Understanding Exception-Related Bugs in Large-Scale Cloud Systems

Haicheng Chen[†], Wensheng Dou[‡], Yanyan Jiang[*], Feng Qin[†]
[†] Department of Computer Science and Engineering, The Ohio State University, United States
[‡] State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences, China
[*] State Key Lab for Novel Software Technology, Nanjing University, China
[†] {chen.4800, qin.34}@osu.edu, [‡] wsdou@otcaix.iscas.ac.cn, [*] jyy@nju.edu.cn

***Abstract*—Exception mechanism is widely used in cloud systems. This is mainly because it separates the error handling code from main business logic. However, the huge space of potential error conditions and the sophisticated logic of cloud systems present a big hurdle to the correct use of exception mechanism. As a result, mistakes in the exception use may lead to severe consequences, such as system downtime and data loss. To address this issue, the communities direly need a better understanding of the exception-related bugs, i.e., *eBugs*, which are caused by the incorrect use of exception mechanism, in cloud systems.**

In this paper, we present a comprehensive study on 210 eBugs from six widely-deployed cloud systems, including Cassandra, HBase, HDFS, Hadoop MapReduce, YARN, and ZooKeeper. For all the studied eBugs, we analyze their triggering conditions, root causes, bug impacts, and their relations. To the best of our knowledge, this is the first study on eBugs in cloud systems, and the first one that focuses on triggering conditions. We find that eBugs are severe in cloud systems: 74% of our studied eBugs affect system availability or integrity. Luckily, exposing eBugs through testing is possible: 54% of the eBugs are triggered by non-semantic conditions, such as network errors; 40% of the eBugs can be triggered by simulating the triggering conditions at simple system states. Furthermore, we find that the triggering conditions are useful for detecting eBugs. Based on such relevant findings, we build a static analysis tool, called *DIET*, and apply it to the latest versions of the studied systems. Our results show that DIET reports 31 bugs and bad practices, and 23 of them are confirmed by the developers as "previously-unknown" ones.

I. Introduction

Exception mechanism is widely used to handle errors in cloud systems. At the time of this writing, about 7% of the source code in twelve popular open source distributed systems [1] involves exception mechanism (Figure 1), i.e., throwing exceptions, or being enclosed in *try*, *catch*, or *finally* code blocks. Such a widespread use of exception mechanism is mainly due to its advantages over the traditional checking-return-value mechanism [2]. First, it separates the error handling code from main business logic, making programs easier to reason about. Second, the runtime system automatically propagates exceptions up along the call stacks until they are caught, so that error conditions will not remain unnoticed. Third, it allows developers to combine multiple exceptions by using their common superclass exception, providing greater flexibility to write the error handling code.

Unfortunately, highly diverse environments and system complexity make exception handling in cloud systems prone to errors, which can severely hurt system reliability [3], [4]. When implementing a cloud system, developers need to constantly anticipate various conditions that may cause exceptions. Such conditions can either come from the external environment (e.g., a remote node is unreachable), or be caused by internal program states (e.g., a variable is set to a wrong value). Furthermore, the sheer scale of cloud systems (both hardware size and software complexity) dramatically increases the hurdle of correct exception handling. In this paper, we refer to the mistakes in using exception mechanism as exception-related bugs, or *eBugs*.

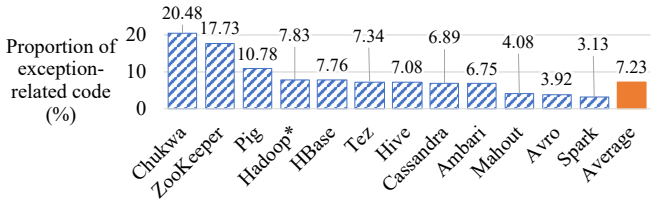


Fig. 1. Statistics of source code (excluding empty lines) that involves exception mechanism in twelve cloud systems. * The Hadoop project includes Hadoop common, Hadoop MapReduce, HDFS, and YARN.

Existing studies on eBugs mainly investigate the root causes based on source code patterns [5]–[8], the relation between eBugs and certain language features (e.g., aspect-oriented programming [9] and Android abstractions [6], [8]), and developers' perception on eBugs [5], [6]. While having discovered useful characteristics of eBugs, none of these studies consider the exception triggering conditions and their relations with eBugs. These relations are essential for understanding the root causes and facilitating exposure and detection of eBugs. Cloud systems often encounter complicated external and internal triggering conditions for exceptions. This unique characteristic motivates us to investigate the root causes of eBugs through understanding their exception triggering conditions.

In this paper, we perform a comprehensive study on eBugs in cloud systems from the perspective of triggering conditions. In particular, this study covers 210 well-documented eBugs selected from about 5,000 exception-related JIRA [10] issues across six popular open source cloud systems, including Cassandra [11], HBase [12], HDFS [13], Hadoop MapReduce [14], YARN [15], and ZooKeeper [16]. We thoroughly analyze both their bug reports and fixing patches to answer

the following key research questions:

- ***RQ1****: How are eBugs triggered in cloud systems?* It helps us understand the conditions that trigger eBug-bound exceptions. Our findings can benefit developers to effectively expose eBugs in cloud systems.
- ***RQ2****: What is the relation between the triggering conditions and the root causes of eBugs?* Our in-depth analysis on the root causes and their relations with the triggering conditions can facilitate eBug detection in cloud systems.
- ***RQ3****: What are the impacts of eBugs on cloud systems?* By analyzing their impacts, we can understand the severity of eBugs in cloud systems.

To the best of our knowledge, this is the first comprehensive study on eBugs in real-world cloud systems, and the first one from the perspective of triggering conditions. Through this study, we have obtained many interesting findings that open up new research opportunities to combat eBugs in the cloud. The main findings are: (1) More than half (54%) of the studied eBugs are exposed by non-semantic triggering conditions, most (75%) of which are network errors and file system errors (Finding 1). (2) Most (86%) of the eBugs do not have strong timing requirements on their triggering conditions (Finding 2). (3) 41% of the eBugs are caused by handling an exception in an overly-general (thus incorrect) way. Among them, many (34%) are caused by incorrectly applying the same handling operations to the exceptions that have different triggering condition types (Finding 7). (4) 10% of the eBugs are caused by creating exception objects that do not describe their triggering conditions accurately (Findings 3, 4, and 5). (5) Most (74%) of the eBugs affect the dependability of cloud systems, e.g., by crashing nodes or losing data, and developers consider most (82%) of them severe (Finding 8).

These findings show the severity of eBugs in cloud systems, while revealing many new opportunities to combat them. First, we can expose eBugs by simulating non-semantic triggering conditions at simple system states, e.g., consistent global states. Second, we can detect eBugs by analyzing exception triggering conditions. For instance, we can detect inaccurate exceptions by checking whether the exceptions accurately describe their triggering conditions. As another example, we can detect overly-general handlers by examining if the same handling code is applied to exceptions with different triggering condition types. Based on these findings, we build a static analysis tool, called *DIET*, to detect inaccurate exceptions by analyzing their exception classes and error messages. By applying DIET to the latest versions of the studied systems, we find 31 new bugs and bad practices. At the time of this writing, developers have confirmed 23 of them. Note that, an inaccurate exception can be an *eBug* if it affects the correctness or performance of a checked system or a *bad practice* if it may introduce a potential eBug in future system versions.

