DScope: Detecting Real-World Data Corruption Hang Bugs in Cloud Server Systems

Ting Dai
North Carolina State University
Raleigh, NC, USA
tdai@ncsu.edu

Jingzhu He
North Carolina State University
Raleigh, NC, USA
jhe16@ncsu.edu

Xiaohui Gu
North Carolina State University
Raleigh, NC, USA
xgu@ncsu.edu

Shan Lu
University of Chicago
Chicago, Illinois, USA
shanlu@cs.uchicago.edu

Peipei Wang
North Carolina State University
Raleigh, NC, USA
pwang7@ncsu.edu

ABSTRACT

Cloud server systems such as Hadoop and Cassandra have enabled many real-world data-intensive applications running inside computing clouds. However, those systems present many data-corruption and performance problems which are notoriously difficult to debug due to the lack of diagnosis information. In this paper, we present DScope, a tool that statically detects data-corruption related software hang bugs in cloud server systems. DScope statically analyzes I/O operations and loops in a software package, and identifies loops whose exit conditions can be affected by I/O operations through returned data, returned error code, or I/O exception handling. After identifying those loops which are prone to hang problems under data corruption, DScope conducts loop bound and loop stride analysis to prune out false positives. We have implemented DScope and evaluated it using 9 common cloud server systems. Our results show that DScope can detect 42 real software hang bugs including 29 newly discovered software hang bugs. In contrast, existing bug detection tools miss detecting most of those bugs.

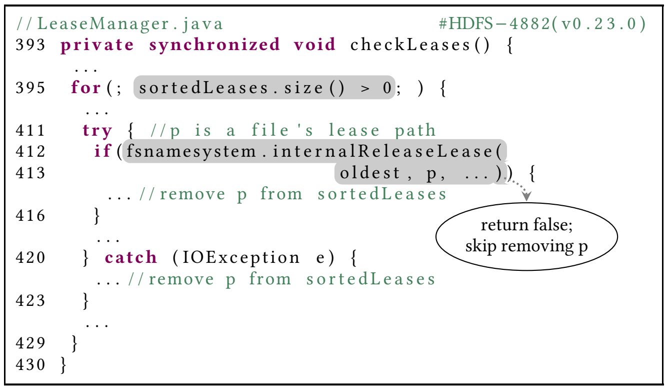
CCS CONCEPTS

• **Computer systems organization → Cloud computing**; **Reliability**; • **Software and its engineering → Automated static analysis**; **Software performance**;

KEYWORDS

static analysis, data corruption, performance bug detection

ACM Reference Format:
Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. 2018. DScope: Detecting Real-World Data Corruption Hang Bugs in Cloud Server Systems. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC'18),* 13 pages.



```
//LeaseManager.java                          #HDFS-4882(v0.23.0)
private synchronized void checkLeases() {
  ...
  for(; sortedLeases.size() > 0; ) {
    ...
    try { //p is a file's lease path
      if(fsnamesystem.internalReleaseLease(
                                   oldest, p, ...)) {
        ...//remove p from sortedLeases
      }
      ...
    } catch (IOException e) {
      ...//remove p from sortedLeases
    }
    ...
  }
}
```

Figure 1: A real-world data corruption hang bug from HDFS. A corrupted file `f` associated with the lease path `p` makes the `internalReleaseLease` function fail for recovering the lease for `f`. When this failure happens, `p` is not removed from `sortedLeases` (skip updating loop index), LeaseManager keeps recovering lease for the file f endlessly.

https://doi.org/10.1145/3267809.3267844

1 INTRODUCTION

Cloud server systems such as Hadoop and Cassandra [3, 4] have enabled many real-world data-intensive applications ranging from security attack detection to business intelligence. However, due to their inherent complexity, those cloud server systems present many performance challenges. Particularly, previous studies [23, 25] have shown that many tricky performance bugs in cloud server systems are caused by unexpected data corruptions which are more likely to be overlooked by the developer. For example, in May 2017, a data corruption bug triggered in a data center failover operation brought down the British Airway service for hours [2].

Performance bugs[1] are notoriously difficult to debug because they typically produce little useful debugging information. The problem exacerbates in cloud server systems since the developer typically does not have the access to the original input data that triggered the performance bug or the large scale infrastructure

[1]We use performance bugs to broadly refer to all non-functional bugs, which could cause slowdown or system unavailability.

to replay the failed production run. Although previous work has extensively studied data corruptions (e.g., [11, 15, 23, 28, 44]) and performance bugs (e.g., [17, 26, 31, 40]), little research has been done to study the intersection between the two, that is, the performance problems caused by data corruptions. Particularly, our work focuses on detecting software hang bugs that are triggered by data corruptions in cloud server systems. Software hang bugs make the system become unavailable to either part of or all of the users, which is one of the most severe performance problems production systems try to avoid [17–19, 29].

1.1 A Motivating Example

To better understand how real-world data corruption hang bugs happen, we use a known HDFS-4882 [2] bug as one example shown by Figure 1. This hang bug happens when the improper handling of a corrupted file `f` causes the loop to skip updating its loop index. In HDFS, when a client's leases get expired, the lease recovery is triggered by the LeaseManager on the NameNode. The LeaseManager sends a lease recovery request for each lease in the `sortedLeases` set to the `FSNamesystem` via a RPC call (line #412-413). If a lease is successfully recovered (e.g., released or renewed) (line #412-416), or an `IOException` happens during the lease recovery (line #420-423), the lease path `p` is removed from the `sortedLeases`. The LeaseManager keeps recovering and removing the leases until the `sortedLeases` set is empty (line #395). However, the `FSNamesystem` only considers the case where the last block is corrupted if data corruption happens in a file. Thus, when the second-to-last block of a file `f` (i.e., an `INode`) is corrupted but the processing state of the last block of `f` is complete, the `FSNamesystem` improperly handles this case and returns `false` by mistake (line #412). This bug occurs when the HDFS client finished writing the second-to-last block, starts to write the last block and part of the DataNodes experience a shut-down failure. To resume the process, the NameNode marks the second-to-last block as committed, unblocks the HDFS client from writing the last block, and marks the last block as complete [1]. As a result, the lease path `p` is not removed from the `sortedLeases`, and the LeaseManager keeps invoking the lease recovery for the same lease endlessly.

1.2 Our Contribution

This paper presents DScope, an automated corruption-hang bug detection tool for server systems commonly used in computing clouds. DScope is a static analysis tool — it can detect data-corruption related hang bugs without running the target system and it requires no system-specific knowledge. To achieve both high coverage and low false positives, DScope first uses static control flow and data flow analysis to identify loops whose exit conditions may be affected by external data (i.e., I/O operations), and then conducts loop bound and loop stride analysis to filter out loops which are guaranteed not to have hang problems. To support such analysis, DScope models Java data-related APIs which are commonly used in cloud server systems.

This paper makes the following contributions:

- We present a hang-bug detection scheme that identifies potential infinite loops caused by data corruptions in cloud server systems.
- We describe a false-positive pruning technique that identifies always-exit loops through loop stride and bound analysis. Different from generic loop analysis, our analysis focuses on a wide variety of Java I/O APIs widely used in cloud systems, and helps greatly improve the accuracy of our data-corruption hang-bug detection.
- We categorize real-world data-corruption hang bugs into four common types based on DScope detection results. This categorization will help future work on avoiding, detecting, and preventing data-corruption hang bugs.

We have implemented DScope and evaluated it using 9 commonly used cloud server systems (e.g., Cassandra, HDFS, Mapreduce, Hive, etc). DScope reports 42 true data corruption hang bugs, with <u>29</u> of them are newly discovered bugs. We also applied two state of the art static bug detectors, Findbugs [7] and Infer [6], to the same set of systems. They detect very few corruption hang bugs (2 for Findbugs and 1 for Infer), indicating the need for a dedicated corruption hang bug detector like DScope.

The rest of the paper is organized as follows. §2 describes the design of the DScope system. §3 presents the types of the data corruption hang bugs. §4 shows the experimental evaluation. §5 discusses the future work. §6 compares our work with related work. Finally, the paper concludes in §7.

2 SYSTEM DESIGN

This section first provides an overview of DScope (§2.1). It then presents the detailed designs of how to discover corruption-hang bug candidates (§2.2) and how to prune false positives (§2.3).

2.1 Approach Overview

DScope focuses on detecting software hang bugs caused by potential data corruptions in cloud server systems. Our bug detection scheme consists of two major steps: 1) discovering all candidate data corruption hang bugs that aims at maximizing detection coverage; and 2) filtering out false positive detections by identifying code patterns that assure the program will not hang under any circumstance.

