TripleAgent: Monitoring, Perturbation And Failure-obliviousness for Automated Resilience Improvement in Java Applications

Long Zhang\textsuperscript{1} and Martin Monperrus\textsuperscript{1}

\textsuperscript{1}KTH Royal Institute of Technology, Sweden

Abstract—In this paper, we present a novel system for fault injection in production for Java applications. The unique feature of this system is to combine automated monitoring, automated perturbation injection, and automated resilience improvement. The latter is achieved with ideas coming from the failure-oblivious literature. We design and implement the system as agents for the Java virtual machine. We evaluate the system on a real-world application for transferring files with the BitTorrent protocol. Our results shows that it is possible to automatically improve the resilience of Java applications with respect to uncaught exceptions.

Index Terms—fault injection, dynamic analysis, exception-handling, software resilience

I. INTRODUCTION

In modern software, resilience capabilities are engineered through error-handling code, in particular exception handling code in managed languages such as Java and C#. This resilience capability is manually engineered by developers, who write the error-handling code. For example, part of their coding activity is to write try-catch blocks to handle exceptions. The problem is that exception-handling code is notably hard to write and to test [6]. As a result, there often exists corner cases where resilience is not provided by developer-written code. In production, when those corner cases are activated, the software system may simply stop providing its functionality because it crashes after an unhandled exception [25].

In this paper, we research around the idea that developer written exception-handling code can be automatically improved. We aim at designing a technique that automatically finds weaknesses in resilience and improves the exception handling capability of a software system.

In this paper, we present a novel system for automatically assessing and improving resilience for Java applications. Our system, called TRIPLEAGENT, is founded on the combination of automated monitoring, automated perturbation injection, and automated failure-obliviousness [5]. Here, an agent means a component that is dynamically attached to a process (in particular, we consider agents for the Java Virtual Machine). The monitoring agent is responsible for collecting relevant information for resilience analysis, and the perturbation agent injects specific exceptions at systemmatically identified locations. The failure-oblivious agent does failure oblivious computing [5], [15], which is the concept of automatically modifying software for surviving unanticipated errors. In the context of TRIPLEAGENT, the failure-oblivious agent is responsible to inject and evaluate try-catch blocks for exception-handling. If successful failure-oblivious code is identified, it means that a previously fragile method can be automatically improved by being transformed into a failure-oblivious method.

All three agents are controlled by an agent controller, which is a separate module outside the JVM. It analyzes all the monitored data and finally generates a resilience report for developers. The resilience report lists weaknesses in the resilience capabilities and proposes resilience improvements. To the best of our knowledge, TRIPLEAGENT is the first system which actively injects concrete exceptions during execution in order to, at the same time, automatically detect failure-oblivious methods.

We evaluate TRIPLEAGENT on a real-world application for transferring files. The application implements the BitTorrent protocol in 6.5K Java lines of code. We consider a realistic workload of downloading a large file from the Internet. In total, TRIPLEAGENT performs 379 perturbation injection experiments and evaluates 85 perturbation points in the application. Among the perturbation points 61 of them have critical influence on normal execution behaviour. After resilience improvement, we observe that exceptions which are thrown from 5(8\%) perturbation points do not lead to a failure any more, with the help of failure-oblivious methods added by TRIPLEAGENT. Considering that our research subject is a mature open-source system, this can be considered as a milestone for automated resilience improvement.

To sum up, our contributions are the following.

- The concept of joint usage of fault injection and failure-obliviousness to evaluate and improve resilience. We propose an algorithm that realizes this concept.
- A system called TRIPLEAGENT that combines perturbation injection and failure-obliviousness in Java, implemented as agent for the Java Virtual Machine. The system is publicly-available for future research in this area.
- An empirical evaluation of TRIPLEAGENT on a real-world Java application of 6.5K lines of code. By performing 379 executions under a realistic, production-like workload, it shows TRIPLEAGENT’s effectiveness for detecting weaknesses and improving software resilience.
II. BACKGROUND

TRIPLEAGENT is founded on techniques from the fault injection and the failure-oblivious computing literature. This section presents a basic introduction to the core concepts.

