How Far Have We Come in Detecting Anomalies in Distributed Systems? An Empirical Study with a Statement-level Fault Injection Method

Yong Yang\textsuperscript{1}, Yifan Wu\textsuperscript{1}, Karthik Pattabiraman\textsuperscript{2}, Long Wang\textsuperscript{3}, Ying Li\textsuperscript{1}

Peking University\textsuperscript{1}
University of British Columbia\textsuperscript{2}
IBM Watson\textsuperscript{3}
Background

- Distributed systems **widely deployed** in various sectors
- With the **increasing scale** and **complexity**, distributed systems suffering from frequent **software and hardware faults**
- The **early detection of the symptoms of failures**, *i.e.* anomalies, can **mitigate or even prevent severe failures**

The evolvement of faults in distributed systems (Hadoop)
A variety of Anomaly Detection (AD) methods

- Log-based methods: Deeplog\textsuperscript{[2]}, PCA approach\textsuperscript{[3]}, \textit{etc.}
- Metrics-based methods: LSTM-AD\textsuperscript{[4]}, Information-theoretic approach\textsuperscript{[5]}, \textit{etc.}
- Trace-based methods: READ\textsuperscript{[6]}, Path similarity approach\textsuperscript{[7]}, \textit{etc.}

What are the advantages and the disadvantages of various anomaly detectors?

No one has tried to systematically evaluate anomaly detectors of distributed systems to explore how far we have come and how we should move forward.
Motivation

A fault injection method that can simulate realistic faults to generate a wide variety of anomalies is the prerequisite for comprehensively evaluating anomaly detectors

- Bit-flip FI techniques, inefficient in distributed systems
- Injecting failures cannot simulate realistic faults
- Existing code-change FI techniques, only covering few types of faults

Limited coarse-grained failures cannot represent the diversity of anomalies. The process of a fault evolving into a failure is missing.

A code snippet from Hadoop(NodeManager):

```java
public class DeprecatedKeyInfo {
    private static final String[] newKeys;
    private final String getWarningMessage(String key) {
        String warningMessage;
        if(customMessage == null) {
            String message = new StringBuilder(key);
            message.append(deprecatedKeySuffix);
            for (int i = 0; i < newKeys.length; i++) {
                message.append(newKeys[i]);
            }
            warningMessage = message.toString();
        }
        return warningMessage;
    }
}
```

A code snippet from Hadoop
Overview

- A systematic approach to evaluate the efficacy of anomaly detectors

**RQ1:** What’s the pattern of anomalies in distributed systems?

**RQ2:** To what extent do distributed systems, by themselves, report the anomalies?

**RQ3:** To what extent do state-of-the-art anomaly detectors detect anomalies of different types?
Fault Injection Methodology

■ Fault Model

■ Faults on a single statement: based on an analysis of elements of 8 fundamental statements

■ Faults on multiple statements: based on an analysis of the real software bugs found in the recent bug study\(^8\) of Openstack

