Detecting Unchecked Exception-Related Behavioural Breaking Changes with UnCheckGuard

by

Vinayak Sharma

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2025

© Vinayak Sharma 2025

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis primarily comprises chapters derived from our research paper which has been accepted at the 25th IEEE International Conference on Source Code Analysis and Manipulation (SCAM 2025), incorporating wording changes, stylistic updates, and other modifications.

I have added the following additional work:

- Added information about software libraries and an overview of the thesis in Chapter 1.
- Added explanation to more core concepts related to the thesis in Chapter 2.
- Elaborated further on the motivating example in Chapter 3.
- Described the problem definition and setup information related to taint analysis in Chapter 5.
- Included examples and explanations on analysis setup in Chapter 5.
- Expanded the table 6.2 in Chapter 6 to showcase a bigger set of result.
- Added one more related work in Chapter 7

Abstract

The ubiquitous use of third-party libraries in software development has enabled developers to quickly add new functionality to their client software. Unfortunately, library usage also carries a cost in terms of software maintenance: library upgrades may include breaking changes, in which client expectations about library behaviour are no longer met in new library versions. Behavioural breaking changes can be particularly insidious, and in their full generality, could require sophisticated program analysis techniques to (approximately) detect.

In this work, we present our UnCheckGuard tool, which detects a class of behavioural breaking changes—those related to exceptions thrown by Java libraries. UnCheckGuard analyzes both sides of the library/client duet. On the library side, UnCheckGuard creates a list of new exceptions that may be thrown by methods in a library's public API, including by its transitive callees. On the client side, UnCheckGuard identifies client methods that call library methods with new exceptions. To reduce false positives, UnCheckGuard additionally filters out new exceptions that cannot be triggered by particular clients, using taint analysis. It therefore can be used by client developers as a tool to screen library updates for relevant incompatibilities.

We have evaluated UnCheckGuard on 302 libraries and 352 library-client pairs drawn from the DUETS collection and found 120 libraries with newly-added exceptions, as well as 1708 callsites to library methods which, when upgraded to the latest version, may introduce a behavioural breaking change in the client due to a newly added unchecked exception. These findings highlight the practical value of UnCheckGuard in identifying exception-related incompatibilities introduced by library upgrades.

Acknowledgements

First and foremost, I would like to thank Dr. Patrick Lam for his unwavering support throughout every stage of my master's journey. His constant positivity and encouragement toward excellence made it possible for me to work on such an interesting topic. From narrowing down the research focus to perfecting the tool, he has guided me at every step of the research process. His mentorship has not only shaped the direction of my thesis but has also played a significant role in my growth as a researcher and academic.

I would also like to thank my parents and my sister for their continuous support. As professors, my parents have always inspired me to strive for academic excellence and have been a constant source of motivation throughout my life. My sister, in particular, has provided me with valuable advice on how to improve academically, which has been instrumental during this journey.

Lastly, I would like to thank Dr. Weiyi Shang and Dr. Derek Rayside for their time and helpful comments, particularly the incitement to increase the size of the evaluation, making the results significantly stronger than in the initial submission.

Dedication

This thesis is dedicated to my family and Addy.

Table of Contents

Author's Declaration **ii**

Statement of Contributions **iii**

Abstract **iv**

Acknowledgements **v**

Dedication **vi**

List of Figures **ix**

List of Tables **x**

List of Abbreviations **xi**

1 Introduction **1**

2 Background **4**

3 Motivating Example **12**

3.1 Detecting Invocation of Library Methods in Client 12

3.2 Detecting Newly Added Exceptions in Library 13

3.3 Verifying If the Client Can Trigger the Exception 15

4 Data Collection **17**

5 Methodology **18**
5.1 Problem Definition 18
5.2 Analysis Setup 19
5.2.1 Library Version Resolution 19
5.2.2 Mapping External Method Invocations to Library Methods 21
5.3 Finding Newly Added Unchecked Exceptions 21
5.3.1 Exception Discovery 21
5.3.2 Exception Filtering with Taint Analysis 22
5.4 Filtering Untriggerable Unchecked Exceptions 26

6 Results **29**
6.1 Client Calls to Newly-added Exceptions 29
6.2 Newly-added Unchecked Exceptions in Java Libraries 39
6.3 Discussion: Developer-Facing Implications 40

