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

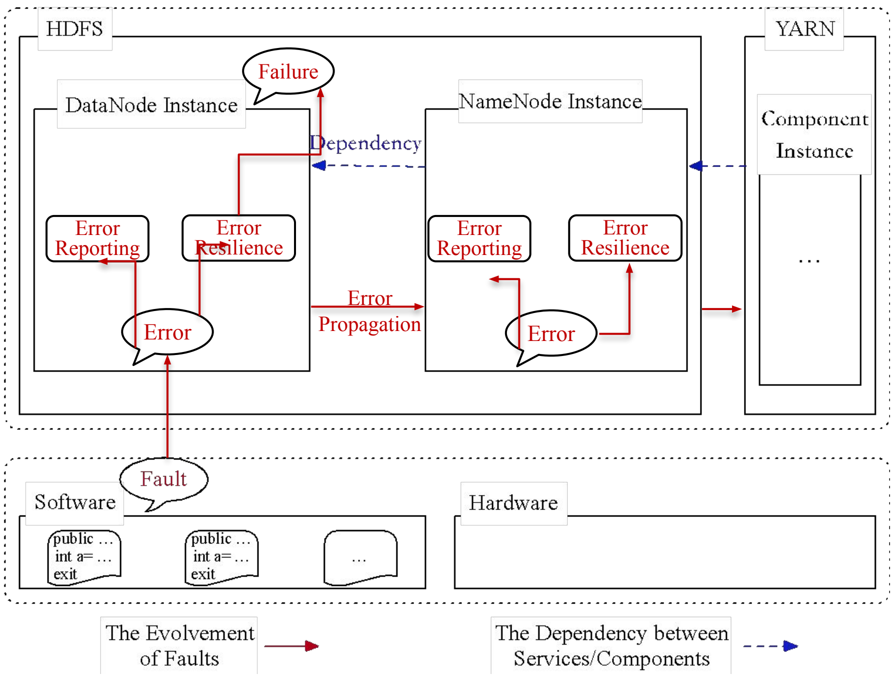
Yong Yang[1], Yifan Wu[1], Karthik Pattabiraman[2] , Long Wang[3], Ying Li[1]

Peking University[1]
University of British Columbia[2]
IBM Watson[3]

UBC IBM Watson 北京大學

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)

Background

- **A variety of Anomaly Detection(AD) methods**
  - Log-based methods: Deeplog[2], PCA approach[3], *etc.*
  - Metrics-based methods: LSTM-AD[4], Information-theoretic approach[5], *etc.*
  - Trace-based methods: READ[6], Path similarity approach[7], *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