In summary, we make the following key contributions:

- We present the first comprehensive study on eBugs from the perspective of triggering conditions in six widely-deployed cloud systems.

TABLE I
INVESTIGATED BUG REPORTS IN THE STUDIED SYSTEMS

System	CA	HB	HF	MR	YN	ZK	Total
Retrieved	1,336	1,576	763	460	457	210	**4,802**
Studied	40	92	31	16	23	8	**210**

- We unveil many interesting findings and explain their implications for combating eBugs in cloud systems. For example, we find that triggering conditions and their relations with root causes provide valuable information for the developers of cloud systems.
- Based on our findings, we build a static analysis tool, called DIET, and evaluate it using the latest versions of the studied systems. We have exposed many new bugs and bad practices that have been confirmed by the developers of these systems.
- We provide a large benchmark of eBugs in cloud systems, which can be used to evaluate the effectiveness of the tools that expose and detect eBugs in cloud systems. Our eBug database is available at [17].

II. METHODOLOGY

A. Target Systems

To understand the characteristics of eBugs in real-world cloud systems, we select the target systems based on three criteria: (i) The systems must be diverse for an unbiased dataset. (ii) The systems should be mature and popular, so that we can understand the real problems faced by developers. (iii) The systems should be open source and have public issue tracking systems.

With these requirements in mind, we identify the following six cloud systems: (i) Cassandra [11], a highly available peer-to-peer NoSQL database; (ii) HBase [12], a master-slave NoSQL database; (iii) HDFS [13], a distributed file system; (iv) Hadoop MapReduce [14], a distributed data processing framework; (v) YARN [15], a distributed resource management system; and (vi) ZooKeeper [16], a distributed coordination service.

B. EBug Collection

All the target systems use JIRA [10] to manage their issues. Over a time span of more than ten years (from 2007 to 2018), more than 60,000 issues were submitted for these systems. It is time-consuming and impractical to manually inspect all these issues and identify eBugs from them. We therefore apply a few filtering rules to identify the relevant issues.

First, we use the following JQL (JIRA Query Language) statement to retrieve potentially relevant issues:

```
issuetype = Bug AND status IN (Resolved,
Closed) AND resolution = Fixed AND text
~ "exception*"
```

With this JQL statement, we narrow down all the issues to only *fully-resolved* (*Resolved* or *Closed* in *status*) and *fixed* bugs

that contain the keyword *exception* or *exceptions*. This JQL returns us with 4,802 issues (Row "Retrieved" in Table I[1]).

We further narrow down the retrieved issues by requiring each selected bug report to include a full exception stack trace and a fixing commit. The exception stack trace and the fixing commit are critical for us to fully understand an eBug because they contain information like what exception is thrown and how the eBug occurs. We further remove the ones whose bugs are located in test files or non-Java files, because they are not related to the core functionality of the target systems. This leaves us with 1,561 reports.

Finally, we manually inspect the remaining 1,561 reports and keep the ones that are related to exception mechanism as discussed in §III. This leaves us with 210 issues for further analysis (Row "Studied" in Table I). In this paper, we denote an eBug using its bug ID in JIRA, i.e., `SYS-###`, where `SYS` is the system name and `###` is the eBug's issue ID.

C. EBug Analysis

To answer our three research questions, we perform an in-depth analysis on each eBug based on the bug description (including the exception stack trace), the discussion among developers in the report, and the source code of the target system (including the fix). We also refer to online resources, e.g., documentation, to facilitate our understanding on eBugs. Through this process, we recover the full picture of how each eBug is triggered (RQ1), how the target system incorrectly uses exception mechanism (RQ2), and how each eBug affects the system (RQ3). Then, we classify eBugs according to their triggering conditions, root causes, and bug impacts.

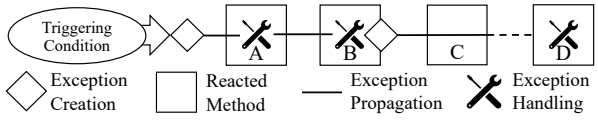
D. Threats to Validity

To maintain the accuracy of our study, we employ different measures to improve our understanding. For example, we use the final fixing commit and sometimes reproduce an eBug to confirm its triggering condition and root cause.

Even though we try to be unbiased, readers need to understand the following limitations of our study. First, all of our subject systems are open source cloud systems. EBugs in commercial cloud systems or other types of software systems may have different characteristics. Additionally, we may miss some eBugs due to our selection process. For example, we may exclude eBugs that do not have full exception stack traces in their bug reports. Finally, we only study eBugs in Java. While the exception mechanism in Java is generally similar to the ones in other languages (e.g., C++, C#, and Python), Java has a few unique features, which may affect developers' practice in using exceptions. Therefore, readers need to be cautious when extending our findings to other scenarios.

III. Exception Mechanism

Figure 2 shows the model of exception mechanism. During program execution, an unexpected error, referred to as a **triggering condition**, occurs. Based on the error type, Java runtime creates and propagates an exception. The **propagation** starts from a `throw` statement and ends at a `catch` statement, along the call stack in the reverse call order. After a method catches the exception and performs exception handling (Boxes A, B, and D), it can optionally re-throw the caught exception (Box A), or wrap the caught exception in a new exception before re-throwing it (Box B). Either way, the re-thrown exception starts a new propagation. Along the propagation path, some methods (including the one where the `throw` statement locates) may not catch the exception. Instead, these methods can specify the exception class in the method signatures using the `throws` keyword. In this way, a method notifies its callers that it can propagate the specified exception, so that the callers can implement proper exception handling as needed. Box C represents the scenario where a method does not catch the exception but specifies it in the signature. We consider both catching an exception and specifying an exception in the signature as a method **reacting** to the exception.




```
1 void B(...) throws OtherException {
2   try { A(...);
3   } catch (SomeException e) {
4     someHandling(...);
5     throw new OtherException(e);
6   } }
```

Fig. 2. The model of exception mechanism, and the code snippet of method `B()` that handles the exception thrown from method `A()`.

The lower-half in Figure 2 shows the code snippet of `B()`. When `A()` throws a `SomeException`, `B()` catches and handles it (Lines 3-4). `B()` then wraps the `SomeException` in a newly created `OtherException`, which it throws at Line 5. `B()` also specifies the re-thrown `OtherException` (Line 1) so that its callers can react to the exception accordingly.

We further define the following terms used in our study:

- **Root exception**: The initially-created exception due to the triggering condition, e.g., the left diamond shape in Figure 2.
- **Cause exception**: If an exception e_a is wrapped by another exception e_b, e_a is the *cause exception* of e_b. In the example above, the `SomeException` is the cause exception of the `OtherException`.
- **EBug**: A bug that occurs in creating, throwing, catching, or handling an exception.

IV. Triggering Conditions

The triggering condition is the key to trigger an eBug. In this section, we study both the types of triggering conditions (§IV-A) and their timing requirements (§IV-B).

A. Triggering Condition Types

We first examine whether the triggering conditions are related to program semantics. Semantic conditions are specific to each individual target system, while findings on non-semantic

[1]For simplicity, we use CA, HB, HF, MR, YN, and ZK to represent Cassandra, HBase, HDFS, MapReduce, YARN, and ZooKeeper, respectively.