Since many software hang problems are caused by infinite loop bugs [19, 29], our work focuses on detecting possible infinite loops caused by data corruptions in cloud server systems written in Java. To detect those loop bugs, DScope leverages the Soot compiler framework [8] to compile application bytecode into intermediate representation (IR) code (i.e, Soot Jimple) and perform static analysis over the IR code in three steps: 1) loop path extraction, 2) I/O depdent loop identification, and 3) loop stride and bound analysis.

Specifically, DScope first extracts different execution paths which start from the loop header and end at the loop header by traversing the control flow graph (CFG) of all loops. Next, DScope derives the exit conditions of each loop path and checks whether those exit conditions depend on any I/O operations. The rationale is that if the loop exit condition depends on an I/O operation, a data corruption (e.g., hardware failure [11, 12, 27, 30, 36, 39], software

[2] We use "system name-bug #" to denote different bugs.

```
549 for (int j = 0; j < length; j++) {
550   String rack = oneblock.racks[j];
      ...
559 }
560
```

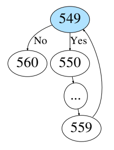


Figure 2: The example of a simple loop with the source code block and the corresponding CFG in the `CombineFileInputFormat` **class in Hadoop v0.23.0.**

fault [44]) can cause the loop exit condition to be never met and thus an infinite loop software hang bug.

After discovering candidate data corruption hang bugs, DScope performs false positive pattern filtering to improve the bug detection precision. The false positive filtering is based on the loop index, loop stride (i.e., the delta value applied to the loop index in each iteration) and loop bound analysis on every loop path. For example, if the loop stride is always positive when the loop bound is an upper bound (or if the loop stride is always negative when the loop bound is a lower bound), and if the loop bound is unchanged during each loop iteration and the loop exit conditions involve bound checking, we say that the detected hang bug is a false positive because the loop will always exit without causing a software hang.

2.2 Identify Bug Candidates

DScope discovers candidate corruption-hang bugs by 1) traversing the CFGs of all loops in application functions to derive their loop paths and 2) checking whether the exit conditions of those loop paths are I/O dependent.

Loop path extraction. For a simple loop, the execution path within one loop iteration, called a *loop path*, consists of all the statements that start from the loop header and end at the loop header. We can extract the loop path easily by traversing the CFG of the loop. For example, Figure 2 shows the source code and its CFG of a simple loop[3]. DScope generates a loop path {549, 550, ..., 559, 549}.

For a nested loop, the loop path consists of concatenations of the execution paths of both inner loops and outer loops. DScope extracts all loop paths using three steps. First, for the execution path, denoted as P_{outer} whose tail is the outer loop header, we add the path into a path set called S_{path}. Second, for the execution path P_{outer} whose tail is a loop body statement, we infer this statement must be the header of an inner loop and extracts the inner loop execution path denoted as P_{inner} from P_{outer}. Third, for each loop path in S_{path}, DScope first clones it and replaces any statement s_i with P_{inner} if s_i is an inner loop header and the current loop path does not contain P_{inner}. This new concatenated loop path is then added to the path set S_{path}. DScope repeats the third step until there is no more new loop path generated. Figure 3 shows an example of nested loops. First, DScope extracts one loop path {544, ..., 549, 560, ..., 571, 544}. Second, DScope extracts {549, 550, .., 559, 549} as an inner loop path. Third, DScope replaces 549 with {549, 550, ..., 559, 549} on {544, ..., 549, 560, .., 571, 544}, to

[3]DScope analyzes IR code directly to extract the execution path of different loops. For easy understanding, we illustrate the execution paths using source code in the rest of the paper.

```
544 for(OneBlockInfo oneblock : blks){
      ...
549   for (int j = 0; j < length; j++){
550     String rack = oneblock.racks[j];
        ...
559   }
560
      ...
571 }
```

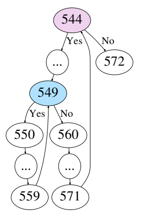


Figure 3: The example of nested loops with the source code block and the corresponding CFG in the `CombineFileInputFormat` **class in Hadoop v0.23.0.**

```
120 while (!dataFile.isEOF()){
      ...
128   DecoratedKey key = null;
129   try{
130     key =...; //throws Exception
        ...
139   } catch (Throwable th){
140     ...
141   }
      ...
185   try{
186     if (key == null)
187       throw new IOError(...);
188     if (dataSize > length)
        ...
206     outputHandler.warn(...);
207   } catch (Throwable th) {
        ...
255   }
256 }
257
```

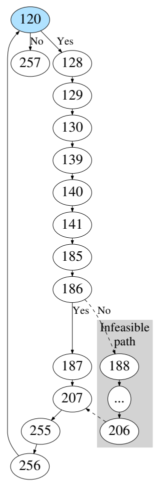


Figure 4: The example of a loop containing exception handling constructs with the source code block and the corresponding CFG in the `Scrubber` **class in Cassandra v2.0.8.**

create a new loop path {544, ..., 549, 550, ..., 559, 549, 560, ..., 571, 544}.

The third group of complicated loops involve exceptions. For those loops, some sub-paths become infeasible due to the exception handling, which should not be considered in our loop exit condition checking. For example, Figure 4 shows a `while` loop containing exception handling. The assignment statement at line #130 can throw an exception when the operation on the right hand side processes a `null` argument. As a result, the variable `key` is not updated and remains to be the default value which is `null`. So when the exception is triggered, the `if` statement (line #186) always returns `true`. Thus, all the statements in the else branch (line #188-206) are unreachable and any path consists of those statements are infeasible paths. In this example, DScope only generates the loop path as {120, 128, 129, 130, 139, 140, 141, 185, 186, 187, 207, 255, 256, 120}.

It can be computationally expensive to traverse the CFG of the loops containing multiple exceptions because every statement in the `try` block has two branches (i.e., triggering or not triggering the exception) resulting in a large CFG. DScope addresses the problem by grouping all the statements based on the data they process. Specifically, DScope identifies all the statements which involve function invocations in the `try` blocks and groups them based on the *arguments* of those function invocations. Since DScope aims

```
//Soot IR
198 $i1 = r0.<InputStream: read()>(r2) //$i1 is an I/O
                                        //related variable
199 if $i1 == -1 goto line #203 //$i1 == -1 is the
    ...                         //exit condition
202 goto line #198
```

Figure 5: The example of the loop's exit condition directly depends on I/O operations. It is in the `IOUtils` class of Compress v1.0.

```
//Soot IR
 3  if l8 >= l0 goto line #12 //l8 >= l0 is the
    ...                       //exit condition
 5  $l2 = l0 - l8
 6  $l4 = $r2.<InputStream: skip>($l2)//$l4 is an I/O
 7  $b5 = $l4 cmp 0L                  //related variable
 8  if $b5 == 0 goto line #12 //$b5 == 0 is the
 9  $l7 = $l8 + $l4           //exit condition
10  i8 = $l7
11  goto line #3
```

Figure 6: The example of the loop's exit condition indirectly depends on I/O operations. It is in the `NonSyncDataInputBuffer` class of Hive v2.3.2.

at detecting data corruption hang bugs, we can assume all the statements in the same group throw exceptions when their *arguments* get corrupted. Suppose there are m statements in the `try` blocks. DScope divides all m statements into n groups and runs the loop path discovery algorithm 2^n times. Thus, DScope can reduce the loop path search space from 2^m to 2^n ($n \ll m$), which reduces DScope's analysis time and resource requirements (e.g., avoiding analysis failures caused by `OutofMemoryException`).

I/O dependent loop identification. To discover candidate data corruption hang bugs, DScope identifies those loops whose exit conditions depend on I/O operations, which are called *I/O dependent loops*. After extracting a loop path, DScope identifies all the loop exit instructions and derives the loop path's exit conditions by performing a union over the exit conditions of all the branch statements. We consider a loop path is I/O dependent if *any* of its exit conditions depend on I/O operations. The rationale is that a data corruption can cause the corresponding I/O operations to return unexpected values or throw exceptions, making the loop never exit and thus software hang. DScope considers the operations performed on *I/O classes* via virtual invocations or on the *I/O variables* via instance invocations as *I/O operations*. The I/O classes include all classes and interfaces in `java.io` and `java.nio` packages and their subclasses and implementation classes. The instances of the I/O classes are called *I/O variables*. Additionally, DScope allows users to easily add application I/O classes in the configuration files to maximize detection coverage by identifying more application I/O dependent loops.