A. Fault Injection

Fault injection is a popular research topic in software testing and dependability evaluation. Fault injection techniques actively inject different kinds of errors into a target system in order to assess its dependability [22], [27]. This can happen in several phases: 1) during unit testing, fault injection generates more test cases so that corner cases are detected, and the coverage of testing is improved. 2) during integration, fault injection can trigger different failure scenarios so that developers gain more confidence in their system’s error-handling design.

The kinds of failures that can be injected vary depending on the considered dependability aspect. For example, injecting processor errors or hardware-based errors is often done for evaluating the dependability of operating systems [9], [16]. Injecting an exception in a certain method is useful for validating an application’s error-handling capability [26].

B. Failure-oblivious Computing

In order to improve the robustness of an application, different techniques can be applied to prevent the application from crashing when an error occurs [15]. Failure-oblivious computing [5] is one of these approaches to overcome software failures at runtime. The main idea of failure-obliviousness is to ignore certain failures in a principled way. For example, if a method tries to write data into an invalid memory address, with failure-oblivious computing, the writing operation is ignored. It has been shown that failure-oblivious computing is able to increase availability [21], e.g. to serve requests to more users despite errors.

In this paper, we use the concept of failure-oblivious computing for the Java programming language. In Java, there is no invalid memory addresses, but the biggest reason for crashing are unhandled exceptions. In this paper, we study failure-oblivious computing with respect to unhandled exceptions.

III. DESIGN OF TRIPLEAGENT

This section presents the design of TRIPLEAGENT, including its architecture, input and output, and main algorithm. At the end, we explain how we implement the concept of TRIPLEAGENT in the context.

A. Definitions

a) Exception: All major programming languages provide a way to signal problems through so-called exceptions. Some statically typed languages such as C#, OCaml and Java, exceptions are typed, and some of them are statically verified at compile-time (aka checked exceptions in Java). For these checked exceptions, developers need to either handle them at the call site or explicitly declare them in the method signature (with the keyword throws in Java [8]).

b) Perturbation point: In this paper, a “perturbation point” is a precise location in code where a fault can be injected. In TRIPLEAGENT, the considered perturbations are injected exceptions, which means that a perturbation point is a statement that throws an exception.

c) Perturbation search space: We define the “perturbation search space” as the Cartesian product of all possible perturbation points and all injection timings with respect to a workload [4].

d) Exception handling point: When an exception is handled in a catch block, the catch block is called the “exception handling point” for this exception. It is defined as a tuple <exception source, exception type, catch location>. In this paper, we make a distinction between normal exception handling points and failure-oblivious exception handling points. The former refers to manually written code while the latter refers to automatically injected code.

e) Failure-oblivious method: A method m is said to be failure-oblivious with respect to a pair <exception source s, exception type t>. This is noted < s, t > → m (here → denotes “thrown at s and caught at m”), it means that: when an exception is thrown at s, it is possible to handle it in m when m is higher in the call stack. Here handling the exception means that the observable behaviour is acceptable.

f) Fault injection experiment: Given an application a and a workload w, every execution of a under w is called an experiment when a fault is injected. In this paper, TRIPLEAGENT conducts a series of experiments to gather enough data in order to evaluate an application’s error-handling capability.

Example. Let us assume an invocation chain across three methods: m2 → m1 → m0. Method m0 can throw IOException, and the developers write a try-catch block to handle it in m2 (the levels up in the stack). Consequently, m2 is the default exception handling point for this exception. If the exception is caught and silenced in method m1 and it does not change the behaviour of the execution, method m1 is a failure-oblivious method for <m0, IOException>. If an IOException is thrown from m1, and successfully caught in m2, the perturbation point in m1 is said to be immunized by the catch block in m2. Assume that under a given workload, those three methods are executed three times, consequently, the perturbation search space consists of $3 \times 2 = 6$ possible exception injections.

