.4 does not recognize commit logs generated by Cassandra 4.0 and crashes.

CASSANDRA-16301 is discovered by loading the data generated by an existing unit test. If Cassandra 3.11.9 is started with a configuration called OldNetworkTopologyStrategy, when it is upgraded to Cassandra 4.0, the node crashes because OldNetworkTopologyStrategy is removed in Cassandra 4.0. This configuration is not tested in Cassandra's stress testing tool, but it is exercised in a unit test function called `testUpdateKeyspace` in Cassandra 3.11.9.

CASSANDRA-16292 is a case that can only be discovered using the unit test translator. Our translator successfully translated a unit test called `testCachedPreparedStatements` in Cassandra 3.10. It creates two Cassandra KeySpaces, creates one table in each KeySpace, and later drops one of the KeySpaces. Executing these translated commands in the upgrade procedure from Cassandra 3.0 to Cassandra 3.2 crashes Cassandra 3.2. *DUPTester*'s *stress testing* strategy does not trigger this failure, because the operation - `DROP KEYSPACE` - is not exercised in stress testing. In addition, the data generated by `testCachedPreparedStatements` cannot be used to perform upgrade, because this unit test uses internal functions and only generates corrupted data. This case also shows that translated unit tests could be applied to testing a wider range of versions, because `testCachedPreparedStatements` was only introduced in Cassandra 3.10 but could be used to test Cassandra 3.0 and 3.2.

HBASE-25238 is discovered by *DUPTester* when it applies the *stress testing* strategy between HBase 2.2.0 and 2.3.3. The upgraded node fails to start with an `InvalidProtocolBufferException`. The root cause is that the upgraded HBase master node tries to parse data generated and serialized by 2.2.0, but the parsing fails because the format of serialized data is changed between 2.2.0 and 2.3.3.

```
message ReplicationLoadSink {
  required uint64 ageOfLastAppliedOp = 1;
+ required uint64 timestampStarted = 3;
}
```

Figure 2. Incompatibility in data serialization protocols of HBase 2.2.0 and 2.3.3. The added a data member `timestampStarted` makes HBase 2.2.0 and 2.3.3 incompatible.

Figure 2 shows the usage of Protocol Buffers [26] in HBase 2.2.0 and 2.3.3 that declared the format of serialized data `ReplicationLoadSink`. It includes several data members such as `ageOfLastAppliedOp`. The `required` keyword means this data member must appear exactly once in the serialized data. However, HBase 2.3.3 adds a required data member in `ReplicationLoadSink` and breaks its compatibility with 2.2.0 causing the upgraded node to crash.

After we reported HBASE-25238, developers quickly raised its priority to "Critical", and we saw positive comments from HBase lead developers like "we need to be careful while merging PRs which modify existing proto message to avoid such issues in future" and "looks like we might need source compatibility report for protos".

False positives and negatives. DUPTester experienced four false positives. Three of them are because the stress testing tool is incompatible with the other version during rolling upgrade. One is because the upgrade procedure between specific versions has unique requirements that *DUPTester* did not follow. To evaluate false negatives, we apply *DUPTester* on 15 randomly sampled cases from our study, DUPTester can trigger 5 of them. Half of the false negatives are because unit tests translated by *DUPTester* do not cover parameters introduced by the new version for some software internal functions. The remaining ones require better input generation — the effectiveness of *DUPTester* is limited by the target system's stress testing and unit test suite.

6.2 *DUPChecker* Static Upgrade Bug Detector

Checker design Guided by our root-cause study in Section 4, we designed a static checker *DUPChecker* to search for two types of data-syntax incompatibility across versions: (type-1) on data defined by serialization libraries and (type-2) on data of `enum` types, as suggested by **Finding 6**.

Note that, we later found that similar incompatibility checkers for serialization libraries (type 1) already exist in open-source world [26, 29], although we could not find records about whether they have been applied to large distributed systems and what are the checking results if they have been. Therefore, our contribution in that front is mainly on applying the checkers to see how widely this type of problems exist in today's large distributed systems, which can then provide guidance for future research.