DScope checks whether the loop exit conditions *directly* depend on I/O operations by identifying the appearance of I/O classes in the exit checking statements. Figure 5 shows an example where the loop exit condition directly depends on the I/O operations. In this example, the variable `$i1` in the exit condition checking statement (line #199) is directly derived from a Java I/O class called `InputStream`.

DScope checks whether the loop exit conditions *indirectly* depend on I/O operations by performing data dependency analysis

```
//Soot IR
10  $r13 = new java.util.HashMap
```

```
//Java source code
269 HashMap<OneBlockInfo, String[]> blockToNodes =
270   new HashMap<OneBlockInfo, String[]>();
```

```
//Java bytecode
Constant pool:
#219 = blockToNodes
#232 = Ljava/util/HashMap<
         Lorg/apache/hadoop/mapreduce/lib/
         input/CombineFileInputFormat$OneBlockInfo;
         [Ljava/lang/String;>;
...
LocalVariableTypeTable:
name index     signature index
#219           #232
```

Figure 7: The `java.util.HashMap<K,V>` example in the `CombineFileInputFormat` class in Hadoop v0.23.0.

on all the statements of the corresponding application function. Specifically, DScope first identifies all the I/O related variables which are assigned with the return values of the I/O operations. Second, for each assignment statement of the application function that involves any I/O related variables on its right-hand-side, DScope iteratively labels the variable on the left-hand-side of the assignment statement as I/O related variables as well. After identifying all the I/O related variables, DScope checks whether the loop exit conditions are I/O dependent by identifying the appearance of I/O related variables in the exit checking statements. Figure 6 shows an example of indirectly I/O dependent loop exit condition. In this example, the loop exit checking involves `$l8` (line #3) and `b5` (line #8) whose value is derived from `l4` which is derived from a Java I/O operation `InputStream.skip()`.

To further check whether the loop exit conditions depend on I/O operations conducted on *complex I/O related variables* (i.e., variables with composite types), DScope performs an integrated analysis by linking variable information from IR code, Java source code, and Java bytecode [4]. DScope considers a variable with composite type as I/O related if any of the variable's elements is I/O related. Note that, by checking only the IR code, DScope might miss identifying some complex variables as I/O related. For example, in Figure 7, by checking only the IR code, DScope cannot identify the variable `$r13` as an I/O related variable, thus all the operations conducted on `$r13` will not be considered as I/O operations. This is because `$r13` is of type `HashMap` and `HashMap` is not an I/O class. To identify complex I/O related variables, DScope needs to retrieve the full type information (i.e., class path) in the Java bytecode for a target variable in the IR code. However, there is no direct mapping from IR code to Java bytecode. So, DScope has to leverage the source code to establish the mapping from. Specifically, DScope first retrieves the source code line number from Soot via `getLineNumber()` API for each variable val_{IR} in the IR code. DScope then analyzes the corresponding source code and extracts val_{IR}'s name in the source code, denoted as val_{src}. In Figure 7, DScope extracts that the variable `$r13` is defined at line #269 in the source code with name `blockToNodes`. Next, DScope

[4]DScope mainly works on Soot IR code, except the integrated analysis in the I/O dependent loop identification module.

Table 1: The 60 commonly used Java classes and interfaces which contain APIs related to the loop index, stride and bound.

Prefix	Class	# of classes or interfaces
java.io	DataInput family	2
	File	1
	InputStream family	12
	Reader family	10
java.nio	Buffer family	8
	channels.Channel family	20
java.util	Iterator, Enumeration	2
	List, Queue, Set, Stack	4
	StringTokenizer	1

leverages Coffi [43], a Java bytecode parser, to extract the full type information for the target variable val_{src}. Specifically, the constant pool provides index lookup for each variable and LocalVariableTypeTable provides the mapping from the variable's index to the index of its signature which contains the full type information. In Figure 7, the constant pool indicates that the index of the variable `blockToNodes` is #219 and the LocalVariableTypeTable lookup tells its corresponding signature index is #232. DScope checks the constant pool using the index number #232 to derive the full type information of `blockToNodes`. Since `blockToNodes` consists of `CombineFileInputFormat$OneBlockInfo` class which is I/O related (i.e, an application I/O class), DScope infers `blockToNodes` is also I/O related. Thus the operations conducted on `blockToNodes` are I/O operations and the loops whose exit conditions depend on those I/O operations are I/O dependent loops.

2.3 Prune False Positives

Since our goal of candidate bug detection is to maximize coverage, false positives can be inevitably included in the candidate list. To improve DScope's bug detection precision, we further develop false positive pattern filtering schemes by identifying those loops which will always exit without causing any software hang. Our false positive filtering is achieved by analyzing the loop stride and loop bounds. DScope prunes false positive candidates by checking whether 1) the loop stride is *always* positive when the loop has an upper bound or the loop stride is always negative when the loop has a lower bound; 2) the loop bound value is unchanged in *every* loop iteration; and 3) the loop exit conditions contain bound checking. Intuitively, any loops satisfying all those conditions will always exit without causing software hang, which should be pruned from DScope's detection list.

DScope's loop stride and bound analysis schemes consider two cases: a) the loop index, stride, and bounds are denoted by numeric primitives (e.g., integer); and b) the loop index, stride, and bounds are denoted by APIs in 60 commonly used Java classes and interfaces, shown in Table 1. Note that those Java classes and interfaces are not necessarily the *I/O classes* but appear frequently in the *I/O dependent loops*. Moreover, they do not include all the Java classes and interfaces which contain the loop related APIs. We plan to further extend our analysis to cover other Java classes

```
//Soot IR
127 $b6 = i1 cmp i0              //i1 is the loop index
128 if $b6 >= 0 goto line #139//i0 is the upper bound
129 ...
130 i7  = i1  + 1
131 i12 = i7  + 1
132 i17 = i12 + 1
133 i22 = i17 + 1
134 i27 = i22 + 1
135 i32 = i27 + 1
136 i37 = i32 + 1
137 i1  = i37 + 1 //all the 1's are the strides
138 goto line #127
```

Figure 8: The example of multiple strides. It is in the `OffHeapBitSet` class of Cassandra v2.0.8.

```
//Soot IR
530 $i5=r1.<ByteBuffer:limit>   //$i5 is the upper bound
531 if i1 >= $i5 goto line #536//i1 is the loop index
532 $i9 = <STEP_LENGTH>;        //$i9 is the stride
533 i1 = i1 + $i9
    ...
535 goto line #531
```

Figure 9: The example of the stride is assigned outside of the function `total` where the loop resides. It is in the `CounterContext` class of Cassandra v2.0.8. The stride `STEP_LENGTH` is a static variable, which is assigned with 34 in the class initializer.

in our future work, which can further improve our false positive filtering efficacy.

When the loop index, stride, and bounds are denoted by numeric primitives, DScope first extracts the loop index variable from the loop exit conditions. The *loop index* is a variable that appears in the loop exit conditions and is updated by another variable in an assignment statement via arithmetic operations (e.g., addition and subtraction). After identifying the loop index, the variable to which it compares in the exit conditions is the *loop bound* and the variable which is added to or subtracted from the loop index is the *loop stride*. DScope further checks whether the numeric stride is positive when the loop has an upper bound or negative when the loop has a lower bound. For example, in the expression "*index = index op stride*", if the "*op*" is an addition operation, the loop stride is positive. In the expression "*index symbol bound*", if the "*symbol*" is ≤ or <, the loop has an upper bound. Finally, DScope examines all the loop paths and checks whether the bound is unchanged within every loop path. Based on all the extracted information, DScope can make decision whether a discovered loop bug is a false positive.

When the loop index, stride, and bounds are denoted by *multiple* numeric primitives or the numeric primitives *outside* the current application function where the loop resides, DScope performs intra-procedure data flow analysis on all the statements of the corresponding application class to achieve accurate false positive filtering. For example, Figure 8 shows a loop with multiple strides in the `OffHeapBitSet` class in Cassandra. The variable `i1` is the loop index while the variable `i0` is the upper bound. The loop index `i1` is updated multiple times from line #130 to line #137. DScope recursively applies all the assignments from line #130 to line #137 to get `i1 = i1 + 8`, and extracts the aggregated stride (i.e, 8). Figure 9 shows an example where the stride variable `$i9` is updated by `STEP_LENGTH` in the `CounterContext` class in Cassandra.