B. Goals

TRIPLEAGENT aims at improving the exception-handling of Java applications. The main goals of TRIPLEAGENT are: 1) give developers feedback about the effectiveness of their exception-handling design in production 2) automatically identify improvements of exception handling.

The former is about detecting the weakness points of the system under consideration and the latter is about finding new failure-oblivious methods that improve the system’s resilience.

C. Input to TRIPLEAGENT

TRIPLEAGENT takes arbitrary software written in Java as input. We note that no manual change is required from the
D. Output for the developer

The output of TRIPLEAGENT is a report for the developer. The report gives three pieces of information: 1) the weak resilience points; 2) the potential failure-oblivious methods, i.e. the potential resilience improvements; 3) insights about the application behavior based on metrics collected during the injection experiments.

Regarding the weak resilience points, TRIPLEAGENT classifies all perturbation points into three categories as follows:

- Fragile points: injecting one exception at this point results in the application crashing or freezing.
- Sensitive points: injecting one exception at this point does not influence the application. However, if one injects more exceptions, say $n$ exceptions, the application fails to fulfill expected functions (for a given $n$).
- Immunized points: No matter how many exceptions TRIPLEAGENT injects at this point, the application still behaves acceptably.

Let us consider a concrete example and assume a program used to download files. In order to conduct a series of $n$ injection experiment, we attach TRIPLEAGENT to the program and we run the program $n$ times to download a file. Eventually, TRIPLEAGENT would tell the developers of the program: how injecting exceptions influences the file downloading procedure and how many failure-oblivious methods have been identified in the program.

E. Core Algorithm

Algorithm 1 describes the whole procedure of TRIPLEAGENT to detect failure-oblivious methods. There are 3 steps in the algorithm: 1) Execute the application normally, in order to monitor and record the application’s normal behaviour. 2) Conduct a series of perturbation-only experiments, by triggering different exceptions and analyzing normal exception handling points. 3) For every potential failure-oblivious methods, TRIPLEAGENT injects exceptions in order to validate its effectiveness: if the exception is correctly handled by this failure-oblivious method, this point is marked as failure-oblivious. By correctly handled we mean that the application behavior remains the same from an end-user perspective.

F. Architecture of TRIPLEAGENT

Figure 1 presents the architecture of TRIPLEAGENT. TRIPLEAGENT considers a Java application in a JVM, such as a standard 3-tier web application, or a Java micro-service.

When an application is loaded into the JVM, TRIPLEAGENT attaches to it three different agents: a monitoring agent, a perturbation agent and a failure-oblivious agent. The monitoring agent is responsible for collecting the information needed by TRIPLEAGENT to evaluate the system’s resilience capabilities. The perturbation agent injects failures into the application in order to identify and validate failure oblivious points. The failure-oblivious agent instruments the code with potential failure-oblivious methods.