The first type is about data defined by serialization libraries. As discussed in Section 4.1.1 and Figure 2, a message/file format can be defined in a language-neutral protocol file using serialization library. When a new version changes a format in one of the following ways, upgrade failures could happen [26, 30]: 1) the tag number, which indicates the position of the data member in the serialized data, is changed; 2) a `required` data member is added or removed; 3) the `required` qualifier of a data member is changed to non-`required`, which may cause failures if the new version generates data that does not contain its no-longer-required data member, 4) there should be an 0 value in an `enum` if it has a data member added or deleted.

DUPChecker searches for such syntax incompatibility for popular serialization libraries, including Protocol Buffers and Thrift [30]. Specifically, *DUPChecker* creates a parser for protocol files by extending the PyParsing module [27] with the serialization library's grammar, which then allows *DUPChecker* to compare the data format of the same data member from different versions and report any incompatible change of the first two categories as an error, and any incompatible change of the third category as a warning. For example, *DUPChecker* easily detects HBASE-25238 (Figure 2), as it breaks the second rule.

System	Bug Ticket	# of ERR.	# of WARN.
HBase	HBASE-25340	7	23
HDFS	HDFS-15700	21	47
Mesos	MESOS-10202	8	12
YARN	YARN-10508	42	0
Accumulo	Accumulo-13946	20	0
Hive	HIVE-HIVE-24454	260	0
Impala	IMPALA-10370	342	96
	Total	700	178

Table 6. Bugs found by *DUPChecker* in 7 systems.

As shown in Table 6, we applied *DUPChecker* to 7 distributed systems, comparing every pair of currently maintained versions that are not declared as incompatible for each system. HBase, HDFS, Mesos, and YARN are all the systems in our initial failure study (Table 1) that use any type of popular serialization libraries: they all use Protocol Buffer library. We added Accumulo and Impala, which use Apache Thrift library extensively, and Hive, which uses Protocol Buffer library extensively, to see whether similar problems exist beyond the initial set of systems under study.

In total, *DUPChecker* found 878 incompatibilities, including 700 violations of category 1 and 2 (i.e., errors) and 178 violations of category 3 and 4 (i.e., warnings), as shown in Table 6. These are all previously unknown problems, never reported to these systems before we did. Particularly, a violation of category 1 or 2 is guaranteed to cause system failures, as long as the corresponding serializer and deserializer are executed. Developers from HBase and Impala quickly confirmed all our reports, acknowledging these indeed break version compatibility; HBase developers also requested *DUPChecker* to be incorporated to their toolchain. The other systems' developers have not responded. Finally, *DUPChecker* is also able to detect already known bugs caused by such incompatibility, such as HDFS-9788, IMPALA-8243, MESOS-2371, MESOS-3989, YARN-5632, etc.

False positives and negatives. In theory, *DUPChecker* can report a false positive when the new-version software check-and-rejects a message with a particular version number. Our manual checking of *DUPChecker*'s found no such false positives. By design, *DUPChecker* only searches for bugs that are related to the four types of changes to message/file formats using Protocol Buffers or Apache Thrift libraries, and is unable to detect other types bugs.

The second type is about incompatibility of `enum`-typed data, as discussed in Section 4.1.1. To detect such incompatibility, *DUPChecker* first identifies the `enum` class whose member's index has been written to a serialized output stream through data flow analysis. *DUPChecker* checks the parameters of the serialized output's write function to see whether they contain the index of any variable whose type is the *enum*. *DUPChecker* assumes if the index of the `enum`-typed variable is written, it could be any member from it. For serialized outputs, we currently only consider variables of `DataOutput` type in Java since they are the most common type used as output in serialization functions. And then *DUPChecker* checks whether the `enum` class has member addition or deletion across two versions. If so *DUPChecker* considers it as a bug. If the `enum` class doesn't have member addition or deletion, *DUPTester* treats it as vulnerabilities to future changes which should have comments to inform developers to preserve the order and add index range check.

We applied *DUPChecker* to the same set of applications as the first type and found 2 new bugs one which has already been confirmed and fixed by the developers, and 6 vulnerabilities, and 3 of them are already confirmed and fixed. In addition, *DUPChecker* is also able to detect already known bugs of this type, such as HBase-15624. In comparison, this type of incompatibility problems are much rarer than the first type, as there are much fewer uses of enum-typed data in messages and files than those using serialization libraries.

