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

Yong Yang†, Yifan Wu†, Karthik Pattabiraman[1‡], Long Wang[1§], Ying Li[1¶]
†School of Software and Microelectronics, Peking University,
‡ University of British Columbia, §IBM TJ Watson Research,
¶National Engineering Center for Software Engineering, Peking University
{yang.yong, yifanwu, li.ying}@pku.edu.cn, karthikp@ece.ubc.ca, wanglo@us.ibm.com

***Abstract*—Anomaly detection in distributed systems has been a fertile research area, and a range of anomaly detectors have been proposed for distributed systems. Unfortunately, there is no systematic quantitative study of the efficacy of different anomaly detectors, which is of great importance to reveal the deficiencies of existing anomaly detectors and shed light on future research directions. In this paper, we investigate how various anomaly detectors behave on anomalies of different types and the reasons for the same, by extensively injecting software faults into three widely-used distributed systems. We use a statement-level fault injection method to observe the anomalies, characterize these anomalies, and analyze the detection results from anomaly detectors of three categories. We find that: (1) the distributed systems' own error reporting mechanisms are able to report most of the anomalies (from 82.1% to 92.8%) but they incur a high false alarm rate of 26.6%. (2) State-of-the-art anomaly detectors are able to detect the existence of anomalies with 99.08% precision and 90.60% recall, but there is still a long way to go to pinpoint the accurate location of the detected anomalies, and (3) Log-based anomaly detection techniques outperform other anomaly detection techniques, but not for all anomaly types.**

***Index Terms*—anomaly detection, distributed systems, fault injection, dependability**

I. Introduction

Computer systems suffer from software faults (i.e., bugs). These faults result in severe system failures, and cost $1.25 billion to $2.5 billion per year [1]. Tellme Networks estimated that 75% of the time in recovering from system failures is spent in just detecting the failures [2]. This situation is getting more severe in modern distributed systems, e.g., Hadoop, Spark, Openstack. The increasing scale and the intrinsic complexity of distributed systems makes it challenging to detect failures and identify their root causes. Prior work [3] has shown that the early detection of the symptoms of failures, i.e. anomalies, can mitigate or even prevent severe failures by taking early action on these detected anomalies. Therefore, there is a compelling need to detect anomalies promptly and precisely in distributed systems.

[1]Corresponding Authors. Part of this work was done when Yong Yang visited UBC in 2019.

There has been significant effort to design smarter anomaly detection approaches for distributed systems. Existing anomaly detection methods can be grouped into three categories: (1) metrics-based [4]–[6], (2) log-based [7]–[9], and (3) trace-based [10], [11]. Various detectors of each category have been proposed and evaluated with different datasets. Unfortunately, these anomaly datasets are either collected by injecting simple failures into the system, which produces unrealistic anomalies, or by collecting real anomalies in a production environment, which comprises anomalies of limited amounts and types. Thus, there is a gap in terms of systematically evaluating anomaly detection systems, and understanding the advantages and disadvantages of the anomaly detectors.

In this paper, we propose a systematic approach to evaluate the efficacy of anomaly detectors. With the proposed approach, we empirically study the merits and limitations of the state-of-the-art anomaly detectors, and explore the reasons behind them, in order to provide guidelines for improving anomaly detection methods. The approach consists of two steps.

In the first step, faults are injected into popular distributed systems with a *realistic* fault injection (FI) method to obtain a wide variety of anomalies for comprehensively evaluating anomaly detectors. To achieve this, we propose SSFI (Statement-level Software FI), a FI method to emulate real software faults at the program statement level. Because all programs can be regarded as a sequence of statements, and software bugs arise at the statement level, *SSFI* is able to inject a wide variety of faults comprising both single and multiple statements. For the software faults involving only a single statement, we analyze the elements for each statement to determine the possible fault types that developers can introduce. For the software faults involving multiple statements, we analyze the possible combinations of statements based on the real bug types in a recent study of distributed systems [12].

In the second step, a data analysis process is conducted on the anomalies gathered from the FI process. The characteristics of anomalies in distributed systems are analyzed and state-of-the-art anomaly detection approaches are evaluated with these anomalies. These anomalies are analyzed from two aspects: 1) the anomaly distribution patterns for distributed

2332-6549/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSRE5003.2020.00015

systems, and 2) the ability of the chosen distributed systems to report these anomalies by themselves. The detection results of anomaly detectors in three categories are compared and dissected in detail in terms of detection precision/recall/F1-measure, detection latency, and the ability to locate the root cause(s) of detected anomalies.

The main findings from the data analysis process include:

- Distributed systems exhibit more frequent Silent Early Exit anomalies (i.e., distributed systems failed to finish processing the requests, and did not emit any observable alarms). These anomalies, whose root causes are challenging for administrators to determine, are mainly caused by incomplete error-handling mechanisms.
- Most anomalies (82.1% - 95.8%) are reported by distributed systems' own error reporting mechanisms, but 26.6% reported anomalies are false alarms. Further, the error reporting mechanisms are able to correctly localize the root causes of more than 90% detected anomalies to the software components. However, when it comes to localizing the root causes to the source code file (e.g., *.class* file, *.cpp* file), which is essential for understanding and solving the problems quickly, less than 20% anomalies are localized correctly.
- Existing anomaly detectors perform well when judging whether there are anomalies (with an F1-measure score of 94.65%), but are weak in locating the root causes of the detected anomalies. Only 29.34% of the detected anomalies are able to localize the root causes to the correct class (i.e., the source code file).
- Log-based anomaly detection methods have better overall detection F1-measure scores than trace-based and metrics-based anomaly detection methods. However, for anomalies of some types, such as Silent Early Exit, trace-based anomaly detection methods have higher recall scores. Further, trace-based anomaly detection methods have the smallest detection latency among the detection methods, which leaves more time for the system to avoid error propagation and further escalation.

The rest of the paper is organized as follows. Section II clarifies the concepts and the motivation of this paper. The FI method is elaborated in Section III. Section IV gives an overview of the evaluation approach. In Section V, the anomalies generated by the FI experiments on targeting systems are analyzed and three anomaly detectors are evaluated. Related work in FI and anomaly detection are discussed in Section VI. Section VII concludes the paper.