TABLE II
TRIGGERING CONDITIONS OF THE STUDIED EBUGS AND THEIR TYPICAL SCENARIOS IN EACH SYSTEM

Triggering Condition (# eBugs)		Scenario (# eBugs)	CA	HB	HF	MR	YN	ZK
Non-semantic condition (114)	Network error (46)	Premature disconnection (17)	0	8	5	1	1	2
		Local timeout (12)	2	5	2	0	3	0
		Connection refused (11)	1	8	0	1	1	0
		Other network errors (6)	0	6	0	0	0	0
	File system error (40)	File corrupted (23)	10	11	1	1	0	0
		File not found (13)	3	5	3	2	0	0
		Other file system errors (4)	2	0	2	0	0	0
	Out of resource (16)	Out of memory (5)	1	2	1	1	0	0
		Out of disk space (5)	1	0	1	1	1	1
		Port conflicted (3)	1	1	1	0	0	0
		Out of other resources (3)	0	0	3	0	0	0
	Untimely interrupt (12)	Thread interrupted when invoking a blocking method (12)	2	1	0	3	6	0
Semantic condition (96)	-	-	17	45	12	6	11	5
Total	**210**		**40**	**92**	**31**	**16**	**23**	**8**

conditions are general to a broader class of cloud systems. We notice that the majority of the studied eBugs (114 out of 210) are triggered by non-semantic conditions such as a node being unreachable. The remaining eBugs (96 out of 210) are triggered by semantic conditions such as a variable being assigned with an incorrect value.

To better understand non-semantic conditions, we further classify them into the following four categories based on the error types: (i) *Network error*: The system encounters a failed network connection. (ii) *File system error*: The system encounters a failed file system operation, e.g., opening a non-existent file. (iii) *Out of resource*: System resource, such as memory space, is used up. (iv) *Untimely interrupt*: A system interrupt occurs when a thread is sleeping or waiting, e.g., after calling `Thread.sleep()`. Table II shows the typical scenarios of each condition type and their distribution.

Finding 1: *Most (75%) of the non-semantic conditions are either network errors or file system errors.*

1) Network Error: Cloud systems are deployed on network and thereby strive to handle network errors in both design and implementation phases. However, network error is still the most common (40%) type of non-semantic condition in the studied eBugs. With further investigation, we find that most (87%) of the network errors occur in the following three scenarios: (i) *Premature disconnection*. For example, in HDFS-7009, a DataNode throws an `EOFException` when a NameNode terminates the connection after sending partial data to the DataNode. (ii) *Local timeout*. For instance, in HBASE-6299, an HMaster throws a `SocketTimeoutException` if it times out before receiving a response from a RegionServer. (iii) *Connection refused*. In YARN-196, a NodeMagager tries to connect to a ResourceManager that has not started. As a result, the remote operating system rejects the connection, causing the NodeManager to throw a `ConnectException`. The remaining (13%) network errors are caused by various reasons, such as failed routing and unsuccessful host name resolution.

2) File System Error: Cloud systems store critical user and system data on file systems. Since file systems are never perfect [18], cloud systems are expected to correctly react to file system errors. For example, when failing to read a data chunk, a distributed file system (e.g., HDFS) may start the data recovery process using a remote data replica. However, we find that a significant number of eBugs are triggered by file system errors. Among these errors, the most frequent scenarios are files being corrupted and files not found: (i) *File corrupted*. For example, in CASSANDRA-12728, a Cassandra server throws an `EOFException` when it unexpectedly reads the end of a truncated hint file. (ii) *File not found*. For instance, in HBASE-9563, a restarted HMaster throws a `FileNotFoundException` when it fails to find a znode file to read. Other file system errors include access mode violation, file path resolution failure, and disk failure.

3) Out of Resource: Cloud systems interact with system resources extensively. However, the desired resource may not be always available. 14% of the non-semantic eBugs are triggered when certain resource is exhausted. Among the 16 out-of-resource eBugs, 13 are triggered due to running out of (i) *memory space*, (ii) *disk space*, or (iii) *host ports*. For example, in CASSANDRA-11540, a Cassandra server throws a `BindException` when it tries to bind an occupied port. Other resources include opened file handlers and disk quotas.

4) Untimely Interrupt: Cloud systems are highly concurrent. They may try to perform two conflicting operations at the same time. When this happens, one operation may stop the other via interrupt. We find that, a considerable amount (11%) of non-semantic eBugs are triggered by untimely interrupts. For example, YARN-2846 is triggered when one thread of a

TABLE III
TIMING REQUIREMENTS ON EBUG TRIGGERING CONDITIONS

Condition Type	Timing Requirement		
	Weak	Moderate	Strong
Network error	9	36	1
File system error	32	5	3
Out of resource	5	11	0
Untimely interrupt	0	10	2
Semantic condition	39	34	23
Total	**85**	**96**	**29**

NodeManager process is checking the status of a container, while another thread is shutting down the whole process.

5) Semantic Condition: About half (46%) of the root exceptions are triggered by *semantic conditions*. Semantic conditions are closely related to program logic, and thereby can be specific to each individual system. For example, in CASSANDRA-5725, a Cassandra server throws an `UnknownColumnFamilyException` when it tries to access a nonexistent column family. From this example, we can see that triggering this type of eBugs needs domain knowledge about the target systems, e.g., column family is a data structure used in Cassandra to organize both user data and system data. However, other systems, e.g. HDFS, do not have this concept. Since 46% of the conditions are relevant to semantics, they call for more attention from our communities.

B. Timing Requirements on Triggering Conditions

Finding 2: *Most (86%) of the eBugs do not have strong timing requirements on their triggering conditions.*

EBugs can be triggered only when triggering conditions occur at certain system states. To understand the difficulty of triggering eBugs in cloud systems, we further analyze each eBug's timing requirement on its triggering condition. We use a similar measurement as a previous bug study in cloud systems [19], and classify the timing requirements into three categories, as shown in Table III.

- **Weak** (85 eBugs). To trigger this type of eBugs, the triggering condition can occur at any consistent global state, or before the system starts. For example, ZOOKEEPER-2757 can be triggered whenever a user issues a delete command with an invalid pathname.
- **Moderate** (96 eBugs). The triggering condition needs to occur on a node when it is in certain states. To trigger these eBugs, we only need to consider the runtime state of one node. For example, MAPREDUCE-5251 can be triggered by simulating out of disk space when a reduce task tries to write a map output to disk. There is no need to check the states of other nodes in the system.
- **Strong** (29 eBugs). The triggering condition needs to occur on a node when both the current node and other nodes are in certain states. To trigger these eBugs, we need to consider the states of multiple nodes. For example, YARN-3842 can only be triggered when a MapReduce ApplicationMaster asks a NodeManager to start a container, while the NodeManager is still in the initialization phase.

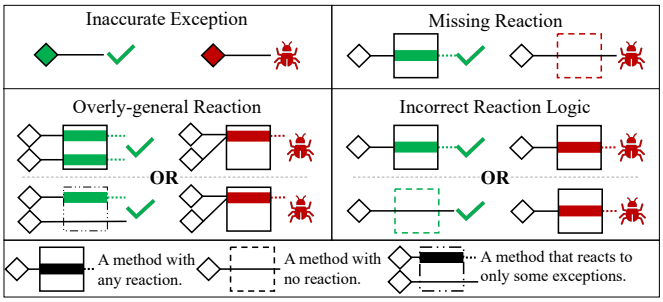


Fig. 3. Four different types of eBug root causes. For each type, we show the correct version on the left (green) and the buggy version on the right (red).