Table 2: The APIs that are related to loop stride and bound update in 60 commonly used Java classes and interfaces. "*": a set of APIs perform similar operations; and "-": does not contain the corresponding type APIs.

Class	The type and name of APIs				
	Forward index	Reverse index	Reset index	Check bounds	Update bounds
File	create*	get*	new	is*, can* exists, get*	-
InputStream & Reader family	read*	reset	new	read*	new
DataInput family	read*	-	new	read*	new
Buffer family	position get* put*	position reset clear	duplicate allocate new	has* remaining	flip limit clear new
Channel family	read write	-	-	read write	-
List & Set	-	remove	new	is*	add clear new
Queue	-	poll remove	-	poll remove	add offer new
Stack	-	pop	-	empty pop	push new
Iterator & Enumeration	next	-	iterator elements new	has*	new
StringTokenizer	next	-	new	has*	new

The loop resides in the function `total()` while `STEP_LENGTH` is a static variable defined in the class initializer. DScope performs data flow analysis to extract the value of `STEP_LENGTH` from the class initializer and then checks whether the stride is positive because the loop index has an upper bound.

We now describe how to perform false positive filtering when the loop index, stride and bounds are denoted by the APIs in 60 commonly used Java classes and interfaces, listed in Table 1.

We classify those APIs into five categories, shown by Table 2: 1) the APIs which move the index forward when the Java class/interface has an upper bound; 2) the APIs which move the index backward when the Java class/interface has a lower bound; 3) the APIs which reset the index; 4) the APIs which check bounds; and 5) the APIs which update bounds. The APIs' names ending with "*" denote those APIs which perform similar operations and share the same prefix in their names. For example, in the `InputStream` family, there are `read()` and `readLine()` functions which both perform read operations on the corresponding `InputStream`.

DScope first extracts all the invoked APIs for each of the 60 Java classes and interfaces in the loop paths, and then prunes false positive candidates by checking whether 1) the "forward index" APIs or the "reverse index" APIs are invoked; 2) the "reset index" APIs and "update bounds" APIs are not invoked; and 3) the "check bounds" APIs are invoked in the exit conditions. Note that, DScope cannot prune the case where both the "forward index" APIs and the "reverse index" APIs are invoked in the loop paths because the loop stride cannot be guaranteed to be *always* positive or negative.

```
//DFSOutputStream.java                         #HDFS-5438(v0.23.0)
1665 private void completeFile(ExtendedBlock last) ... {
       ...
1667   boolean fileComplete = false;
1668   while (!fileComplete) {
1669    fileComplete = dfsClient.namenode.complete(src,
                            dfsClient.clientName, last);
        ...
1689   }}
```

Figure 10: The code snippet of the HDFS-5438 Bug. When the `ExtendedBlock last` is corrupted, the `fileComplete` variable is never set to be true, causing an infinite loop in DFSOutputStream.

The APIs in the five categories do not necessarily change the loop index or bounds. Those APIs' arguments should also be considered when DScope performs the false positive filtering. For example, `ByteBuffer` contains overloading methods which have an attribute called *relative* or *absolute*. The `ByteBuffer.get()` is a relative method while the `ByteBuffer.get(int)` is an absolute method. Invoking a relative method can change the loop index (i.e., `ByteBuffer.position`) while invoking absolute methods cannot. Another example is the `InputStream` class. Invoking the `InputStream.read(byte[], int, int)` with a zero size byte array or with 0 as the third parameter cannot change the loop index.

To achieve accurate pruning, DScope first annotates all the commonly used Java APIs with the attribute *change-positive*, *change-negative* or *change-possible*. The positive APIs can change the loop index (or bounds). The negative APIs cannot change either one. The possible APIs can possibly change the loop index (or bounds). For positive APIs, DScope's pruning steps are the same. For negative APIs in the type of "forward index", "reverse index", "reset index" or "update bounds", DScope ignores them when performing the pruning. For possible APIs, DScope performs intra-procedural data flow analysis on their parameters to decide whether these APIs change loop index/bounds or not.

DScope's false positive filtering only considers the commonly used Java APIs. If the loop index, stride or bounds are *only* related to specific application functions, which means the loop paths do not invoke any Java APIs in Table 2, DScope skips analyzing the loop and simply considers it as a false positive — this design decision may introduce false negatives, but greatly help the efficiency and accuracy of DScope.

One false negative example is the HDFS-5438 bug, shown by Figure 10. This hang bug is caused by a corrupted block, i.e, `last`. `DFSOutputStream` keeps polling `NameNode` to check the completeness of the committing block operation (line #1669). When the `last` block is corrupted, `NameNode` fails to commit it to the disk but returns `false` instead. This results in an infinite loop (line #1668-1689) causing a software hang in `DFSOutputStream`. DScope prunes this case because the loop paths do not invoke any Java APIs in Table 2. In fact, the loop path only invokes a specific application function, i.e., `complete()`. DScope should be able to detect this bug after adding inter-procedural analysis, which is however beyond the scope of this work.

```
//IOUtils.java                                   #Hadoop-8614(v0.23.0)
183 public static void skipFully(InputStream in, long len
     ) throws IOException {
184  while (len > 0) {
185   long ret = in.skip(len); /* in is corrupted */
186   if (ret < 0) {            /* ret = 0 */
187     throw new IOException(...);
188   }
189   len -= ret;
    }}
```

Figure 11: The example when error code returned by I/O operations directly impacts the loop stride. Data corruption causes the I/O function, `InputStream.skip` **returns 0, and 0 is used as the stride.**

3 DATA CORRUPTION HANG BUG TYPES

This section summarizes common types of corruption-hang bugs based on the detection results of DScope (the details of all the bugs detected by DScope will be presented in §4). Although DScope design was **not** affected by these types, this categorization can help future work on avoiding, detecting, and fixing corruption-hang bugs, and help developers better understand the impact of data corruption and corruption-hang bugs.

Our categorization is along two dimensions:

- What is the cause — is it specific error code returned by data operations (Type 1), or specific corrupted data content (Type 2), or specific exception thrown by data operations (Type 3, Type 4)?
- How did the cause lead to an infinite loop — is it through a direct data assignment (Type 1) or control-flow change (Type 3), or indirect data and control flow (Type 2, Type 4)?

Type 1: Error codes returned by I/O operations directly affect loop strides. For this type, the loop stride is directly assigned with a return value of an I/O operation. An infinite loop occurs when an unexpected error code is returned due to underlying data corruption. For example, as shown by Figure 11, when the log file (`InputStream in` at line #183) is corrupted due to bad encoding (Yarn-2724) or corruption propagation (Yarn-7179), the `InputStream in` becomes null. The `skip()` function returns 0 instead of the EOF indicator -1 (line #185). The return value `ret` is then used as the stride at line #189, which makes `len` never get updated but always stay larger than the lower bound (`len > 0`). As a result, the `skipFully()` function causes the system to hang by spinning in the loop forever. Variations of this hang bug type include the cases where the stride is always negative when the loop exit condition contains an upper bound or the stride is always positive when the loop exit condition contains a lower bound.

Type 2: Corrupted data content indirectly affects loop strides. This type of bugs occur when a specific piece of data is corrupted to certain unexpected values. Those values will then affect loop strides through data and/or control flow propagation and lead to infinite loops. For example, as shown by Figure 12, when a configuration file (`conf` at line #190) is corrupted, the variable `BUFFER_SIZE` read from `conf` becomes 0. Calling `read()` function on a zero-size byte array at line #87 causes the loop stride to be zero and zero is then returned, which makes the loop's exit condition (`size < 0`) never be satisfied.

