USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UpFuzz: Detecting Data Format Incompatibility Bugs during Distributed Storage System Upgrade

Ke Han and Sruthi P C, *Purdue University;* Yayu Wang, *The University of British Columbia;* Yaoxu Song and Bishal Basak Papan, *Purdue University;* Junwen Yang, *Meta;* Pedro Fonseca and Yongle Zhang, *Purdue University*

https://www.usenix.org/conference/nsdi26/presentation/han

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

UPFUZZ: Detecting Data Format Incompatibility Bugs during Distributed Storage System Upgrade

Ke Han[†] Sruthi P C[†] Yayu Wang[‡*] Yaoxu Song[†] Bishal Basak Papan[†] Junwen Yang[§]
Pedro Fonseca[†] Yongle Zhang[†]

† *Purdue University* ‡ *The University of British Columbia* § *Meta*

Abstract

Data format incompatibility is a significant cause of cloud incidents during distributed system upgrades, often resulting in severe consequences such as data corruption and service unavailability. A majority of such bugs are only discovered post-release, largely due to the lack of automated testing techniques tailored specifically for the upgrade process. Traditional automated test generation methods face a unique challenge when applied to upgrade testing: the high cost associated with upgrading distributed storage systems due to system initialization. Therefore, the accurate selection of potential failure-inducing tests from the extensive pool of automatically generated tests becomes critical.

In this work, we address this problem by proposing a novel approach to prioritize upgrade tests through analyzing data format properties over *transitively persisted states*: program states that are persisted to disk, directly or indirectly, through chains of memory copies by the old version, and eventually read by the new version after upgrade. Because data format incompatibility bugs happen due to translation errors of such states across versions, *transitively persisted states* satisfying unique *data format properties* related to *changed data formats* are particularly essential for testing.

We build a likely invariant analysis engine that captures such properties as feedback for seed test selection in UPFUZZ, the automated testing engine for the distributed storage system upgrade procedure. UPFUZZ has detected 15 previously unknown upgrade failures caused by data format incompatibilities in the latest stable versions of Cassandra, HBase, and HDFS; developers have confirmed 8 of them. 7 are triggered exclusively with UPFUZZ's data format analysis. The detected bugs have severe consequences, with 6 crashing the cluster and 4 causing data loss or corruption.

1 Introduction

Modern data center workloads heavily rely on distributed storage systems such as distributed file systems [40, 70] and databases [4, 28, 31] to manage critical data. These systems must therefore be correct.

There is extensive research on addressing the traditional challenges of developing distributed systems, such as ensuring correctness under diverse interleavings or in the presence of faults [45, 57, 66, 73, 77, 78, 87]. However, modern distributed system implementations evolve continuously, introducing challenges that are largely orthogonal to those traditionally studied. Developers continuously add new features, improve performance, and fix bugs, leading to frequent production upgrades (e.g., in large data centers, thousands of distributed system upgrades are routinely observed in a single day [68]). Consequently, developers must implement upgrade mechanisms that correctly migrate large and complex states of distributed systems across versions while considering the myriad combinations of states that the system might be in.

Recent work [80] has shown that subtle inconsistencies in the expected format of the system state are a major cause of cloud incidents during distributed system upgrades, accounting for almost two thirds of the distributed system upgrade failures. Such bugs, which we refer to as **data format incompatibility bugs (DFI bugs)**, often result in severe consequences such as data loss and corruption, as well as large-scale service unavailability [80]. For example, on June 12, 2025, Google Cloud experienced a massive outage that impacted 1.4 million users [12]. The root cause was an inconsistency in the data format of policy data structure between the old and new versions of a control service. Similarly, the 2024 CrowdStrike global outage crashed 8.5 million systems and was also caused by a format inconsistency during an upgrade [1].

To test for upgrade bugs, developers [4, 70] often write upgrade tests; however, these tests are manually written and generally test only very limited upgrade scenarios. One recent work, DUPTester [80], aims to automate test generation for upgrades, but relies on transforming existing stress tests and unit tests; thus, it inherits the limitations of the manual tests.

The key challenge in testing for upgrades in a distributed system is that the upgrade operation is exceptionally expensive. In fact, while distributed systems can often execute hundreds of thousands of typical operations per second (e.g., put or get operations in a key-value store), an upgrade operation may take tens of seconds or even minutes to complete. This poses a massive challenge for automated testing systems, which have to explore the vast state space and apply upgrades on each state to check for failures.

TPState: program states closely related to DFI bugs (§4). To address this challenge, we propose a novel approach to prioritize system states more likely to expose data format incompatibility bugs (DFI bugs) in automated testing. Our key insight is that *DFI bugs occur when applications fail to correctly translate, during upgrades, persisted in-memory*

*Work done as a remote research intern at Purdue University.

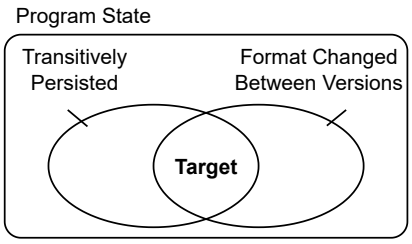

Figure 1: Target subsets of program state.

state, whose in-memory representation's specification (e.g., class definitions) changes across versions. Such a state, which we term **Transitively Persisted State (TPState)**, is a subset of the program state (e.g., an integer, a string, or a buffer) persisted to external storage directly or transitively through chains of memory copies at one point in time during execution. Collectively, TPStates persisted during the entire execution produce the program's persisted state, which eventually will be read by the new version after an upgrade. These persisted subsets of program state are organized in memory as memory reference graphs (e.g., the in-memory representation of a database table). When the specification of such in-memory representation changes between versions (indicating a data format change), they require careful transformation that is prone to errors. Therefore, tests that produce a diverse set of such states are especially valuable for uncovering DFI bugs. Figure 1 shows these target states.

Specifically, we design Transitively Persisted State Analysis to identify tests generating special, previously unexplored TPStates whose format has been changed (thus, prone to DFI bugs), and prioritize tests that generate such TPStates during feedback-based fuzz testing [2, 3, 13, 17, 24, 37, 62].

Selective TPState Property Analysis: a new feedback for DFI bugs (§5). Traditional state-based feedback fuzzing suffers from the state explosion problem [37]: simply rewarding fuzzers for exploring new states has proven counterproductive in practice, leading to lower bug detection rates. This is because for non-trivial applications, the number of program states is effectively infinite. To reduce the states while preserving the fuzzer's ability to trigger DFI bugs, we design the following novel techniques to analyze TPState selectively:

• *Selective TPState Monitoring (§5.1)*. TPStates contain subsets of program states frequently accessed throughout a system's codebase, as they often represent critical system components (e.g., database tables). Monitoring them at every location where they are accessed introduces unaffordable overhead and could represent the persisted state inaccurately, as they might be modified before eventually serialized to disk. We design program analysis techniques to (1) identify such transitively serialized subsets of program states and (2) the program location between their modification and serialization sites, where we selectively insert TPState monitors.

• *Selective Property Inference (§5.2)*. Not all properties of TPState affect the translation (i.e., serialization and de-

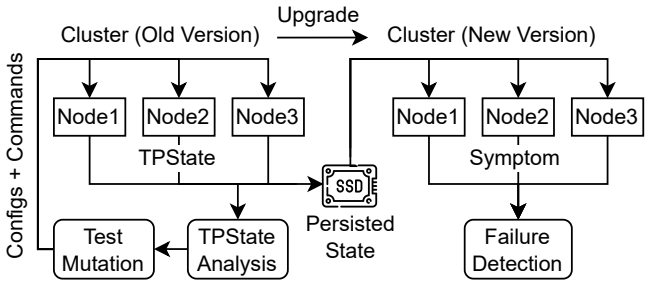

Figure 2: Overview of utilizing *TPState Analysis* in upgrade fuzzing.

serialization) process during upgrade. From an empirical study of real-world DFI bugs, we derive a taxonomy of templates of **data format properties** that directly affect data transformation. We then infer properties using these templates, prioritizing those that are closer to modified format specifications (e.g., changed class definitions) by reference distance, as they are more likely to impact state translation.

• *Selective Structural Conjunction (§5.3)*. A characteristic of DFI bugs is that they often arise under conjunctive properties of TPStates, where multiple independent properties simultaneously hold within a single data instance. For instance, a real-world DFI bug [9] was triggered only when a single table contained both a dropped column and multiple indexes, rather than two separate tables each satisfying one of the properties. We term such cases **structural property conjunction**: conjunctions of properties observed on program states when traversing a data instance's memory reference graph (e.g., a database table's memory reference graph contains its metadata, rows, columns, and cells). However, structural conjunction results in exponential state-space growth, and are typically not supported by existing techniques [25, 36]. To bound the state space, we implement a new property monitoring engine supporting structural property conjunction as a first-class citizen with widely adopted optimizations including temporal and spatial state reduction, as well as a new frequency-based conjunction reduction strategy to selectively apply conjunction to infrequently observed properties.

UpFuzz: a fuzzer built on the TPState analysis results (§6). We integrate our techniques in UpFuzz, the first feedback-based fuzzing engine for testing cross-version data-format compatibility of cloud storage systems during upgrades. Figure 2 shows the overview of our proposed approach. UpFuzz generates configurations for the target distributed system and the tests to be executed in the cluster. Each test includes a sequence of user-level commands scheduled to be executed by different nodes in the cluster. The fuzzing engine collects TPStates and analyzes their data format properties. UpFuzz leverages TPState analysis in two ways: (1) it prioritizes mutating tests that generate TPStates with uncommon data-format properties, and (2) it decides whether to run the costly upgrade based on whether a test exhibits interesting TPState properties.

UpFuzz has uncovered 15 previously unknown DFI bugs

in the latest stable versions of Cassandra, HBase, and HDFS, with 8 of them confirmed and 3 fixed. Note that Cassandra, HDFS, and HBase have on average about one cross-version data format incompatibilities reported each year, but they often have severe consequences and are extremely difficult to discover [80]. Among the 15 DFI bugs discovered by UpFuzz, 6 of them cause cluster crash and 4 of them cause data corruption or data loss. UpFuzz has also detected 7 upgrade failures due to logic errors or concurrency bugs (i.e., in total 21 upgrade bugs). Our evaluation also demonstrates significant advantages for UpFuzz with TPState analysis over traditional approaches. Among the detected DFI bugs: 7 were triggered exclusively when UpFuzz's TPState analysis was enabled, 3 were triggered faster, 3 showed comparable speed, and 2 experienced small slowdowns. Compared to traditional feedback[1], UpFuzz reveals on average 11% more states related to data format incompatibility and achieves a 4.7× speedup in reaching the same number of such states.