TABLE IV
STATISTICS OF EBUG ROOT CAUSES

Root Cause	eBug #	CA	HB	HF	MR	YN	ZK
Inaccurate exception	21	3	8	4	3	3	0
Missing reaction	36	12	11	3	3	4	3
Overly-general reaction	87	13	42	14	6	11	1
Incorrect reaction logic	66	12	31	10	4	5	4
Total	**210**	**40**	**92**	**31**	**16**	**23**	**8**

V. ROOT CAUSES

Based on the exception mechanism shown in Figure 2, eBugs can be classified into three categories: (1) **Inaccurate exception**, if the eBug is caused by creating an inaccurate exception; (2) **Missing reaction**, if the eBug is caused by neither catching nor specifying an exception in the method signature; and (3) **Incorrect reaction**, if the eBug incorrectly reacts to an exception. Incorrect reaction can be further broken down into **overly-general reaction**, where different exceptions are incorrectly handled in the same way, and **incorrect reaction logic**, where the reaction logic is incorrect for all exceptions. Figure 3 illustrates these four types of eBugs, and Table IV shows their distribution. Note that, previous studies have also classified eBugs based on their root causes [5]–[8]. Unlike them, the focus of our classification is on the relation between the triggering conditions and the root causes (RQ2).

A. Inaccurate Exception

A newly-created exception is expected to accurately represent the triggering condition so that the system can correctly react to the error. This requires the exception to instantiate the correct *class* and contain correct information such as an *error message* and a *cause exception*. Table V shows the number of eBugs where the exception instantiates a wrong class, has a wrong error message, or misses a cause exception.

1) Wrong Exception Class: Exception class is the primary source to indicate the triggering condition. An exception instance with a wrong class will not be handled correctly.

TABLE V
THE TYPE DISTRIBUTION OF INACCURATE EXCEPTION EBUGS

Type	Wrong Class	Wrong Message	Lacking Cause	Total
eBug #	13	5	3	21

```
1  void updateMetaLocation() throws IOException {
2    if (waitForRootServerConnection() == null)
3  -   throw new NullPointerException(...);
4  +   throw new IOException(...);
5    }
6    void process() {
7      try { updateMetaLocation();
8      } catch (IOException e) { cleanup(); }
9    }
```

Fig. 4. EBug HBASE-3164. A NullPointerException is thrown when a remote node is unreachable.

We find two ways of creating incorrect exceptions: (i) In five eBugs, the incorrect exception class is the superclass of the intended one. (ii) In the other eight eBugs, the incorrect exception class has no relation with (i.e., neither a superclass nor a subclass of) the intended one.

Finding 3: *Using a superclass of the intended exception makes it difficult to perform correct exception handling.*

We find that IOException is the only culprit class in all the five eBugs that use a superclass of the intended exception. For example, in HDFS-8224, a DataNode throws an IOException when it tries to read the checksum granularity from a corrupted file. Since other disk failures, e.g., out of disk space, also trigger IOException, the DataNode cannot differentiate the exceptions to perform data recovery only for file corruption. To fix this bug, a dedicated subclass exception, InvalidChecksumSizeException, is used to denote the case where the file storing the checksum granularity is corrupted.

Finding 4: *In half of the eBugs that create a totally misleading exception, the exception class is inconsistent with its triggering condition.*

When the newly-created class is neither a superclass nor a subclass of the intended one, the exception cannot be correctly caught by its intended catch block. Take eBug HBASE-3164 in Figure 4 as an example. When a RegionServer opens a META region, it needs to report the update to the RootServer. If the RootServer is currently unreachable, waitForRootServerConnection() will return a null (Line 2). Instead of throwing an IOException that semantically matches the triggering condition (i.e., network error), the buggy code throws a NullPointerException (Line 3). "We actually throw the NPE [when] it's not an actual NPE", a developer also points out. As a result, even though a proper catch block exists (Line 8), it will not catch the exception, leaving the META region inaccessible.

TABLE VI
THE TRIGGERING CONDITIONS AND EXCEPTION CLASSES OF FOUR EBUGS WITH TOTALLY MISLEADING EXCEPTION CLASSES

Bug ID	Triggering Cond.	Exception Class
CASSANDRA-11448	Out of resource	RuntimeException
HBASE-3164	Network error	NullPointerException
HDFS-2484	File system error	LeaseExpiredException
YARN-2846	Untimely interrupt	IOException

TABLE VII
EXCEPTION CLASSES THAT ARE TRIGGERED MORE THAN ONCE BY NON-SEMANTIC TRIGGERING CONDITIONS

Triggering Cond.	N†	P*	Top 4 Exception Class (#)♣
Network error	8	91%	ConnectException (11) IOException (10) SocketTimeoutException (6) EOFException (5)
File system error	4	78%	EOFException (11) FileNotFoundException (10) IOException (8) IllegalArgumentException (2)
Out of resource	3	75%	OutOfMemoryError (5) IOException (4) BindException (3)
Untimely interrupt	1	75%	InterruptedException (9)

† N is the number of exception classes. * P is the percentage of eBugs the classes cover the triggering condition. ♣ Due to space limit, we only show the top four exception classes for network errors.

Among the eight eBugs in this category, four of them are triggered by non-semantic conditions. We find that all the incorrect classes are inconsistent with their conditions (Table VI). This makes us wonder: Does each type of non-semantic condition have a set of frequently triggered exception classes? If so, the inconsistency between the common classes and the triggered ones may help detect inaccurate exception eBugs.

We analyze all the 114 eBugs with non-semantic triggering conditions. For each eBug, we use the root exception class because it is directly related to the triggering condition. To prevent using the wrong exception classes in inaccurate exception eBugs, we employ their fixing patches to retrieve the correct exception classes.

Finding 5: *For non-semantic triggering conditions, a few (1-8) exception classes can cover most eBugs (75-91%).*

As Table VII shows, each condition type triggers only a few exception classes more than once, and these classes cover a majority of eBugs with the corresponding condition type. For example, three exception classes cover 75% of the eBugs that are exposed by the triggering condition of "out of resource". We also notice that these frequently triggered classes do not include the misleading ones shown in Table VI. For example, the RuntimeException in CASSANRA-11448 is not a frequently triggered class for the "out of resource"

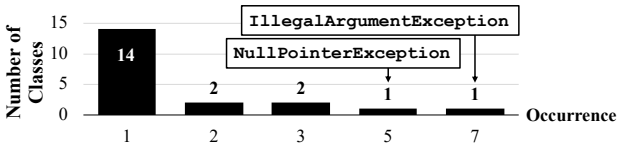


Fig. 5. Exception classes that occur in missing reaction eBugs.

```
1   void call() {
2     try { reacquireContainer(...);
3   + } catch (InterruptedIOException e) {
4   +   LOG.warn(...);
5     } catch (IOException e) {
6       // FileNotFoundException can reach here.
7       deactivateContainer(...);
8     } }
9
10  int reacquireContainer(...) throws IOException {...}
```

Fig. 6. EBug YARN-5103. Handling IOException and its subclasses in the same way stops a running YARN container prematurely.

triggering condition.