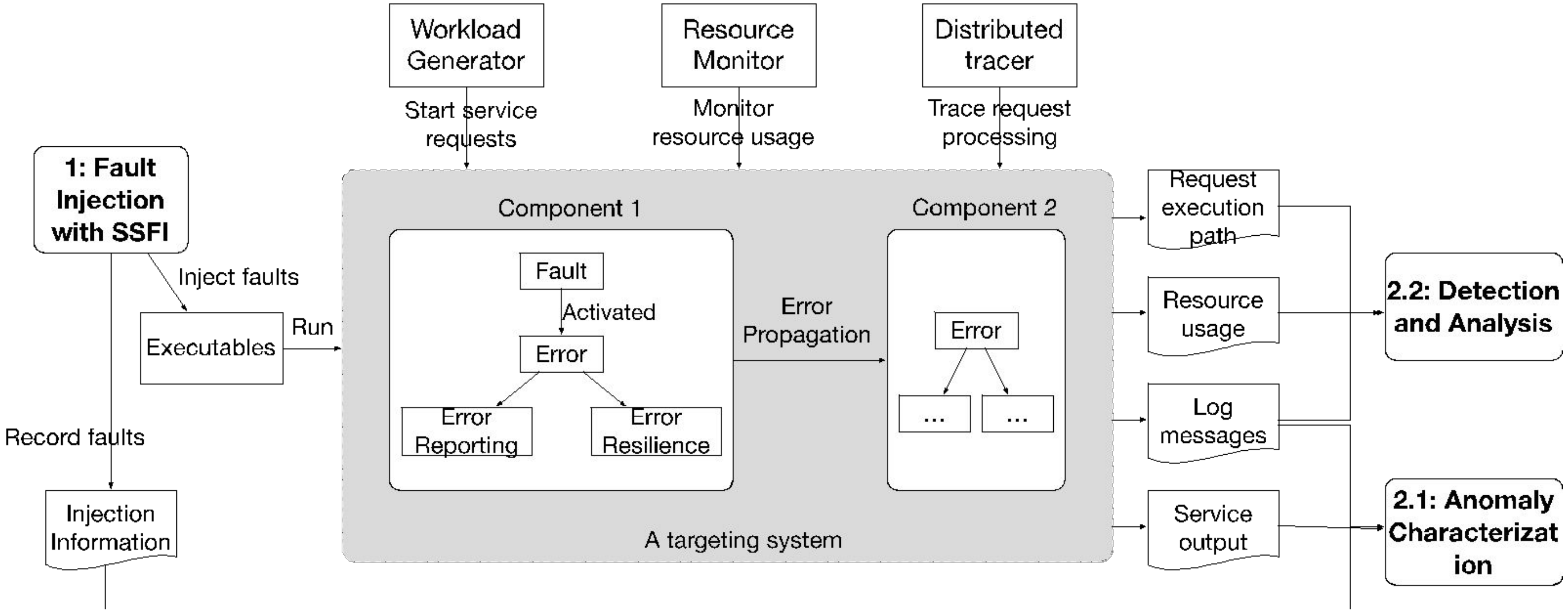
```
private static class DeprecatedKeyInfo {
    private final String[] newKeys;
    private final String getWarningMessage(String key) {
      String warningMessage;
      if(customMessage == null) {
        StringBuilder message = new StringBuilder(key);
        message.append(deprecatedKeySuffix);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

A code snippet from Hadoop

A code snippet from Hadoop(NodeManager)

Overview

- **A systematic approach to evaluate the efficacy of anomaly detectors**



An overview of the evaluation approach

- **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

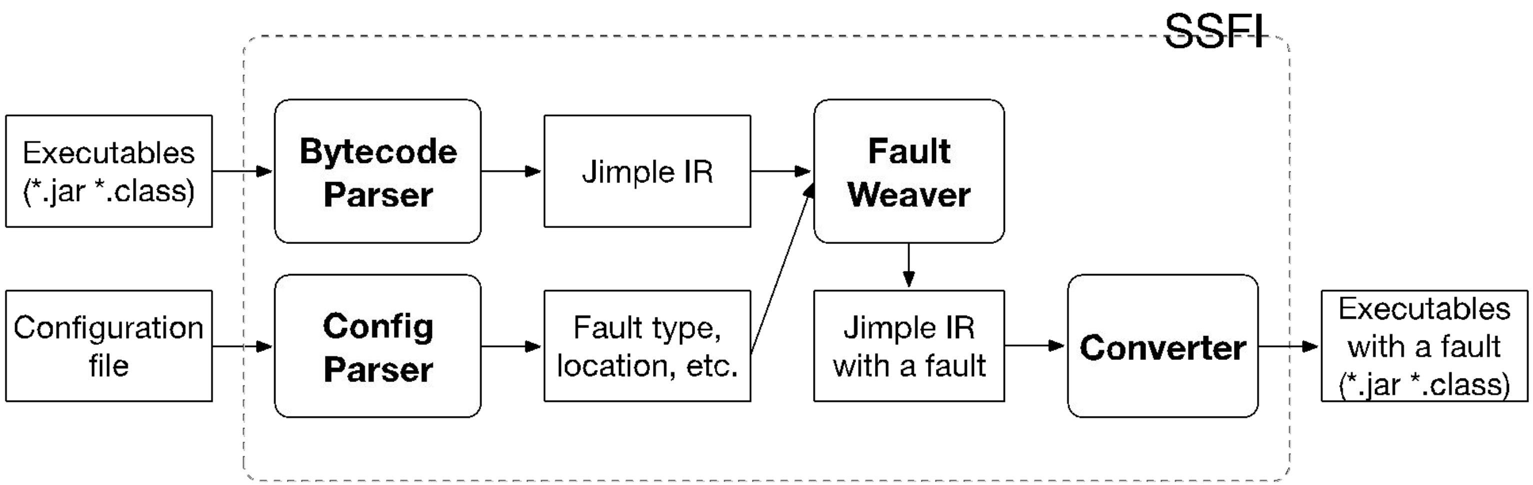
The fault model of SSFI

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

Fault Injection Methodology

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