Summary. The paper makes the following contributions:

- We propose Selective TPState Analysis to target upgrade-relevant persisted states whose formats change across versions, while controlling state explosion.
- We build UpFuzz, a feedback-driven upgrade testing framework that uses TPState properties to guide exploration and to skip unnecessary upgrade checks.
- We evaluate UpFuzz on three popular distributed storage systems and find 15 previously unknown DFI bugs.

UpFuzz is publicly available at https://github.com/zlab-purdue/upfuzz

2 Background

In this section, we provide background on (1) data format and its role in distributed system upgrade failures, as well as state-of-the-art and challenges in (2) upgrade testing in distributed systems and (3) feedback-based fuzzing.

Data Format in Modern Distributed Systems. Data formats define how data is structured and interpreted, such as the tree-like format of XML [19] or the tabular format of CSV [10]. For programs to exchange data successfully, they must adhere to the same format specifications. Unlike standardized formats such as XML or CSV, modern distributed systems often employ customized, system-specific data formats tailored for performance or scalability. These formats evolve frequently across versions and lack explicit specifications, resulting in particular difficulty in checking their consistency across versions. During upgrade, the new version needs to transform data generated by the old version to the new-version format. Thus, inconsistencies in such data formats across versions can introduce severe failures [71, 80].

[1]To ensure a fair comparison, we implemented UpFuzz baseline version (with traditional feedback) using state-of-the-art fuzzing techniques, including grammar-aware input mutation [11, 82], configuration mutation [85], and stateful fuzzing [23, 86]. UpFuzz has uncovered a total of 38 previously unknown bugs, 18 of which have been confirmed.

```
1  /* Create a table with a composite key. */
2  CREATE TABLE t (k1 INT, k2 INT, v1 TEXT, v2 TEXT,
3                  PRIMARY KEY (k1, k2));
4  /* Insert 2 rows with the same k1. */
5  INSERT INTO t VALUES (0, 1, 'a..', 'b..');
6  INSERT INTO t VALUES (0, 2, 'c..', 'd..');
7  /* Drop the column v2 of the table */
8  ALTER TABLE t DROP v2;
9  /* Read the first row. */
10 SELECT * FROM t WHERE k1=0 and k2=1;
```

Figure 3: A test that triggers Cassandra-13939, a real-world failure when upgrading Cassandra from 2.2 to 3.0.

Distributed System Upgrade Testing. A recent study [80] reveals that most distributed systems do not have dedicated upgrade test facilities, and develops DUPTester, a testing tool for distributed system upgrade. For every pair of versions to test, old and new, DUPTester tests three upgrade scenarios including full-stop upgrade, rolling upgrade, and new-version nodes joining an old-version cluster. DUPTester utilizes the target system's stress tests and integration tests as testing workloads, and uses error log messages, exceptions, and crashes as test oracles to detect upgrade failures. However, DUPTester's reliance on manually crafted stress and integration tests limits its effectiveness. As a result, it can only trigger one-third of their studied upgrade failures [80]. Our evaluation also reveals DUPTester's inefficiency in exposing DFI bugs (details in § 7). This highlights the necessity for developing automated upgrade test generation techniques.

Feedback-Based Fuzzing. The recent development of feedback-based fuzzing tools [2, 3, 13, 17, 24, 29, 30, 37, 39, 41–43, 46, 49, 53, 58, 62, 64, 65, 69, 74–76, 79, 81], such as AFL [3] and Syzkaller [17], has been proven to be effective in automated test generation. These tools repeatedly execute the target program with randomly generated program inputs, and use collected information about the program execution, which is called feedback, to guide future input generation. Therefore, their effectiveness heavily relies on feedback metrics and test throughput (i.e., frequency of the feedback loop). Our experience shows that feedback-based fuzzing faces two challenges when applied to the distributed system upgrade procedure: (1) the slow nature of the distributed system upgrade procedure results in low test throughput and prohibits the efficacy of traditional fuzzing[2]; (2) existing feedback metrics [3, 22, 33, 37, 63, 72] collect general coverage or state information during execution, and as a result, they fail to reward test executions that are likely to produce data triggering DFI bugs, often rewarding unrelated behaviors instead (details in §3). Both challenges highlight the need to develop metrics more sensitive to upgrade-specific behaviors.

3 A Real-World DFI Bug Example

In this section, we present a real-world DFI bug in Cassandra [4], a widely-deployed distributed key value store, to show the insufficiency of traditional feedback. We use this bug as a

[2]Appendix A studies the cost of distributed storage system upgrade.

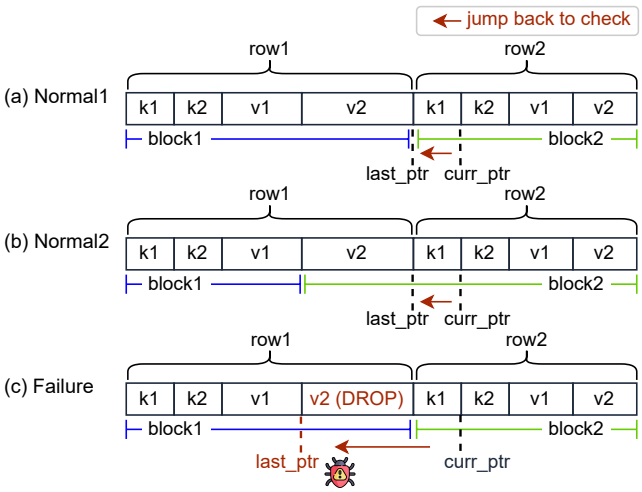

Figure 4: Persisted data in different tests in `Cassandra-13939`. In (c), the `last_ptr` is not updated correctly.

```
1   public Row selectRow(InputStream in) {
2       if (last_ptr >= currentBlock.endOffset)
3           currentBlockIdx++;
4       if (currentBlockIdx > lastBlockIdx)
5           return null;
6       return deserializeRow(in);
7   }
8
9   public Row deserializeRow(InputStream in) {
10      Row r = new Row();
11      RowHeader h = deserializeRowHeader(in);
12      for (int i = 0; i < h.rowSize; i++) {
13          Cell c = deserializeCell(in);
14          if (c.isDropped())
15              continue;                    // BUG
16          r.addCell(c);
17          last_ptr = in.curr_ptr;          // BUG
18      }
19      return r;
20  }
```

Figure 5: Simplified code of Cassandra 3.0 deserialization.

running example in this paper to illustrate the intuition behind our solution (§4) and the design of specific techniques (§5).

`Cassandra-13939` [9] is an upgrade failure caused by data format incompatibility, the buggy state of which cannot be captured by existing coverage-based and state-based feedback metrics. Figure 3 shows a simplified test triggering it. It inserts two rows with specific keys, drops the last column, and selects the first row. On any version of Cassandra, the `SELECT` command correctly retrieves only the first row. However, if the test is run on Cassandra 2.2 and the system is upgraded to version 3.0, issuing the same `SELECT` command on version 3.0 incorrectly returns both rows.

Root Cause. Cassandra has two different organizations of tables: row-based and cell-based. In row-based tables, each row is stored as a single record in the table. In contrast, cell-based tables store each cell (i.e., column name, value, and timestamp) as a separate record in the table, sorted by column name and then timestamp. In addition, as shown in Figure 4, Cassandra partitions tables into index blocks (referred to as `IndexBlk`) to support efficient search.

In Cassandra-13939, the system is upgraded from Cassandra 2.2, which uses a cell-based table layout, to Cassandra 3.0, which adopts a row-based layout that reduces metadata duplication and enables faster batched access. As shown in Figure 4 (a) and (b), in Cassandra 3.0's deserialization process, when parsing data written in the old-version cell-based table format, the system can determine the end of the current row only when the current pointer (curr_ptr) reaches the first cell of the next row. Then, it uses the backtracking pointer (last_ptr) to check whether the end of an index block has been reached. However, if the row ends at the block boundary and its last column is dropped (Figure 4 (c)), the system fails to update last_ptr correctly, which is the bug. Thus, the system incorrectly concludes that the last row (row1) has not ended, and incorrectly continues to read row2 for `SELECT` command.

Figure 5 shows the simplified code snippet of Cassandra 3.0 deserialization. Function `selectRow()` is invoked iteratively by the `SELECT` command in the test in Figure 3. If the update to `last_ptr` at line 17 is skipped due to a dropped cell (line 15), the check of `last_ptr` at line 2 would return false in the next invocation to `selectRow()`, causing the check of `currentBlockIdx` at line 4 to return false, and, thus, continue to read the next row (row2) incorrectly.

Developers fixed this bug by explicitly accounting for the dropped cell when computing `last_ptr`, so the new-version deserializer can correctly detect the end of the current row.

Insufficiency of Coverage-Based Feedback. Typical fuzzing tools [3, 17, 24] rely on branch coverage as feedback to generate tests, but, unfortunately, the bug-triggering conditions are not reflected in branch coverage.

To trigger Cassandra-13939, two conditions[3] need to be satisfied: ① the last row to read ends at the boundary of an index block, and ② the last column is dropped. A test that satisfies condition ① is shown in Figure 4 (a), and this has the same branch coverage during deserialization as a test has a row exceeding an index block, as shown in Figure 4 (b). Since Cassandra 2.2 is cell-based, it could have a row split across two blocks (row1 in Figure 4 (b)). In this case, if a read is indeed reading two rows, when it peeks row2, both branches at line 2 and line 4 take the same direction as in Figure 4 (a). Moreover, condition ② is not reflected in branch coverage either because dropping any cell results in the same branch coverage in `deserializeRow()`.