2) Wrong Error Message or Lacking Cause Exception: When an exception carries a wrong error message, or an exception does not wrap a cause exception, developers may lack critical information about the triggering condition for diagnosing the failure. For example, in HDFS-7899, if a DataNode disconnects with an HDFS client, the client will throw an EOFException with a message stating: "Premature EOF: no length prefix available". As the bug reporter points out, this error message "is not very clear to a user" because it does not indicate that the DataNode is unreachable.

B. Missing Reaction

Once an exception is thrown, some methods in the call stack need to react to it. If these methods neither catch the exception, nor specify it in their signatures, we refer to this type of mistake as a *missing reaction* eBug. We examine the fixing patch of an eBug to identify which methods should react to an exception. To gain more insights, we also analyze the exception classes that are missed.

Finding 6: *IllegalArgumentException (19%) and NullPointerException (14%) are the dominant exception classes that cause missing reactions.*

As shown in Figure 5, we find that many exception classes can be missed by developers. Among them, two classes are more frequent than others: IllegalArgumentException and NullPointerException. For example, in CASSANDRA-5701, a Cassandra server throws an IllegalArgumentException when a client queries a nonexistent column family from it. Since none of the methods in the call stack handles this exception, the Java thread crashes, and the client gets disconnected abruptly.

TABLE VIII
THE RELATION BETWEEN THE TRIGGERING CONDITIONS OF THE INCORRECTLY REACTED EXCEPTION AND THE CORRECTLY REACTED ONES IN OVERLY-GENERAL REACTION EBUGS

Relation	Same Type	Different Types	Unknown	**Total**
eBug #	48	30	9	**87**

C. Overly-General Reaction

If a method can throw multiple exception classes, it is a norm for developers to specify only their common parent class in the method signature. For example, method reacquireContainer() in Figure 6 specifies only an IOException (Line 10), but it can throw subclasses like InterruptedIOException and FileNotFoundException.

However, this common practice poses a big challenge for accurate exception handling, because developers need to be aware of all the potential exceptions a method can throw. As a result, developers can make overly-general reaction eBugs, i.e., incorrectly handling multiple exceptions in the same way while they should be treated differently. We find that overly-general reaction causes many (41%) eBugs.

For example, eBug YARN-5103 in Figure 6 incorrectly applies the same handling to IOException and its subclasses, such as InterruptedIOException and FileNotFoundException. When a NodeManager restarts, it will invoke reacquireContainer() to reload the information of a running container and wait for its completion (Line 2). If the NodeManager interrupts the waiting thread because it needs to restart again, reacquireContainer() will throw an InterruptedIOException. The method call() incorrectly catches and handles this exception in the same way as other IOExceptions (Lines 5-7). Therefore, the running container stops prematurely for a benign interrupt (Line 7). Instead, the method call() should catch the InterruptedIOException separately and let the container continue running (Lines 3-4).

Finding 7: *In many (34%) overly-general reaction eBugs, the incorrectly reacted exception and the correctly reacted ones are caused by different types of triggering conditions.*

Although multiple exceptions can be combined and handled in the same way, exceptions with different triggering condition types (i.e., network error, file system error, out of resource, untimely interrupt, and semantic condition in §IV-A) usually represent different errors and may require different handling. We analyze the relation between the correctly reacted exceptions (whose reaction is not changed in the fixing patch) and the incorrectly reacted one (as specified in the eBug report) in each eBug to see if they are triggered by different triggering condition types (Table VIII). If the incorrectly reacted exception and the correctly reacted ones are caused by different triggering condition types, we label the eBug as *different type*.

```
1  List<ServerName> fetchServerAddresses() {
2    try { return listServerNames();
3    } catch (KeeperException e) {
4  -   return null;
5  +   return new ArrayList<ServerName>(0);
6    } }
```

Fig. 7. EBug HBASE-4045. Instead of a null, the handler should return an empty ArrayList when server names cannot be retrieved.

TABLE IX
EBUG FAILURE SYMPTOMS

Symptom	eBug #
Node downtime	48
Incorrect error message	44
Data loss or potential data loss	31
Hang or performance downgrading	26
Resource leak/exhaustion	10
Operation failure†	51
Total	**210**

† We only consider an eBug as causing operation failure if it does not have any other symptom.

If their triggering condition types overlap, we label the eBug as *same type*. We cannot infer the triggering conditions of the correctly handled exceptions in nine eBugs due to insufficient information around the throw statements. So, we label them as *unknown*. We observe that, in many (34%) overly-general reaction eBugs, the exceptions are triggered by different condition types. For example in YARN-5103 (Figure 6), the InterruptedIOException is triggered by an untimely interrupt, while other IOExceptions are triggered by different condition types, like a FileNotFoundException triggered by a file system error.

D. Incorrect Reaction Logic

If a handler is incorrect for all the exceptions it handles, we say that it has *incorrect reaction logic*. Take HBASE-4045 in Figure 7 as an example. In HBase, when a RegionServer tries to replicate its data to a different cluster, it needs to fetch the destination server names from ZooKeeper (Line 2). If ZooKeeper is unreachable, an exception will be thrown. The handler catches it (Line 3) and returns a null (Line 4). The caller of fetchServerAddresses() does not expect the return value to be null, and dereferences it (not shown in the figure), which crashes the thread. Instead, the handler should return an empty ArrayList, which the caller can handle properly. Detecting this type of eBugs requires understanding the system logic, which remains a challenge for future work.

VI. BUG IMPACTS

We study the eBug impacts from two perspectives. First, we analyze their failure symptoms to understand how they affect the systems (Table IX). Then, we use the issue priority to infer if developers consider an eBug as a severe defect (Table X).

TABLE X
JIRA ISSUE PRIORITY OF EBUGS

Priority	Blocker	Critical	Major	Minor	Trivial	Total
eBug #	21	42	110	33	4	**210**

Finding 8: *74% of the eBugs affect the availability (e.g. node downtime) and integrity (e.g., data loss) of cloud systems. Moreover, developers consider most (82%) of them as severe defects (i.e., priority not lower than major).*

Overall, we find that, eBugs have various failure symptoms. Many of them affect the system availability (e.g., node downtime) and integrity (e.g., data loss). Sometimes, eBugs can turn a transient and benign error (i.e., an exception) into a severe failure. For example, in YARN-196, a NodeManager aborts only because it fails to register itself with the ResourceManager due to a transient network partitioning. A simple retry fixes the eBug, and allows the NodeManger to start.

We also find that, developers consider most (82%) of the eBugs as severe defects, i.e., having a priority of *blocker*, *critical*, or *major*. Even the seemingly most benign type of symptoms, i.e., *incorrect error message*, can cause much trouble for end users of cloud systems: two thirds of these cases are marked as major or a higher priority in JIRA.

VII. LESSONS LEARNED AND APPLICATIONS

Our study shows that eBugs seriously affect the dependability of cloud systems (Finding 8). In this section, we discuss implications for existing approaches and opportunities for new research to combat eBugs in cloud systems (§VII-A, §VII-B, and §VII-C). Additionally, we discuss our experiences in applying the findings to detect new inaccurate exceptions in the studied cloud systems (§VII-D).

