Understanding Real-World Vulnerabilities in Distributed Cloud Systems

Authors: EXCERPT, Lian Li

Date: 2025-05-08T19:07:10+00:00

Abstract

Distributed cloud systems are facing great security challenges because of widely-existing vulnerabilities. These vulnerabilities are often easily exploitable, leading to numerous cloud breaches. In this paper, we present VulCloud, the most comprehensive study on 243 vulnerabilities from 16 widely-deployed distributed cloud systems. Each vulnerability is studied in depth along 5 dimensions: root causes, triggering conditions, security impacts, observability, and fixing strategies. From our study, we obtain many interesting findings that can open up new research directions for combating vulnerabilities in distributed systems.

Full Text

Preamble

Understanding Real-World Vulnerabilities in Distributed Cloud Systems

Jie Lu (lujie@ict.ac.cn)
SKLP, Institute of Computing Technology, Chinese Academy of Sciences, China

Lian Li (lianli@ict.ac.cn)
SKLP, Institute of Computing Technology, Chinese Academy of Sciences, China

ABSTRACT

Distributed cloud systems face significant security challenges due to widespread vulnerabilities that are often easily exploitable, leading to numerous cloud breaches. In this paper, we present VulCloud, the most comprehensive study to date of 243 vulnerabilities across 16 widely-deployed distributed cloud systems. Each vulnerability is examined in depth across five dimensions: root causes, triggering conditions, security impacts, observability, and fixing strategies. Our

study yields numerous important findings that open new research directions for combating vulnerabilities in distributed systems.

1 INTRODUCTION

Security requirements intensify as systems become more distributed. Distributed systems have been widely adopted in industry, with enterprises increasingly migrating data, applications, and business elements to these platforms. For instance, Ant Group has moved all their data—including sensitive information such as bank account details—to OceanBase, a massively parallel distributed database developed by the group.

However, distributed systems face substantial security challenges. The open cloud environment and the vast amounts of valuable data naturally attract attackers, as evidenced by numerous recent cloud breaches. The underlying infrastructure of cloud platforms must be strengthened to mitigate these security challenges.

Various techniques have been proposed to secure distributed systems. Classic mechanisms such as Secure Sockets Layer (SSL) and Key Management Service (KMS) are adopted for communication via REST APIs and for accessing encrypted user data, respectively. Additionally, new security mechanisms are implemented to better address distributed systems' unique requirements. For example, many systems (e.g., Kubernetes and MongoDB) implement security policies such as authentication protocols and permission control as centralized services available to other components via network requests. This centralized approach is believed to offer better security guarantees than distributed alternatives where each component implements its own policy. When correctly implemented and configured, these mechanisms can effectively protect systems from external attacks and prevent breaches.

Unfortunately, security mechanisms in distributed systems are notoriously difficult to implement correctly. These systems often run across thousands of nodes with distinct execution environments, making it extremely complicated for programmers to reason about complex inter-node interactions and cover all security concerns. Consequently, security vulnerabilities are widespread in real-world distributed systems, often causing catastrophic failures. For example, a major factor in the infamous iCloud photo leak was a security flaw in the cloud API that allowed unlimited password guessing attempts.

While substantial research has studied various bugs in distributed systems, these efforts have focused on reliability rather than security vulnerabilities. For instance, CBSDB investigated 3,655 bugs but intentionally excluded security issues. Many studies have examined cloud security problems at a high level, but no comprehensive analysis exists of vulnerabilities specifically within distributed cloud systems. To our knowledge, only limited work has examined vulnerabilities in Hadoop, and we lack a comprehensive understanding of vulnerabilities across different types of distributed systems.

This paper presents the most comprehensive study to date of 243 vulnerabilities from 16 widely-deployed systems. The selected systems span cloud computing infrastructure, with vulnerabilities collected from all 168 CVE entries reported since 2012, plus 75 issues from bug tracking systems. Our study addresses five research questions:

- **RQ1**: Which security mechanisms are vulnerable and what are their root causes?
- **RQ2**: What are the triggering conditions for vulnerabilities?
- **RQ3**: Do these vulnerabilities have severe impacts?
- **RQ4**: Are there observable symptoms when vulnerabilities are exploited?
- **RQ5**: How do developers fix these vulnerabilities?