All the agents are controlled by a controller which makes two kinds of decision: 1) when, where and what kind of exception are to be injected, 2) whether a specific failure-oblivious method should be switched on. Finally, the controller generates the feedback to the developer based on data gathered for some fixed period of time.

```
Algorithm 1 Auto-detection of Failure-oblivious Methods
Input:  
    An application $A$;
    A repeatable workload for this application $W$;
Output:  
    $R$, a set of failure-oblivious methods;
1:    $P$, a set of perturbation points
2:    $Q$: Set of potential failure-oblivious methods
3:    // Gather normal behaviour
4:    Execute the application normally under $W$;
5:    for each method $m$ loaded into JVM do
6:      if $m$ throws a checked exception $e$ then
7:          $P \leftarrow < m, e >$;
8:        end if
9:    end for
10:   // Find potential failure-oblivious methods to be assessed
11:  $r$: The chance of injecting the exception
12:  $Q$: Set of potential failure oblivious methods
13:  for each point $< m, e > \in P$ do
14:    Execute the application under the following condition:
15:    if $m$ is executed AND ifPerturbation(r) then
16:        Inject the exception $e$ in the beginning of $m$;
17:        $s \leftarrow$ the call stack
18:        for each method $n$ in $s$ do
19:            $Q \leftarrow (< m, e >\mapsto n)$;
20:        end for
21:    end if
22:  end for
23:   // Assess all potential failure-oblivious methods
24:  for each perturbation point $< m, e > \in P$ do
25:    for $< m, e >\mapsto n$ in $Q$ do
26:      Execute the application;
27:      if $m$ is executed AND ifPerturbation(r) then
28:        Inject the exception $e$ at the beginning of $m$;
29:        Catch and silence the exception $e$ in $n$;
30:        Observe the application behaviour;
31:        if the behaviour is acceptable then
32:            $R \leftarrow (< m, e >\mapsto n)$;
33:          end if
34:      end if
35:    end for
36:  end for
37:  return $R$;
```
1) Monitoring agent: In order to study the influence of perturbations and evaluate all possible failure-oblivious methods on a system, it is necessary to collect different kinds of monitoring information. For this, we propose to use a monitoring agent that is attached to the runtime process. Our monitoring agent works as follows. For each method in code loaded in the JVM, the agent collects static and dynamic information. The static information is: 1) its position in the code, 2) whether it declares checked exceptions to be thrown (throws keyword in Java, the information is available through reflection)

The collected dynamic information is: 3) the number of method executions over a period of time, 4) each time an exception is caught, the agent collects the stack information, including the stack distance between the method raising the exception and the method catching it. This includes exception caught by failure-oblivious catch blocks (as defined in Section II).

The TRIPLEAGENT monitoring agent also collects the following generic information:
- The set of classes that have been loaded so far into the JVM.
- Whether a service has exited normally or crashed due to an unhandled exception.
- A set of operating system metrics including CPU usage, memory usage, and peak thread number.
- The application logs.

2) Perturbation agent: The perturbation agent is responsible for injecting specific perturbations at a specific point in time. The perturbation order comes from the agent controller. In our implementation, the type of a perturbation is a thrown exception. However, developers and users of TRIPLEAGENT can implement their own perturbation strategy. The perturbation agent detects every method with a throws keyword and attaches itself into this method by rewriting the bytecode. When the perturbation mode is switched on for this method, the agent throws the declared exception at the beginning of the method, as first executed statement, effectively resulting in skipping the execution of the whole method body.

This perturbation strategy enables it to have a tractable perturbation search space (defined in Section III-A): there is $n$ possible perturbations by method, where $n$ is the number of declared exceptions in the signature, usually small ($n = 1$ in 78% of the cases). This perturbation strategy is also interesting because it represents a worst-case: none of the code in this method is executed.

Listing 1. The Perturbation Strategy in TRIPLEAGENT

Listing 1 gives an example about how this perturbation agent works. When a method like exampleMethod() throws multiple exceptions, the corresponding number of perturbation points is automatically injected with code transformation. The perturbation agent controls every perturbation point separately. When a specific point is activated, it throws an exception in the beginning of the method.

3) Failure-oblivious agent: The failure-oblivious agent attaches failure-oblivious capability at specific positions in the code. For reasoning about resilience with respect to uncaught exceptions, the failure-oblivious agent injects a try-catch wrapper in all methods. Basically, the whole method body is wrapped with a try-catch block which handles all types of exceptions (Exception in Java). By default, the catch block simply throws again the exception. When the failure-oblivious mode is activated, the catch block silences the exception and prevents it (coming this method or from code called from this method) from propagating. When an exception is silenced by the inject catch block, there are three possible outcomes: 1) the application runs normally; 2) the application runs in a gracefully degraded mode; 3) the application crashes.
Listing 2. The Failure-oblivious Strategy in TRIPLEAGENT

Listing 2 illustrates how this is done. In `Class2` method `callExampleMethod` the exampleMethod mentioned in Listing 1 is invoked. The failure-oblivious agent detects it as a possible failure-oblivious method. So the whole method body is wrapped with a try catch block. When agent controller activates this failure-oblivious method, it silences all exceptions come from the method. Otherwise it throws the caught exception so that it is propagated as usual.