False positives and negatives. DUPChecker reports a violation if a data member is changed in a enum and *any* member of this enum is serialized. Thus, when only the members whose indices are not changed are serialized, *DUPChecker* could report false positives. In the 8 issues we reported to developers, 7 of them are already confirmed and fixed. For false negatives, by design, *DUPChecker* only checks serialized output of *DataOutput* type, so it is unable to detect enum classes serialized to other type of output.

7 Future Research Directions

Our study and tool evaluation aim to offer motivation and guidance for future research to tackle the challenging problem of upgrade failures along several directions.

Testing. *DUPTester* demonstrates that it is feasible to achieve better testing guided by the triggering conditions revealed in our study. However, relying on workloads in default stress testing and existing unit tests limits *DUPTester*'s ability in exploring the test space. Techniques such as fuzz testing [1, 44, 81] and symbolic execution [39, 40, 55, 74] could be applied to automatically explore the test space and trigger more upgrade failures.

Static analysis. *DUPChecker* demonstrates that applying static analysis on data format declarations could detect hundreds of incompatibilities. In addition to changed enum-typed data, our study also reveals upgrade failures caused by changed file names, changed default configuration values, missing deserialization functions, and changed constants. More static analysis techniques could be developed to detect these incompatibilities in the future, but will face challenges like identifying data that affects (de)serialization through inter-procedural dependencies, matching code regions that get refactored between versions, etc.

Data serialization library. Although using data serialization libraries does not eliminate upgrade failures, it does make data-incompatibility checking much easier, as demonstrated by *DUPChecker*. Unfortunately, in the systems that we have studied, many message classes and file classes use custom serializers and deserializers, which contribute to the majority of data syntax incompatibilities in our study. A broader adoption of potentially more flexible and efficient serialization libraries could help eliminate upgrade failures in the field.

8 Related Work

8.1 Studies on Software System Failures.

Software failures have been thoroughly studied [41, 54, 62–64, 66–68, 70, 73, 75, 78]. Different subcategories of failures have been analyzed, including distributed system failures [58, 67, 70], concurrency bugs [56, 63], OS bugs [41, 68], configuration errors [66, 76], bugs introduced by bug fixes [79], and other bugs [42, 53, 60, 77]. To the best of our knowledge, this paper is the first to analyze software upgrade failures.

Studies mentioning upgrade failures. Liu *et al.* [61] studied 112 high-severity incidents from Microsoft Azure production cluster. They pointed out that software upgrade is one of the reasons for incompatible data-formats. However, they did not offer any details regarding software upgrade problems and did not offer solutions to address data-format incompatibility. Gunawi *et al.* [46] studied 597 publicly available post-mortem reports about cloud service outages. One of their findings is that 16% of these outages involve hardware or software upgrade. They did not conduct detailed root cause, severity, or triggering study for these upgrade problems. Tudor *et al.* [43] analyzed 55 upgrade failures from a e-commerce system, a database system, and Apache web server. Their study focuses on upgrade failures caused by misconfiguration, broken dependency, and operator error. Our study is the first to focus on upgrade failures caused by software defects in distributed systems.

Studies on distributed system failures. Yuan *et al.* [80] analyzed 198 failures on five open source distributed systems. Their study focuses on how a problem propagates from a component error to a system failure, and found that error handling is both the last line of defense and the weakest link. Some other studies [45, 70] focused on understanding major root causes of cloud failures (e.g., software bugs and misconfigurations), without discussion about upgrade failures.

8.2 How to Perform Upgrade?

Others have proposed more robust ways to perform system upgrade [32, 33, 43, 59, 65, 69]. Some of the upgrade failures could be eliminated if such proposals are used in practice. Our study provides motivation and guideline for future research on system upgrade techniques.

Ajmani *et al.* [32, 33] propose a centralized upgrade database to schedule upgrade operations on each node in a distributed system. It requires software developers to provide a thorough specification that details the behavior of each object and how the abstract state of one object in the new version maps to that in the old version. It could help avoid upgrade failures under the condition that such a detailed specification is correctly provided.

Imago [43] supports atomic upgrade for distributed systems. It starts the new version of the target system in a set of new physical or virtual machines (i.e., a shadow of the running old-version system), transfers persistent data from the running old-version system to the new-version cluster, and finally switches over to the new-version cluster to finish the upgrade. If any problem occurs during the launching of the new system, Imago gives up the upgrade. Imago can mitigate the impact of an upgrade failure in production—the system simply sticks to the old version, at the cost of running a second copy of the whole cluster. Orthogonal from Imago, *DUPTester* and *DUPChecker* aim to effectively expose and detect threats to upgrade failures before code release, and hence lower the cost of system upgrade.