Our findings reveal that authentication, authorization, input validation, information flow, and fault tolerance are the five security mechanisms most threatened by vulnerabilities. For example, we find that in token-based authentication, tokens are often not invalidated when associated identities change. Additionally, 28.8% of vulnerabilities result in information leaks, with 72.9% disclosing sensitive data in logs due to missing or insufficient sanitization. While rich log information is essential for diagnosis, we must be aware of the risk of leaking sensitive data. We also find that 90.9% of vulnerabilities involve complex interactions between two or more components, and 8.6% relate to fault-tolerance schemes. For instance, when authentication or authorization services become unavailable, systems may mistakenly believe security mechanisms are not enforced and handle subsequent requests without proper checks. Finally, we find attacks exploiting these vulnerabilities are difficult to detect, as 50.9% are exploited silently with no explicit exceptions or errors in logs.

Based on these findings, we offer lessons for researchers and developers. For example, to detect information leak vulnerabilities, researchers should focus on log contents and effectively model inter-component communications, as sensitive information often propagates across multiple components. Developers should place audit logs around privileged operations to avoid silent exploitation, making automatic audit log generation a worthwhile research direction. Our studied vulnerabilities are publicly available at https://github.com/anonymous for future research.

Our contributions are:

- We present VulCloud, the most comprehensive study of vulnerabilities in distributed systems.
- We provide a large-scale vulnerability benchmark for evaluating tools.
- We offer takeaways to guide future research and help developers build more secure systems.

The remainder of this paper presents our methodology (§2), answers RQ1-RQ5 (§3-§7), discusses challenges and solutions (§8), reviews related work (§9), and concludes (§10).

2 METHODOLOGY

We conducted a comprehensive empirical study of 243 vulnerabilities in distributed systems to answer our five research questions. This section explains our system selection, vulnerability collection process, analysis methodology, and threats to validity.

2.1 Target Systems

We selected a diverse set of 16 popular open-source distributed systems (Table 1) covering the entire cloud computing software stack. From top to bottom: coordination services (ZooKeeper and Etcd), distributed message queue (Kafka), data processing frameworks (MapReduce and Spark), resource manager systems (YARN, Kubernetes, and Mesos), in-memory cache systems (Memcached and Redis), distributed databases (HBase and MongoDB), distributed file systems (HDFS and Ceph), and cloud computing platforms (OpenStack and CloudStack). Many cloud providers employ this stack to deliver services.

2.2 Collecting Vulnerabilities

We collected vulnerabilities from two sources: CVE entries and bug tracking systems.

First, we gathered all CVEs reported since 2012 for each system, yielding 168 vulnerabilities. For security reasons, many CVEs (82 of 168) are only vaguely described, often without patches or technical details. We manually compared vulnerable and patched versions to locate corresponding patches, requiring three man-months to study numerous git commits.

Second, we queried issue tracking systems using keywords: “vulnerability,” “authorization,” “authentication,” “access control,” “sensitive,” “leak,” and “security.” After manually reviewing thousands of returned issues to identify real exploitable vulnerabilities (excluding functional bugs), we found an additional 75 vulnerabilities.

In total, we collected 243 real-world vulnerabilities across our 16 systems (Table 1).

2.3 Analyzing Vulnerabilities

For each vulnerability, we examined source code, comments, patches, and related discussions. In answering RQ1-RQ5, we categorized vulnerabilities by affected security mechanism (with root cause), triggering conditions, impacts, observability, and fixing strategy. Each vulnerability is assigned to exactly one security mechanism based on its root cause. For example, Kubernetes-CVE-2018-1002105 disables authentication, but its root cause is authentication code failing to handle exceptions, so we classify it as a fault tolerance vulnerability rather than an authentication vulnerability.

2.4 Threats to Validity

As with previous bug studies, potential threats include representativeness of systems and vulnerabilities, and subjectivity of our methodology.