7 Related Work **41**

8 Conclusion **43**

References **44**

List of Figures

2.1 Call graph for the `Example` program showing method calls and exception sources. 10

5.1 Pipeline of UnCheckGuard for detecting behavioural breaking changes due to newly added unchecked exceptions. 20

5.2 Taint propagation from a client parameter (source) to an exception site (sink) occurs via either *data dependence* (solid lines, e.g., a value passed into the exception constructor arguments) or *control-flow dependence* (dashed lines, e.g., a value influencing a branch that leads to the `throw` statement). 23

List of Tables

6.1 Exception Analysis Funnel 29
6.2 Selected clients, libraries, versions, and counts of callsites reaching newly-added exceptions 30
6.3 Distribution of reachable newly-added exceptions across version types . . . 39

List of Abbreviations

API Application Programming Interface 1, 2, 6, 14, 42

BBC behavioural breaking change 2, 4–7, 10, 12, 43

Chapter 1

Introduction

Software libraries are mainly collections of code components designed to be reused across projects at different levels. They help developers save time and effort by eliminating the need to rewrite code, and their functionalities can be directly integrated into software applications. The use of libraries developed by others is ubiquitous in modern software development [21, 45]. In recent years, the number of libraries available in different software ecosystems—such as npm for JavaScript [14], Maven Central for Java [6], and PyPI for Python [17]—has grown significantly. For example, PyPI, the Python software ecosystem, has exhibited a 47% compound annual growth rate in active packages [4].

A wide range of contributors—including: (i) Individual developers, (ii) Open-source communities, (iii) Companies (such as Google, Facebook, and Microsoft) and (iv) Academic and Research institutions [29]—develop, maintain, and upgrade software libraries. These contributors create libraries to meet specific needs and continue refining them over time.

The dependence on software libraries has increased over time because of the increase in open source software OSS movement. Therefore, software development has become a highly modular activity and has a huge reliance on third-party libraries [10, 47, 7]. However, libraries developed by others are also updated by others, on schedules that are not controlled by the client developers.

When a client develops software exposed to the internet, they take on a moral responsibility for ensuring that the application integrates security updates for all utilized software libraries [46]. Failing to apply the latest updates can leave the application vulnerable to well-known security threats such as Log4Shell (Log4j), Heartbleed (OpenSSL), and similar critical vulnerabilities [20, 48, 1]. As such, clients must regularly update third-party libraries, since neglecting updates constitutes a form of technical debt that accumulates automatically over time.

However, upgrading a library to the latest version is not painless [15, 11, 8]: new versions of libraries can include breaking Application Programming Interface (API) changes [12]. In the case of method signature-related changes, the library developer may have modified method signatures, retracted previously available methods, or made other interface-level changes that are no longer compatible with the client code. Worse yet, non-signature

related breaking changes may trigger behavioural changes in client application. A behavioural breaking change (BBC) caused by a non-method signature change can result from modified function logic or a newly added unchecked exception in the function. Clients can often catch these breaking changes with the help of test cases, provided they have created tests to verify the behaviour of their application, especially as it relates to its use of the library. Upgrading the library can be inconvenient at best and may require nontrivial changes in the client code at worst. Keeping libraries upgraded ensures that the client code continues to work properly, and hopefully in a less-vulnerable manner.

Compilers and simple static checkers (including japicmp[1] and Revapi[2] for Java as well as [5, 16]) can verify the absence of syntactic breaking changes in libraries. The situation is worse for semantic/behavioural breaking changes: there do not exist techniques for reliably detecting such changes. Of course, in its full generality, the problem is undecidable, though breaking change detection could be estimated using static and dynamic program analysis techniques. One such technique has been implemented by CompCheck [49]. The tool utilizes pre-written test cases present in the client code and runs them while updating the library versions to find behavioural breaking changes (BBC). Then, the tool uses the pattern in which the API was used to perform pattern matching and identify more clients that may have a BBC.

In this work, we contribute a novel way to detect one type of BBC in a library. Our approach enables client developers to inspect relevant changes to the set of unchecked exceptions that a Java library may throw—particularly by the APIs actually used by specific client code. A new unchecked exception thrown by a library constitutes a BBC; uncaught exceptions can cause the client code to crash or exhibit unexpected behaviour. First, we collect all the APIs used by the client and analyze them for any newly added unchecked exception(s). We perform this analysis using static analysis tools and libraries such as SootUp [25], Soot [44], and FlowDroid [2]. Then, we conduct a taint analysis to ensure that the unchecked exception is actually triggerable by the client.