MVEDSUA [69] supports reliable, low-latency updates to stateful services by augmenting dynamic software updating (DSU) with multi-version execution (MVE). MVEDSUA forks the current program (the leader) and connect it via MVE with the child (the follower), which will perform the update. When the update on the follower completes, the MVE system feeds it the events it missed, already processed by the leader. MVE system compares the responses of both versions. In response to any disagreement during the comparison, MVEDSUA can terminate the follower, effectively rolling back the update. MVEDSUA can mitigate the impact of an upgrade failure in production just like Imago, and is orthogonal to *DUPTester* and *DUPChecker*.

Li et al. [59] discusses the state-of-the-art update deployment practices in Microsoft Azure, including stage and canary deployment, and develops a tool called Gandalf which improves the safety of update deployment. Gandalf continuously monitors system signals, including service-level logs and performance counters, to detect anomalies. Gandalf analyzes whether an anomaly is caused by an update deployment using correlation analysis and decides if the deployment should be stopped. Orthogonal from Gandalf, *DUPChecker* and *DUPTester* aim to expose upgrade failures before they happen in stage or canary deployment.

9 Conclusions

This paper presents the first in-depth analysis of upgrade failures from widely-deployed distributed systems. We found that the majority of upgrade failures have severe consequences, while only a small portion of them were caught before code release. Guided by our findings about root causes and triggering conditions of upgrade failures, we designed an upgrade testing framework *DUPTester* and a static upgrade bug checker *DUPChecker*. Both have found previously unknown upgrade bugs in multiple distributed systems. Particularly, *DUPChecker* has been requested by HBase developers to integrate into their toolchain. We believe *DUPTester* and *DUPChecker* are just the starting point in tackling this critical problem of software upgrade failures. We release our dataset and tools at "https://github.com/zlab-purdue/ds-upgrade" to help followup research.

Acknowledgements

We greatly appreciate the insightful feedback from our shepherd Minlan Yu and the anonymous reviewers. We thank the SOSP Artifact Evaluation Committee for their precious time and effort to validate our work. We thank Yu Luo for promptly providing the hardware resource used in our artifact evaluation. This research is supported by a Canada Research Chair grant, an NSERC Discovery grant, NSF (CNS-1764039, CCF-2028427, CCF-2119184), the CERES Center for Unstoppable Computing, the Marian and Stuart Rice Research Award, and research gifts from Facebook and Microsoft.