A. Testing Cloud Systems under Adversarial Conditions

Software testing is critical in exposing bugs before software release. Many testing techniques have been proposed for exposing or detecting software bugs [20]–[22], but few are designed for cloud systems [23]–[25].

Cloud systems usually run in complex cluster environments, and may encounter different kinds of adversarial conditions, e.g., network errors and file system errors. Improperly handling these conditions can lead to severe consequences. As Figure 1 shows, cloud systems often use exception mechanism to handle adversarial conditions. However, Finding 1 indicates that there are issues in handling some conditions. Existing testing techniques on cloud systems have tried to inject adversarial conditions such as network partitioning [23], [25]–[28], file corruption, and out of disk space [29]. However, other adversarial conditions in Table II, such as connection refused, file not found, port conflict, and untimely interrupt, have not been attempted in cloud systems. Moreover, researchers and developers can use the triggering conditions summarized in §IV as a checklist to test cloud systems. For example, by

limiting the available memory and disk space during normal testing, eBugs that are triggered by either "out of memory" or "out of disk space" become more likely to be exposed.

To trigger an eBug, testing tools need to simulate the triggering condition at proper system states. Luckily, Finding 2 shows that most (86%) eBugs, e.g., ZOOKEEPER-2757 and MAPREDUCE-5251, have weak or moderate requirements on system states. This finding indicates that simple simulation of triggering conditions can expose most eBugs in cloud systems.

B. Avoiding eBugs in Cloud Systems

Our findings imply that enhancing exception flow analysis can help avoid eBugs. Exception flow analysis [30]–[35] helps developers better understand the exception propagation in the system and thereby better react to exceptions. Throughout our study, we consistently observe that the root causes of many eBugs are related to their exception triggering conditions (Findings 1, 4, 5, and 7). Therefore, by combining exception flow analysis with the related triggering conditions, developers can obtain deeper understanding about what error triggers an exception, and thus handle it correctly. For example, HBase defines 72 exception subclasses that extend `IOException`. For a method handling `IOException`, it will be greatly helpful to avoid overly-general reactions if developers can know each concrete exception and its triggering condition.

C. Detecting eBugs in Cloud Systems

Unlike existing eBug detection tools that focus on empty or incomplete exception handlers [3], overly-general handlers that stop the system [3], missing recovery operations [36], or violations of predefined exception propagation rules [37], our findings (Findings 3, 4, 5, and 7) reveal the important correlation between the root causes and triggering conditions, which suggests new opportunities for detecting eBugs.

Findings 3-5 indicate that inaccurate exceptions do not accurately describe their triggering conditions. Therefore, it is possible to detect these eBugs by checking if the exceptions are consistent with their triggering conditions. In this way, we can detect eBugs like HBASE-3164, where HBase incorrectly throws a `NullPointerException` for a network error. Table VII also provides the commonly triggered exceptions for each non-semantic condition type. Detection tools can use it as a checklist to detect inaccurate exceptions.

Similarly, Finding 7 shows that some overly-general reactions incorrectly apply the same handling code to the exceptions that are caused by different triggering condition types. This suggests a new way to detect overly-general reactions, i.e., by checking if the handled exceptions are triggered by different condition types. Therefore, we can detect eBugs like YARN-5103, where two exceptions (`InterruptedIOException` and `FileNotFoundException`) are triggered by different condition types (untimely interrupt and file system error, respectively) but are handled in the same way.

D. DIET: Detecting Inaccurate Exceptions using Triggering Condition Types

Findings 3 and 4 in §V-A show that inaccurate exceptions cannot describe the triggering conditions precisely, and may mislead developers to handle them incorrectly. We also obtain two interesting observations from our study: (1) There exist strong relations among exception classes and their triggering condition types (Finding 5). (2) An exception's error message usually convey information about its triggering condition type. Ideally, an exception's class and error message should convey the consistent information about its triggering condition type. If an exception's class and its error message imply different types of triggering conditions, the exception is likely to be inaccurate. Inspired by these observations, we build a static analysis tool, *DIET*, to automatically detect inaccurate exceptions, by inspecting the inconsistency between an exception's class and its error message.

1) DIET's Approach: DIET works in two phases: a learning phase and a detection phase. In the learning phase, given a set of (e, t) pairs, where e denotes a root exception and t denotes e's triggering condition type, DIET learns two probabilities: (i) $P_{c,t}$: the probability that the triggering condition of an exception with class c is of type t, and (ii) $P_{w,t}$: the probability that the triggering condition of an exception containing keyword w in its error message is of type t. In the detection phase, for a given root exception, DIET employs the above probabilities to examine whether its exception class and error message imply different triggering condition types. Note that, DIET focuses on root exceptions because they usually describe the triggering conditions more accurately than their wrapper exceptions. DIET identifies root exceptions by finding the ones that have no cause exceptions.

In the DIET design and experiment, we use the five triggering condition types summarized in §IV-A, i.e., network error, file system error, out of resource, untimely interrupt, and semantic condition. We further use all the 210 eBugs in our empirical study to learn $P_{c,t}$ and $P_{w,t}$.

Learn $P_{c,t}$: We first extract the root exception's class c_i and its corresponding triggering condition type t_i from each studied eBug e_i. This step generates 210 pairs of (c_i, t_i). For each exception class c in these 210 pairs, the probability that c's triggering condition is of type t is computed using the following equation:

$$P_{c,t} = \frac{\text{number of } (c_i, t_j) \text{ where } c_i = c, t_i = t}{\text{number of } (c_i, t_i) \text{ where } c_i = c} \quad (1)$$

Learn $P_{w,t}$: We first extract the root exception's error message m_i and its triggering condition type t_i from each studied eBug e_i. Since the root exceptions in 44 eBugs do not have error messages, this step generates 166 pairs of (m_i, t_i). Next, DIET extracts the unique keywords from each error message. Note that, DIET does not consider numbers, conjunctions, determiners, and adverbs as keywords because they do not reflect the essence of triggering conditions. For each (m_i, t_i) pair, DIET can generates multiple $(w_{i,j}, t_i)$ pairs,

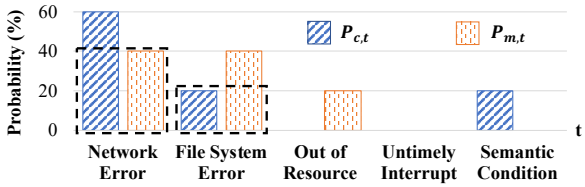


Fig. 8. The $P_{c,t}$ and $P_{m,t}$ of an exception. For example, when t is network error, $P_{c,t}$ is 60% and $P_{m,t}$ is 40%. The overlapping areas are highlighted with the dashed boxes. The total overlap is 60%.

where $w_{i,j}$ denotes the j_{th} unique keyword extracted from message m_i. For each keyword w in all these $(w_{i,j}, t_i)$ pairs, the probability that the triggering condition of keyword w is of type t is computed using the following equation:

$$P_{w,t} = \frac{\text{number of } (w_{i,j}, t_i) \text{ where } w_{i,j} = w, t_i = t}{\text{number of } (w_{i,j}, t_i) \text{ where } w_{i,j} = w} \quad (2)$$

Detect inaccurate exceptions: Given a target system, DIET extracts all its root exceptions, which have no cause exceptions. DIET then analyzes each root exception as follows.