Branch coverage in the old-version serialization code does not capture these conditions either: as shown in Figure 6, Cassandra 2.2 serializes a `ColumnFamily` (i.e., a table) cell by cell at line 9 and creates index blocks when the accumulated size exceeds a configured index block size at line 13, and then serializes all index blocks at line 20. Branch coverage cannot capture whether a row ends at an index-block boundary, because the serialization process does not have the concept of row. It also cannot capture the presence of dropped cells,

[3]We use satisfied conditions and observed properties interchangeably.

```
1  public void serialize(ColumnFamily cf) {
2    CFIndex index = serializeCF(cf);
3     // TPSTATE MONITOR
4    serializeIndex(index);
5  }
6  // Serialize the table and create index blocks.
7  public CFIndex serializeCF(ColumnFamily cf) {
8    CFIndex ret = new CFIndex();
9    for (Cell c : cf.cells()) {
10     serialize(c, this.output);
11     size += c.size();
12     if (size >= Config.blockSize()) {
13       ret.add(new IndexBlk(...));
14       size = 0;
15     }
16   }
17   return ret;
18 }
19 // Serialize the index blocks.
20 public void serializeIndex(CFIndex index) {
21   for (IndexBlk i : index.getBlocks())
22     serialize(i, this.output);
23 }
```

Figure 6: Simplified code of Cassandra 2.2 serialization.

as they are serialized in the same way as non-dropped ones.

Insufficiency of State-Based Feedback. The state-of-the-art state-based feedback metric [37] monitors likely invariants [36] over directly accessed program variables within the same basic block. Unfortunately, the program states compared in both conditions are only indirectly referenced within a function (`serialize()` at line 1 in Figure 6), but there are no directly accessed variables within the same basic block that can capture these two conditions.

4 Motivation of Proposed Approach

Since upgrade operations are extremely expensive, it is critical to guide testing towards persisted states more likely to trigger DFI bugs. In this section, we explain our core idea by illustrating how it captures the bug-triggering conditions in the motivating example following two observations:

Observation 1: *Data format properties of persisted state that affect the deserialization process are typically preserved in their in-memory representations.*

As shown in Figure 7, the two required conditions to trigger Cassandra-13939 can be observed as properties of the table and index in-memory data structures which will be persisted to disk during the serialization process (Figure 6). These properties can be expressed using the following logical predicates (`a==b` denotes `Objects.equals(a,b)` when applied to object references). The corresponding class definitions for tables and index are shown in Figure 8:

① $\exists$ `i` $\in$ `index.indexBlks`,
`i.end == cf.metadata.columns.last`
This predicate states that an index block ends at the last column; equivalently, a row ends at a block boundary.

② $\exists$ `c` $\in$ `cf.metadata.droppedColumns`,
`c == cf.metadata.columns.last`
This predicate captures the dropped column being the last in the table.

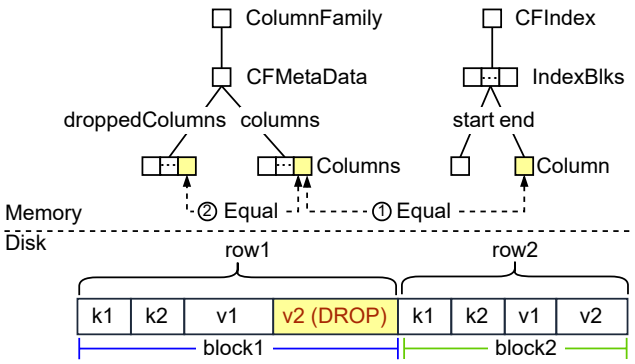

Figure 7: Incompatibility-related format patterns on disk (①, ②) can be captured by observing properties of the table and index in-memory data structure in Cassandra-13939.

```
// Database Table
class ColumnFamily
 CFMetaData metadata;
 Array<Cell> cells;
class CFMetaData
 Set<ColumnDef> columns;
 Set<ColumnId> droppedColumns;
class ColumnDef
 ColumnId name;
```

```
// Index
class CFIndex
 List<IndexBlk> indexBlks;
class IndexBlk
 Cell start;
 Cell end;
class Cell
 ByteBuffer value;
 ColumnId name;
```

Figure 8: Cassandra 2.2 data structures of the table and index.

Surprisingly, it is not obvious what program states will be persisted by solely examining the serialization function. For example, though table metadata is involved in both bug-triggering conditions, the serialization function of a table (`serialize()` at line 1 in Figure 6) only serializes the table and its index, but not its metadata. Instead, in Figure 11, the table's metadata is serialized to be stored as cells in system tables and will be persisted transitively to disk when system tables are serialized.

Following the definition of transitive relation [18], we refer to subsets of program state serialized to external storage directly or transitively, through in-memory data copy and serialization, as **Transitively Persisted State (TPState)**.

Observation 2: *Modifications to the specification (i.e., class definition) of persisted states' in-memory representation reflect corresponding changes in the underlying data format.*

Figure 9 shows that condition ① involves a class definition change across versions: the type of data member `.end` is changed. This data format change is directly related to the root cause of the failure: in the new version, `ClusteringPrefix` is an identifier for an entire row, preventing an `IndexBlk` ending in the middle of a row.

Therefore, although Cassandra lacks a formal specification of the on-disk data formats for tables and indexes across versions, we can leverage modifications of their in-memory specifications to guide testing.

Our Approach. The observations inspire us to guide fuzz testing towards program states with incompatible cross-version data formats by monitoring data format properties over TPStates whose class definitions are changed across versions.

```
class IndexBlk                    class IndexBlk
  Cell start;                       ClusteringPrefix start;
  Cell end;                         ClusteringPrefix end;
```

Figure 9: Difference of index between Cassandra 2.2 and 3.0.

5 Transitively Persisted State Analysis

Since the massive number of program states renders state-based feedback fuzzing ineffective [37], we carefully design a TPState Analysis that selectively monitors program states while preserving the ability to trigger DFI bugs. This section introduces our methods for: (1) selecting subsets of program states and their monitoring locations, (2) choosing data format properties to monitor, and (3) efficiently inferring these properties while supporting structural property conjunction.

5.1 Selective TPState Monitoring

In this section, we introduce our selective TPState monitoring approach by addressing two questions: (1) how to identify TPStates, and (2) where to monitor TPStates.

How to identify TPStates. As defined in §4, TPStates are serialized directly or transitively to persisted storage. We use the following transitive serialization analysis to capture TPStates:

We perform an inter-procedural static backward data flow analysis starting from the parameters of invocations to annotated disk I/O operations (e.g., `OutputStream.write()` in Java). For each data flow, the backward tracking ends at (1) initialization statements (e.g., `new`) or (2) non-copy assignment statements whose right-hand side is an expression involving multiple variables (e.g., `a = b + c`). If the accessed field `field` is a buffer (e.g., `ByteBuffer` in Java), we continue tracing the program states serialized into that buffer. For example, as shown in Figure 11, because every table's metadata is *serialized* to be stored in cells' byte buffers and later *serialized transitively* to disk, we annotate in-memory buffer serialization functions `ByteBufferUtils.serialize()` and trace data flow across them – establish a data flow from `cd.name` to `cell.value` at line 5. Finally, when this adapted data flow analysis stops, it reports the data types (signature of classes and data members) encountered as the result.

Since TPStates are subsets of program state serialized to disk, and these subsets of program state are organized as reference graphs in memory, we also infer their specifications (i.e., class definitions): During the backward data flow analysis, for field (data member) access assignments (`var = obj.field`), we also track the parent object `obj`. For the motivating example, the analysis captures all the classes and their data members referenced by tables and indexes in Figure 8.

Where to monitor TPStates. TPStates are frequently accessed throughout the target system's codebase, as they often represent critical system components (e.g., tables). Monitoring such states at every location they are accessed is impractical and could represent the disk state inaccurately, as they might be modified before they are eventually serialized to disk. We aim to cover all upgrade-relevant TPStates that are eventually persisted, but selectively choose how to capture properties that reflect what is actually persisted while avoiding pervasive instrumentation on hot paths that would reduce fuzzing throughput and dilute the feedback signal.

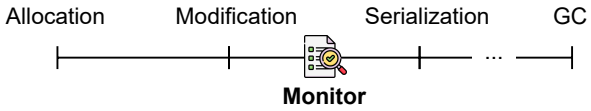

Figure 10: Life cycle of transitively persisted states.

Figure 10 shows the life cycle of TPStates. After being allocated in memory, it will be modified and serialized (potentially multiple times) before garbage-collected. We insert a monitor in between modification and serialization. Metrics inferred during this phase reflect the state at the point of its conversion into disk format.

We perform an inter-procedural backward control-/data-flow analysis on top of Soot [16, 61] compiler framework, to identify such monitoring points by locating the merge point of the control flow from TPStates' modification site and its serialization site. The analysis first identifies the modification sites of TPStates: assignment statements to data members and item insertion into the collections belonging to the transitively persisted states (e.g., line 13 in Figure 6). Starting from a TPState's modification sites and sink disk I/O operations, it performs a backward control flow analysis to find where the control flow merges in the call graph of the target system. For example, in Figure 6, suppose line 22 is the disk I/O operation that writes `IndexBlk` (items in `CFIndex`) to the output stream (in the authentic Cassandra implementation, the write to disk happens in its callee function) and line 13 is the statement that modified `CFIndex`, we insert a monitor for the `index` object (all program states reachable in `index`'s reference graph) at line 3. This merge point analysis ensures that the monitor is placed on the execution path after modification and before serialization of the TPState (`CFIndex index`). The detailed algorithm is in Appendix C.

Reference aliasing needs to be resolved in order to determine whether a modification site is modifying the state that will be written by a sink disk I/O operation. We adopt a conservative approach: any assignment to the data member identified with its signature (`T.f` assume `obj` is of type `T`) and any insertion to a collection with the same type (e.g., `List<Integer>`) will be considered a *candidate modification point*, and if later in the backward analysis the reference of object or collection used in the modification statement is possible to be an alias of the object (`obj`) or collection (`c`) being serialized, we conservatively consider the object or collection may be modified and mark the candidate modification point as a modification point.