Representativeness of Systems. We selected 16 systems covering most key cloud computing elements: distributed file systems, coordination services, resource managers, key-value databases, in-memory caches, data processing systems, and cloud platforms. These systems are written in multiple languages (C/C++, Java, Scala, Go, Python) and implement distinct security mechanisms (e.g., Kerberos in Hadoop, LDAP in CloudStack, custom identity services in OpenStack). However, we cannot cover all systems, particularly closed-source ones.

Representativeness of Vulnerabilities. Our vulnerabilities are collected without bias from widely-used systems. While we cannot compile all vulnerabilities (some are silently fixed, and our search method may miss some), we believe our collection is representative.

Subjectivity. Due to complexity and manual effort, subjectivity may exist. To mitigate this, each vulnerability was inspected and cross-validated by at least two authors, with disputes discussed until agreement.

3 RQ1: SECURITY MECHANISMS

Distributed systems adopt classic authentication and authorization mechanisms for access control. Authentication verifies user identity, while authorization specifies access permissions to resources. Vulnerabilities arise from flaws in protocol implementation or policy enforcement. Without sufficient input validation, malformed inputs threaten data integrity and availability. Distributed systems also have two implicit security mechanisms: fault tolerance and information flow. When services implementing security policies throw exceptions or crash, fault tolerance should ensure policies remain enforced. Information flow mechanisms guide data propagation, but if sensitive data reaches unexpected sinks (e.g., I/O), confidentiality is broken.

Table 2 categorizes our 243 vulnerabilities across five security mechanisms: authentication, authorization, input validation, information flow, and fault tolerance. While conceptually similar to client-server applications, these vulnerabilities exhibit new characteristics (e.g., multi-component interactions) discussed throughout this paper.

3.1 Authentication

Most authentication vulnerabilities (15/22, 68.2%) are traditional, including 9 authentication bypasses (e.g., default passwords), 3 man-in-the-middle vulnerabilities, and 3 missing authentication issues. The remaining 7 (31.8%) are introduced by token-based authentication mechanisms.

3.1.1 Token-based Authentication In distributed systems, both external user requests and internal node/component requests require authentication. Many systems adopt a centralized design. For example, HDFS uses an external Kerberos server (Figure 1 [Figure 1: see original paper]). Administrators configure legitimate users and nodes. When users log in or nodes join, tickets are assigned. These tickets establish connections between clients and servers or among cluster nodes.

The problem emerges with high authentication request volumes: large MapReduce jobs create thousands of sub-tasks, each requiring authentication. The Kerberos server cannot handle this load, and distributing tickets to all subtasks is undesirable. Token mechanisms provide finer-grained authentication. Different token types are issued for distinct requests, limited to specific services. In Figure 1, a "read file" request: (1) a user communicates with NameNode using an authenticated ticket; (2) NameNode returns a block access token with file locations; (3) the user requests data from DataNode using the token; (4) DataNode returns file data. The token only permits block access, not administrative tasks like node shutdown.

3.1.2 Missing Token Invalidation Despite benefits, token-based authentication introduced 7 vulnerabilities: 3 from distributed data races (previously studied) and 4 from missing token invalidation when token-associated identities change (e.g., identity deletion or permission revocation). Figure 2 [Figure 2: see original paper] illustrates OpenStack-CVE-2013-2059: (1) user1 requests a token; (2) OpenStack generates and stores the token; (3) the token is returned; (4) admin deletes user1; (5) due to a bug, the token remains; (6) user1 uses the token; (7-8) the request is handled because the token still exists, breaking authentication protocols.

Finding 1: The token-based mechanism introduced 31.8% (7/22) of authentication vulnerabilities, with 4 caused by missing token invalidation when identities change.

Implication 1: Tokens must be invalidated when associated identities change.

3.2 Authorization

Authorization in distributed systems determines resource access for authenticated users. Given a user identity, the system checks permission policies (often ACLs) and grants appropriate access. The difference lies in execution: requests involve long operation chains across components with thousands of interactive events (RPCs, messages). Authorization may occur anywhere along these chains, locally or via remote services, creating many failure points.