The fault model of SSFI

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Fault Source</th>
<th>Statements</th>
<th>Description</th>
<th>Corresponding Bugs in Openstack ([8])</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE_CHANGE</td>
<td>left/right operand</td>
<td>AssignStmt</td>
<td>Add/subtract/zero/negative/change a variable to a certain value</td>
<td>Wrong SQL Value, Wrong Parameter Value, Wrong SQL Where, Wrong SQL Column, Wrong Value, Missing Parameters, Wrong Parameter Order, Wrong Table, HOG</td>
</tr>
<tr>
<td>NULLIFY</td>
<td>left/right operand</td>
<td>AssignStmt</td>
<td>Set an object(pointer) to NULL</td>
<td>Missing Key Value Pair, Missing Dict Value</td>
</tr>
<tr>
<td>EXCEPTION_SHORTCIRCUIT</td>
<td>the only operand</td>
<td>ThrowStmt</td>
<td>Directly throw one of the declared exceptions or the exceptions in try-catch block</td>
<td>Wrong Return Value</td>
</tr>
<tr>
<td>INVOKE_REMOVAL</td>
<td>-</td>
<td>InvokeStmt</td>
<td>Remove a method invoking statement without return values</td>
<td>Missing Function Call, Missing Method Call</td>
</tr>
<tr>
<td>ATTRIBUTE_SHADOWED</td>
<td>the left operand</td>
<td>AssignStmt</td>
<td>Exchange the field and the local variable (with same name and type)</td>
<td>Wrong Variable Value</td>
</tr>
<tr>
<td>CONDITION_INVERSED</td>
<td>binary logical operation</td>
<td>IFStmt</td>
<td>Inverse the if-else block</td>
<td>Wrong API use</td>
</tr>
<tr>
<td>CONDITION_BORDER</td>
<td>binary logical operation</td>
<td>IFStmt</td>
<td>Replace the logical operation with one arithmetic operation including/excluding the border value</td>
<td>Wrong Access Method</td>
</tr>
<tr>
<td>SWITCH_FALLTHROUGH</td>
<td>destination label/address</td>
<td>GotoStmt</td>
<td>Add/Remove a break between two cases of the switch structure</td>
<td>Wrong SQL Column</td>
</tr>
<tr>
<td>SWITCH_MISS_DEFAULT</td>
<td>destination label/address</td>
<td>SwitchStmt</td>
<td>Remove the default case process block of the switch structure</td>
<td>Wrong Access Key</td>
</tr>
<tr>
<td>SYNCHRONIZATION</td>
<td>-</td>
<td>SyncStmt</td>
<td>Delete the synchronization modifier for a method/block</td>
<td>Missing Sync Annotation</td>
</tr>
<tr>
<td>EXCEPTION_UNCAUGHT</td>
<td>bugs in Openstack</td>
<td>ThrowStmt</td>
<td>Directly throw an undeclared exception for a method or a try-catch block</td>
<td>Missing Exception Handlers</td>
</tr>
<tr>
<td>EXCEPTION_UNHANDED</td>
<td>bugs in Openstack</td>
<td>AssignStmt</td>
<td>Remove all the statements in the catch block</td>
<td>Inject Resource Leak</td>
</tr>
</tbody>
</table>
SSFI Overview

SSFI (Statement-level Software FI), able to inject **12 different types of software faults** into software systems that can be compiled into Bytecode. SSFI also provides **always/random activation mode** for each fault.

An overview of SSFI’s fault injection process.
Fault Injection Methodology

```java
public long calculate(int testNumber) {
    for (int i = 0; i < 5; i++) {
        testNumber = testNumber + 1;
    }
    return testNumber;
}
```

A: Source code

```java
public long calculate();

Code:
0:  iconst_0
1:  istore_2
2:  goto   11
5:  iinc   1, 1
8:  iinc   2, 1
11: iload_2
12: iconst_5
13: if_icmplt 5
16: iload_1
17: iinc

Fault injection parameters from Config Parser (fault type, location, etc.)
B: Bytecode

Bytecode Parser parses runnable bytecode into Jimple code

```java
public long calculate(int)
{
    WorkBench this;
    long $stack5;
    int testNumber, i;
    this := @this: WorkBench;
    testNumber := @parameter0: int;
    i = 0;
    goto label2;

label1:
    testNumber = testNumber + 1;
    i = i + 1;