5.2 Selective Property Inference

In this section, we introduce our *selective property inference* by answering two questions: (1) What kinds of data-format properties should we infer? (2) How to associate data format properties with data format changes?

```
public void serialize(CFMetaData md) {
  for (ColumnDef cd : md.columns)
    systemTable.addCell(
      new Cell(
        "name", ByteBufferUtils.serialize(cd.name))
    );
  for (ColumnId ci : md.dropColumns)
    systemTable.addCell(
      new Cell(
        "drop_cols", ByteBufferUtils.serialize(ci))
    );
}
```

Figure 11: Simplified code of CFMetaData serialization.

Property Type	Examples	#	%
Boundary	$x_{offset} \geq b$; INV(seq.first/seq.last)	8	23%
Special Value	x=0; x=null; x.length() = 1	13	37%
Equivalence	object.equals(x,y)	6	17%
Type	x.getClass() $\in \{\tau_1, \tau_2, \ldots\}$	11	31%
Structural Conj	$\text{INV}_1(x.f_1) \wedge \text{INV}_2(x.f_2)$	12	34%
Value Conj	$\text{INV}_1(x) \wedge \text{INV}_2(x)$	4	11%
Total		35	

Table 1: Taxonomy of data format properties' templates. INV(x) represents a property over x. The sum of # and % exceed the total number and 100% respectively, because one property could belong to multiple types (e.g., a special value on the boundary). The # of conjunctions represents the # of properties that need to be joined conjunctively with additional properties.

What data format properties to monitor. To understand what properties over TPStates reflect data formats, we sampled and studied 12 upgrade failures induced by DFI bugs across widely-deployed distributed systems, following the bug-collection methodology in [74] to collect bugs from 2018 to 2022. For each case, we derived bug-triggering properties as predicates over TPStates the same as in Cassandra-13939 (§4), resulting in a total of 35 data format properties. We categorize them according to their semantic meanings and how they can be monitored, and derive a set of property templates that can be used in automated property inference in Table 1. We describe our techniques to infer them efficiently in §5.3. Note that our study cannot cover all possible data format properties, but our analysis shows they cover conditions checked during parsing in almost all data types in generic data description languages [38] (Appendix D).

• *Boundary Property* checks whether the current offset has reached a boundary ($x_{offset} \geq b$) or another condition is related to the item residing on the boundary INV(seq.first/last) of a sequence. It captures incorrect computation of boundaries during parsing, such as off-by-one mistakes.

• *Special Value Property* checks special values (e.g., 0, null) of serialized variables, as well as the size of serialized sequences and collections. It captures corner cases of variable values and structures that impact translation, such as index corner cases (empty vs. single-element collections).

• *Equivalence Property* checks whether two variables are equivalent using (object.equals(x,y)). It captures semantic equivalence of serialized states that are critical during data format translation, such as the equivalence between the index boundary and dropped column in the motivating example.

• *Type Property* checks whether a previously unobserved type of TPState is serialized, which will directly result in a previously unobserved data format on disk.

• *Conjunction*[4] checks and captures corner cases when multiple properties must hold simultaneously over the same variable (i.e., *value conjunction*), such as x!=0 $\wedge$ x!=1 and the reference graph rooted at the same object (i.e., *structural conjunction*), such as $\text{INV}_1(x.f_1) \wedge \text{INV}_2(x.f_2)$.

How to relate properties to format changes. Although TPState analysis reveals format-related properties of program states, they do not determine which formats have changed or which data needs to be translated across versions. We address this by identifying modified type definitions – termed *Version Diff* – and using them to steer exploration toward format changes that may require data translation.

We compare the class definitions of TPStates identified in §5.1 across versions (e.g., Figure 8), examining (1) whether each class definition exists and (2) whether the data members of a class are changed (e.g., removal, name change, type change). During the testing phase, when a previously unobserved data format property is detected, we check its associated reference chain (e.g., CFIndex.indexBlks[0].end in Figure 12 (a)), and use the number of references between the program state monitored by the likely invariant and the closest version diff as the reference distance, and prioritize states with a smaller reference distance.

For example, in Cassandra-13939, condition ① i.end == cf.metadata.columns.last involves a data format changed across versions: the type of data member .end is changed from Cell to ClusteringPrefix as in Figure 9. Because *i.end*'s type is changed across versions and *i.end* is the object being monitored, the reference distance is 0, indicating that this previously unobserved property is strongly associated with changed data format.

We adopt an exponential probabilistic model to prioritize tests involving states close to the changed data formats: $P(N) = c \cdot e^{-kN}$ where N is the reference distance, and c and k are constants. The rationale is that larger distances imply weaker relevance to the format change, thereby rapidly decreasing the probability of selecting a test as the distance increases. We find setting $c = 0.8$ and $k = -0.5$ provided effective prioritization in practice.

5.3 Efficient Property Inference with Selective Structural Conjunction

To infer data format properties over TPStates, we adopt a widely used approach for automatically uncovering properties of program states called likely invariant inference [36]. Such

[4] We do not list **disjunction** separately, because it simply denotes that a TPState can satisfy one among multiple properties, which applies to all data format properties by default.

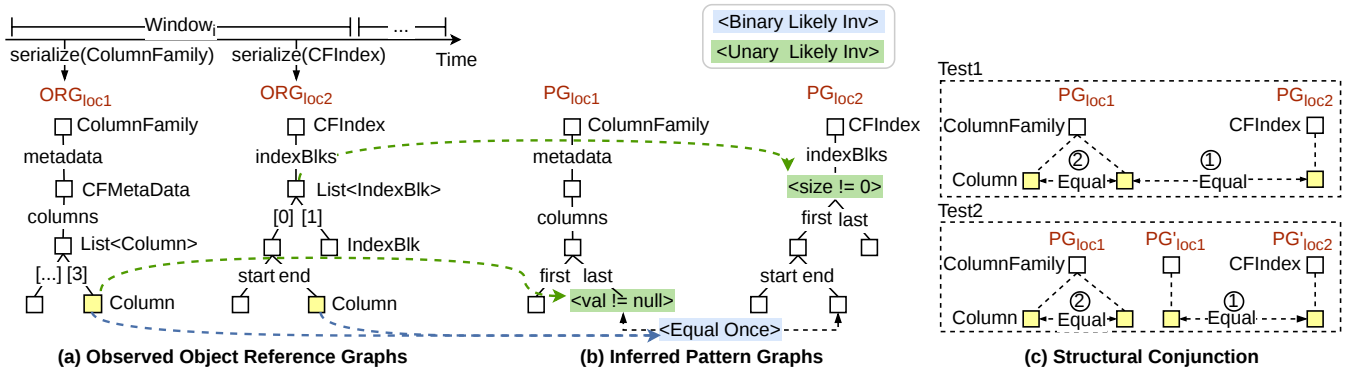

Figure 12: Likely Invariant Inference. (a) (b): Object Reference Graph (ORG) of Transitively Persisted States are monitored, and violated likely invariants are organized in the corresponding pattern graph. Binary likely invariants across ORGs are inferred only within a serialization window. (c): Multiple pattern graphs are maintained at each monitoring location for Structural Conjunction. Test1 observes properties ① and ② on the same ORG while test2 observes them on different ORGs.

techniques monitor program executions and infer invariants that are likely to hold. These likely invariants are derived from finite executions and are not always guaranteed. When an inferred likely invariant is violated, it signals the discovery of a previously unobserved program behavior. Thus, we also refer to data format properties as data format likely invariants. A violation does not imply a bug; it marks the execution as interesting for further exploration and for downstream cross-version checking. This signal is sound as a novelty detector as it reliably indicates "not seen before".

However, existing likely invariant inference engines by default do not infer **structural conjunction** of likely invariants, due to the combinatorial explosion of possible conjunctions when applied to real-world programs [25]. In this section, we describe our new likely invariant inference and monitoring engine that scales to real-world distributed systems.

Summarize TPStates into likely invariants. Specifically, our inference engine infers data-format likely invariants over transitively persisted states and their structure – the object reference graph (ORG), obtained by systematically unfolding all data structures encountered when traversing an object's data members (object refers to an object in Java or a memory reference in other languages). Violations of likely invariants indicate previously unobserved data format properties and enable systematic exploration of data format.

As shown in Figure 12 (b), for each ORG monitored at a location, we construct pattern graphs which summarize the violated likely invariants observed for this ORG. For example, because the `CFIndex`'s ORGs never have `null` in any of its `IndexBlk`'s `.start` field, the engine learns a likely invariant $val != \texttt{null}$ and labels it in the corresponding position in the pattern graph. Similarly, because an *Equal* relationship is observed between the `CFIndex`'s and `ColumnFamily`'s ORGs, the pattern graphs record a satisfied *Equivalence Likely Invariant* in the corresponding positions (*Equal Once* in Figure 12 (b)). Because data format properties typically exist among TPStates serialized close to each other, we only infer binary likely invariants for (1) (*temporal locality*) objects serialized within a temporal serialization window (i.e., object reference graphs serialized within the same function invocation are considered within the same serialization window) and (2) (*spatial locality*) data members that can be reached within a limited depth (default set to five).

Structural likely invariant conjunction A unique feature in our likely invariant inference engine is to perform likely invariant conjunction at object level, as structural conjunction is needed in 34% of our studied properties. For example, our motivating example happens only if both required properties are observed in the same table. Therefore, the conjunction of the two likely invariants over the same object (i.e., table) captures the failure-inducing state accurately.

Specifically, we capture them by recursively applying a conjunction operation across all likely invariants violated within an object's data members, and the analysis extends to the data members of each respective member, provided that they are identified as TPStates. Figure 12 (c) shows that multiple pattern graphs are maintained at each monitoring location (`PG` and `PG'` in Test2) and they are used to distinguish when multiple properties are observed in the same ORG (Test1) and different ORGs (Test2).

To avoid excessive memory overhead, we avoid duplicating the shared structural information among pattern graphs (e.g., reference edges). Instead of storing complete pattern graphs, we assign a unique identifier to each vertex (in a class' reference graph) and invariant, and represent each instance as a set of $< vertex\_id, invariant\_id >$ pairs. This design also enables fast pattern graph comparison using set comparison instead of costly graph comparison.

Frequency-based conjunction reduction Unfortunately, structural conjunction results in the potential exponential growth in the combinations of violated likely invariants within a single test, leading to a "corpus explosion" where a large number of tests are deemed significant for upgrade, thus rendering the feedback metric ineffective.