4) Agents controller: The agent controller is responsible for conducting a series of controlled experiments (see Section III-A). It controls every agent and gathers all the information to analyze the system resilience. Additionally, the controller also takes configurations decided by developers. For example, developers can define a filter to focus on resilience improvement for a specific package.

G. Implementation

The monitoring agent is implemented with the help of JVM Tool Interface (JVMTI) ¹. The perturbation agent and failure-oblivious agent are implemented as JVM agents, using the ASM library for binary code transformation ². The agents controller is a standalone service, it communicates with the JVM and the agents through local files.

For sake of open-science, the code is made publicly available at https://github.com/KTH/chaos-engineering-research.

IV. EVALUATION

For our evaluation, we apply TRIPLEAGENT to a real-world Java project called T Torrent. T Torrent is a peer-to-peer file downloading tool based on the BitTorrent protocol.

We choose T Torrent because it is representative of a Java software packages of medium size and complexity, while being open-source. In addition, we note that: 1) T Torrent is popular: it has more than 1,000 stars on GitHub. 2) Its application domain, file downloading is appropriate for generating realistic production traffic.

A. Research Questions

Our evaluation on T Torrent answers the following research questions:

- What is the effectiveness of TRIPLEAGENT to evaluate an application’s resilience with respect to exception handling?
- Does TRIPLEAGENT identify other locations in code that provide alternative resilience capabilities?
- Is TRIPLEAGENT able to automatically improve resilience?

¹See https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html
²See http://asm.ow2.org

B. Experiment Protocol

We apply Algorithm 1 to T Torrent. The application A is T Torrent version 1.5.

The workload W for T Torrent consists of downloading a large file (ubuntu-14.04.5-server-i386.iso, a Linux distribution installer of 623.9MB). Since BitTorrent is peer-to-peer protocol this workload involves other machines on the internet which serve (aka “seed”) the file. To that extent, the workload is a production one.

We configure TRIPLEAGENT to only consider classes and methods in package com/turn/ttorrent for fault injection.

Finally, we perform an injection experiment with multiple injected exceptions, as described in Section III-E.

C. Experimental Results

Per Algorithm 1, the first step of TRIPLEAGENT is to execute T Torrent normally and to monitor all possible perturbation points. It detects 85 points in total within the package com/turn/ttorrent.

Then, TRIPLEAGENT performs two series of experiments: 1) it injects one exception per perturbation point and compares the behaviour between these experiments and the normal execution 2) it injects multiple exceptions per perturbation point and also compares the behaviour.

As a result, all perturbation points get classified in the 3 categories defined in Section III-D: there are 38 fragile points, 23 sensitive points and 24 immunized points. Figure 2 shows the distribution of these perturbation points, which are used as a base line for the following experiments.

Fig. 2. Results of perturbation-only experiments on T Torrent

The next step of TRIPLEAGENT’s main algorithm is to compute and assess the possible failure-oblivious methods. As explained in Section III-F3, for a perturbation point, a set of failure-oblivious methods is identified. In our experiment, TRIPLEAGENT detects 104 possible failure-oblivious methods for the 85 perturbation points. The minimum, median and maximum number of potential failure-oblivious methods per perturbation point is respectively 0, 1.2, 6.

Then, 208 executions are made to assess the failure-obliviousness of the candidate points. The result is described in Figure 3. Bars marked as perturbation mode are the same with Figure 2 as a baseline. The others marked as failure-oblivious mode show the improvement of resilience. TRIPLEAGENT successfully transforms 1 fragile point into a sensitive one, 1 fragile point and 4 sensitive points into immunized ones.
Failure-oblivious improvement

Listing 3

```java
try
// a failure oblivious try-catch can be

this.raf.close(); // IOException may occur
// a failure oblivious try-catch can be
```

shows method

shows a failure-oblivious method

Listing 5

```java
IOException{
int bytes = 0;
...}
if (bytes < requested) {
    throw new IOException("...");
}
return bytes;
}
```