First, DIET extracts the root exception's class c, and looks up the learned probability $P_{c,t}$ for each triggering condition type t summarized in §IV-A, which indicates how likely the triggering condition of the exception is of type t.

Second, DIET extracts the exception error message m, and unique keywords $(w_1, ..., w_n)$ from the error message m. For each keyword w_i, DIET looks up the learned probability $P_{w_i,t}$ for each triggering condition type t, which indicates how likely the triggering condition of keyword w_i is of type t. DIET computes the probability that the triggering condition of exception e is of type t by averaging $P_{w_i,t}$ for all unique keywords in the error message m.

$$P_{m,t} = \frac{\sum_{w_i \in m} P_{w_i,t}}{n} \quad (3)$$

Finally, DIET uses the following equation to compute the probability that the root exception's class and error message imply the same triggering condition type:

$$P_{\textit{same-type}} = \sum_{t \in \text{Five types}} \min(P_{c,t}, P_{m,t}) \quad (4)$$

Intuitively, $P_{\textit{same-type}}$ is the minimal common probability for all five triggering condition types. As shown in Figure 8, $P_{\textit{same-type}}$ can be represented as the overlapping area when plotting the $P_{c,t}$ and the $P_{m,t}$ for all five condition types in the same histogram. The smaller $P_{\textit{same-type}}$ is, the more likely the root exception's class and error message imply different types of triggering conditions, and the more likely the root exception is inaccurate.

2) Experiments on Cloud Systems: We evaluate DIET using the latest versions of the studied systems (Table XI). For these cloud systems, DIET extracts 18,125 exceptions in total (Row `Throw`), and 5,905 of them are considered as root exceptions (Row *Root ex.*). For each root exception, DIET calculates its $P_{\textit{same-type}}$ using Equation 4 (Row *Calculated*). If its $P_{\textit{same-type}} \leq 0.2$ (a configurable threshold), DIET will report it as an inaccurate exception. Finally, DIET reports 314 inaccurate exceptions (Row *Reported*). We manually inspected all the reports, and identified 31 candidates for real inaccurate exceptions (Row *Candidate*). Note that, DIET fails to calculate $P_{\textit{same-type}}$ for half of the root exceptions (Row *Calculated*), and DIET's false positive rate is high (283 out of 314 reported exceptions). This is mainly because we use only a small dataset (the studied 210 eBugs) to train DIET, and many exception classes and keywords of error messages in our experimental subjects are not contained in these 210 eBugs. This can be improved by training DIET with a larger dataset of eBugs.

Among the 31 candidates, we found two eBugs in which the inaccurate exceptions can cause failure symptoms. The remaining 29 candidates are bad practices, in which their exception classes and error messages are indeed misleading. Although a bad practice has not caused failure symptoms, it may introduce eBugs in the future because it is difficult to correctly handle an inaccurate exception. For example, since the inaccurate exceptions in eBugs `HDFS-8224` and `HBASE-3164` (discussed in §V-A1) are misleading, when developers implemented the corresponding exception handlers later, the exceptions were not handled properly.

We report all these bugs and bad practices to developers of these cloud systems. So far, developers have confirmed the two bugs and 21 bad practices (Table XII). More importantly, all of these 23 confirmed issues are "previously-unknown". At the time of writing, developers have fixed 10 issues, and will fix another six issues in next major updates.

TABLE XI
APPLYING DIET ON REAL-WORLD CLOUD SYSTEMS

System (Version)	Cassandra (3.11)	Hadoop† (3.1.2)	HBase (2.1.4)	ZooKeeper (2.4.14)	**Total**
`Throw`	2,823	9,853	5,020	429	**18,125**
Root ex.	1,282	3,090	1,374	159	**5,905**
Calculated	550	1,579	716	84	**2,929**
Reported	100	136	73	5	**314**
Candidate	9	20	2	0	**31**

† Hadoop includes Hadoop common, HDFS, MapReduce, and YARN.

TABLE XII
BUGS AND BAD PRACTICES DETECTED BY DIET

System	**Bug**	**Bad Practice**		
	Confirmed	**Confirmed**	**Pending**	**Rejected**
Cassandra	0	8*	1	0
Hadoop	2 (1 fixed)	13 (9 fixed)	2	3
HBase	0	0	0	2
ZooKeeper	0	0	0	0
Total	**2 (1 fixed)**	**21 (9 fixed)**	**3**	**5**

* Cassandra developers will fix six out of eight confirmed bad practices in the next major update.

Bugs: DIET detected two bugs, which are confirmed by the developers. Both eBugs are caused by using incorrect classes. For example, in one of them, HADOOP-16295, a DataNode throws an `IOException` when it is interrupted during file renaming. This leads to a checking of disk health, which is necessary only when the `IOException` is triggered by a file system error, e.g., when the renaming actually fails.

These bugs highlight the importance of throwing exceptions with accurate classes. When the exception class does not match its triggering condition, the system may misbehave in two ways. First, the unintended handling operations may be executed, such as both bugs found by DIET. Second, the intended handling operations may be skipped. Although DIET has not found any new bugs with this symptom, they do exist in our eBugs dataset, e.g., CASSANDRA-11448.

Bad practices: DIET detected 29 bad practices, where 21 have been confirmed by developers. Although these bad practices have not caused any failure symptoms, developers act proactively to these reported bad practices. For example, Hadoop developers have fixed nine out of thirteen confirmed bad practices [38], and Cassandra developers will fix six out of eight confirmed bad practices in their next major update [39]–[43]. Developers rejected five bad practices, because they believed that these reported candidates work as intended. For example, in HBase, a `RuntimeException` is used to represent a file system error. Developers thought it is a norm in HBase to use `RuntimeException` for a fatal file system error [44].

As a preliminary attempt to detect eBugs in cloud systems, DIET has found many unknown issues in popular and mature cloud systems. We believe that, by integrating other findings in our study, DIET can be further extended to detect more eBugs and bad practices. For instance, by integrating Finding 7 and exception flow analysis [30]–[35], DIET can help detect exception handlers that apply the same handling to exceptions triggered by different condition types, which is an indicator for overly-general reaction eBugs (Finding 7).

VIII. Related Work

In this section, we discuss related work that are not discussed in previous sections.

EBug studies in other systems: Prior studies have examined the root causes of eBugs in general systems and Android applications from source code patterns [5]–[8]. These studies provide valuable insights on developers' common mistakes when handling exceptions in their target systems. However, due to the inherent system differences, their findings may not be applicable to cloud systems. Some studies have analyzed the relation between eBugs and certain language features, e.g., aspect-oriented programming [9] and Android abstractions [6]. Other studies try to understand developers' perception on eBugs [5], [6], and the common exception handling practices [45]–[47]. Complementary to these studies, our eBug study focuses on analyzing the relations between triggering conditions and the root causes in cloud systems, which are critical in eBug exposure and detection.

EBug detection: A few tools have been designed to detect eBugs. Aspirator detects empty or incomplete (e.g., containing "TODO") exception handlers, as well as overly-general handlers that abort the whole system [3]. CAR-Miner detects missing recovery operations by inferring methods that should have executed together when exceptions occur [36]. EPE detects incorrect exception propagation by finding exceptions that are thrown or caught in unintended methods [37]. Unlike these tools, DIET detects inaccurate exceptions by analyzing the inconsistency between the triggering conditions inferred from exception classes and error messages.