II. Background and Motivation

A. Background

A *fault* is the adjudged or hypothesized cause of an error, including software faults and hardware faults [13]. An *error* is the part of the total state of the system that may lead to its subsequent failures. A *failure* is an event that occurs when the delivered service deviates from the correct service [13]. A fault causes an error if it is activated. Otherwise it stays dormant. An error can be tolerated by the systems' error resilience mechanisms. If an error is tolerated, then it goes latent. Otherwise it induces a failure.

Anomalies are the unusual behaviors of the system after a fault is activated. The goal of anomaly detection is to detect anomalies in the system *as early as possible* after an anomaly appears, and locate the root cause of the detected anomalies.

While many anomaly detection methods have been proposed, the way to evaluate anomaly detectors remains primitive. The anomalies used for evaluation are either obtained by injecting simple failures, or from collecting real anomalies from the production environment. The former results in unrealistic anomalies. The latter contains anomalies of a limited amount and type due to the low occurrence rate of anomalies. Furthermore, the evaluation often does not distinguish between anomalies of different types, which may lead to biased detection results, and fail to reveal the blind-spots of anomaly detection methods.

Thus, it is challenging to comprehensively evaluate different anomaly detectors. To address this challenge, there are mainly two requirements. First, the anomaly detectors should be applied to the same platform and datasets. Only in this way, can the difference between different anomaly detectors be shown in an unbiased manner. Second, the anomaly datasets must contain anomalies of diverse types. Only with anomalies of diverse types, can the merits and limitations of different anomaly detectors be comprehensively analyzed.

```
public class NonAggregatingLogHandler {
  public void handle(LogHandlerEvent event) {
    switch (event.getType()) {
      case APPLICATION_STARTED:
        this.dispatcher.getEventHandler().handle(
            new ApplicationEvent(appStartedEvent.getApplicationId(),
                ApplicationEventType.APPLICATION_LOG_HANDLING_INITED));
        break;
      case CONTAINER_FINISHED:
        break;
      case APPLICATION_FINISHED:
        LogDeleterRunnable logDeleter =
            new LogDeleterRunnable(appOwners.remove(appFinishedEvent
                .getApplicationId()), appFinishedEvent.getApplicationId());
          logDeleter.run();
        break;
      default:
        ;
    }
  }
```

Fig. 1. Injecting a fault preserves the process of faults evolving into failures

B. Motivation

FI techniques have been the predominant way to obtain the evaluation datasets for anomaly detection techniques. Unfortunately, most of the existing work on evaluating anomaly detectors injects rather simple failures, which often do not represent the manifestation of software bugs. We illustrate why failure injection is inadequate to evaluate anomaly detectors using two real examples below.

1) **The process of a fault evolving into a failure is missing, which is a critical time window for anomaly detectors to detect anomalies promptly.** An FI example is shown in Fig.

1 from *org.apache.hadoop.yarn.server.nodemanager. containermanager.loghandler.NonAggregatingLogHandler* in Hadoop. When a failure is injected into the process (a *NodeManager* process) running this code snippet by directly killing it, the running job will hang there immediately. On the other hand, when a software fault (e.g., mistakenly missing a *break* between the second and the third switch branch) is injected, it takes around 15 seconds for the *NodeManager* to exit after the fault is activated, and then the running job starts to hang. During this time, a *ClassCastException* is thrown and some container clean-up actions are taken. This would be a golden time window for anomaly detectors to detect the anomaly before the irreversible failure (i.e., the job hangs). Thus, directly injecting the failure short-circuits this process, and prevents the anomaly detection technique from detecting failures.

```
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;
    }
  }
```

Fig. 2. Injecting failures/faults results in different job execution results

2) **Limited coarse-grained failures cannot represent the diversity of anomalies**. For example, consider the code snippet in Fig. 2 from *org.apache.hadoop.conf.Configuration$ DeprecatedKeyInfo* in Hadoop. If the task(YarnChild process) running this code snippet is crashed by directly killing it, the *ApplicationMaster* will restart another YarnChild process successfully. However, if a software fault (e.g., mistakenly using $<=$ instead of using $<$ when comparing *i* and *newKeys.length*) is injected, the *ApplicationMaster* tries several times to restart this task, but none of the attempts are successful as they all raise exceptions. In this case, injecting a software fault results in the failure of the job execution, while directly injecting the failure does not reveal this failure mode. Thus, injecting failures directly may present an overly optimistic view of system failures detected by the anomaly detection method.

Challenge: Thus, we see that injecting failures directly instead of faults can present a misleading view of the system as far as anomaly detection techniques are concerned. As software faults are a leading cause of distributed system failures [3], FI techniques for software faults are gaining momentum. Existing software FI methods are only able to inject few types of software faults (e.g., removing a function call [14], modifying the value of a variable [15], [16]), which fails to generate anomalies of diverse types. *Therefore, we need an FI technique for systematically and comprehensively evaluating anomaly detectors, under realistic software faults.*

III. Fault Injection Methodology

In this section, we first introduce the fault model used by our FI tool, SSFI (Statement-level Software Fault Injection), and then describe the design and implementation of SSFI.

A. Fault Model

Each program can be regarded as a sequence of statements. Software faults are mistakes made by developers on a single statement or a combination of multiple statements. By enumerating the possible mistakes on a single statement and those involving multiple statements, we can emulate all code-level software faults for a system. To determine mistakes that are commonly made by developers on a single statement and a combination of multiple statements, we propose the fault model used in SSFI.

Because statements of different programming languages in the source-code level are different, we adopt the well-known three-address code model [17] to unify the statements of different programming languages. Each statement in three-address code has at most three operands, and statements are not nested. There are 6 different types of statements in three-address code [17] as follows:

- *AssignStmt*, contains a left operand, at most two right operands and at most one binary arithmetic or logical operation. This statement is used to assign the value from the right side to the operand on the left side.
- *GotoStmt*, contains a destination label/address to jump to. This statement is used to change the sequential execution order of statements.
- *IfStmt*, contains at most two operands and at most one logical operation. It is used to check whether a condition is true.
- *InvokeStmt*, contains at most one left operand, a function address and a list of operands as the parameters of the function. It is used to call another function and get the results.
- *ReturnStmt*, contains at most one operand. It is used to return the result to the function caller.
- *SwitchStmt*, contains one operand. While it can be translated and represented with other statements, many three-address codes like Jimple [18] still use it for convenience.