Listing 4. IOException in FileCollectionStorage/write

```java
Listing 4. IOException in FileCollectionStorage/write

2) Improving Resilience under a High Number of Exceptions: Listing 4 shows method write in class FileCollectionStorage. This method is executed 1620 times during a normal execution. If TRIPLEAGENT injects one single exception in this method, the application still downloads the file correctly. However, when the perturbation agent injects more than 30 exceptions, the application gets stalled.

TRIPLEAGENT detects and evaluates 1 single alternative failure-oblivious method in the stack in method Piece/record. With a new execution, TRIPLEAGENT observes that if the exception is silenced in method Piece/record, the application downloads the file successfully, no matter how many exceptions are thrown.

In this case, TRIPLEAGENT succeeds in detecting a failure-oblivious method that provides better resilience compared to the normal error-handling code written by the developer.

Listing 5. IOException in SharedTorrent/validatePieces

```java
Listing 5. IOException in SharedTorrent/validatePieces

3) Improving Resilience from Crashing to Robust: Let now us consider Listing 5. With one perturbation-only experiment in method validatePieces, TRIPLEAGENT identifies that an exception IOException thrown in this method crashes the whole process.

Then, TRIPLEAGENT performs 1 failure-oblivious experiment for this very method. It basically assess whether not throwing an exception is valid option. Indeed, TRIPLEAGENT observes that when a silencing catch block is automatically injected there, the application does not crash anymore, and even better, the resulting behavior is correct (the file is

D. Case Studies

In the following we detail 3 case studies where resilience is improved.

Listing 3. IOException in FileStorage/close

```java
void close() throws IOException {
...
this.raf.close(); // IOException may occur
// a failure oblivious try-catch can be
```

1) Failure-oblivious Method as Alternative to Normal Resilience: First, Listing 3 shows a failure-oblivious method with respect to exception IOException. This method is executed only 1 time during normal download of the file. Under perturbation, TRIPLEAGENT identifies that if one single exception is thrown from this method, there exists in the stack an exception handler such that allows downloading the file correctly.

By activating a failure-oblivious try-catch block in this very method (i.e. if the method body is wrapped with a try-catch block and silence its exception), the T Torrent still succeeds in downloading the file. It means that TRIPLEAGENT successfully detects an alternative method in the stack that provides the same resilience.

Under a realistic workload, TRIPLEAGENT performs 379 experiments to evaluate 85 perturbation points spread over 6.5kLOC. TRIPLEAGENT identifies 38 fragile points, 23 sensitive points and 24 immunized points. After analyzing all methods with respect to failure-obliviousness, TRIPLEAGENT locates 11 failure-oblivious methods, which represents an automated resilience improvement. Overall, as shown in Figure 3, TRIPLEAGENT decreases fragility by 5% and improves resilience by 21%.

Table 1 presents a sample of perturbation points and their corresponding failure-oblivious points. Every row describes 1) a perturbation point (the class name, method name), the thrown exception type, the default exception handler written by developers; 2) the failure-oblivious improvement (failure oblivious method and concrete change of the perturbation point’s category).

| Perturbation Point | Exception Type | Failure-Oblivious Method | Change |
|--------------------|----------------|--------------------------|--------|
| FileStorage/close  | IOException    | False                    | -      |
| SharedTorrent/validatePieces | IOException | True | implemented try-catch block |

Fig. 3. Resilience Improvement on T Torrent. There are fewer sensitive and fragile points, and more immunized ones.

Quantitative insights from the T Torrent experiment

Under a realistic workload, TRIPLEAGENT performs 379 experiments to evaluate 85 perturbation points spread over 6.5kLOC. TRIPLEAGENT identifies 38 fragile points, 23 sensitive points and 24 immunized points. After analyzing all methods with respect to failure-obliviousness, TRIPLEAGENT locates 11 failure-oblivious methods, which represents an automated resilience improvement. Overall, as shown in Figure 3, TRIPLEAGENT decreases fragility by 5% and improves resilience by 21%.