Among 48 authorization vulnerabilities, 7 (14.9%) involve confused identity, 12 (25.5%) missing permission checks, 15 insufficient checks, and 13 overly permissive ACLs. The latter two are traditional vulnerabilities; we focus on the first two.

3.2.1 Confused Identity This vulnerability occurs when permission checks are performed against a delegated sub-component instead of the original requester—a classic confused deputy problem. In HBase-15132, when a regular user submits a region merge command to HMaster, HMaster forwards the request to RegionServer as the admin user running HMaster, not the requester. Thus, the merge always succeeds regardless of the actual requester.

With many components handling requests, developers easily forget to check the original requester, causing confused identity vulnerabilities. HBase addresses this by setting the original requester as identity for each dispatched request and performing end-to-end authorization.

Finding 2: When forwarding requests, developers often forget to check the original requester's identity, causing confused identity vulnerabilities (14.9% of authorization issues).

Implication 2: When dispatching requests, developers should verify the original requester's identity and provide end-to-end authorization.

3.2.2 Missing Permission Checks These vulnerabilities arise when developers forget permission checks for privileged operations (12 vulnerabilities). In ZooKeeper-CVE-2019-0201, a read request passes through three components without any permission checks, allowing users to read others' sensitive information.

Existing research targets these vulnerabilities but requires manual API specification and assumes privileged operations are guarded on most paths. However, these approaches are ineffective for distributed systems for three reasons: (1) 28/47 vulnerabilities involve operations never guarded anywhere; (2) 34.0% (16/47) use ad hoc permission checks via direct user variable tests, making manual specification difficult; (3) static analyses must handle inter-component interactions since checks and operations often occur on remote components.

Finding 3: Missing permission checks account for 25.5% (12/47) of authorization vulnerabilities, and existing detection techniques are ineffective.

Implication 3: Effective detection requires new techniques to automatically discover privileged operations (especially unguarded ones) and address inter-component interactions.

3.3 Information Flow

Information flow leaks (70/243, 28.8%) are prevalent due to diverse sensitive information sources/sinks and complex propagation across components via RPCs, bash commands, temporary files, etc., making proper sanitization challenging.

3.3.1 Various Sources and Sinks Table 3 shows leak distribution by source and sink types. 10 vulnerabilities leak configuration files (e.g., Kafka-4056 stores SSL passwords in configs printed to logs). 15 leak private user data; 19 leak

confidential system data (tokens, secret keys). The most common source is command-line arguments (26), as users frequently send credentials via command line to access external services like Amazon S3.

There are six sink types, with log files (51) dominating. Systems extensively log events for monitoring and diagnosis, accidentally printing sensitive data like passwords. 3 vulnerabilities leak to environment variables—in resource managers like YARN/Mesos, daemon processes fork children that inherit parent environment variables, exposing sensitive data. The remaining 4 sinks are traditional (e.g., Web UIs), previously researched.

Finding 4: Four source types and six sink types exist, with 72.9% (51/70) of leaks exposing sensitive data in logs.

Implication 4: Researchers should use our compiled sources/sinks to develop detection tools, paying special attention to log leaks.

3.3.2 Cross-Component Propagation All 70 vulnerabilities propagate sensitive information across components before reaching sinks. In MapReduce-CVE-2015-1776, an encryption secret key is encapsulated in objects and propagated across 20 components via 4 communication types (RPC, Thread, Event, Bash).

Static taint analysis is widely used for leak detection but typically tracks single-component flows. While some work models inter-component communication for Android, it doesn't address inter-node communication. Existing analyses must be extended for distributed systems.

Finding 5: All information flow leak vulnerabilities involve sensitive data spreading across multiple components.

Implication 5: Static taint analyses must be extended to handle inter-component communications.

Sanitization: 63 leaks (90.0%) stem from missing sanitization; 7 from incorrect sanitization. Systems often default to DEBUG/TRACE logging levels for diagnosis, causing leaks. Interestingly, 95% of logging-related leaks are fixed by masking sensitive data using regular expressions like "password:.*".