Nowadays, most programming languages provide exception handling and synchronization mechanisms. Therefore, we decided to include two more statements in our analysis:

- *ThrowStmt*, contains an operand(exception). It is used to raise an exception and transfer the control to corresponding exception handlers.
- *SyncStmt*, contains two labels/addresses denote the statement range of synchronization.

We determine the fault model in SSFI by enumerating and then summarizing the possible mistakes that developers may make on a single or multiple statements introduced above.

TABLE I
FAULT MODEL OF SSFI

Fault Type	Fault Source	Statements	Description	Corresponding Bugs in Openstack [12]
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

For example, in an *IfStmt*, programmers may use other logical operations instead of the correct one (such as misusing $<=$ when $<$ is expected, i.e., *CONDITION_BORDER* fault type).

Our fault model is actually based on a recent study of software bugs in Openstack [12]. Table I presents the fault types that SSFI is able to inject. The last column in the table illustrates the mappings of our fault types to the bug types in the study [12], which shows that the majority of software faults in distributed systems are due to minor code problems. In that study, the bugs were classified into 23 minor code problems. Although the bugs are classified at a different level with our work, 21 of the 23 minor code problems can be mapped to our fault model. Only the *Wrong Parameter Type* and *Missing Import* are not covered by the proposed fault model as they will likely be detected at the compilation stage for many programming languages. In these 23 bug types, most of them are caused by minor problems on a single statement. Only *Missing exception handlers*, which can be further classified into two categories, involves multiple statements: (1) the program throws undeclared exceptions, and (2) the program catches the exceptions but simply neglects them. The former (*EXCEPTION_UNCAUGHT* in Table I) involves a combination of *ThrowStmt* and *GotoStmt* and the latter(*EXCEPTION_UNHANDLED* in Table I) involves a combination of *AssignStmt*, *ThrowStmt* and *GotoStmt*.

B. SSFI

SSFI is able to inject 12 different types of software faults in Table I into software systems that can be compiled into Java Bytecode. Note that SSFI is customizable to other fault types as well. We have also made SSFI publicly available [19].

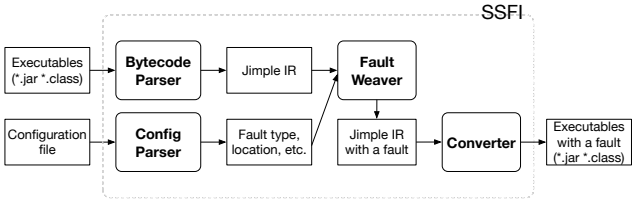

Fig. 3. An overview of SSFI's operation

Software faults can be permanent or transient [20]. Thus SSFI provides two different fault activation modes for each type of fault, namely *always* activation mode and *random* activation mode. In the *always* mode, the injected fault is activated every time the code snippet is run. In the *random* mode, the injected fault is activated when the code snippet is run for the nth time (n is chosen at random if not specified) to simulate transient software faults. SSFI is highly configurable with a simple configuration file specifying which type of fault will be injected to which package/class/method/variables/code blocks with which activation mode. By default, SSFI randomly chooses a combination of the fault type, fault location, and activation mode for the fault.

Fig. 3 shows an overview of SSFI's structure, consisting of 4 components: Bytecode Parser, Config Parser, Fault Weaver and Converter. SSFI takes a bytecode-based runnable file and a configuration file as inputs, and outputs the modified bytecode-based runnable file with the fault specified in the configuration file. Bytecode Parser leverages Soot [18] framework to parse the bytecode into a Jimple intermediate representation (IR), which is a type of three-address code. The Config Parser interprets the configuration file and determines the fault type,

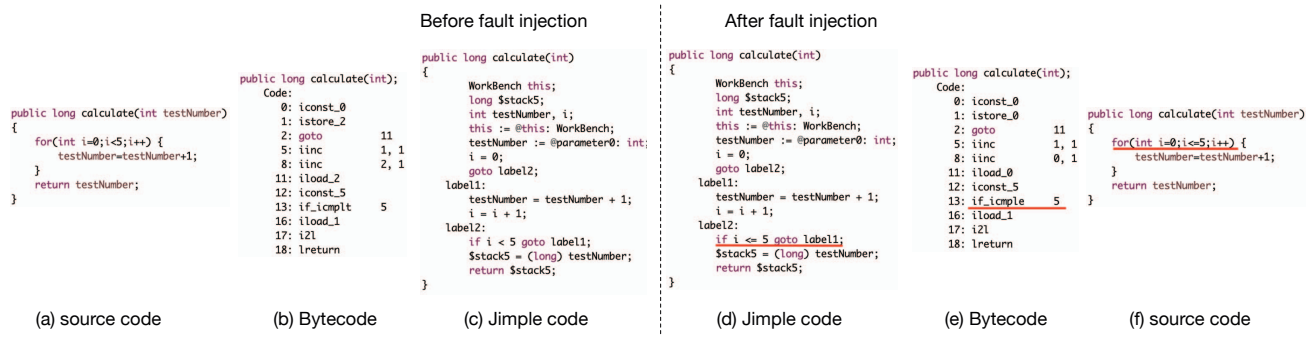

Fig. 4. An example fault injected using SSFI

fault location and fault activation mode for the fault to be injected. The Fault Weaver contains transformation rules for each fault type in Table I, and makes the corresponding changes to the Jimple IR based on the fault type, fault location and fault activation mode. It outputs the Jimple IR with the specified fault being injected. Fault Weaver also scans the program's code automatically to decide which types of faults can be injected (i.e., are valid) into each program statement. Finally, the modified Jimple IR is converted by the Converter, which utilizes Soot to convert Jimple IR to bytecode, into a bytecode-based runnable file with the fault injected.