Other bug studies: In cloud systems, prior works have focused on other types of bugs, including general bugs [4], concurrency bugs [48], crash recovery bugs [19], timeout related bugs [49], and system failures [3]. They have identified invaluable observations on different types of bugs, helping improve cloud system reliability in many ways. Complementary to these studies, our work examines a different and important threat in cloud systems, i.e., eBugs. We hope the combined efforts can greatly help developers improve the reliability of cloud systems.

Previous works have also studied other types of bugs in general systems [50]–[52]. These studies have inspired lots of research that combat bugs in various ways [53]–[55]. We believe our study can help better understand eBugs in cloud systems and reveal opportunities to alleviate them.

IX. Conclusion

In this paper, we present a comprehensive analysis of 210 eBugs in six popular cloud systems, from the perspective of triggering conditions. Most of these eBugs affect the availability or integrity of the cloud systems. Through this study, we have made many interesting findings, which reveal important opportunities for combating eBugs in cloud systems. Based on our findings, we develop DIET to detect inaccurate exceptions in cloud systems. DIET has detected 31 eBugs and bad practices, and developers have confirmed 23 of them.

Acknowledgment

We would like to thank the anonymous reviewers for their thorough and insightful comments. We are grateful to Dejun Teng for helping us calculate the proportion of exception-related code in popular cloud systems. In this work, Haicheng Chen and Feng Qin were partially supported by National Science Foundation grants #CNS-1513120 and #CCF-0953759 (CAREER Award). Wensheng Dou and Yanyan Jiang were partially supported by National Key R&D Program of China (2017YFB1001800), National Natural Science Foundation of China (#61732019, #61932021, #61802165), Youth Innovation Promotion Association at Chinese Academy of Sciences, Alibaba Innovative Research Program, and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Both Haicheng Chen and Wensheng Dou are the corresponding authors of this paper.

REFERENCES

[1] Apache Hadoop. [Online]. Available: https://hadoop.apache.org

[2] Advantages of exceptions. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html

[3] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014, pp. 249–265.

[4] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin *et al.*, "What bugs live in the cloud? A study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–14.

[5] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in Java programs," *Journal of Systems and Software*, vol. 106, pp. 82–101, 2015.

[6] J. Oliveira, D. Borges, T. Silva, N. Cacho, and F. Castor, "Do Android developers neglect error handling? A maintenance-centric study on the relationship between Android abstractions and uncaught exceptions," *Journal of Systems and Software*, vol. 136, pp. 1–18, 2018.

[7] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen, "Unveiling exception handling bug hazards in Android based on GitHub and Google code issues," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015, pp. 134–145.

[8] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in Android Apps," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 408–419.

[9] R. Coelho, A. Rashid, A. von Staa, J. Noble, U. Kulesza, and C. Lucena, "A catalogue of bug patterns for exception handling in aspect-oriented programs," in *Proceedings of the 15th Conference on Pattern Languages of Programs*, 2008, p. 23.

[10] Jira Software. [Online]. Available: https://www.atlassian.com/software/jira

[11] Apache Cassandra. [Online]. Available: http://cassandra.apache.org

[12] Apache HBase. [Online]. Available: http://hbase.apache.org

[13] HDFS architecture. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[14] MapReduce tutorial. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[15] Apache Hadoop YARN. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[16] Apache ZooKeeper. [Online]. Available: http://zookeeper.apache.org

[17] EBugs in cloud systems. [Online]. Available: https://hanseychen.github.io/eBugs/

[18] O. R. Gatla, M. Hameed, M. Zheng, V. Dubeyko, A. Manzanares, F. Blagojević, C. Guyot, and R. Mateescu, "Towards robust file system checkers," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, 2018, pp. 105–122.

[19] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu, "An empirical study on crash recovery bugs in large-scale distributed systems," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 539–550.

[20] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[21] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing," in *Proceedings of the 16th Network and Distributed System Security Symposium*, 2008, pp. 151–166.

[22] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh, "Torturing databases for fun and profit," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014, pp. 449–464.

[23] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A framework for cloud recovery testing," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011, pp. 1–18.

[24] R. Alagappan, A. Ganesan, Y. Patel, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Correlated crash vulnerabilities," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016, pp. 151–167.

[25] H. Liu, X. Wang, G. Li, S. Lu, F. Ye, and C. Tian, "FCatch: Automatically detecting time-of-fault bugs in cloud systems," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 419–431.

[26] Jepsen. [Online]. Available: https://jepsen.io/

[27] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, 2018, pp. 51–68.

[28] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014, pp. 399–414.

[29] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, 2017, pp. 149–166.

[30] G. B. de Pádua and W. Shang, "Revisiting exception handling practices with exception flow analysis," in *Proceedings of 17th International Working Conference on Source Code Analysis and Manipulation*, 2017, pp. 11–20.

[31] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio, "Understanding the exception handling strategies of Java libraries: An empirical study," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 212–222.

[32] S. Liang, W. Sun, M. Might, A. Keep, and D. Van Horn, "Pruning, pushdown exception-flow analysis," in *Proceedings of 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 265–274.

[33] H. Melo, R. Coelho, U. Kulesza, and D. Sena, "In-depth characterization of exception flows in software product lines: An empirical study," *Journal of Software Engineering Research and Development*, vol. 1, no. 1, p. 3, 2013.

[34] P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivančić, and A. Gupta, "Interprocedural exception analysis for C++," in *Proceedings of the 25th European Conference on Object-Oriented Programming*, 2011, pp. 583–608.

[35] M. Bravenboer and Y. Smaragdakis, "Exception analysis and points-to analysis: Better together," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*, 2009, pp. 1–12.

[36] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 496–506.

[37] T. Montenegro, H. Melo, R. Coelho, and E. Barbosa, "Improving developers awareness of the exception handling policy," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 413–422.

[38] HDFS-14486. [Online]. Available: https://issues.apache.org/jira/browse/HDFS-14486

[39] CASSANDRA-15111. [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-15111

[40] CASSANDRA-15112. [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-15112

[41] CASSANDRA-15114. [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-15114

[42] CASSANDRA-15116. [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-15116

[43] CASSANDRA-15117. [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-15117

[44] HBASE-22369. [Online]. Available: https://issues.apache.org/jira/browse/HBASE-22369

[45] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in Java projects: An empirical study," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 500–503.

[46] M. Monperrus, M. G. de Montauzan, B. Cornu, R. Marvie, and R. Rouvoy, "Challenging analytical knowledge on exception-handling: An empirical study of 32 Java software packages," Tech. Rep. hal-01093908, 2014.

[47] M. B. Kery, C. Le Goues, and B. A. Myers, "Examining programmer practices for locally handling exceptions," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 484–487.

[48] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 517–530.

[49] T. Dai, J. He, X. Gu, and S. Lu, "Understanding real-world timeout problems in cloud server systems," in *Proceeding of the IEEE International Conference on Cloud Engineering*, 2018, pp. 1–11.

[50] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 329–339.

[51] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001, pp. 73–88.

[52] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 26–36.

[53] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 25–36.

[54] W. Zhang, C. Sun, and S. Lu, "ConMem: Detecting severe concurrency bugs through an effect-oriented approach," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 179–192.

[55] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: A technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 344–360.