Finding 6: Missing sanitization accounts for 90.0% (63/70) of information flow leaks, with systems typically using regex to filter logs.

3.4 Vulnerable Input Validation

Like other services, distributed systems provide interfaces for job submission, data upload, etc., and are threatened by malformed inputs. Table 4 classifies 83 input validation vulnerabilities by input type: web, SQL, Bash, XML, file, and memory.

Vulnerable input validation is a classic problem causing various injections. Web interfaces face XSS threats like general web applications. Spark SQL's SQL interface is subject to SQL injection. Command-line injections occur when user inputs are passed directly as arguments. XML-like injections happen when parsing malformed config files (XML/JSON).

18 vulnerabilities involve user input files; 4 cause out-of-memory errors from large files. Memory-corruption errors from incorrect command-line inputs can cause serious vulnerabilities like Cloudbleed.

While similar to traditional applications, distributed system input validation vulnerabilities also involve multiple components. MR-CVE-2017-15173 is triggered when: (1) a user submits a job to YARN; (2) YARN uploads an XML job history file to HDFS; (3) an attacker replaces this file with one containing malicious "xi:include" commands; (4) when JobHistory reads the file, it leaks sensitive data. This requires cross-component interaction across two systems (YARN and HDFS).

Finding 7: Vulnerable inputs can propagate across multiple components before being handled.

Implication 6: Existing techniques can detect input validation vulnerabilities but need extension for inter-component communications.

3.5 Fault Tolerance

21 vulnerabilities relate to fault-tolerance schemes: 13 from vulnerable exception handling and 8 from vulnerable crash-recovery.

3.5.1 Vulnerable Exception Handling Distributed systems must handle exceptions from component failures or network issues, but not all are properly handled. 13 vulnerabilities exist: 7 from erroneous exception propagation, 6 from unhandled exceptions.

Erroneous Exception Propagation: While propagating exceptions to callers prevents catastrophic failures, it may unintentionally reveal sensitive information. In OpenStack-CVE-2019-14433, a "reboot vm" request is dispatched to admin-only components. When it fails (e.g., server already shutdown), an exception containing confidential information is thrown back to the user, leaking sensitive data.

Existing tools detect incorrect exception types but cannot recognize sensitive information propagation.

Finding 8: Exceptions with sensitive data can propagate from higher to lower security level components.

Implication 7: Tools must analyze exception propagation from high-security components.

Unhandled Exceptions: Uncaught exceptions from security mechanisms (authentication/authorization) cause vulnerabilities. In Kubernetes-CVE-2018-1002105 (Figure 3 [Figure 3: see original paper]): (1) a user sends an "upgrade:server1:websocket" request; (2) APIServer authenticates and forwards to Server1; (3.1) Server1 throws an unsupported operation exception; (4.1) APIServer doesn't handle it and creates an unprotected bridge port; (5) the user sends unauthorized requests to Server2 via this bridge; (6.1) APIServer forwards them. Thus, unauthorized requests can reach any server.

Finding 9: Unhandled exceptions from security mechanisms cause serious vulnerabilities.

Implication 8: These can be prevented using tools like Apirator to detect uncaught exceptions.

3.5.2 Vulnerable Crash Recovery 8 vulnerabilities arise during crash recovery. Distributed systems often experience node crashes, requiring recovery mechanisms. Previous work shows crash recovery bugs affect reliability; we focus on security impacts.

Since distributed systems prefer centralized security policy services, when nodes hosting these services crash, recovery may improperly bypass security policies. In MongoDB-SERVER-12616, when users send "admin commands" to Mongos, it queries authorization rules in Config Servers. If all Config Servers crash, Mongos cannot retrieve authorization info and erroneously treats access control as "disabled," granting permissions even when "authorization-enable" is true.

Finding 10: When security policy service nodes are unavailable, recovery can mistakenly bypass security policies.

Implication 9: Recovery must restore security policies for each unhandled request when policy service nodes crash.