D. Case Studies

In the following we detail 3 case studies where resilience is improved.

Listing 3. IOException in FileStorage/close

```java
void close() throws IOException {
...
this.raf.close(); // IOException may occur
// a failure oblivious try-catch can be
```

1) Failure-oblivious Method as Alternative to Normal Resilience: First, Listing 3 shows a failure-oblivious method with respect to exception IOException. This method is executed only 1 time during normal download of the file. Under perturbation, TRIPLEAGENT identifies that if one single exception is thrown from this method, there exists in the stack an exception handler such that allows downloading the file correctly.

By activating a failure-oblivious try-catch block in this very method (i.e. if the method body is wrapped with a try-catch block and silence its exception), the T Torrent still succeeds in downloading the file. It means that TRIPLEAGENT successfully detects an alternative method in the stack that provides the same resilience.

Listing 4. IOException in FileCollectionStorage/write

```java
Listing 4. IOException in FileCollectionStorage/write

2) Improving Resilience under a High Number of Exceptions: Listing 4 shows method write in class FileCollectionStorage. This method is executed 1620 times during a normal execution. If TRIPLEAGENT injects one single exception in this method, the application still downloads the file correctly. However, when the perturbation agent injects more than 30 exceptions, the application gets stalled.

TRIPLEAGENT detects and evaluates 1 single alternative failure-oblivious method in the stack in method Piece/record. With a new execution, TRIPLEAGENT observes that if the exception is silenced in method Piece/record, the application downloads the file successfully, no matter how many exceptions are thrown.

In this case, TRIPLEAGENT succeeds in detecting a failure-oblivious method that provides better resilience compared to the normal error-handling code written by the developer.

Listing 5. IOException in SharedTorrent/validatePieces

```java
Listing 5. IOException in SharedTorrent/validatePieces

3) Improving Resilience from Crashing to Robust: Let now us consider Listing 5. With one perturbation-only experiment in method validatePieces, TRIPLEAGENT identifies that an exception IOException thrown in this method crashes the whole process.

Then, TRIPLEAGENT performs 1 failure-oblivious experiment for this very method. It basically assess whether not throwing an exception is valid option. Indeed, TRIPLEAGENT observes that when a silencing catch block is automatically injected there, the application does not crash anymore, and even better, the resulting behavior is correct (the file is

D. Case Studies

In the following we detail 3 case studies where resilience is improved.
correctly downloaded, its content is the expected one). In this case, TRIPLEAGENT has automatically transformed a crashing exception into acceptable behavior.

| Perturbation Point | Exception | Default Handler Point | Failure-oblivious Method | Category (before - after) |
|--------------------|-----------|-----------------------|--------------------------|--------------------------|
| FileCollectionStorage/write | IOException | SharingPeer/handleMessage | Piece/record | sensitive - immunized |
| SharedTorrent/handlePieceCompleted | IOException | SharingPeer/handleMessage | SharedTorrent/handlePieceCompleted | sensitive - immunized |
| Piece/record | IOException | SharingPeer/handleMessage | Piece/record | sensitive - immunized |
| FileStorage/write | IOException | SharingPeer/handleMessage | SharedTorrent/handlePieceCompleted | sensitive - immunized |
| SharedTorrent/validatePieces | IOException | SharedTorrent/validatePieces | FileCollectionStorage/close | immunized - immunized |
| FileStorage/close | IOException | SharedTorrent/validatePieces | FileCollectionStorage/close | immunized - immunized |
| FileStorage/close | IOException | SharedTorrent/validatePieces | FileCollectionStorage/close | immunized - immunized |
| HTTPTrackerClient/announce | AnnounceException | Announce/exception | HTTPTrackerClient/announce | immunized - immunized |
| ConnectionHandler/close | IOException | Client/runt | ConnectionHandler/close | immunized - immunized |
| Announcement/promoteCurrentTrackerClient | IOException | Announcement/promoteCurrentTrackerClient | Announcement/promoteCurrentTrackerClient | immunized - immunized |
| FileCollectionStorage/close | IOException | SharedTorrent/close | FileCollectionStorage/close | immunized - immunized |

TABLE 1

SAMPLE OF PERTURBATION POINTS AND THE CORRESPONDING FAILURE-OBVIOUS METHODS

V. RELATED WORK

In this section, we discuss the related work along three aspects.