```
public long calculate(int);
  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: i2l
    18: lreturn
```

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

B: Bytecode

Bytecode Parser

parses runnable bytecode into 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;
}
```

C: Jimple code

An example fault injected using SSFI

Fault Injection Methodology

```
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

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

Fault Weaver modifies the Jimple code to injected a specified fault

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

```
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
}
```

E: Modified Bytecode

D: Modified Jimple code

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

Source code

An example fault injected using SSFI

Evaluation Results

Evaluation Setup

Systems used for evaluation

Systems	Workload	Descrition
Hadoop	Wordcount	A data processing system with MapReduce programming model and HDFS
HaloDB	CRUD	A key-value store written in Java
Weka	Bayes Classification	A program that implements a collection of machine learning algorithms
Spark	Wordcount	A cluster-computing framework with HDFS
Flink	Wordcount	A stream-processing framework

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

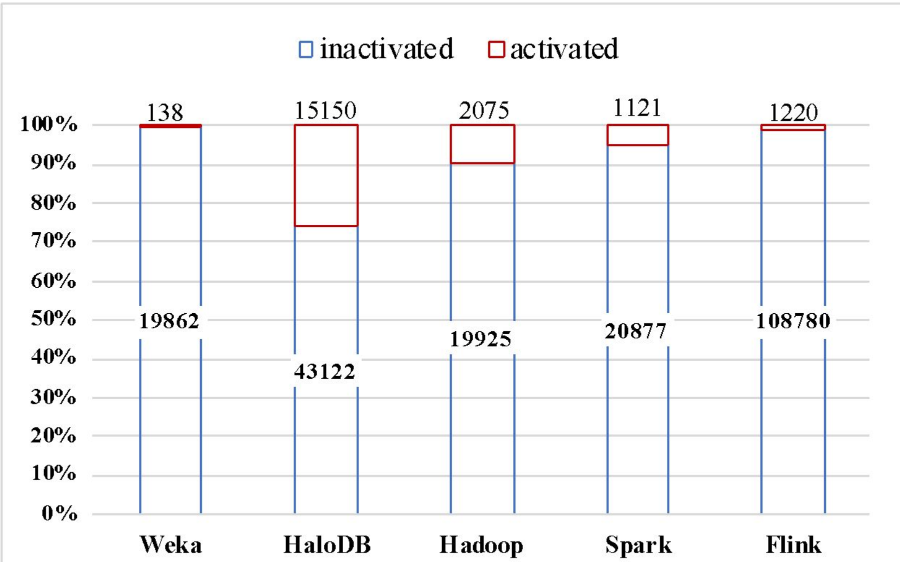


Fig.6 An overview of the injected faults

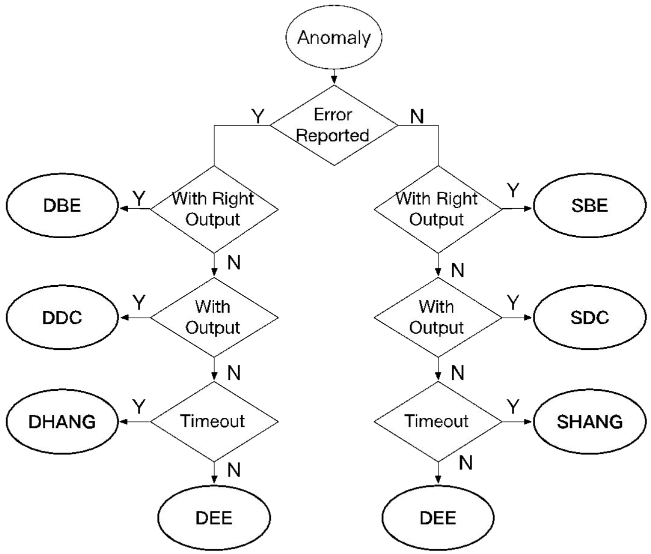
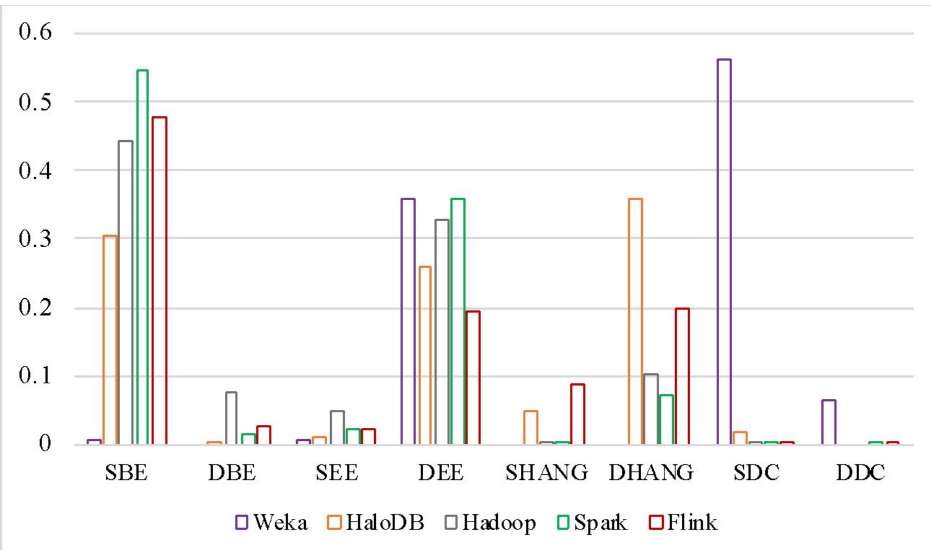


Fig. 7 Different types of anomalies

Evaluation Results

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



The anomaly distribution in target systems

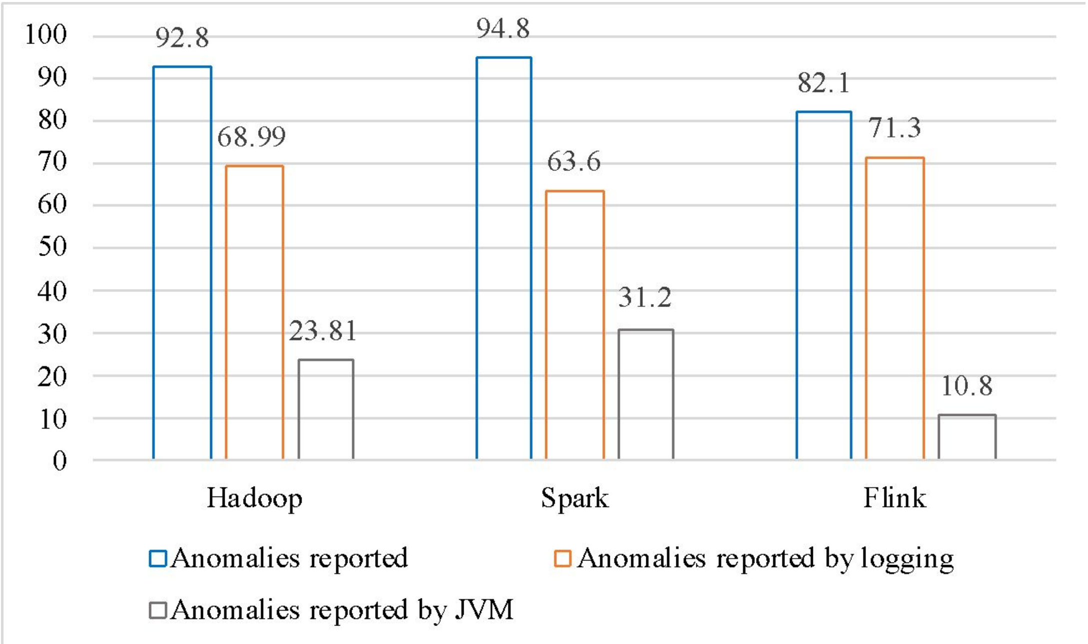
```java
public class DefaultSpeculator extends AbstractService
    implements Speculator {
  protected void serviceStart() throws Exception {
    Runnable speculationBackgroundCore
        = new Runnable() {
          @Override
          public void run() {
            ...
          }
        };
    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**