4 RQ2: TRIGGERING

To understand triggering, we manually reproduced 109/243 vulnerabilities with custom PoC exploits, documenting reproduction steps. The remaining 134 could not be reproduced due to unmet configuration, input, or environment requirements.

Figure 4 [Figure 4: see original paper] shows triggering condition distributions by inputs/events, components, and nodes. Most (73/109, 67.0%) require fewer than 2 inputs/events; 33/109 (30.3%) need only one, suggesting high exploitability. All 109 reproducible vulnerabilities can be triggered in clusters of $ $4 nodes, and 99 (90.8%) involve >2 components.

Finding 11: Most vulnerabilities (73/109, 67.0%) trigger with one or two inputs/events, and all can be reproduced in small clusters ($ $4 nodes).

Implication 10: Given triggering inputs, vulnerabilities are easily exposed in small clusters.

Table 5 summarizes input/event types. Since vulnerabilities may depend on multiple inputs, totals exceed 109. 49/109 (45.0%) require specific configurations; 24/109 (21.8%) need crafted inputs. 55/109 (50.5%) are triggered by user requests (web/command-line/REST API queries); 28/109 (25.7%) require job execution. The remaining 51/109 (46.8%) need special events: 24/109 (22.0%) node changes (reboot), 15/109 (13.8%) security operations (permission changes), 3/109 (2.8%) upgrades, and 9/109 (8.3%) uncategorized events like network aborts.

Finding 12: 55/109 (50.5%) vulnerabilities require user requests; 46.8% require special events like node reboots or permission changes.

5 RQ3: IMPACTS

We classify security impacts into five categories (Table 6), characterizing each by direct impact. For example, a password exposure is "information disclosure" even though attackers could use it for further damage.

112/243 (46.5%) vulnerabilities cause information disclosure: 78 from information flow leaks, 34 from authorization/input validation issues. 58/243 (24.1%) enable privilege escalation (elevated access to protected resources) from erroneous authentication/authorization or insufficient input validation.

47/243 (19.5%) cause denial of service, often receiving high CVE severity scores. These typically crash primary nodes/daemons or exhaust resources, bringing down clusters. 13/243 (5.4%) enable remote code execution, all from insufficient input validation except Spark-CVE-2020-9480 (authentication bypass). One vulnerability (Kafka-CVE-2018-1288) causes data loss by allowing authenticated users to interfere with replication.

Finding 13: Vulnerabilities have significant impacts: information disclosure (112/243, 46.5%), privilege escalation (58/243, 24.1%), denial of service (47/243, 19.5%), and others.

6 RQ4: OBSERVABILITY

There are three symptom types: explicit errors (immediately observable, e.g., crashes), gray errors (diagnosable via careful log examination), and silent errors (no useful logging). In Kubernetes-CVE-2018-1002105, attack requests appear as normal user requests, making detection impossible via logs alone.

Most vulnerabilities (124/243, 50.9%) are exploited silently; 42/243 (17.3%) are gray errors, highlighting the need for improved accountability. However, this conflicts with information flow leak prevention, as logging sensitive data is risky.

Finding 14: Most vulnerabilities are exploited silently (124/243, 50.9%) or only observable through careful log examination (42/243, 17.3%). Developers should add audit logs around privileged operations.

7 RQ5: FIXING

7.1 Fixing Strategies

95.1% (231/243) of vulnerabilities are fixed by patching root causes. 8 are patched by disabling vulnerable components without fixing root causes (e.g., ZooKeeper-CVE-2018-8029 adds a config to disable problematic "4-letter word" commands). 4 information flow leaks are fixed by lowering logging levels rather than removing sensitive data. However, this is incomplete: YARN-5549 initially changed log level from INFO to DEBUG, but this hindered monitoring and didn't prevent leaks if admins set DEBUG. The final patch used a backend storage server with ACLs to store task launch logs, allowing users to view only their own logs.

Implication 11: Administrators must be cautious with configuration parameters, as misconfigurations create vulnerabilities.