Based on the insight that likely invariants that are violated infrequently are more interesting to test, we measure each

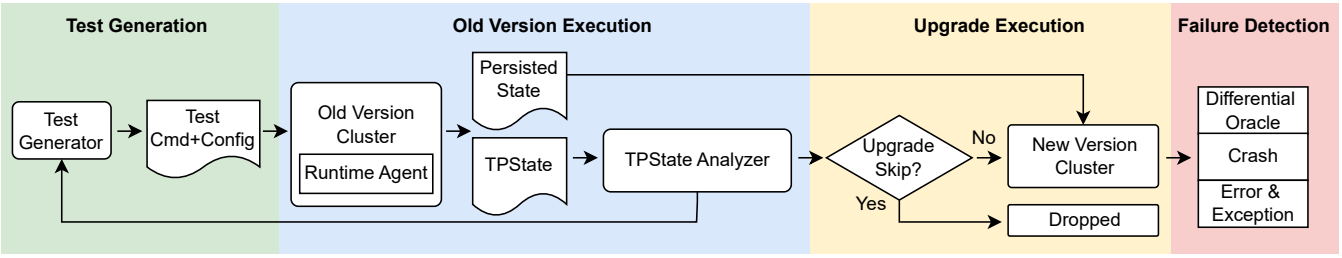

Figure 13: Overview of UPFUZZ architecture.

invariant's violation frequency and perform conjunction for those with low violation frequency. For example, property ① – a row ending at an index block boundary – is observed rarely in Cassandra-13939, a conjunction of it and property ② – last column is dropped is considered much more interesting (rare) than a conjunction of any frequently observed properties (e.g., a table contains one row and a row contains one cell). In practice, we rank likely invariants by their inverse violation frequency, and only report the violation if the conjunction includes an invariant for the highest ranked five.

6 Architecture and Implementation

In this section, we explain UPFUZZ's architecture and implementation details. Figure 13 shows the architecture of UPFUZZ. In the test generation stage (~9k LoC), UPFUZZ generates a system configuration file for both old-version and new-version systems and integration test input – a sequence of user-level commands (e.g., Cassandra SQL commands). In *Old-version Test Execution* stage (~7k LoC), the test is executed on the old-version while UPFUZZ collects feedback metrics including branch coverage and violated data format likely invariants over identified TPStates. Based on the collected feedback, UPFUZZ decides whether to add the test to the test corpus for future test generation and whether to perform upgrade operation for this test in *Upgrade Test Execution* stage (~2k LoC). For tests that are upgraded, UPFUZZ uses failure detection to detect upgrade failures.

Corpus management: The test corpus maintains tests triggering new feedback metrics and is divided into three sub-corpus: Branch Coverage Corpus (with new branch coverage), Format Corpus (violating new data-format likely invariants), and Version Diff Corpus (violating invariants related to a version diff). The test mutator selects seeds with a 70% probability from the Version Diff Corpus, 20% from the Format Corpus, and 10% from the Branch Coverage Corpus. This approach prioritizes exploring the testing space near uncommon data format related to version diff over new branch coverage.

Feedback with TPState analysis: UPFUZZ maintains an accumulated set of graph patterns produced by TPState analysis, aggregating patterns observed across all tests (analogous to accumulated code coverage). After each test, UPFUZZ detects previously unobserved patterns by comparing the patterns from the current execution against this accumulated set. Specifically, when a test violates a previously unbroken likely invariant, we mark the test as interesting and add the invariant to the accumulated set of broken invariants. Subsequent violations of the same invariant are not considered interesting. The invariant template/conjunction component is extensible and supports adding new templates when needed.

Upgrade test skip and execution: UPFUZZ skips upgrades for tests less likely to yield upgrade bugs. Upgrades are always performed for tests with new feedback. For others, UPFUZZ uses a probabilistic model based on mutation depth d: $P_{drop}(d) = c \cdot e^{-k \cdot d}$, where $c = 0.4$, $k = 0.3$. Mutation depth is the number of mutations from the test's root in the *mutation tree* (randomly generated tests are roots, and the parent of a test is its seed used in mutation). Randomly generated tests have a 40% chance to skip, but as mutation depth increases, skipping becomes less likely as deeper mutations are harder to generate and more valuable for exploration.

Failure detection: UPFUZZ includes a basic error checker [80] that detects crashes, exceptions, and error logs. To reduce the efforts for failure investigation, UPFUZZ groups failures with similar error logs and exceptions. UPFUZZ also uses its random input generator to create read commands correlated with the test's write commands and differential oracles to detect inconsistent read results between versions.

7 Evaluation

We apply UPFUZZ to three widely deployed distributed systems to detect DFI bugs, including Cassandra [4] (peer-to-peer distributed database), HDFS [70] (distributed file system), and HBase [5] (master-worker distributed database). Each system has multiple maintained stable versions and we test full-stop system upgrade across these versions.

Research questions: We focus on analyzing (1) *Bug detection*: whether exploration of incompatibility-related format properties enables efficient detection of DFI bugs, (2) *State exploration*: how well our technique explores incompatibility related states and properties, as well as (3) *Runtime overhead* and *false positive rate*.

Testbed: We conducted experiments on x86_64 machines with 192GB ECC DDR4-2666 memory and two Intel Xeon Silver CPUs (10 cores, 2 hyperthreads each core, 2.20 GHz). UPFUZZ sets up each system using a cluster of containers respecting its minimum node requirement: 1 node for Cassandra, and 3 nodes for HDFS and HBase. UPFUZZ parallelizes test execution with multiple clusters on a physical machine.

Bug Id	Summary	Consequence	Status	Avg Triggering Time						
				DUPTester	Base	Base+S	DF	DF+S	DF+VD	DF+VD+S
CS-18105	Truncated data comes back after upgrade	data corrupt.	F	×	7.75h	10.16h	5.34h	12.32h	9.26h	8.35h
CS-18108	Data loss after a system upgrade	data corrupt.	C	×	20.88h	×	15.59h	18.01h	5.14h	16.83h
CS-19590	Error deserializing mutation when upgrade	crash	C	×	×	×	14.55h	10.00h	15.59h	6.93h
CS-19591	MarshalException during migration	crash	R	×	×	×	×	×	×	18.66h*
CS-19623	Illegal RT bounds sequence after migration	read err.	C	×	×	×	×	×	×	120.00h*
CS-19639	Legacy data read with NullPointerException	read err.	R	×	×	×	×	×	×	21.20h
CS-19689	Wrong reverse read results after upgrade	read inconsist.	R	×	×	×	×	×	×	19.95h
CS-20182	Legacy data migration with assertion error	crash	R	×	×	×	×	×	18.58h	21.16h
HB-28583	Upgrade crash with InvalidProtocolBuffer	crash	R	×	×	×	×	×	20.39h	15.29h
HB-28812	Upgrade crash due to deprecated comparator	crash	F	Up	Up	Up	×	Up	Up	Up
HB-28815	HMaster aborted after upgrade	crash	C	Up	Up	Up	×	Up	Up	Up
HB-29021	Wrong read for legacy data after upgrade	read inconsist.	F	×	0.62h	1.41h	0.57h	1.34h	1.20h	0.33h
HD-16984	Directory timestamp lost during upgrade	data loss	C	×	0.06h	0.06h	0.06h	0.06h	0.06h	0.06h
HD-17219	Wrong count results after upgrade	read inconsist.	R	×	10.53h	7.35h	6.38h	5.14h	4.86h	4.82h
HD-17686	Wrong stat results after upgrade	data loss	R	×	15.49h	17.66h	12.32h	10.27h	13.68h	10.95h

Table 2: 15 new DFI bugs detected by UPFUZZ. F: Fixed. C: Confirmed, R: Reported. × indicates that the bug is never triggered within 24 hours. Star (*) means the bug cannot be triggered consistently within 24 hours and we record the shortest amount of time we observed. Up means the bug will occur as long as an upgrade is performed with any workload. Each entry reports the average number across 3 runs.

7.1 Detecting New Format Incompatibilities

As shown in Table 2, UPFUZZ has detected 15 data format incompatibilities, which is a significant number considering these systems have on average about 1 to 2 DFI bugs reported each year. 8 of them have been confirmed by the developers and 3 have been fixed. As revealed in the summary of each case, these failures often cause severe consequences such as cluster outage and data loss. In addition to the 15 new DFI bugs, UPFUZZ also found 23 other new bugs. Among them 7 occur during the upgrade process, while the remaining 16 occur in single version. We present them in Appendix B.

There are three key decision points in UPFUZZ's design, namely (1) whether to use data format properties as feedback to discover *new formats*, (2) whether to use version diff to guide testing towards potentially *incompatible formats*, and (3) whether to skip upgrade according to the feedback to save testing resources. Table 2 presents our detailed ablation study to understand the benefits of these techniques (represented as *DF, VD, S* respectively). *DUPTester* refers to the state-of-the-art upgrade testing tool for distributed systems [80]. *Base* refers to a baseline implementation of UPFUZZ with traditional code coverage feedback using state-of-the-art fuzzing techniques, including grammar-aware input mutation [11, 82], configuration mutation [85], and stateful fuzzing [23, 86]. Because traditional fuzzers, such as AFL [3] and Syzkaller [17], do not apply to distributed systems, we implemented UPFUZZ from scratch. We evaluated UPFUZZ in all settings for 24 hours, with each evaluation repeated three times.

Findings. Our main findings are: (1) Enabling all three techniques yields the best result: compared to *Baseline*, *DF+VD+S* accelerates the detection of 73% DFI bugs (9 accelerated consistently, and 2 inconsistently marked with star *). (2) 47% of DFI bugs can only be detected with our techniques. (3) Upgrade skip is beneficial only when guided by TPState analysis; without it, skipping can reduce overall test efficiency when combined with the baseline.

Comparison with DUPTester. DUPTester cannot detect most DFI bugs because they require unique input to trigger or a differential test oracle to detect. DUPTester lifts developer-written unit tests into upgrade tests, enabling highly specialized scenarios and assertions. In contrast, UPFUZZ relies on an input generator and a differential test oracle, and is therefore better suited for bugs that require specific persisted states and cross-version behavioral divergence. Conversely, bugs that require unit-test-only APIs, special setups, or tightly orchestrated concurrency may fall outside UPFUZZ's current input model; in these cases, DUPTester is complementary.