1) Fault injection: Fault injection is a popular and widely-researched topic in the area of software dependability. In the 1990s, the research was mostly about hardware implemented fault injection tools. For example, Madeira et al. [13] invented RIFLE, a pin-level fault injector to generate processor errors. Next, more software-based fault injection tools were invented. Kanawati et al. [10] proposed FERRARI, a tool for the emulation of hardware faults as well as control flow errors. Han et al. [9] designed DOCTOR, a tool for injecting hardware failures and network communication failures. Wei et al. [24] built a software-based hardware faults injector called LLFI, and quantitatively compared the accuracy of fault-injection with assembly code level injector PINFI. Lee et al. [12] presented SFIDA, a tool to test the dependability of distributed applications on the Linux platform. Kao et al. [11] invented “FINE”, a fault injection and monitoring tool to inject both hardware-induced software errors and software faults. Since the target of the study is the UNIX kernel, the assembly code of the kernel is also needed. Kouwe and Tanenbaum [23] proposed HSFI, a fault injection tool that injects faults with all context information from the source level and applies fault injection decisions efficiently on the binary. The novelty of TRIPLEAGENT is that it is designed to inject application-level exceptions (and not hardware faults). TRIPLEAGENT is able to give developers concrete insights at the source code level about their exception-handling implementation.

2) Failure-oblivious computing: Rinard et al. [21] invented a safe compiler for C to enable servers to execute despite memory errors. Perkins et al. [17] proposed ClearView, a system for automatically patching errors in deployed software. It observes values of registers and memory locations and tries to detect any violations of invariants at this level. Qin et al. [18] invented Rx, which enables the program to rollback to a recent checkpoint upon a software failure, and then to re-execute in a modified environment. Rigger et al. [19] presented an approach that allows C programmers to perform explicit sanity checks and to react according to invalid arguments or states. They also designed a C dialect called Lenient C [20] that checks undefined behaviours in the C standard including memory management, pointer operations and arithmetic operations. None of these tools combine fault injection and failure-oblivious computing together as we do in TRIPLEAGENT. They do not actively inject failures into the system, nor do they conduct application-level analysis to detect valuable failure-oblivious positions.

3) Exception analysis: Byeong-Mo et al. [2] gave a comprehensive review on exception analysis. Magiel Bruntink et al. [1] proposed a characterization and evaluation method to statically discover faults in idiom-based exception handling. Fu and Ryder [7] described a static analysis method for exception chains in Java. Alexandre L. Martins et al. [14] presented VerifyEx to test Java exceptions by inserting exceptions at the beginning of try blocks. Zhang and Elbaum [26] presented an approach that amplifies test to validate exception handling. Cornu et al. [3] proposed a classification of try-catch blocks at testing time. Those tools rely on test suites to analyze resilience with respect to error-handling. On the contrary, TRIPLEAGENT analyzes the system behaviour based on end-user traffic and usage.

VI. CONCLUSION

In this paper, we have presented TRIPLEAGENT, a system which combines automated monitoring, automated perturbation injection and automated resilience improvement. By evaluating TRIPLEAGENT on a real-world Java application, we have shown that it is able to detect weaknesses in exception-handling of Java code. In the future, we will further explore the design space of perturbation and failure-obliviousness strategies. For instance, we would like to consider timeout-based perturbations for concurrent software. Our long-term goal is to use TRIPLEAGENT in production, and consequently, we will also investigate the overhead of TRIPLEAGENT at runtime.