Implication 12: Backend storage with proper ACLs provides detailed logging without information leaks.

7.2 Fixing Complexity

Following previous studies, we measured fixing complexity across five axes (Table 7): conversations, patches, files modified, LOC in final patch, and days to fix. Results from 82 vulnerabilities with complete data show fixes can be complex: up to 3,764 LOC across 50 files, with 152 conversations. Average patches/conversations are similar to other bugs, but average LOC is higher (60 vs. 34).

13 incomplete fixes introduced new vulnerabilities: 8 from flaws in original patches, 5 from fixing only one component while others had the same vulnerability. For example, YARN-CVE-2016-3086 was patched on all platforms except Linux, creating YARN-CVE-2017-15718.

Finding 15: Fixes can be complex (up to 3,764 LOC, 50 files).

Finding 16: 5 incomplete fixes failed to patch all vulnerable components.

8 LESSONS LEARNED

8.1 Static Analyses

Our taxonomy of root causes across five security mechanisms suggests specific detection techniques:

Missing Token Invalidation Detection: Tokens are widely used for authentication. Finding 1 shows tokens may not be invalidated when identities change. A static analysis could identify identity-modifying operations and check for consistent token invalidation. Similar issues affect sessions and cookies, warranting further investigation.

Confused Identity Detection: Requests forwarded across components cause confused identity vulnerabilities. We can enforce APIs to forward original requester identity and ensure authorization checks occur before request handling.

Missing Permission Check Detection: Finding 3 highlights that many bugs involve operations never guarded anywhere, making existing consistency-checking approaches ineffective. New techniques must automatically discover privileged operations, especially unguarded ones, and address inter-component interactions.

Information Flow Leak Detection: Static taint analysis is widely used but must be extended for inter-component communications, as sensitive data propagates across components (Finding 5). Our compiled sources/sinks (Finding 4) can aid practical taint analysis development.

Fix-Guided Detection: Complex fixes often miss vulnerable components (Finding 15). Clone detection techniques can identify similar vulnerabilities across components from known bugs.

8.2 Testing

Testing approaches like fuzzing have important implications:

Interaction Coverage: Finding 7 shows vulnerabilities commonly involve multiple components, suggesting interaction coverage as an effective test metric. Finding 10 indicates fault injection on security policy components can expose vulnerable recovery processes.

Environment and Inputs: All vulnerabilities reproduce in small clusters ($ $4 nodes) and trigger with 1-2 inputs/events (Finding 11). Inputs should include user requests, permission changes, etc. (Finding 12).

Testing Oracles: Most vulnerabilities exploit silently (Finding 14), making oracle development challenging. Audit logs must be enriched to detect violations. While existing studies enhance logs for diagnosis, none focus on audit logs for security violations—an important research direction.

9 RELATED WORK

Empirical Studies on Vulnerabilities. Liu et al. studied 3,938 vulnerabilities across five projects, analyzing distribution, dependency, and recurrence. Other work examined Android, web application, and Docker vulnerabilities. Kim and Lee analyzed cloned vulnerabilities. Li and Paxson studied patching

characteristics for 3,000+ vulnerabilities. However, none focus on distributed system vulnerabilities. While many studies discuss cloud security generally, only limited work examines Hadoop vulnerabilities from CVE. Our work is the most comprehensive study of distributed system vulnerabilities.

Empirical Studies on Bugs in Distributed Systems. CBS studied 3,000+ distributed bugs; CREB examined 103 crash recovery bugs; Lu et al. studied node change bugs; TaxDC developed a taxonomy of concurrency bugs; Yuan et al. found improper exception handling causes critical failures; Chen et al. showed 74% of exception bugs affect availability. Our work demonstrates improper exception handling also causes security vulnerabilities.

10 CONCLUSION

We present VulCloud, the most comprehensive study of vulnerabilities in distributed cloud systems. Analyzing 243 vulnerabilities from 16 widely-deployed systems, our findings reveal important implications for combating these threats. We believe this work will inspire new research to better secure distributed systems.

Note: Figure translations are in progress. See original paper for figures.