We present 3 case studies that illustrate how our techniques effectively detect new DFI bugs, as well as the limitations.

Bug #3: States captured exclusively by data-format properties. In Bug #3, an assertion error is raised when the new-version Cassandra reads the legacy data. This bug is triggered by the following properties of the old version: (1) In a table, the user drops a column and adds a column with the same name and a different type. (2) The table contains data in the dropped column before being removed. These properties do not trigger new branch coverage and cannot be rewarded by *Baseline*. For property (1), there is no branch comparing the names of the inserted column and the dropped column. In comparison, when utilizing data-format properties, because the column and the added column are two vertices in their table's reference graph, UPFUZZ infers whether their names are equal (i.e., $metadata.regColumns.item == metadata.dropColumns.item$). Moreover, UPFUZZ triggers the bug by capturing the conjunction of property (1) and property (2) (i.e., $!columnFamily.cells.size > 0$). As a result, UPFUZZ strategies with *DF* enabled can trigger this bug, whereas *Baseline* fails to trigger it within a 24-hour budget.

Bug #9: States prioritized by version diff. Bug #9 occurs when the new version HBase replays the Write Ahead Logs

Bug Id	Avg Trigger Time					
	Base	Base+S	DF	DF+S	DF+VD	DF+VD+S
CS-13939	×	×	20.59h	16.48h	18.99h	13.41h
CS-14803	11.24h	8.49h	5.40h	2.04h	3.74h	3.35h
CS-14912	15.53h	8.05h	10.93h	14.52h	4.25h	4.59h
CS-15970	5.58h	4.15h	2.32h	1.51h	3.55h	2.21h
HB-22503	×	×	15.24h	19.11h	9.79h	14.43h
HB-25902	Up	Up	Up	Up	Up	Up
HB-26021	Up	Up	Up	Up	Up	Up
HB-25238	0.07h	0.07h	0.07h	0.07h	0.07h	0.07h
HD-14831	Down	Down	Down	Down	Down	Down
HD-15624	18.36h	15.35h	16.55h	12.24h	14.96h	5.29h

Table 3: Evaluation of existing DFI bugs. "Up" and "Down" mean the bug occurs once an upgrade or a downgrade is performed with any workload, respectively. When computing the average triggering time across bugs, we use × if any bug is not triggered within the time budget. DUPTester only triggers HB-25902, HB-26021, HB-28583, and HD-14831, using the same time as other approaches.

(WALs) of the old version. The failure-inducing condition is related to a data structure with a Version Delta – the protobuf [15] message for restoring snapshots and replaying WALs with the new version introduced an extra required field. UPFUZZ *DF* does not prioritize such violations, which explores over 40k distinct format states in 24 hours but fewer than 5% are relevant to cross-version modifications. In contrast, UPFUZZ with *VD* prioritizes these relevant states, enabling it to explore upgrade-related states more efficiently. As a result, UPFUZZ with *VD* enabled can trigger the bug, while UPFUZZ *Baseline* and *DF* fail to do so in the 24-hour budget.

Bug #1: Reason for slowdown. In Bug #1, deleted data reappears in the new-version Cassandra when the following commands are executed to the same table in the old version: create a row, create an index, truncate the table and drop the index. UPFUZZ *Baseline* triggers the bug slightly faster because the critical bug-triggering conditions are reflected in code coverage (e.g., truncating a table directly results in new code coverage). *DF+VD* slightly outperforms *DF* because the index data structure changes across versions.

We performed the same evaluation on our studied DFI bugs (2 out of 12 are not evaluated due to deprecated dependencies). The results show the same trend (presented in Table 3).

7.2 State Exploration

To evaluate UPFUZZ's ability to explore incompatibility-related program states, we compared the three different settings and measured the total number of distinct violated likely invariants observed (and close to version diff). A higher number of distinct violations indicates UPFUZZ's ability to cover more unusual format patterns related to incompatibility. The results represent the average of three runs.

As shown in Figure 14, testing strategy (*DF+VD+S*) outperforms both *DF+S* and *Baseline* across all evaluated systems. Compared to *Baseline*, *DF+VD+S* and *DF+S* identified on average 10.9% and 6.4% more distinct likely invariant violations across evaluated systems, respectively: 13.5% and 9.1%

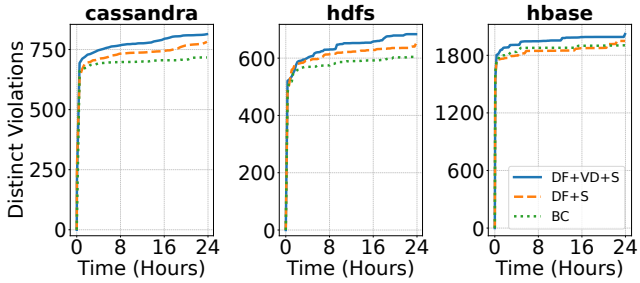

Figure 14: The number of violated data-format likely invariants close to version diff.

System	Version	Start-up	Command Exec.
Cassandra	2.2.19	0.18%	11.44%
	4.1.6	0.08%	10.79%
HBase	2.6.0	2.05%	22.03%
	2.5.9	0.08%	12.78%
HDFS	2.10.2	11.47%	10.26%

Table 4: Overhead (%) of system start-up and command execution.

more in Cassandra, 13.1% and 7.6% more in HDFS, and 6.1% and 2.4% more in HBase. Baseline takes about 24 hours to peak and reach 750, 650, and 1940 violations for Cassandra, HDFS, and HBase. *DF+S* takes 17.9, 23.8, and 22.4 hours to reach the same amount of violations. *DF+VD+S* speeds it up to 5 times, with 4.8, 5.6, and 5.0 hours.

7.3 Performance Overhead

To quantify the performance cost of invariant inference, we measure overhead as the percentage latency increase of *DF+VD+S* over the *Baseline* UPFUZZ. We report overhead separately for system startup and command execution.

Table 4 shows that invariant inference incurs $< 12\%$ startup overhead and up to 23% command-execution overhead. Startup overhead is minimal in Cassandra/HBase but higher in HDFS (11.47%), likely due to metadata serialization. HBase has the highest command overhead (22.03%), consistent with Figure 14 where it yields the most violations. The end-to-end overhead is modest, since command execution accounts for only a small fraction of the total runtime compared to system startup and the upgrade process.

7.4 False Positives

We say that a false positive occurs when the failure detector flags a test as buggy, but manual inspection finds no underlying bug. UPFUZZ achieves a low false positive rate, typically below 1%. This low rate stems from our failure-detection design: the differential oracle compares the new version's behavior against the old version's behavior, treating the old-version result as the testing oracle. Since storage systems generally aim to preserve user-visible behavior across versions, most differential mismatches correspond to real bugs.

The false positives primarily arise from non-deterministic outputs (e.g., `SCAN` outputs a non-deterministic field: command execution time) and version-dependent output formats

(e.g., in Cassandra, the message of `InvalidRequest` exceptions is changed across versions). To mitigate these cases, UPFUZZ provides a post-processing output normalization mechanism that extracts semantic content by masking nondeterministic fields and normalizing version-specific formats.

7.5 Applying UPFUZZ to New Systems

Applying UPFUZZ to a new system mainly requires (1) development of a system-specific input and configuration generator, along with scripts to perform upgrades, and (2) adaptation of UPFUZZ's data flow analysis. The steps in (1) are common to fuzzers for complex systems, such as Syzkaller [17].

For step (2), UPFUZZ relies on static analysis (Soot [16]) to identify TPStates and place selective monitors. In systems that use reflection or dynamic class loading, the static call graph can be incomplete, which may cause UPFUZZ to miss some TPStates unless we supplement it. In practice, porting typically requires lightweight annotations for system-specific I/O sinks (beyond UPFUZZ's built-in common write APIs) and for a small number of hard-to-resolve call-graph edges. To further reduce manual effort and missed paths, UPFUZZ can incorporate a dynamic call graph collected from unit-test execution to complement the static call graph.

Optionally, users may tune hyper-parameters, such as the array sampling rate during invariant inference. Typically, integrating a system into UPFUZZ takes a graduate student two weeks, enabling automated testing across multiple versions after this one-time setup. We expect UPFUZZ to be used for each release version of the target system.

8 Limitations and Future Work

UPFUZZ currently targets a limited set of upgrade scenarios: our evaluation focuses on full-stop upgrades on small clusters and does not cover all real-world settings (e.g., rolling upgrades, larger clusters, or background tasks such as compaction and rebalancing). Extending UPFUZZ to these scenarios is a direction for future work.

9 Related Work

Performing Upgrade Safely: Researchers and industry practitioners have proposed and adopted many robust approaches to perform system upgrade [7,8,20,21,34,50,56,67]. *Modular Upgrade and Canary Deployment.* Modular Upgrade [20, 21] and Canary Deployment [8] enable safe upgrade by gradually rolling out an update to the entire cluster. Unfortunately, they typically take a significant amount of time [6] and prevent fast upgrades which are desired in many scenarios (e.g., deploying a fix to eliminate service outage).

Reverse Protocol Engineering. Dynamic data format analysis has been investigated in the context of network messages [26, 27, 32, 47, 51, 60], typically referred to as reverse engineering of protocol syntax. However, these techniques often incur high overhead, making them unsuitable for integration with fuzz testing in large-scale cloud systems [83, 84]. They infer message formats through data flow analysis: either taint tracking the data flow from an input stream [27, 32] or reconstructing the data flow to an output stream with recorded instruction-level execution trace [26], which often incur two to three times execution slowdown. In addition, they aim to reverse engineer the *complete* data format by inferring *disjunctions* of format patterns instead of *conjunctive* ones.