Fig. 4 presents an example of how SSFI injects a *CONDITION_BORDER* fault into *calculate* method. First, the bytecode snippet shown in Fig. 4(b) is parsed by the Bytecode Parser into the Jimple IR in Fig. 4(c). For ease of understanding, the equivalent source code of Fig. 4(b) is shown in Fig. 4(a) (this is not required by SSFI). Then, the Fault Weaver makes changes to the Jimple IR based on the transformation rules for *CONDITION_BORDER* fault(replacing the $<$ operation with $<=$ operation) and generates the modified Jimple IR with the *CONDITION_BORDER* fault injected in Fig. 4(d). Finally, the Converter employs Soot to convert the modified Jimple IR back into bytecode in Fig. 4(e) with a *CONDITION_BORDER* fault. The equivalent source code of Fig. 4(e) after injection is shown in Fig. 4(f) for ease of understanding (again, this is not required by SSFI).

IV. Evaluation Approach

Fig. 5 shows an overview of the proposed evaluation approach. First, emulated software faults are injected into the targeted systems. The injection information, such as the fault type and the fault location, is recorded. Then the application injected with faults, starts to run, and a workload generator sends service requests to the running application. Meanwhile, a resource monitor and a distributed tracer are set to separately generate the resource usage data and the request execution path of the system processing the requests. The service request output/result and log messages are recorded as well.

The injected fault may or may not be activated to cause an error. The fault activation information (such as the activation time and the component) is also recorded. The error may be reported or ignored by systems' error reporting mechanisms. The error may also propagate to other components and cause anomalies. The anomalies of different systems are classified and characterized based on the recorded injection information, service output and log messages.

Finally, three state-of-the-art anomaly detectors are applied to these anomalies based on the resource usage data, log messages and request execution path for each service request. The detection results are analyzed. By characterizing anomalies and analyzing the detection results from different anomaly detectors, this paper asks three research questions (RQs).

RQ1: What's the pattern of anomalies in distributed systems? By answering this question, we aim to understand whether there is a common pattern for the anomalies in distributed systems, and what the pattern is (if it exists). The answer will help researchers and system administrators obtain an overview of anomalies and comprehend the types of anomalies that are commonly seen in distributed systems. To answer this question, the anomaly patterns of both distributed systems and non-distributed systems are compared and analyzed. The correlation between faults and anomalies is also dissected.

RQ2: To what extent do distributed systems, by themselves, report the anomalies? Distributed systems have their own error reporting mechanism to report the anomalies. By answering this research question, we aim to evaluate the efficacy of these error reporting mechanisms. In this way, we try to figure out why an external anomaly detector is required instead of directly relying on these mechanisms, and what types of anomalies really need an external anomaly detector. These error reporting mechanisms are evaluated in terms of recall, false alarm rate, reporting latency and locating accuracy.

RQ3: To what extent do state-of-the-art anomaly detectors detect anomalies of different types? Three state-of-the-art anomaly detectors, each belonging to one of the three categories, are applied to the anomalies of different types, and evaluated in terms of detection precision/recall/F1-measure, detection latency and locating accuracy. The detection results for each type of anomaly, are compared and analyzed to gain insights into the strengths and drawbacks of the chosen anomaly detectors. By answering this research question, we aim to understand the current pain spots (if any), and shed light on the future directions of improvement in detecting anomalies in distributed systems.

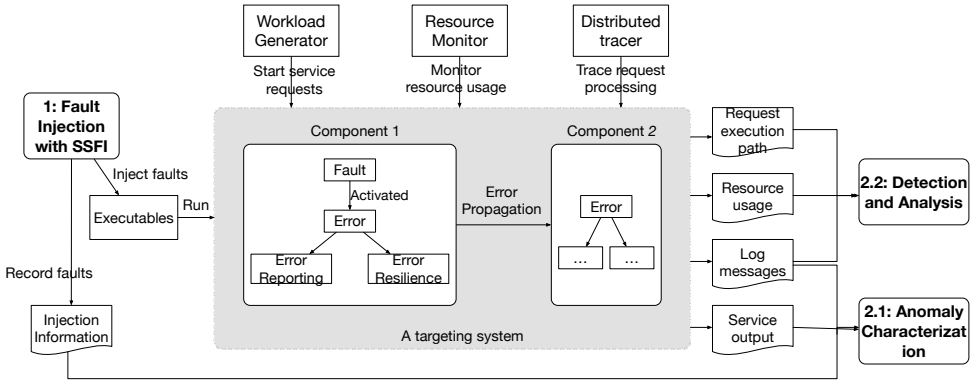

Fig. 5. An overview of the evaluation approach

V. Experimental Evaluation

We first describe the experimental setup, and the types of anomalies observed. We then present our results organized by research question (RQ), followed by the threats to the validity.

A. Experimental Setup

TABLE II
Systems used for evaluation and their workload

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

We use three principles to choose the systems to evaluate: (1) whether it is available as open-source or upon request, (2) whether it is widely used (e.g., based on the stars of the project in Github), and (3) whether it increases the diversity of all chosen systems. Table II lists the chosen systems and their workloads (service requests) in the experiments. We choose both distributed systems and non-distributed systems in order to understand if there are any differences between them. Hadoop, Spark and Flink are the selected distributed systems, and Weka and HaloDB are the non-distributed systems.

For each service request, SSFI randomly chose a combination of fault type, fault location and activation mode to inject a fault into the targeting system. To eliminate the influence from other injected faults, 1) for each service request, only one fault is injected; 2) after a service request finishes, systems are restarted and all the temporary and persistent data are cleaned.

We used Docker Stats as the resource monitor, and REPTrace [21] the distributed tracer in Fig. 5. To determine the timeout threshold for each service request, we ran each request for 100 times without injecting any faults, and chose double the average running time of these requests as the timeout threshold for service requests of this type.

We chose three detectors to be applied to detect anomalies, each in a different category, as follows. We chose the three detectors in our studies because they are state-of-the-art in each of the categories considered, and their source code was available to us (either open source or available upon request).