Figure 13 shows an example when the loop stride is indirectly impacted by the corrupted data content which involves *multiple*

```
//BenchmarkThroughput.java                  #HDFS-13514(v2.5.0)
172 public int run(...) throws IOException {
190  Configuration conf = getConf();/*conf is corrupted*/
     ...
194  BUFFER_SIZE = conf.getInt(...);/*BUFFER_SIZE = 0*/
     ...
229 }
```

```
78  private void readLocalFile(Path path, ...) throws
     IOException {
     ...
83   InputStream in = new FileInputStream(...);
84   byte[] data = new byte[BUFFER_SIZE];
85   long size = 0;
86   while (size >= 0) {  /* size = 0 */
87    size = in.read(data);
    }}
```

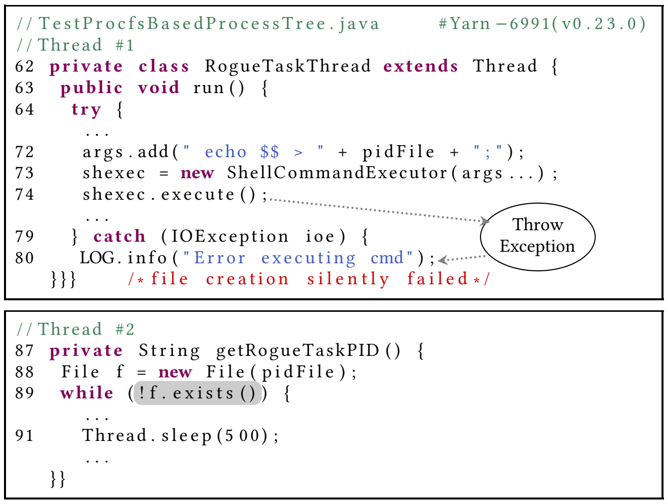
Figure 12: The example when corrupted data content indirectly impacts the loop stride. The corrupted configuration file causes "BUFFER_SIZE = 0", which in turn makes the `InputStream in` **perform read operation on a zero-size byte array and return 0. The loop's exit condition become infeasible because "size < 0" is never satisfied.**

```
//CombineFileInputFormat.java          #Mapreduce-2185(v0.23)
477 private static class OneFileInfo {
     ...
544  for (OneBlockInfo oneblock : blocks) {
545   blockToNodes.put(oneblock, oneblock.hosts);
      ...      /*corrupted block's racks.length is 0*/
549   for (int j = 0; j < oneblock.racks.length; j++) {
550    String rack = oneblock.racks[j];
       ...
554    rackToBlocks.put(rack, blklist);
       ...
    }}}
```

```
255 private void getMoreSplits(...) throws ... {
     ...
348  while (blockToNodes.size() > 0) {
      ...
359   for (Iterator<...> iter = rackToBlocks.
360        entrySet().iterator(); iter.hasNext();) {
361    Map.Entry<...> one = iter.next();
       ...
363    List<OneBlockInfo> blocks = one.getValue();
       ...
369    for (OneBlockInfo oneblock : blocks) {
370     if (blockToNodes.containsKey(oneblock)){
371      blockToNodes.remove(oneblock);
         ...
    }}}}}
```

Figure 13: The example when corrupted data content indirectly impacts the loop stride. Data corruption causes `blockToNodes` **and** `rackToBlocks` **to be different on the dimension of the blocks' number. This difference makes the corrupted block never been removed from the** `blockToNodes` **(i.e., zero-stride), causing the loop's exit condition to be infeasible. This is because "**`blockToNodes.size() <= 0`**" is never satisfied.**

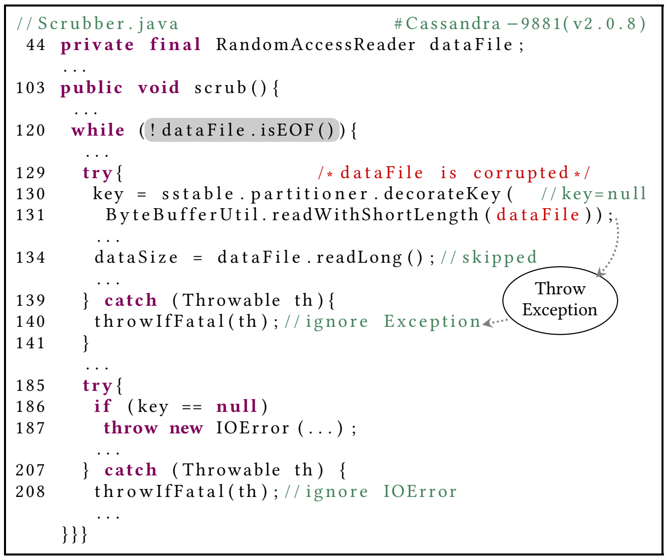
I/O related variables. The `blockToNodes` and `racktoBlock` are two maps which store different metadata information about every data block. If everything works correctly, these two maps should contain information about exactly the same set of blocks (i.e., every record in the `blocks` on line #544). However, if a block (`oneblock` at line #549) is corrupted and its `racks.length` becomes 0, this block will still be inserted into `blockToNodes` at line #545, but not



```
//TestProcfsBasedProcessTree.java      #Yarn-6991(v0.23.0)
//Thread #1
62 private class RogueTaskThread extends Thread {
63  public void run() {
64   try {
     ...
72    args.add(" echo $$ > " + pidFile + ";");
73    shexec = new ShellCommandExecutor(args...);
74    shexec.execute();
     ...
79   } catch (IOException ioe) {
80    LOG.info("Error executing cmd");
  }}}     /*file creation silently failed*/
```

```
//Thread #2
87 private String getRogueTaskPID() {
88  File f = new File(pidFile);
89  while (!f.exists()) {
     ...
91    Thread.sleep(500);
     ...
  }}
```

Figure 14: The example when improper exception handling directly impacts the loop stride. `ShellCommandExecutor.execute()` **causes IOException. The exception is simply logged, and the creation of the** `pidFile` **is silently failed (i.e., zero-stride), which makes** `File.exists()` **always be false.**



```
//Scrubber.java                   #Cassandra-9881(v2.0.8)
 44 private final RandomAccessReader dataFile;
     ...
103 public void scrub(){
     ...
120  while (!dataFile.isEOF()){
      ...
129   try{                       /*dataFile is corrupted*/
130    key = sstable.partitioner.decorateKey(   //key=null
131     ByteBufferUtil.readWithShortLength(dataFile));
       ...
134    dataSize = dataFile.readLong(); //skipped
       ...
139   } catch (Throwable th){
140    throwIfFatal(th); //ignore Exception
141   }
      ...
185   try{
186    if (key == null)
187     throw new IOError(...);
       ...
207   } catch (Throwable th) {
208    throwIfFatal(th); //ignore IOError
       ...
  }}}
```

Figure 15: The example when improper exception handling indirectly impacts the loop stride. Data corruption causes the I/O function `decorateKey()` **to throw exception at line #130-131, which makes the loop skip the index updating statement (i.e., zero-stride) at line #134.**

be put into the `rackToBlocks` map with line #554 skipped. This would eventually cause the `while` loop on line #348 to hang. The reason is that this while loop keeps iterating until every block in `blockToNodes` is removed. Unfortunately, since only blocks that also exist in `rackToBlocks` map can be removed (line #369 – #371), the corrupted block will never be removed from `blockToNodes` and cause an infinite loop.

Type 3: Improper exception handling directly affects loop strides. Sometimes, a data-related operation itself is expected to update the loop stride. When this operation throws an exception, an improper exception handling may give up the operation, together with the associated stride updates, causing infinite loops. For example, the Yarn-6991 bug belongs to this type, shown by Figure 14. The `ShellCommandExecutor.execute()` function is expected to create a `pidFile`, whose existence will help a while loop (line #89) to exit. When the disk is full, `ShellCommandExecutor.execute()` throws an exception at line #74. This exception is simply logged. Consequently, without the creation of `pidFile`, the while loop at line #89 never exits.

Type 4: Improper exception handling indirectly affects loop strides. For this type of bugs, the stride-update operation itself did not raise any exceptions. However, an exception handling of another operation, a data-related operation, changes the control flow and causes the stride update to be skipped. For example, the Cassandra-9881 bug matches this type, shown by Figure 15. When the `dataFile` (`RandomAccessReader` at line #131) is corrupted, the `decorateKey()` function cannot recognize it, thus throws an exception without assigning `key` at line #130 (i.e., `key == null`), or executing `dataFile.readLong()` at line #134. But this exception is simply ignored because it's not fatal at line #140. When the `key` is null, the `scrub()` function throws an `IOError` (line #187), catches it (line #207), and ignores it because it's not a fatal error (line #208). Without moving the index (i.e., zero-stride) by calling `dataFile.readLong()` at line #134, the `scrub()` function keeps reading from the same place, looping forever.

Discussion Theoretically, other types of corruption-hang bugs could exist, like corruption affecting loop bounds, instead of loop strides, or corrupted data content directly, instead of indirectly, affects loop strides. DScope bug detection algorithm *can* detect those types of bugs too. However, we did not observe them in the real-world bugs that we have encountered.

Table 3: The cloud server systems used in our evaluation and the number of detected data corruption hang bugs in each system.