References

[1] American fuzzy lop. https://lcamtuf.coredump.cx/afl/.
[2] Application deployment and testing strategies. https://cloud.google.com/architecture/application-deployment-and-testing-strategies.
[3] Canary deployment. https://cloud.google.com/blog/products/gcp/how-release-canaries-can-save-your-bacon-cre-life-lessons.
[4] CASSANDRA-10652. https://jira.apache.org/jira/browse/CASSANDRA-10652.
[5] CASSANDRA-10822. https://jira.apache.org/jira/browse/CASSANDRA-10822.
[6] CASSANDRA-13441. https://issues.apache.org/jira/browse/CASSANDRA-13441.
[7] CASSANDRA-5102. https://issues.apache.org/jira/browse/CASSANDRA-5102.
[8] CASSANDRA-6678. https://issues.apache.org/jira/browse/CASSANDRA-6678.
[9] Docker hub. https://www.docker.com/products/docker-hub.
[10] Dropbox upgrade failure. https://dropbox.tech/infrastructure/outage-post-mortem.
[11] HDFS-11856. https://issues.apache.org/jira/browse/HDFS-11856.
[12] HDFS-14726. https://issues.apache.org/jira/browse/HDFS-14726.
[13] HDFS-156224. https://issues.apache.org/jira/browse/HDFS-15624.
[14] HDFS-1936. https://issues.apache.org/jira/browse/HDFS-1936.
[15] HDFS-5988. https://issues.apache.org/jira/browse/HDFS-5988.
[16] HDFS-8676. https://issues.apache.org/jira/browse/HDFS-8676.
[17] Java virtual machine. https://en.wikipedia.org/wiki/Java_virtual_machine.
[18] JUnit 5. https://junit.org/junit5/.
[19] KAFKA-10173. https://issues.apache.org/jira/browse/KAFKA-10173.
[20] KAFKA-6238. https://issues.apache.org/jira/browse/KAFKA-6238.
[21] KAFKA-7403. https://jira.apache.org/jira/browse/KAFKA-7403.
[22] Linux containers. https://linuxcontainers.org/.
[23] MESOS-3834. https://issues.apache.org/jira/browse/MESOS-3834.
[24] Microsoft Azure Blog. https://azure.microsoft.com/en-us/blog/.
[25] Microsoft says 11-hour azure outage was caused by system update. https://www.entrepreneur.com/article/240029.
[26] Proto Buffer Guide. https://developers.google.com/protocol-buffers/docs/proto.
[27] PyParsing. https://github.com/pyparsing/pyparsing.
[28] Summary of windows azure service disruption on feb 29th, 2012. https://azure.microsoft.com/en-us/blog/summary-of-windows-azure-service-disruption-on-feb-29th-2012/.
[29] Thrift Compatibility Checker. https://github.com/brunorijsman/thrift-compatibility.
[30] Thrift Guide. https://diwakergupta.github.io/thrift-missing-guide/.
[31] ZOOKEEPER-1805. https://issues.apache.org/jira/browse/ZOOKEEPER-1805.
[32] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Scheduling and simulation: How to upgrade distributed systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 8–8, 2003.
[33] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming*, pages 452–476, 2006.
[34] Apache Cassandra. http://cassandra.apache.org.
[35] Apache HBase. http://hbase.apache.org.
[36] Apache Kafka. https://kafka.apache.org/.

[37] Apache Mesos. https://mesos.apache.org/.

[38] Apache ZooKeeper. https://zookeeper.apache.org/.

[39] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[40] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, 2008.

[41] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, 2001.

[42] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. Understanding real-world timeout problems in cloud server systems. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 1–11, 2018.

[43] Tudor Dumitraş and Priya Narasimhan. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 18:1–18:20, 2009.

[44] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium, NDSS'08*, pages 416–426, 2008.

[45] 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. 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, pages 7:1–7:14, 2014.

[46] 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 Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 1–16, 2016.

[47] Hadoop Distributed File System (HDFS) architecture guide. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[48] Hadoop MapReduce. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[49] Apache Hadoop YARN. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html.

[50] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 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, pages 1–16, 2018.

[51] 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 '17, pages 150–155, 2017.

[52] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley, 2015.

[53] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, 2012.

[54] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. In *Proceedings of the ACM SIGCOMM 2009 conference*, SIGCOMM '09, pages 243–254, 2009.

[55] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[56] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 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, pages 517–530, 2016.

[57] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, 2011.

[58] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A characteristic study on failures of production distributed data-parallel programs. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, pages 963–972, 2013.

[59] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. Gandalf: An intelligent, end-to-end analytics service for safe deployment in large-scale cloud infrastructure. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 389–402, 2020.

[60] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 25–33, 2006.

[61] 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, pages 155–162, 2019.

[62] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 31–44, 2013.

[63] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS'08, pages 329–339, 2008.

[64] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding BGP misconfiguration. In *Proceedings of the ACM SIGCOMM 2002 conference*, SIGCOMM '02, pages 3–16, 2002.

[65] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 287–296, 2003.

[66] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, OSDI'04, 2004.

[67] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, USITS'03, pages 1–15, 2003.

[68] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 305–318, 2011.

[69] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the Twenty-Fourth International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 573–585, 2019.

[70] A. Rabkin and R.H. Katz. How Hadoop clusters break. *Software, IEEE*, 30(4):88–94, 2013.

[71] Redis: an open source, advanced key-value store. http://redis.io/.

[72] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at facebook and oanda. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 21–30, 2016.

[73] B. Schroeder and G.A. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, 2010.

[74] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331, 2010.

[75] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 193–204, 2010.

[76] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 244–259, 2013.

[77] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How not to structure your database-backed web applications: A study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 800–810, 2018.

[78] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, 2011.

[79] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 26–36, 2011.

[80] 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 Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 249–265, 2014.

[81] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book.* CISPA Helmholtz Center for Information Security, 2021.