- Deeplog [9] is a log-based anomaly detector. It takes anomaly detection as a prediction problem which predicts the next log key/template based on the previous n log keys. The prediction model is a Long Short-Term Memory (LSTM) neural network, learning the log key transition probability from the log messages of normal service requests for each component. If the next log key is not in the predicted log key candidates list, an anomaly is detected. We obtained the open-sourced implementation of Deeplog from its authors [22].
- MRD is a metrics-based anomaly detector. MRD leverages a Gated Recurrent Unit (GRU) neural network to train a model to predict the resource consumption sequence of a service request. It takes the resource consumption sequences from normal service requests as the training data. If the similarity between the real resource consumption sequence and the predicted resource consumption sequence is lower than a threshold, an anomaly is detected. MRD is publicly available [23].
- READ is a trace-based anomaly detector. READ builds a core finite state automaton (FSA) and a full FSA for each type of service request from the request execution paths generated by REPTrace [21]. The core FSA represents the states and transitions that a normal service request must go through, while the full FSA represents the states and transitions that a normal service request can go through. In the detection stage, an anomaly is detected either if a state or transition in the core FSA does not appear, or a state or transition that is not in the full FSA appears.

Fig. 6 presents the number of faults injected in each system, and the activated faults. The inactivated faults and corresponding service requests are used as the training data for anomaly detectors to learn a detection model (Section V-E).

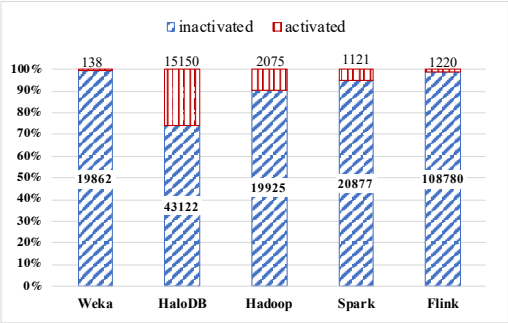

Fig. 6. An overview of the fault injection data

B. Anomaly Model

As discussed in section II, an activated fault may cause a failure or just a latent error. Based on the final result of the anomaly, anomalies are classified into 4 types: Benign, Early Exit, Data Corruption, Hang. For Benign anomalies, the activated faults cause latent errors instead of failures. Based on whether the system's own error reporting mechanisms can report these anomalies, each anomaly type can be further classified into Detected and Silent. In our experiments, the systems report errors when stack traces are printed or there is at least one log message whose level is above *WARN*, such as *ERROR* and *FATAL*. Because we observed that even during the normal execution of service requests, there are a few log messages with *WARN* level, we exclude the *WARN* level. Therefore, there are 8 types of anomalies in total, namely Detected Benign (DBE), Silent Benign (SBE), Detected Early Exit (DEE), Silent Early Exit, Detected Hang (DHANG), Silent Hang (SHANG), Detected Data Corruption (DDC), and Silent Data Corruption (SDC). These are shown in Fig. 7.

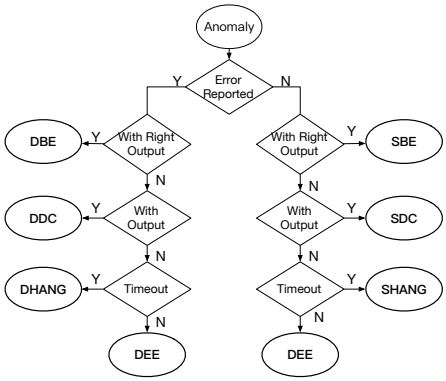

Fig. 7. Different types of anomalies

C. RQ1: What's the pattern of anomalies in distributed systems?

The anomaly distribution of each system is shown in Fig. 8. The Detected Early Exit anomaly is common for all the systems. As can be observed, compared to non-distributed systems, distributed systems are more error-resilient. This is because the proportion of Benign anomalies (including SBE and DBE) are much bigger in distributed systems. Further, the error-resilience mechanisms in distributed systems exhibit higher percentages of Silent Early Exit anomalies, which indicates that the error resilience mechanisms attempted to tolerate the errors (without reporting any problems), but failed.

An example is shown in Fig. 9. The *Speculator* mechanism shown in *org.apache.hadoop.mapreduce.v2.app.speculate. DefaultSpeculator* is an optimization mechanism to replace the tasks that execute much slower than others. The *serviceStart* method declares an *Exception*. However, after injecting an *EXCEPTION_SHORTCIRCUIT* into *serviceStart*, which directly throws an *EXCEPTION* at the beginning of the method, the job execution failed silently in the end. The error handling mechanism caught the declared exception, but neither did it explicitly report that an exception was raised nor did it recover from the injected fault, which results in a Silent Early Exit anomaly in the end. This kind of anomaly is difficult for anomaly detection techniques to determine the root cause, as there is little useful information available in the corresponding error messages.

Therefore, we advocate that developers should explicitly record the error messages when designing the error-handling mechanisms, regardless of whether the error is believed to be tolerated. Further, distributed applications are unlikely to suffer from Data Corruption anomalies (both Detected Data Corruption and Silent Data Corruption) under software faults, which is quite different from other non-distributed applications.

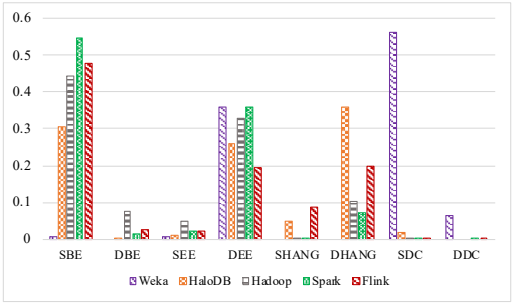

Fig. 8. Anomaly distribution of each system

```
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();
  }
```

Fig. 9. Program fails to tolerate a fault without reporting it

We also analyzed the correlation between faults and anomalies. The results show that there is not a strong correlation

between faults and anomalies, which means that each fault type can result in anomalies of different types. For example, Fig. 10 presents the correlation analysis result in Hadoop. The upper x-axis denotes the total number of faults of each type, and each bar denotes the number of each type of anomalies caused by one type of faults. Silent Benign, Detected Early Exit and Detected Benign are the most common anomalies caused by each type of faults. Other types of anomalies are relatively evenly distributed to different types of faults. Note that we do not consider the correlations between faults of type *EXCEPTION_UNHANDLED* and anomalies, as there are very few cases of activated *EXCEPTION_UNHANDLED* faults.