Evaluation Results

- 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%)



Anomalies reported by distributed systems' error reporting mechanisms

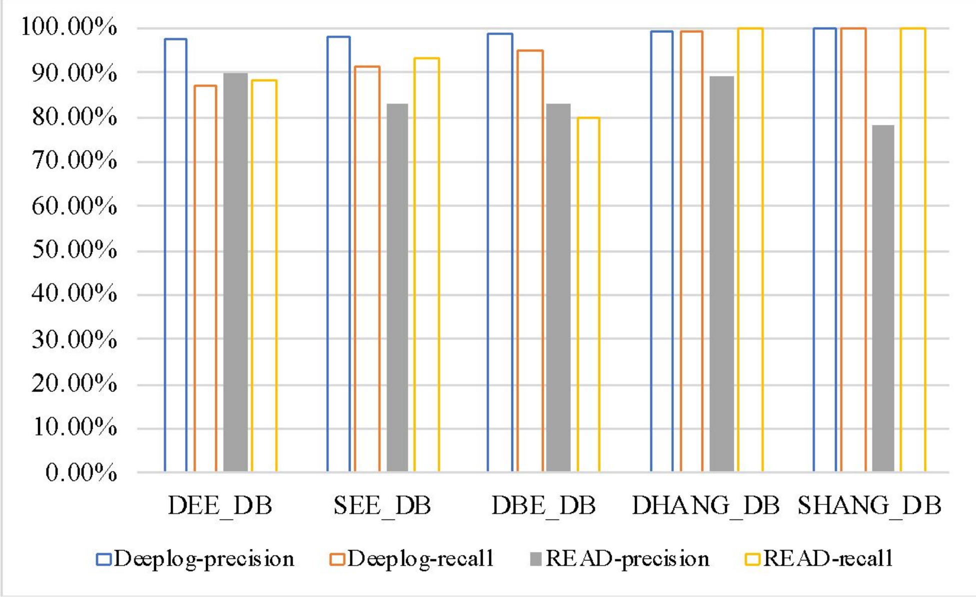
Simple methods are feasible, but get ready for **frequent false alarms**

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

The detection result of three anomaly detectors

detector	precision	recall	f1-measure
Deeplog	99.08%	90.60%	94.65%
MRD	68.85%	79.52%	71.77%
READ	87.13%	90.10%	88.59%



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

The detection latency and locating accuracy of Deeplog and READ

detector	detection latency	locating accuracy at class-level	locating accuracy at component-level
Deeplog	3.42%	29.34%	71.23%
READ	2.11%	-	78.32%

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

[1] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, “Why do internet services fail, and what can be done about it?” in USENIX symposium on internet technologies and systems, vol. 67. Seattle, WA, 2003.

[2] M. Du, F. Li, G. Zheng, and V. Srikumar, “Deeplog: Anomaly detection and diagnosis from system logs through deep learning,” in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017, pp. 1285–1298.

[3] Wei Xu, Ling Huang, Armando Fox, David Paerson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In Proc. ACM Symposium on Operating Systems Principles (SOSP). 117–132.

[4] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, “Long short term memory networks for anomaly detection in time series,” in Proceedings. Presses universitaires de Louvain, 2015, p. 89.

[5] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward, “Effi- cient fault detection and diagnosis in complex software systems with information-theoretic monitoring,” IEEE Transactions on Dependable and Secure Computing, vol. 8, no. 4, pp. 510–522, 2011.

[6] Y. Yang, L. Wang, J. Gu, and Y. Li, “Transparently capturing request execution path for anomaly detection,” 2020.

[7] Y.-Y. M. Chen, A. J. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E.A.Brewer,Path-basedfailureandevolutionmanagement. University of California, Berkeley, 2004.

[8] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, “How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform,” in Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 200–211.