System	Description	# of bugs
Cassandra	Distributed database management system	2
Compress	Libraries for I/O ops on compressed file	2
HD Common	Hadoop utilities and libraries	10
Mapreduce	Hadoop big data processing framework	5
HDFS	Hadoop distributed file system	4
Yarn	Hadoop resource management platform	4
Hive	Data warehouse	12
Kafka	Distributed streaming platform	1
Lucene	Indexing and search server	2
Total		42

4 EVALUATION

In this section, we present our experimental evaluations on DScope. We first describe our evaluation methodology and then discuss our evaluation results in detail.

4.1 Evaluation Methodology

DScope is implemented on top of Soot v2.5.0 [8], a Java bytecode analysis infrastructure, with the latest Coffi library [43], written in Java language with about 18,000 lines of code. Our experimental evaluation covers a wide range of popular cloud server systems listed in Table 3: Cassandra is a distributed key-value store; Compress provides libraries for I/O operations on compressed files; Hadoop common provides utilities and libraries for all Hadoop projects; Hadoop MapReduce is a big data processing platform; HDFS is a distributed file system; Hadoop Yarn is a distributed resource management service; Hive is a data warehouse; Kafka is a distributed streaming system; and Lucene is a data indexing and searching server. We try to cover as many cloud server systems as possible to show that data corruption hang bugs are widespread in the real world.

All the experiments were conducted in our lab machine with an Intel® Xeon® E5-1630 Octa-core 3.7GHz CPU, 16GB memory, running 64-bit Ubuntu 16.04 with kernel v4.13.0. Our evaluation considers both coverage (i.e., true positives) and precision (i.e., false positives) of data corruption hang bug detection. We also compare DScope with two state-of-the-art static bug detection tools, Findbugs(v3.0.1) [7] and Infer(v0.9.2) [6].

For all the hang bugs reported by DScope, we first manually validate them by checking whether we can reproduce the software hang symptom after injecting data corruption into the corresponding data. We first check DScope's analysis results to identify which faulty I/O operations affect the loop strides. We then inject the faults (e.g., corrupted data content, corrupted configuration files, disk exhaustion) into the corresponding I/O operations. If the software hang does happen, we mark the bug as a true positive. Otherwise, we consider it as a false positive. For all the true positives, we then search the bug repository (i.e., JIRA [5]) to see whether they are already reported. If they are, we mark them as the existing data corruption hang bugs. Otherwise, we report them in the bug repository and mark them as newly discovered data corruption hang bugs.

We then use the true positives detected by DScope as the benchmark to evaluate the detection efficacy of Findbugs and Infer. For these two tools, if they report at least one line of the code related to a data corruption hang bug (e.g, a line of the data corruption loop body or a line contains a variable which is then used in the loop), we consider the reported issue as a true positive. We omit the false positives of Findbugs and Infer in our evaluation because these two generic bug detection tools can report hundreds or thousands of suspicious issues. For example, Findbugs and Infer reports 5,434 and 13,993 issues in Hive v2.3.2, respectively. It is extremely time-consuming to validate all of their detection results manually. It is also wrong to label all the issues identified by Findbugs or Infer but not DScope as false positives since some of those issues are true bugs although they are not related to data corruption hang bugs.

4.2 Bug Detection and Precision Results

Table 4 and 5 show the detection results achieved by different schemes. DScope reports 79 data corruption hang bugs, with 42 of them being true bugs, and 29 out of the 42 bugs are newly discovered bugs. Note that, we ran DScope on the target cloud server systems and identified those 42 bugs. But it does not mean that those 42 bugs include *all* the data corruption bugs in those systems. There are some other types of data corruption hang bugs that we cannot identify. For example, data corruption causes the recursive functions never end, making system hang. However, it is out of the scope of this paper, which is part of our future work.

Table 4: The detection comparison of DScope with Findbugs and Infer on all the 9 systems. "TP": the number of true positive bugs by each scheme; "FP": the number of false positive bugs reported by DScope; "-": runtime execution errors (Infer).

System		Release date	DScope		Findbugs	Infer
			TP	FP	TP	TP
Cassandra	v2.0.8	2014/05/29	2	1	0	1
Compress	v1.0	2009/05/21	2	2	0	-
HD Common	v0.23.0	2011/11/11	4	6	0	0
	v2.5.0	2014/08/11	6	6	0	0
Mapreduce	v0.23.0	2011/11/11	3	0	0	0
	v2.5.0	2014/08/11	2	0	0	0
HDFS	v0.23.0	2011/11/11	1	1	0	0
	v2.5.0	2014/08/11	3	5	1	-
Yarn	v0.23.0	2011/11/11	2	2	1	0
	v2.5.0	2014/08/11	2	5	0	0
Hive	v1.0.0	2015/05/20	7	6	0	-
	v2.3.2	2017/11/18	5	1	0	0
Kafka	v0.10.0.0	2016/05/22	1	1	0	0
Lucene	v2.1.0	2007/02/17	2	1	0	0
Total			42	37	2	1

In contrast, existing generic bug detection tools cannot detect most of those 42 data corruption hang bugs. Findbugs only identifies the HDFS-5892 and Yarn-163 bugs while Infer only identifies the Cassandra-9881 bug. Those results are expected because no previous static analysis tools, including Findbugs and Infer, have targeted data corruption hang bugs. Findbugs targets bugs that follow specific anti-patterns in Java programs, such as "private method is never called", "method concatenates strings using + in a loop", and "unchecked type in generic call", none of which are related to data corruption hang bugs detected by DScope. Note that, Findbugs does contain one specific anti-pattern called "an apparent infinite loop" which is related to data corruption hang bugs. However, Findbugs only reports two suspicious issues on the target cloud server systems and both issues involve a `while(true)` type loop. After further inspection, these two loops can exit eventually due to timeouts. Infer mostly focuses on memory and resource leak bugs, and hence cannot detect most corruption-hang bugs shown in Table 5.

Findbugs identifies the HDFS-5892 bug, as it discovers `getFinalizedDir()` can be "null" in the loop body. This bug happens when corrupted data content indirectly affects the loop stride (i.e, the `getFinalizedDir().length` becomes 0). Indeed, the `getFinalizedDir()` function is called during the loop's execution, but it is not the root cause of this data corruption hang bug. Findbugs identifies the Yarn-163 bug, as it discovers that encoding the `InputStreamReader reader` to a `FileReader` can corrupt the `reader`, which

Table 5: The detection comparision of DScope with Findbugs and Infer on all the 42 data corruption hang bugs.

#	Bug name	System version	Bug type	Known or new	Deteced		
					DScope	Findbugs	Infer
1	Cassandra-7330	v2.0.8	#1	known	✓	✗	✗
2	Cassandra-9881	v2.0.8	#3	known	✓	✗	✓
3	Compress-87	v1.0	#1	known	✓	✗	✗
4	Compress-451	v1.0	#2	new	✓	✗	✗
5	Hadoop-8614	v0.23.0	#1	known	✓	✗	✗
6	Hadoop-15088	v2.5.0	#1	new	✓	✗	✗
7	Hadoop-15415	v0.23.0	#2	new	✓	✗	✗
8		v2.5.0	#2	new	✓	✗	✗
9	Hadoop-15417	v0.23.0	#2	new	✓	✗	✗
10		v2.5.0	#2	new	✓	✗	✗
11	Hadoop-15424	v2.5.0	#1	new	✓	✗	✗
12	Hadoop-15425	v2.5.0	#1	new	✓	✗	✗
13	Hadoop-15429	v0.23.0	#2	new	✓	✗	✗
14		v2.5.0	#2	new	✓	✗	✗
15	HDFS-4882	v0.23.0	#3	known	✓	✗	✗
16	HDFS-5892	v2.5.0	#2	known	✓	✓	✗
17	HDFS-13513	v2.5.0	#2	new	✓	✗	✗
18	HDFS-13514	v2.5.0	#2	new	✓	✗	✗
19	Mapreduce-2185	v0.23.0	#2	known	✓	✗	✗
20	Mapreduce-2862	v0.23.0	#2	known	✓	✗	✗
21	Mapreduce-6990	v0.23.0	#1	new	✓	✗	✗
24	Mapreduce-7088	v2.5.0	#1	new	✓	✗	✗
25	Mapreduce-7089	v2.5.0	#1	new	✓	✗	✗
26	Yarn-163	v0.23.0	#1	known	✓	✓	✗
27	Yarn-2905	v2.5.0	#1	known	✓	✗	✗
22	Yarn-6991	v0.23.0	#4	new	✓	✗	✗
23		v2.5.0	#4	new	✓	✗	✗
28	Hive-5235	v1.0.0	#1	known	✓	✗	✗
29	Hive-13397	v1.0.0	#2	known	✓	✗	✗
30	Hive-18142	v1.0.0	#2	new	✓	✗	✗
31	Hive-18216	v2.3.2	#1	new	✓	✗	✗
32	Hive-18217	v2.3.2	#1	new	✓	✗	✗
33	Hive-18219	v1.0.0	#2	new	✓	✗	✗
34		v2.3.2	#2	new	✓	✗	✗
35	Hive-19391	v1.0.0	#2	new	✓	✗	✗
36	Hive-19392	v1.0.0	#2	new	✓	✗	✗
37		v2.3.2	#2	new	✓	✗	✗
38	Hive-19395	v1.0.0	#1	new	✓	✗	✗
39	Hive-19406	v2.3.2	#2	new	✓	✗	✗
40	Kafka-6271	v0.10.0	#1	new	✓	✗	✗
41	Lucene-772	v2.1.0	#2	known	✓	✗	✗
42	Lucene-8294	v2.1.0	#2	new	✓	✗	✗
Total #					42	2	1