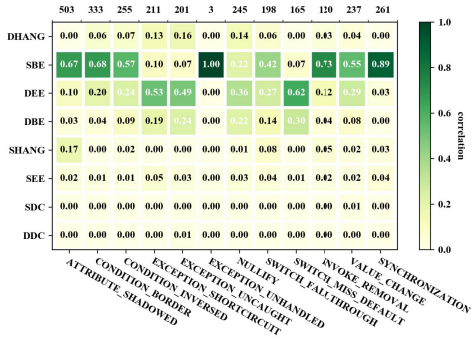

Fig. 10. The correlation between faults and anomalies in Hadoop

D. RQ2: To what extent do distributed systems report the anomalies by themselves?

Fig. 11 demonstrates the ability of distributed systems' error reporting mechanisms to report anomalies. Generally speaking, the error reporting mechanisms are able to report the majority of the anomalies (recall ranging from 82.1% to 92.8%). This explains why simple ways to detect anomalies such as monitoring the key words in logs are still widely used in industry. However, among all the reported anomalies, 26.6% are false alarms. An error report or a detection result is counted as a false alarm if the injected fault is not activated or manifested but an anomaly is reported by either the error reporting mechanisms or anomaly detectors.

The high false alarm rate is a cause for concern for the error reporting mechanisms. Further, a large portion (ranging from 10.8% to 31.2%) of the reported anomalies are reported by JVM's *stderr* and *stdout* instead of by logging tools, such as Log4J. This indicates that diverse log sources should be considered when applying log-based anomaly detection methods, which make the log parsing problem more difficult with more heterogeneous log sources.

The reporting latency and locating accuracy of error reporting mechanisms are summarized in Table III. At a component-level, error reporting mechanisms can accurately localize the root cause (i.e., the injected fault) of the anomalies with 94.2% accuracy in Hadoop and 93.6% accuracy in Spark. However, when it comes to localizing the faults to the source code file (i.e., class-level), the mechanisms have low accuracy.

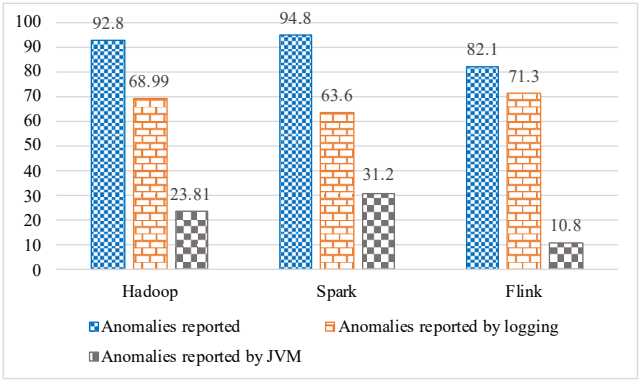

Fig. 11. Anomalies reported by distributed systems' error reporting machanisms

TABLE III
REPORTING LATENCY AND LOCATING ACCURACY OF ERROR REPORTING MECHANISMS

Systems	Reporting Latency	Locating Accuracy in class-level	Locating Accuracy in component-level
Hadoop	5.22s	10.2%	94.2%
Spark	6.34s	12.4%	93.6%
Flink	2.12s	18.2%	-

E. RQ3: To what extent do state-of-the-art anomaly detectors detect anomalies of different types?

In this RQ, we evaluate three anomaly detectors in terms of precision, recall, F1-measure, detection latency, and their fault localization accuracy at class and component levels We first adjusted the hyperparameters and the sizes of training datasets of the MRD and Deeplog detectors to get the best F-measure value. Then we calculate the other evaluation metrics using the values obtained. For READ, we adjusted the parameters and the sizes of training datasets to get the best F1-measure.

To evaluate these anomaly detectors, we first built 7 anomaly datasets of Hadoop from the service requests with the injected faults, including the 6 test datasets in Table IV, and a training dataset consisting of 16,125 normal service requests. In these datasets, each service request has the corresponding information of the injected fault, the log messages, and log keys generated by the system, the request execution path generated by REPtrace, and the resource consumption of the system during the processing of the request. Table IV lists the number of normal service requests (with inactivated faults), the number of anomalous service requests (with activated faults) and the corresponding anomaly types observed.

As shown in Table V, in terms of overall F1-measure (on the *ALL* dataset), Deeplog outperforms the other two anomaly detection methods. MRD has significantly lower precision and recall values compared to the Deeplog and READ because the anomalous service requests do not necessarily have anomalous resource consumption. Fig. 12 shows the detection results of Deeplog and READ on each type of anomaly. As shown in Fig. 12, Deeplog's detection precision outperforms READ on every anomaly type. However, Deeplog's detection recall is lower

TABLE IV
THE PUBLIC DATASET OF HADOOP

dataset name	anomaly type	# of normal service requests	# of anomalous service requests
DEE_DB	DEE	693	693
SEE_DB	SEE	68	68
DBE_DB	DBE	299	299
DHANG_DB	DHANG	249	249
SHANG_DB	SHANG	5	5
ALL	-	1314	1314

than READ on the Detected Early Exit (DEE) and Silent Early Exit (SEE) anomaly types. We manually analyzed the service requests with DEE anomalies detected as well as normal service requests to determine why. We determined that there are no wrong messages in the logs generated by the Log4J, and the error messages reported by JVM fail to be converted to a log template with the current log template extraction method. Thus, Deeplog considers the log key sequence as normal.

The service requests with SEE anomalies may not contain any log key sequences that Deeplog has never seen in the training stage. Therefore, Deeplog fails to detect these anomalies, which decreases its recall. On the other hand, READ detects anomalies based on the request execution paths i.e., Directed Acyclic Graphs (DAGs) that are composed of system call events and their relationships generated by REPTrace, which only depends on whether its execution behavior deviates from the normal at the system-call level. Therefore, READ has higher recall values on these types of anomalies than others.