label2:
    if i < 5 goto label1;
    $stack5 = (long) testNumber;
    return $stack5;
}
```

C: Jimple code

An example fault injected using SSFI

How Far Have We Come in Detecting Anomalies in Distributed Systems? An Empirical Study with a Statement-level Fault Injection Method
How Far Have We Come in Detecting Anomalies in Distributed Systems? An Empirical Study with a Statement-level Fault Injection Method

Fault Injection Methodology

An example fault injected using SSFI

C: Jimple code

public long calculate(int)
{
    WorkBench this;
    long $stack5;
    int testNumber, i;
    this := @this: WorkBench;
    testNumber := @parameter0: int;
    i = 0;
    goto label2;
label1:
    testNumber = testNumber + 1;
    i = i + 1;
label2:
    if i < 5 goto label1;
    $stack5 = (long) testNumber;
    return $stack5;
}

Fault injection parameters from Config Parser (fault type, location, etc.)

Converter compiles the modified Jimple code into runnable bytecode with an injected fault

Fault Weaver modifies the Jimple code to injected a specified fault

D: Modified Jimple code

public long calculate(int)
{
    Code:
    0: iconst_0
    1: istore_0
    2: goto 11
    5: iinc 1, 1
    8: iinc 0, 1
    11: iload_0
    12: iconst_5
    13: if_icmple 5 ;
    16: iload_1
    17: i2l
    18: lreturn
    Return $stack5;
}

E: Modified Bytecode

public long calculate(int testNumber)
{
    for(int i=0;i<=5;i++) {
        testNumber=testNumber+1;
    }
    return testNumber;
}

Source code
Evaluation Results

■ Evaluation Setup

<table>
<thead>
<tr>
<th>Systems</th>
<th>Workload</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>Wordcount</td>
<td>A data processing system with MapReduce programming model and HDFS</td>
</tr>
<tr>
<td>HaloDB</td>
<td>CRUD</td>
<td>A key-value store written in Java</td>
</tr>
<tr>
<td>Weka</td>
<td>Bayes Classi-</td>
<td></td>
</tr>
<tr>
<td>Spark</td>
<td>Wordcount</td>
<td>A cluster-computing framework with HDFS</td>
</tr>
<tr>
<td>Flink</td>
<td>Wordcount</td>
<td>A stream-processing framework</td>
</tr>
</tbody>
</table>

■ Three anomaly detectors
  - Deeplog (log-based)
  - MRD (metrics-based)
  - READ (trace-based)

Fig. 7 Different types of anomalies

Fig. 6 An overview of the injected faults
Evaluation Results

- **Silent Early Exit** anomalies, **more frequent** in distributed systems due to incomplete error-resilience mechanisms.

The anomaly distribution in target systems

```java
public class DefaultSpeculator extends AbstractService implements Speculator {
    protected void serviceStart() throws Exception {
        Runnable speculationBackgroundCore = new Runnable() {
            @Override
            public void run() {
                ...
            }
        };
        speculationBackgroundThread = new Thread(
            speculationBackgroundCore, "processing");
        speculationBackgroundThread.start();
        super.serviceStart();
    }
}
```

The anomaly distribution in target systems

**Explicitly record the error messages** when designing the **error-handling mechanisms**, regardless of whether the error is believed to be tolerated.
The error reporting mechanisms, able to report the majority of the anomalies (recall ranging from 82.1% to 92.8%) but with a high false alarm rate (26.6%).

Simple methods are feasible, but get ready for frequent false alarms.
Evaluation Results

- **Log-based** method, **better overall detection results** than **trace-based** and **metrics-based** methods, but **not for all anomaly types**

- State-of-the-art anomaly detectors, able to detect the existence of anomalies with **99.08% precision and 90.60% recall**

<table>
<thead>
<tr>
<th>detector</th>
<th>precision</th>
<th>recall</th>
<th>f1-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deeplog</td>
<td>99.08%</td>
<td>90.60%</td>
<td>94.65%</td>
</tr>
<tr>
<td>MRD</td>
<td>68.85%</td>
<td>79.52%</td>
<td>71.77%</td>
</tr>
<tr>
<td>READ</td>
<td>87.13%</td>
<td>90.10%</td>
<td>88.59%</td>
</tr>
</tbody>
</table>

The detection precision and recall for each anomaly type

Existing AD methods are **powerful** to decide **whether there are anomalies**
Evaluation Results

- There is still **a long way to go to pinpoint the accurate location** of the detected anomalies.

<table>
<thead>
<tr>
<th>detector</th>
<th>detection latency</th>
<th>locating accuracy at class-level</th>
<th>locating accuracy at component-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deeplog</td>
<td>3.42%</td>
<td>29.34%</td>
<td>71.23%</td>
</tr>
<tr>
<td>READ</td>
<td>2.11%</td>
<td>-</td>
<td>78.32%</td>
</tr>
</tbody>
</table>
Summary

- A systematic approach to evaluating existing anomaly detectors
- A realistic software fault injection method for distributed systems
- Findings from the comprehensive evaluation give inspiration for developers and researchers
Q&A

SSFI project: https://github.com/alexvanc/ssfi
Email: yang.yong@pku.edu.cn
References