is related to the data corruption hang bugs — performing skip operations on a corrupted `FileReader` can cause the `skip` function to return error code (i.e, 0).

Infer identifies the Cassandra-9881 bug, as it discovers that the `scrub()` function in the `Scrubber` class could invoke a `throwIf-Commutative()` function at line #248 with null parameter, shown by Figure 16. As we discussed in §3, when data corruption happens, key fails to be assigned to new values and sticks with the default

```
//cassandra -2.0.8: Scrubber.java
 41 private boolean isCommutative = false;
    ...
103 public void scrub(){
     ...
120  while (!dataFile.isEOF()){
      ...
127   DecoratedKey key = null;
      ...
248   throwIfCommutative(key, th);//Infer: null parameter
      ...
    }}
```

```
327 private void throwIfCommutative(DecoratedKey key,
328                                 Throwable th) {
329  if (isCommutative && !skipCorrupted){
331    outputHandler.warn(String.format("...", key));
       ...
    }}
```

Figure 16: Infer identifies a null parameter problem in the `throwIfCommutative()` function at line #248. The Cassandra-9881 bug happens at line #103-256.

Table 6: The types of false positives pruned by DScope.

System		Pruned FP	
		Numeric primitives	Java APIs
Cassandra	v2.0.8	386	71
Compress	v1.0	147	20
HD Common	v0.23.0	1023	378
	v2.5.0	1650	790
Mapreduce	v0.23.0	377	363
	v2.5.0	938	641
HDFS	v0.23.0	312	323
	v2.5.0	1723	1073
Yarn	v0.23.0	151	214
	v2.5.0	451	665
Hive	v1.0.0	4268	3003
	v2.3.2	5269	3663
Kafka	v0.10.0.0	186	441
Lucene	v2.1.0	287	44
Total		17168	11689

value, "null". This makes the `scrub()` function skip updating the index, causing an infinite loop. Indeed, the `throwIfCommutative()` function is called during the loop's execution, but it is not the root cause of this data corruption hang bug. In fact, it does not break the loop to prevent `scrub()` from hanging. This is because the `isCommutative` variable is false, which makes the `if` branch at line #329 unreachable. Thus, even with a null parameter, the `throwIfCommutative()` can still execute successfully at line #248.

Table 5 also shows the types of the detected data corruption hang bugs. As we can see, "Type 1" and "Type 2" cover most of the detected bugs — 16 and 22 bugs respectively. This indicates that most of the data corruption hang bugs happen when the data corruption causes the error code returned by I/O operations to directly impact the loop stride or corrupted data content indirectly impacts the loop stride.

To understand how DScope does not prune all the false positives, we manually study those 37 false positives in Table 4. We find most of cases require inter-procedural analysis to identify. We will discuss it in §5.

As shown in Table 6, DScope prunes 28,857 false positives in total, including 17,168 cases where the loop index, stride and bounds are denoted by numeric primitives, and 11,689 cases where the loop index, stride and bounds are denoted by commonly used Java APIs.

We should note that, we do not intend to claim that DScope can replace those generic bug detection tools such as Findbugs and Infer. We believe our bug detection schemes are complementary to those existing tools and could be used in combination by the software developer.

5 DISCUSSION

We observe that in most of the 37 false positive cases, the forwarding-index/reversing-index Java APIs and the checking-bounds Java APIs are located in different application functions. These APIs are indirectly invoked in the application functions which are invoked in the loop paths. To further reduce false positives, we plan to conduct inter-procedural analysis on all the bug candidates to generate the loop paths where the loop index, stride, and bounds are denoted by either numeric primitives or Java APIs. We then adopt DScope's false positive pruning principles to prune the false positives without missing true positives.

DScope focuses on detecting data-corruption hang bugs. We plan to explore auto-fix schemes to correct those bugs based on their types, as described in §3. For example, when the error code returned by the I/O operations directly affects the loop strides, one possible fix is to add extra error code checking statements in the loop exit conditions to avoid the software hang. If the corrupted data content indirectly affects loop strides, one possible fix could be adding additional check over the data content before it is used in the loop body. When the improper exception handling causes the loop stride update to be skipped, one possible fix is to add additional exception handling to properly update the stride when data corruption occurs.

6 RELATED WORK

Data corruption study and detection: Previous work has been extensively studied the data corruption problems in storage systems. Hwang et al. [27] and Schroeder et al. [39] studied the data corruptions in memory devices. They found that DRAM failures occur more frequently than expected. Bairavasundaram et al.[11, 12] and Oleksenko et al. [36] detected the data pointer corruptions on disks. They showed that disk failures are prevalent for data corruptions. Previous works have also been done to detect data corruptions in file systems. ZFS [14] detected file system corruption caused by storage hardware, e.g., latent sector errors. Fryer et al. [22, 41] implemented runtime data corruption detectors for the Ext3 and Btrfs file systems.

The above work provides motivations for us to study data corruption induced performance problems. Our work focuses on detecting data corruption hang bugs in software-level rather than detecting the data corruption itself (hardware-level).

Performance bug detection and diagnosis: Much work has been done to detect and diagnose performance bugs in large scale systems. X-ray [9] uses symbolic execution to automatically identify and suggest fixes to performance bugs caused by configuration or input-based problems. Xu et al. [47] presented a clustering-based scheme to detect system anomalies. Jin et al. [29] employed rule-based methods to detect performance bugs that violate known efficiency rules. Caramel [33] statically detects inefficient loops that can be fixed by adding conditional-breaks. LDoctor [40] provides statistical diagnosis for inefficient loops. Jolt [16] dynamically detects infinite loops by checking each loop iteration's runtime state. Tools also exist to detect inefficient nested loops [34] and workload-dependent loops [46].

In comparison, our work focuses on detecting data corruption induced software hang problems before they are triggered in the production system. We adopt a pattern-driven approach instead of rule-based or anomaly detection approaches to achieving both high coverage and precision for our targeted data corruption hang bugs.

Previous work Carburizer [30] statically analyzes device driver code and identifies infinite driver-polling problems. That is, a driver may wait for a device to enter a given state by polling a device register. Once the register data is corrupted, a buggy driver may be stuck forever. DScope and Carburizer both statically analyze loops and loop-exit conditions. However, they face different design challenges due to the different types of bugs they target. DScope targets cloud systems written in Java, instead of low-level device drivers, and hence needs to handle a much broader set of I/O functions and I/O related data (e.g., not only data retrieved by I/O operations but also status returned by I/O operations), and more complicated control flows caused by Java exceptions. Carburizer false-positive pruning only involves identifying loop timeouts. However, DScope has to conduct sophisticated loop stride and bound analysis in its false-positive pruning. Finally, as indicated in §3, the type of corruption-hang bugs identified by DScope in cloud systems go much beyond simple I/O-state infinite polling problems, where the device register content often directly updates the loop stride.