TABLE V
THE DETECTION RESULTS 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%

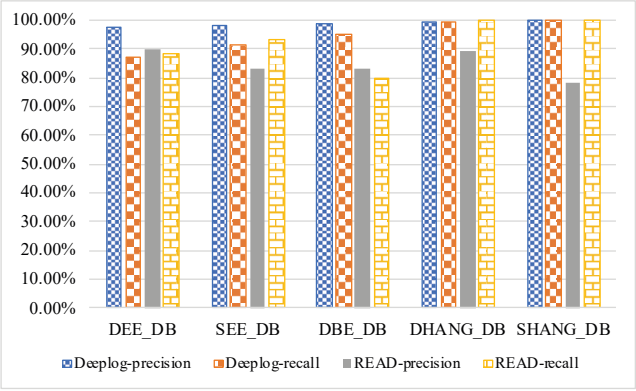

Fig. 12. The detection precision and recall for each anomaly type

Table VI shows the anomaly detection results of Deeplog and READ in terms of detection latency and localization accuracy. MRD is not listed as it is not able to detect anomalies in real time or identify the location of the root cause. If multiple anomalies are detected by Deeplog, the timestamp of the log message with the smallest timestamp is used to calculate the detection latency. i) Compared to Hadoop's own error reporting mechanisms, both Deeplog and READ can detect anomalies with shorter detection latency. READ reduces the detection latency by 59.6%, while Deeplog only reduces it by 34.5%. Thus, READ can detect anomalies much earlier than Deeplog, which allows administrators to take prompt actions to avoid propagation. ii) READ also outperforms Deeplog in accurately localizing the root cause to the correct component. However, READ is not able to localize the root cause to the correct class as it lacks application-level information.

TABLE VI
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%

F. Threats to validity

We use source-code based FI, which has been used in many state-of-the-art FI tools such as SAFE [24], LLFI [25], G-SWFIT [26], etc., in our study. Many studies (e.g., [27]) have found that this method can emulate realistic faults, which justifies the validity of applying FI to mimic real-world anomalies. Table I shows that SSFI is able to cover most of the software faults found in a recent bug study [12]. Our experiments also demonstrate that SSFI generates comprehensive types of anomalies in distributed systems.

The choice of the FI target systems and the anomaly detectors may also affect the experimental results. To mitigate their influence, we chose widely used systems and state-of-the-art detectors in different categories. The source code of both the systems and anomaly detectors is publicly available enabling reproducibility.

VI. RELATED WORK

A. Fault Injection (FI)

FI has been an effective way of software testing and assessing the dependability of software systems [28]. Faults can be coarsely grouped into hardware faults and software faults (also called bugs). Previous research mainly focuses on injecting hardware faults into systems with physical means like heavy-ion radiation [29] or with Software Implemented Fault Injection (SWIFI) techniques [25], [30], [31]. However, with the increasing complexity of software systems, software faults have been recognized as one of the major reasons causing fatal system failures [3]. SWIFI methods can also be used to inject software faults by corrupting states of software systems and the inputs/outputs of software component interfaces. However,

the faults produced by SWIFI methods can emulate software faults only to a limited extent [27].

The most representative way to inject software faults is to directly inject code changes to the systems [32], [33]. The early FI methods inject code changes with orthogonal defect classification (SDC) schema by defining few code change patterns on each fundamental program statement [34]. However, actual software faults are more complex than code changes on single program statement. Duraes et al. [26] study bug fixing data in open-source and commercial software systems, and find that many software faults involve several program statements. They also provide a fault model based on their results. However, a recent study [12] of software bugs in OpenStack finds that the fault model proposed by Duraes et al. [26] cannot cover some frequent bug types in distributed systems. Existing open-sourced FI tools for injecting code changes in Java only cover a few types of faults. J-SWFIT [14] is a FI framework for FI in Java programs, with two fault types. JACA [16], [35] is another FI tool that can change the values of parameters, attributes and returned variables. Byte-monkey [15] is able to inject four types of faults.

B. Anomaly Detection in Distributed Systems

In a running distributed system, there are mainly three types of data available for anomaly detection, namely metrics, log and trace. According to the data that anomaly detectors are based on, anomaly detection methods can be broadly classified into three categories: metrics-based, log-based and trace-based.

Resource usage of systems includes multiple dimensions, such as CPU, network usage recorded by monitoring tools. Threshold-based methods require expertise to set the threshold for each type of resource and cannot deal with cases where there is a huge fluctuation of resource consumption. Machine learning methods have been widely adopted in metrics-based anomaly detectors. Jiang et al. [4] proposed a method to learn the correlation between different metrics at the same time point with an autoregressive model. In another work of Jiang et al. [5], different metrics are grouped into clusters and the entropy of each cluster is updated and monitored. Anomalies are detected when the entropy of a cluster significantly changes. Malhotra et al. [6] employed Long Short Term Memory (LSTM) networks to predict the metrics at a time point based on the metrics values in the previous time windows. If the similarity between the real metrics and predicted metrics exceeds a threshold, an alarm is produced.

Log messages (also called log entries) generated by distributed systems are semi-structured data. Usually a log template (also called a log key) extraction (also called log parsing) method is required before applying log-based anomaly detection methods. Lim et al. [7] use individual message frequency to model systems' behavior and calculates the log templates numbers, log template frequency and their distribution in each time window. Anomalies are judged by users based on the visualization of the individual message frequency. A Finite State Automaton (FSA) is learned from the log sequence [8], in which each state refers to a log template. In the detection stage, an anomaly is found if the transition between two log templates has never appeared before or its transition frequency is under a threshold in the learned FSA. Deeplog [9] trains a prediction model from the normal log key sequences with an LSTM network, and predicts the next log key based on the past log key sequence of a fixed window size. If the next log key is not in the predicted log keys, an anomaly is detected.