Developers often ignore checked exceptions [36], but we contend that informing developers about relevant newly added exceptions as they upgrade the library is more likely to result in developer action, consistent with the design principles of Google's Tricorder tool [42]. Therefore, we utilize taint analysis to reduce the number of false positive reports sent to client developers. By doing so, the number of reported APIs shown to the client includes only those that may realistically throw new exceptions in updated versions of the client code, minimizing false positives [39]. We hope that our reports help client developers better understand how new exceptions affect their own code.

We explore the following research questions:

RQ1. Do library clients call methods with new added exceptions, and is it possible for the clients to trigger these exceptions? Furthermore, is it possible to write client-focussed test cases that trigger the exceptions?

RQ2. For library changes that introduce triggerable new unchecked exceptions, under what circumstances do such exceptions occur (i.e. major/minor/patch versions)?

[1] https://github.com/siom79/japicmp

[2] https://revapi.org/revapi-site/main/index.html

In our corpus of 302 distinct libraries, we found 120 libraries with newly-added exceptions, with at least 47.5% of these exceptions being client-relevant exceptions added in non-major releases. We then investigated 352 client-library pairs to explore the prevalence of potentially breaking behavioural changes. We found that new potentially client-relevant unchecked exceptions occured in 120 of the 302 libraries, and that clients called methods reaching these exceptions at 1708 client callsites. This shows that client applications do in fact call library methods that throw these new exceptions. Furthermore, we demonstrated that it is possible to trigger these exceptions by writing test cases using methods from the client.

The contributions of this work are as follows:

- We implement the *UnCheckGuard* static analysis tool, which traverses bytecode to find newly-added exceptions and filters them using taint analysis, to report relevant newly added unchecked exceptions.
- We conduct an empirical study of libraries to detect potential behavioural breaking changes in libraries caused by newly added unchecked exceptions.
- We evaluate 352 client-library pairs from the DUETS dataset [13] using *UnCheckGuard*, identifying 1708 call sites where libraries' newly added unchecked exceptions could cause behavioural breaking changes in clients, and write test cases showing that the exceptions can be triggered from client code.

Data Availability Statement.

We have made our tool and dataset available at https://doi.org/10.5281/zenodo.16788650

Chapter 2

Background

In this chapter we define core concepts that underpin our approach to detecting BBCs caused by newly added exceptions.

Exception Handling. Exception handling is a tool that allows developers to recover from exceptional or error conditions that may disrupt the intended flow of an application [37]. In particular, Java supports the use of `try-catch` blocks to handle exceptions. The following is an example of exception handling in Java using a `try-catch` block:

```
1  public class ExceptionHandlingExample {
2      public static void main(String[] args) {
3          try {
4              int result = Math.floorDiv(10,0);
5              System.out.println("Result: " + result);
6          } catch (ArithmeticException e) {
7              System.out.println("Something went wrong: " + e.
                   getMessage());
8          }
9      }
10 }
```

In the above example, dividing 10 by 0 using `Math.floorDiv` throws an `ArithmeticException`, as specified in the official documentation of the `Math` class in the Java Standard Library.[1] The `try-catch` blocks catch and handle the `ArithmeticException` to prevent the program from crashing.

Java Exceptions. Java defines two different types of exceptions:

- **Checked Exception**: appears in the throwing method's signature. The client must either catch the exception or declare it to be thrown [43]. The compiler checks this type of exception, as do several tools such as japicmp[2]. The Java Language

[1] https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

[2] https://github.com/siom79/japicmp

Specification [19] formally defines the semantics of checked exceptions and compiler behaviour. Following is an example of a checked exception:

```
public static void readFile() throws FileNotFoundException
    {
    FileReader file = new FileReader("data.txt");
    System.out.println("Reading file...");
    file.close();
}
```

In this example, the public method `readFile()` declares that it throws an `FileNotFoundException`, which is a checked exception in Java. If the method cannot read the file `data.txt` or if the file does not exist, the `FileReader` constructor[3] throws a `FileNotFoundException`. The developer must handle this exception either by using a `try-catch` block or by declaring it in the method signature with the `throws` keyword.