Feedback-Based Testing: Feedback-based fuzzing [2, 3, 13, 17, 24, 29, 30, 37, 39, 41–43, 46, 49, 53, 58, 62, 64, 65, 69, 74–76, 79, 81] has been proven to be effective in generating bug-triggering tests. Recent work [30, 54, 58, 88] have applied feedback-based fuzzing to distributed systems. However, they use network messages and code coverage as feedback, which are oblivious to data formats. A recent work, INVSCOV [37], leverages likely invariants as feedback to guide fuzzing. In comparison, UPFUZZ differs in three important ways. First, to mitigate state explosion, INVSCOV restricts inference to likely invariants over *directly referenced* program variables accessed within the same basic block, which is often insufficient to detect DFI bugs. Second, UPFUZZ supports the structural conjunction of likely invariants and employs a frequency-based reduction strategy to control state explosion. Third, INVSCOV mines its set of likely invariants *offline* from unit tests and reuses this set during fuzzing, whereas UPFUZZ performs inference *online*, continuously discovering properties during fuzzing and thus avoiding dependence on a pre-computed set.

Crash Consistency Testing: Crash consistency testing tools [44, 45, 48, 52, 59, 66, 77, 78] inject crashes at different execution points and check whether recovery preserves crash-consistency guarantees (e.g., atomicity/durability invariants) within a single software version. Crash consistency and DFI testing are complementary: crash consistency checks crash/recovery correctness within one version, whereas UPFUZZ targets cross-version failures when state written by version A is later read by version B after a format change.

10 Conclusion

We present UPFUZZ – the first systematic testing engine for data format incompatibility bugs in the distributed system upgrade procedure through feedback-based fuzzing. We propose *transitively persisted state analysis*, an analysis that captures data format properties of in-memory representation of persisted system state and associate them with changed data formats, to guide feedback-based fuzzing. Evaluation shows UPFUZZ is able to explore data formats related to format change and detects unique upgrade failures induced by data format incompatibility bugs.

Acknowledgments

We thank our shepherd, Eric Eide, and the anonymous reviewers for their constructive suggestions. This work was supported by NSF grants 2300562, 2140305, and a gift from Meta Platforms, Inc. We also thank CloudLab [35] and Chameleon Cloud [55] for providing the computing infrastructure.

References

[1] 2024 crowdstrike-related it outages - wikipedia. https://en.wikipedia.org/wiki/2024_CrowdStrike-related_IT_outages.

[2] Aflplusplus/aflplusplus. https://github.com/AFLplusplus/AFLplusplus.

[3] american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[4] Apache cassandra. https://cassandra.apache.org/_/index.html.

[5] Apache hbase – apache hbase™ home. https://hbase.apache.org/.

[6] Application deployment and testing strategies - canary testing. https://cloud.google.com/architecture/application-deployment-and-testing-strategies#canary_test_pattern.

[7] Application deployment and testing strategies - shadow testing. https://cloud.google.com/architecture/application-deployment-and-testing-strategies#shadow_test_pattern.

[8] Canary deployment. https://cloud.google.com/blog/products/gcp/how-release-canaries-can-save-your-bacon-cre-life-lessons.

[9] [cassandra-13939] mishandling of cells for removed/dropped columns when reading legacy files - asf jira. https://issues.apache.org/jira/browse/CASSANDRA-13939.

[10] Comma-separated values. https://en.wikipedia.org/wiki/Comma-separated_values.

[11] Fuzzing with grammars - the fuzzing book. https://www.fuzzingbook.org/html/Grammars.html".

[12] Google cloud service health. https://status.cloud.google.com/incidents/ow5i3PPK96RduMcb1SsW".

[13] libfuzzer – a library for coverage-guided fuzz testing. — llvm 17.0.0git documentation. https://llvm.org/docs/LibFuzzer.html. (Accessed on 04/15/2023).

[14] Mongodb: The developer data platform | mongodb. https://www.mongodb.com/.

[15] Protocol buffers. https://developers.google.com/protocol-buffers.

[16] Soot: A java bytecode framework for static analysis. https://soot-oss.github.io/soot/.

[17] Syzkaller-kernel fuzzer. https://github.com/google/syzkaller.

[18] Transitive relation - wikipedia. https://en.wikipedia.org/wiki/Transitive_relation.

[19] Xml. https://en.wikipedia.org/wiki/XML.

[20] 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.

[21] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming*, pages 452–476, 2006.

[22] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

[23] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, 2022.

[24] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Jan H Boockmann and Gerald Lüttgen. Comprehending object state via dynamic class invariant learning. In *International Conference on Fundamental Approaches to Software Engineering*, pages 143–164. Springer, 2024.

[26] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 621–634, 2009.

[27] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329, 2007.

[28] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[29] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[30] Yuanliang Chen. Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 109–109. IEEE Computer Society, 2024.

[31] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[32] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402, 2008.

[33] Jared DeMott, Richard Enbody, and William F Punch. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. *BlackHat and Defcon*, 2007.

[34] 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.

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

[36] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[37] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846, 2021.

[38] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. *ACM Sigplan Notices*, 41(1):2–15, 2006.

[39] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594. USENIX Association, August 2020.

[40] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[41] Google. Honggfuzz: A security oriented, feedback-driven, evolutionary, easy-to-use fuzzer with interesting analysis options. https://github.com/google/honggfuzz.

[42] Aki Helin. A crash course to radamsa. https://gitlab.com/akihe/radamsa.

[43] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. Winnie: fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[44] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. Vinter: Automatic Non-Volatile memory crash consistency testing for full systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 933–950, Carlsbad, CA, July 2022. USENIX Association.

[45] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 147–161, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 147–161, 2019.

[47] Yonghwi Kwon, Fei Peng, Dohyeong Kim, Kyungtae Kim, Xiangyu Zhang, Dongyan Xu, Vinod Yegneswaran, and John Qian. P2c: Understanding output data files via on-the-fly transformation from producer to consumer executions. In *NDSS*, 2015.

[48] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 718–733, New York, NY, USA, 2023. Association for Computing Machinery.

[49] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 475–485, New York, NY, USA, 2018. Association for Computing Machinery.

[50] 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.

[51] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *NDSS*, volume 8, pages 1–15, 2008.

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

[53] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association.

[54] Tao Lyu, Liyi Zhang, Zhiyao Feng, Yueyang Pan, Yujie Ren, Meng Xu, Mathias Payer, and Sanidhya Kashyap. Monarch: A fuzzing framework for distributed file systems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 529–543, 2024.

[55] Joe Mambretti, Jim Chen, and Fei Yeh. Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn). In *2015 international conference on cloud computing research and innovation (ICCCRI)*, pages 73–79. IEEE, 2015.

[56] 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.

[57] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox fuzzing of distributed systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1615–1629, New York, NY, USA, 2023. Association for Computing Machinery.

[58] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox fuzzing of distributed systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1615–1629, 2023.

[59] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Crashmonkey and ace: Systematically testing file-system crash consistency. *ACM Trans. Storage*, 15(2), April 2019.

[60] John Narayan, Sandeep K Shukla, and T Charles Clancy. A survey of automatic protocol reverse engineering tools. *ACM Computing Surveys (CSUR)*, 48(3):1–26, 2015.

[61] Rohan Padhye and Uday P. Khedker. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, SOAP '13, page 31–36, New York, NY, USA, 2013. Association for Computing Machinery.

[62] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 398–401, 2019.

[63] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[64] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, 2018.

[65] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[66] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 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)*, pages 433–448, Broomfield, CO, October 2014. USENIX Association.

[67] 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.

[68] 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.

[69] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, 2017.

[70] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[71] Lilia Tang, Chaitanya Bhandari, Yongle Zhang, Anna Karanika, Shuyang Ji, Indranil Gupta, and Tianyin Xu. Fail through the Cracks: Cross-System Interaction Failures in Modern Cloud Systems. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys'23)*, May 2023.

[72] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, 2019.

[73] Haoze Wu, Jia Pan, and Peng Huang. Efficient exposure of partial failure bugs in distributed systems with inferred abstract states. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI '24, April 2024.

[74] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660. IEEE, 2020.

[75] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[76] Wen Xu, Soyeon Park, and Taesoo Kim. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 971–986, 2020.

[77] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. USENIX Association.

[78] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[79] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *OSDI*, pages 349–365, 2021.

[80] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. Understanding and detecting software upgrade failures in distributed systems. In *Proceedings of the 28th Symposium on Operating Systems Principles*, SOSP'21, 2021.

[81] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 745–761, USA, 2018. USENIX Association.

[82] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Generating software tests. In *Generating Software Tests*. Saarland University, 2019.

[83] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 722–733. IEEE, 2020.

[84] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. Efficient fuzz testing for apache spark using framework abstraction. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 61–64. IEEE, 2021.

[85] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. Fuzzing configurations of program options. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–21, 2023.

[86] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. {StateFuzz}: System {Call-Based}{State-Aware} linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3273–3289, 2022.

[87] Yonghao Zou, Jia-Ju Bai, Zu-Ming Jiang, Ming Zhao, and Diyu Zhou. Blackbox fuzzing of distributed systems with multi-dimensional inputs and symmetry-based feedback pruning. Network and Distributed System Security (NDSS) Symposium 2025, 2025.

[88] Yonghao Zou, Jia-Ju Bai, Zu-Ming Jiang, Ming Zhao, and Diyu Zhou. Blackbox fuzzing of distributed systems with multi-dimensional inputs and symmetry-based feedback pruning. Network and Distributed System Security (NDSS) Symposium 2025, 2025.

A Slow Upgrade Procedure

We investigated four widely adopted distributed systems (Cassandra [4], MongoDB [14], HBase [5], HDFS [70]), and the time duration of their upgrade procedure (with minimum workload) varies from 5 seconds to more than 2 minutes. Our analysis shows that many time-consuming tasks during upgrade are difficult to avoid using generic approaches. In particular, our investigation reveals the following typical tasks performed during upgrade: (1) flushing data, (2) coordinated shutdown, (3) process initialization (e.g., process setup, JVM class loading), (4) configuration loading and setup, (5) system-specific health testing, (6) disk checking (e.g., access privilege checking, fsck), (7) system-specific initialization and data loading, and (8) membership establishment. Though theoretically possible, skipping each of these tasks typically requires customized source code changes, which is ad-hoc and could leave the system in an undefined state.

System	Versions	Time
Cassandra	2.2 to 3.0	31s
HDFS	2.10 to 3.3	145s
HBase	2.6 to 3.3	90s
MongoDB	5.0 to 6.0	5s

Table 5: Upgrade duration with minimal configuration.