Distributed tracing techniques [21], [36], [37] trace how requests are processed in distributed systems, and generate the request execution paths. Their potential for accurately modeling the behavior of distributed systems has led to anomaly detectors based on these request execution paths. Chen et al. [38] build a probabilistic context free grammar model for each type of request from the request execution paths to detect anomalies. Sun et al. [10] build an FSA for all the requests to detect anomalies based on the request execution paths, in which each state denotes a component in the distributed system. Our previous work–READ [11] builds a core FSA and a full FSA for each type of service request from the request execution paths from the REPTrace [21] to detect anomalies.

VII. Conclusion and Future Work

In this paper, we proposed a realistic software fault injection method at the program statement level (SSFI) to understand the efficacy of anomaly detection techniques in distributed systems. By answering three research questions with a systematic approach, we explore the pattern of anomalies in distributed systems. We also analyzed the ability of distributed systems' own error reporting mechanisms to report anomalies without extra anomaly detectors and comprehensively evaluated anomaly detectors of three categories. Our experimental evaluation finds that: 1) Silent Early Exit anomalies are more frequent in distributed systems due to incomplete error-resilience mechanisms; 2) Most anomalies (82.1% - 95.8%) can be reported by distributed systems' own error reporting mechanisms. However, the false alarm rate is also high; 3) Log-based anomaly detection methods have better overall detection accuracy than trace-based and metrics-based anomaly detection methods. Trace-based anomaly detectors are able to detect anomalies faster than the other detectors; and (4) None of the detector types is able to accurately localize the root cause of the anomaly in terms of the source line or class.

As future work, we plan to consider a wider range of distributed systems, and also evaluate more anomaly detectors. We also plan to expand the range of faults considered by SSFI, and apply preliminary coverage analysis of FI to improve the SSFI's performance on a broader fault range. We also want to study if a combination of the log analysis and the trace/execution path analysis can better localize the root cause.

Acknowledgements

This work has been supported by Key-Area Research and Development Program of Guangdong Province (No. 2020B010164003), and Natural Science and Engineering Research Council of Canada (NSERC). We thank the anonymous reviewers for ISSRE'20 for their helpful comments.

REFERENCES

[1] S. Elliot, "Devops and the cost of downtime: Fortune 1000 best practice metrics quantified," *International Data Corporation (IDC)*, 2014.

[2] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based faliure and evolution management," in *Conference on Symposium on Networked Systems Design & Implementation (NSDI)*. USENIX Association, 2004.

[3] 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.

[4] G. Jiang, H. Chen, and K. Yoshihira, "Modeling and tracking of transaction flow dynamics for fault detection in complex systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 4, pp. 312–326, 2006.

[5] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward, "Efficient 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] 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.

[7] C. Lim, N. Singh, and S. Yajnik, "A log mining approach to failure analysis of enterprise telephony systems," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 398–403.

[8] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *2009 ninth IEEE international conference on data mining*. IEEE, 2009, pp. 149–158.

[9] 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.

[10] K. Sun, J. Qiu, Y. Li, Y. Chen, and W. Ji, "A state machine approach for problem detection in large-scale distributed system," in *NOMS 2008-2008 IEEE Network Operations and Management Symposium*. IEEE, 2008, pp. 317–324.

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

[12] 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.

[13] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[14] B. P. Sanches, T. Basso, and R. Moraes, "J-swfit: A java software fault injection tool," in *2011 5th Latin-American Symposium on Dependable Computing*. IEEE, 2011, pp. 106–115.

[15] A. Wilson, "Bytecode-level fault injection for the jvm," https://github.com/mrwilson/byte-monkey.

[16] N. G. Leme, E. Martins, and C. M. Rubira, "A software fault injection pattern system," in *Proceedings of the IX Brazilian Symposium on Fault-Tolerant Computing*, 2001, pp. 99–113.

[17] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 363, 1986.

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 214–224.

[19] L. W. Yong Y, Karthik P, "Ssfi: a statement-level software fault injection method," https://github.com/alexvanc/ssfi/tree/multi-mode.

[20] G. Carrozza, D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Analysis and prediction of mandelbugs in an industrial software system," in *2013 IEEE Sixth international conference on software testing, verification and validation*. IEEE, 2013, pp. 262–271.

[21] Y. Yang, L. Wang, J. Gu, and Y. Li, "Transparently capturing execution path of service/job request processing," in *International Conference on Service-Oriented Computing*. Springer, 2018, pp. 879–887.

[22] Y. Wu, "A pytorch implementation of deeplog's log key anomaly detection model." https://github.com/wuyifan18/DeepLog.

[23] A. Y, "Using gru to predict the resource consumption sequence of service requests," https://github.com/alexvanc/gru_resource_prediction.

[24] D. Cotroneo and R. Natella, "Fault injection for software certification," *IEEE Security & Privacy*, vol. 11, no. 04, pp. 38–45, jul 2013.

[25] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "Llfi: An intermediate code-level fault injection tool for hardware faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 11–16.

[26] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *Ieee transactions on software engineering*, vol. 32, no. 11, pp. 849–867, 2006.

[27] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. IEEE, 2000, pp. 417–426.

[28] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 44, 2016.

[29] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo, "Using heavy-ion radiation to validate fault-handling mechanisms," *IEEE micro*, vol. 14, no. 1, pp. 8–23, 1994.

[30] J. Carreira, H. Madeira, J. G. Silva *et al.*, "Xception: Software fault injection and monitoring in processor functional units," *Dependable Computing and Fault Tolerant Systems*, vol. 10, pp. 245–266, 1998.

[31] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Symplfied: Symbolic program-level fault injection and error detection framework," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 472–481.

[32] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 402–411.

[33] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3. ACM, 1996, pp. 158–171.

[34] W.-I. Kao, R. K. Iyer, and D. Tang, "Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1105–1118, 1993.

[35] C. M. F. R. Nelson G. M. Leme, Eliane Martins, "Jaca software fault injection tool," http://www.ic.unicamp.br/ eliane/JACA.html.

[36] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[37] B.-C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vpath: Precise discovery of request processing paths from black-box observations of thread and network activities." in *USENIX Annual technical conference*, 2009.

[38] Y.-Y. M. Chen, A. J. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer, *Path-based failure and evolution management*. University of California, Berkeley, 2004.