Fault injection: Previous work [13, 24, 42] used fault injection techniques to analyze the failure behaviors (e.g., hang, crash) of both software and hardware systems. For example, HSFI [42] injected faults in the source code. Fault injection is also widely used to check whether file systems can handle certain type of data corruptions [21, 23, 38, 48]. For instance, Bairavasundaram et al. [10] used context aware fault injections to find disk errors in virtual memory systems. Zhang et al. [48] conducted a comprehensive reliability case study of local file systems to analyze both on-disk and in-memory data integrity in Sun's ZFS. Their results show that file systems are robust to disk corruption but less resilient to memory corruptions. Cords [23] exposed data losses, block corruptions, and unavailability problems commonly exist in distributed file systems. Cords also indicated that modern distributed file systems are not equipped to effectively use redundancy across replicas to recover from local file system faults. In contrast, our work focuses on detecting potential data corruption hang bugs before they are triggered by the data corruption faults. We only rely on static code analysis, which can be easily applied to different cloud server systems. We believe our work is complementary to the fault injection based approaches which can be used to validate our candidate bugs and further reduce false positives.

Functional bug detection: Apart from performance bugs, recent works have also been done to detect functional bugs. pbSE [45] conducted concolic execution to detect functional bugs and generate test cases for those bugs. Kollenda et al. [32] detected the crash bugs by identifying the crash-resistant primitives via system calls on Linux, Windows API functions, and exception handlers. In contrast, our work focuses on detecting performance bugs, which requires the bug detection system to focus on different aspects of the program such as loop exit checking.

Software testing: DeepXplore [37] is a whitebox framework to test deep learning systems. DeepXplore takes unlabeled test inputs as seeds in DNN systems. It uses gradient ascent to modify the input to maximize chance of finding rare corner cases. Fex [35] is a software system evaluator, which collects a set of reused scripts to develop a matured evaluation framework. Fex addressed the limitation of rigid, simplistic and inconsistent in large system testing. Elia et al. [20] designed an interoperability certification model, which facilitates testing interoperability among different web applications. Our work is complementary to those software testing tools. Our tool can identify potential buggy functions with infinite loops, which can guide the test case generation to further test our detection results.

7 CONCLUSION

In this paper, we have presented DScope, a new data corruption hang bug detection tool for cloud server systems. DScope combines candidate bug discovery and false positive pattern filtering to detect software hang bugs that are related to data corruptions. DScope is fully automatic without requiring any user input or predefined rules. We have implemented a prototype of DScope and evaluated it over 9 commonly used cloud server systems. DScope successfully detects 42 true corruption hang bugs (29 of them are new bugs) while existing bug detection tools can only detect very few of them (2 by Findbugs and 1 by Infer).

ACKNOWLEDGMENTS

We would like to thank Ennan Zhai, our shepherd, and the anonymous reviewers for their insightful feedback and valuable comments. This work was sponsored in part by NSF CNS1513942 grant and NSF CNS1149445 grant. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or U.S. Government.

REFERENCES

[1] 2013. HDFS-4882. https://issues.apache.org/jira/browse/HDFS-4882.

[2] 2017. What Lessons can be Learned from BA's Systems Outage? http://www.extraordinarymanagedservices.com/news/what-lessons-can-be-learned-from-bas-systems-outage/.

[3] 2018. Apache Cassandra. http://cassandra.apache.org/.

[4] 2018. Apache Hadoop. http://hadoop.apache.org/.

[5] 2018. Apache JIRA. https://issues.apache.org/jira.

[6] 2018. Facebook Infer. http://fbinfer.com/.

[7] 2018. Findbugs. http://findbugs.sourceforge.net/.

[8] 2018. Soot: A Framework for Analyzing and Transforming Java and Android Applications. https://sable.github.io/soot/.

[9] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *OSDI*.

[10] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2006. Dependability Analysis of Virtual Memory Systems. In *DSN*.

[11] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. 2008. An Analysis of Data Corruption in the Storage Stack. *TOS* 4, 3 (nov 2008), 8:1–8:28.

[12] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. 2008. Analyzing the Effects of Disk-Pointer Corruption. In *DSN*.

[13] James H. Barton, Edward W. Czeck, Zary Z. Segall, and Daniel P. Siewiorek. 1990. Fault Injection Experiments Using FIAT. *TC* 39, 4 (apr 1990).

[14] Jeff Bonwick and Bill Moore. 2007. *ZFS–The Last Word In File Systems.* https://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf.

[15] Nedyalko Borisov, Shivnath Babu, Nagapramod Mandagere, and Sandeep Uttamchandani. 2011. Dealing Proactively with Data Corruption: Challenges and Opportunities. In *SMDB*.

[16] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. 2011. Detecting and Escaping Infinite Loops with Jolt. In *ECOOP*.

[17] Ting Dai, Daniel Dean, Peipei Wang, Xiaohui Gu, and Shan Lu. 2018. Hytrace: A Hybrid Approach to Performance Bug Diagnosis in Production Cloud Infrastructures. *IEEE Transactions on Parallel and Distributed Systems* (2018).

[18] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. 2014. PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures. In *SOCC*.

[19] Daniel J. Dean, Peipei Wang, Xiaohui Gu, Willam Enck, and Guoliang Jin. 2015. Automatic Server Hang Bug Diagnosis: Feasible Reality or Pipe Dream?. In *ICAC*.

[20] Ivano Alessandro Elia, Nuno Laranjeiro, and Marco Vieira. 2015. Test-Based Interoperability Certification for Web Services. In *DSN*.

[21] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In *SC*.

[22] Daniel Fryer, Mike Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. 2014. Checking the Integrity of Transactional Mechanisms. *TOS* 10, 4 (oct 2014), 17:1–17:23.

[23] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *FAST*.

[24] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhen-Yu Yang. 2003. Characterization of Linux Kernel Behavior Under Errors. In *DSN*.

[25] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 2014. What Bugs Live in the Cloud?: A Study of 3000+ Issues in Cloud Systems. In *SOCC*.

[26] Jian Huang, Xuechen Zhang, and Karsten Schwan. 2015. Understanding Issue Correlations: A Case Study of the Hadoop System. In *SOCC*.

[27] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. 2012. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *ASPLOS*.

[28] Weihang Jiang, Chongfeng Hu, Arkady Kanevsky, and Yuanyuan Zhou. 2008. Is Disk the Dominant Contributor for Storage Subsystem Failures? A Comprehensive Study of Failure Characteristics. In *FAST*.

[29] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *PLDI*.

[30] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. 2009. Tolerating Hardware Device Failures in Software. In *SOSP*.

[31] Jonathan Kaldor, Jonathan Mace, MichałBejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *SOSP*.

[32] Benjamin Kollenda, Enes Göktaş, Tim Blazytko, Philipp Koppe, Robert Gawlik, RK Konoth, Cristiano Giuffrida, Herbert Bos, and Thorsten Holz. 2017. Towards Automated Discovery of Crash-Resistant Primitives in Binary Executables. In *DSN*.

[33] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In *ICSE*.

[34] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-Access Patterns. In *ICSE*.

[35] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, and Christof Fetzer. 2017. Fex: A Software Systems Evaluator. In *DSN*.

[36] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Christof Fetzer, and Pascal Felber. 2016. Efficient Fault Tolerance using Intel MPX and TSX. In *DSN*.

[37] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *SOSP*.

[38] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON File Systems. In *SOSP*.

[39] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2009. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS*.

[40] Linhai Song and Shan Lu. 2017. Performance Diagnosis for Inefficient Loops. In *ICSE*.

[41] Kuei Sun, Daniel Fryer, Dai Qin, Angela Demke Brown, and Ashvin Goel. 2014. Robust Consistency Checking for Modern Filesystems. In *RV*.

[42] Erik van der Kouwe and Andrew S Tanenbaum. 2016. HSFI: Accurate Fault Injection Scalable to Large Code Bases. In *DSN*.

[43] Clark Verbrugge. 1996. *Using Coffi*. http://www.sable.mcgill.ca/~clump/Coffi/Coffi.ps.

[44] Peipei Wang, Daniel J. Dean, and Xiaohui Gu. 2015. Understanding Real World Data Corruptions in Cloud Systems. In *IC2E*.

[45] Qixue Xiao, Yu Chen, Chengang Wu, Kang Li, Junjie Mao, Shize Guo, and Yuanchun Shi. 2017. pbSE: Phase-Based Symbolic Execution. In *DSN*.

[46] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. 2013. Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks. In *ISSTA*.

[47] Kui Xu, Ke Tian, Danfeng Yao, and Barbara G. Ryder. 2016. A Sharper Sense of Self: Probabilistic Reasoning of Program Behaviors for Anomaly Detection with Context Sensitivity. In *DSN*.

[48] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2010. End-to-End Data Integrity for File Systems: A ZFS Case Study. In *FAST*.