B Other Detected New Bugs

Ticket	Simplified Summary	Status
CS-18644	IndexOutOfBounds during migration	C
HB-28167	HMaster crashes due to NPE	C
HB-28590	NPE after upgrade	R
HB-28659	NPE in HMaster (setServerState function)	R
HB-28912	NPE when there's network delay with HDFS	R
ZK-4748	BufferUnderflowException during upgrade	R
HD-17163	IOException during upgrade	R

Table 6: 7 new non-data format bugs detected by UPFUZZ.

In addition to the new data format bugs, UPFUZZ also found 23 other new bugs. Among them 7 are upgrade bugs not due to data format incompatibility (Table 6), while the remaining 16 could only occur in single version (Table 7).

C Monitoring Point Identification

The detailed algorithm is shown in Algorithm 1: During backward data flow analysis, once it identifies a modification site of a transitively persisted state, it performs a forward control flow search to find the control flow merge point with the state's serialization site.

D Data-Format Likely Invariant Templates

Table 8 lists all the types defined in data description calculus, as well as corresponding data-format likely invariant templates.

$x : \tau \mid e$ (Constrained Type) refers to a piece of data x of type τ with constraint e. Since e is developer-specified, we derived a set of such constraints, such as whether a serialized value being 0 and whether a serialized reference being *null*, from our case study. We refer to these particular conditions as *Special Value Likely Invariants*.

τ seq(τ_s, e, τ_t) (Sequence Type) refers to a sequence of items of type τ, where τ_s represents the separator's type, e the terminating condition (e.g., sequence length limit), and τ_t the look-ahead type check. *Type Check Likely Invariant* captures an item's type τ and is derived from the sequence type's intrinsic semantics (τ_s and τ_t are not supported because they are less error-prone according to our study). *Boundary Condition Likely Invariant* checks whether the current offset has reached a boundary ($x_{offset} >= b$), which is an error-prone terminating condition revealed in our study. In addition, we use *Bound-*

Ticket	Simplified Summary	Status
CS-18636	Unknown keyspace after dropping a keyspace	C
CS-18843	NPE when typing an incorrect query statement	C
CS-18897	NPE in getHostId during system start	C
HB-28105	NPE in QuotaCache if Table is dropped from cluster	F
HB-28109	HMaster crash during start up	F
HB-28125	list_quota_table_sizes returns wrong results	R
HB-28159	Table state fetch error during table initialization	R
HB-28545	Worker thread gets stuck during TRUNCATE	R
HB-28660	list_namespace returns wrong results	C
HB-28187	NPE when flushing a non-existing column family	F
HD-16360	Remove local root dir without protection	R
HD-17169	Incorrect result format of du	R
HD-17174	Wrong side effect of checksum command	R
HD-17176	EOF during shutting down	R
HD-17206	Quota not working when appending a file	R
HD-17207	Wrong REM_DISK_QUOTA when exec COUNT	R

Table 7: 16 new single version bugs detected by UPFUZZ. F: Fixed. C: Confirmed, R: Reported.

Algorithm 1: Monitor Location Identification.

Input: *statements*: Program statements.
Input: *DiskOp*: Disk I/O Operations.
Output: $\mathcal{M}$: All monitor locations.

```
1  stmt ← statements.end();
2  objSet, serSites ← ∅, ∅;
3  while stmt ≠ statements.start() do
4      if stmt ∈ DiskOp then
5          foreach obj ∈ stmt.getArguments() do
6              objSet.add(obj);
7          serSites.add(stmt);
8      if stmt is a field access in form obj1 = obj2.getField(f) and
         obj1 ∈ objSet then
9          objSet.add(obj2)
10     if stmt modifies obj and obj ∈ objSet then
11         s ← stmt;
12         c ← stmt.getCallerFunction();
13         while !reachable(s, serSites, c) do
14             s ← c.getCallSite();
15             c ← s.getCallerFunction();
16         M.add(s);
17         objSet.remove(obj);
18     stmt ← stmt.prev();
```

	DDC Type	Semantic	Invariant	Example	Supp?
$x:\tau \mid e$	Constrained Type	A piece of data x of type τ with constraint e.	*Special Value*	$x=0$; $x=1$; $x=null$; $x=$""; $x=true$; x.length() $=1$.	✓
τ seq(τ_s, e, τ_t)	Sequence Type	A sequence of items of type τ. τ_s: separator's type. e: terminating condition. τ_t: look-ahead type check.	*Boundary Condition* *Boundary Distance* **Type Check**	$x_{offset} >= b$ $\lvert x_{offset} - b\rvert >= c$ x.getClass() $\in \{\tau_1, \tau_2, ...\}$	✓ ✓ ✓
$\Sigma x:\tau_1.\tau_2$	Dependent Sum Type	A pair: second's type depends on first's value.	*Type Decl. Update*	x.getClass() $\in \{\tau_1, \tau_2, ...\}_{updated}$	✓
$\mu\alpha.\tau$	Recursive Type	A recursive data structure.	*In-/Out-Degree*	len$(x.children) = 0$; $x.f == null$.	✓
$\tau_1 + \tau_2$	Sum Type	Data matching either τ_1 or τ_2.	**Inv. Disjunction**	$INV_1(x) \mid\mid INV_2(x)$	✓
τ_1 & τ_2	Intersection Type	Data matching both τ_1 and τ_2.	**Inv. Conjunction**	$INV_1(x.f_1)$&&$INV_2(x.f_2)$	✓
absorb(τ)	Active Type	Parses a value of τ without returning anything.	**Type Check**	x.getClass() $\in \{\tau_1, \tau_2, ...\}$	✓
scan(τ)	Active Type	Scans the input for data that matches τ within a boundary.	**Boundary Condition** *Boundary Distance*	$x_{offset} >= b$ $\lvert x_{offset} - b\rvert >= c$	✓ ✓
compute$(e:\sigma)$	Active Type	Compute a value of type σ using expression e.	*Equivalence*	`object.equals(x, y)`	✓
$\lambda x.\tau$ and τe	Parameterized UDF Type	Abstraction and application to parameterize user-defined types.	*Type Expr. Check*	x.getClass() $\in \{\tau_1, \tau_2, ...\}$	✓
$C(e)$	Parameterized Base Type	A base type C parameterized by expression e.	N/A	N/A	N/A
unit and $\perp$	Terminal Types	Parse success and failure.	N/A	N/A	N/A

Table 8: Data Description Calculus and Data-Format Likely Invariant Templates. **Bolded invariants** match conditions in default DDC semantics. *Italic invariants* are derived from real-world case studies. x and y are variables of primitive type or reference type. $INV(x)$ is a likely invariant over x. $x.f$ is a data member of x. b is a boundary and c is a constant representing the smallest distance observed. $\{\tau_1, \tau_2, ...\}$ refers to a set of types and $\{\tau_1, \tau_2, ...\}_{updated}$ a set of types modified in the new version. ✓ means partially supported.

ary Distance Likely Invariant, which measures the distance between the current offset and the boundary ($|x_{offset} - b|$), to speed up the fuzzing process to break *Boundary Condition Likely Invariants*.

$\Sigma x:\tau_1.\tau_2$ (Dependent Sum Type) refers to a pair where the second's type depends on the first's value. One particular application of Dependent Sum Type is class definition: a class definition is a sequence of pairs where the first value is a class member's name and the second value is the member's type. *Type Declaration Update Likely Invariant* captures whether a serialized object's class declaration is modified in the new version (x.getClass() $\in \{\tau_1, \tau_2, ...\}_{updated}$).

$\mu\alpha.\tau$ (Recursive Type) refers to a recursive data structure. *In-/Out-Degree Likely Invariants* captures certain graph properties of this recursive data structure, such as the number of children (len$(x.children) = 0$, $x.f == null$).

$\tau_1 + \tau_2$ (Sum Type) refers to data matching either τ_1 or τ_2. Its semantics yield *Invariant Disjunction* ($INV_1(x) \mid\mid INV_2(x)$) when multiple likely invariants are observed over a variable.

τ_1 & τ_2 (Intersection Type) refers to data matching both τ_1 and τ_2. Its semantics yield *Invariant Conjunction* ($INV_1(x.f_1)$ && $INV_2(x.f_2)$) of likely invariants over different data members of an object.

Active Types – absorb(τ), scan(τ), compute$(e:\sigma)$ – describes certain computations during parsing. Likely invariants derived from absorb(τ) and scan(τ) have been explained. compute$(e:\sigma)$ represents a value computed from some parsed value using the expression e. One example is a search index built from a sequence of records inside distributed storage. Equivalence comparison is often performed when constructing or parsing such indexes, from which we derive *Equivalence Likely Invariant* (`object.equals(x,y)`).

$\lambda x.\tau$ and τe (Parameterized User-Defined Type) represent abstraction and application to parameterize user-defined types. One example is a type parameter used in a Java generic class – *Type Expression Check Likely Invariant*, which is captured the same way as *Type Check Likely Invariant*. $C(e)$ (Parameterized Base Type) represents a base type C parameterized by expression e. It is not supported because our current implementation is on Java-based systems that do not support parameterized base types (e.g., Int and String cannot be parameterized).

unit and *bottom*$(\perp)$ are terminal types representing parse success and failure. They do not have corresponding likely invariants because they do not have in-memory representations.

E Equivalence Likely Invariants Inference

To infer equivalence likely invariants, we compute a *Comparable Set* for monitors: if two program states identified in §5.1 are compared explicitly using `object.equals(x,y)` or stored in a comparable collection (e.g., a set, a map's keys) anywhere in the program, they belong to a *comparable set*. When any of their monitors are triggered, we check whether the other program state is visible in the same context and infers *Equivalence Likely Invariant* by applying `object.equals(x,y)` to them. For example, if a prior test

infers the invariant *obj.f1 == obj.f2*, and a subsequent test determines that *obj.f1 == obj.f2 == obj.f3*, the invariant is violated and updated. If another test finds that *obj.f1* != *obj.f2*, the invariant remains unchanged. At each step, we maintain the largest subset of equivalent states to ascertain whether a special equivalence relationship between two variables has been previously observed.