- **Unchecked Exception**: do not appear in the throwing method's signature, and includes subclasses of RuntimeException or Error. This type of exception does not appear in the method's signature [3]. As a result, the compiler does not check it, and tools such as japicmp do not detect it. These exceptions can cause unexpected runtime failures when the client does not handle them correctly [9]. This type of exceptions often gets overlooked by client developers particularly during testing, especially when they are introduced through library upgrades. Client developers often overlook this type of exception during testing, especially when library upgrades introduce them. The addition of unchecked exceptions to newer versions of libraries can contribute to BBCs in the client applications. Following is an example of an unchecked exception:

```
public class ThrowUncheckedExample {

    public static void main(String[] args) {
        boolean flag = false;
        checkFlag(flag);
    }

    public static void checkFlag(boolean flag) {
        if (!flag) {
            throw new IllegalArgumentException("Flag is
               false");
        }
        System.out.println("Flag is true");
    }
}
```

[3] https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html

In this example, the method `checkFlag` explicitly throws an `IllegalArgumentException`, which is an unchecked exception. Since it is a subclass of `RuntimeException`, the Java compiler does not require the `checkFlag` method to declare it using the `throws` keyword or to check whether the calling function handles it using a `try-catch` block.

Why unchecked exceptions exist. Java includes unchecked exceptions to represent programming errors that developers should fix rather than handle using a `try-catch` block. Unchecked exceptions usually indicate issues such as invalid arguments, logic bugs, or null references. Since these exceptions reflect flaws in the program's logic, Java does not require developers to declare them in a method's signature. This design choice helps avoid forcing developers to catch exceptions they cannot reasonably recover from.

Transitive Throwing. Unchecked exceptions can propagate through multiple method calls without being declared. When a method throws an unchecked exception and does not catch it, the exception moves up the call chain until the program either catches it or crashes. We refer to this behavior as transitive throwing.

This becomes problematic when library methods introduce new unchecked exceptions. Even if the client does not directly call the modified method, the exception can still reach client code through intermediate methods. This silent propagation of transitive unchecked exceptions can potentially cause a BBC.

Breaking Changes. We define a breaking change as a change in the library's API that causes the client code to either break or stop functioning the way it did prior to the library upgrade. Breaking changes fall into two categories:

- **Syntactic Breaking Changes**. This type of change usually occurs when the method signature changes. Library developers may change the method's signature by removing or updating the function name, modifying the input parameters, changing the checked exception(s) associated with the function, or altering the return type [22]. Static tools like japicmp can detect these types of method signature changes. The Java compiler checks for syntactic breaking changes during compilation but it only checks the code that is being recompiled. The Java compiler will not report any errors if the JAR of a library is replaced during runtime without recompiling (drop-in change). An example of a syntactic breaking change follows. We first present the code before changes:

```
1 public class MyLibrary {
2     public void greet() {
3         System.out.println("Hello!");
4     }
5 }
```

  Now, we present the code after the changes, which introduces a syntactic breaking change:

```java
public class MyLibrary {
    public void greet(String name) {
        System.out.println("Hello, " + name + "!");
    }
}
```

In this example, the original version defines the method `greet()` with no parameters. In the updated version, the method `greet()` requires a parameter of type `String`. If the client application, which used the original version, tries to recompile the code against the updated version, the compiler raises a compilation error, indicating that no method `greet()` with zero arguments exists.

- **Behavioural Breaking Changes (BBC).** In this type of change, the syntax remains the same, but the semantics change. Various reasons can cause such changes in semantics, including updates to the function logic, addition of a new unchecked exception, or modification of an existing unchecked exception (for example, changing an `IllegalArgumentException` into a `NullPointerException`). In general, the compiler does not detect these types of changes. They can cause the client's application to crash at runtime. The following is an example of a behavioural breaking change. We first present the original version of the method before any modifications:

```java
public class FeatureToggle {
    public boolean isFeatureEnabled() {
        return true;
    }
}
```

Client:

```java
FeatureToggle ft = new FeatureToggle();
if (ft.isFeatureEnabled()) {
    System.out.println("New feature is active!");
}
```

The client prints "New feature is active!" when using the original version of the `isFeatureEnabled` method.

Now, we present the updated version of the method:

```java
public class FeatureToggle {
    public boolean isFeatureEnabled() {
        return false;  // Changed behaviour
    }
}
```

In this example, the original version of the `isFeatureEnabled` method always returns `true`, which causes the client to print "New feature is active!". After the update,

the method's logic changes to always return `false`. As a result, the client no longer prints the message. Although the method signature remains unchanged, the internal behaviour differs. This change breaks the intended behaviour of the client application.

Semantic Versioning. Software libraries generally follow semantic versioning (semver) [40], where the version number of the library indicates the level of change. Developers structure the numbers as "MAJOR.MINOR.PATCH":

- **MAJOR** version number flags breaking changes in the library.
- **MINOR** version number indicates the introduction of new features while ensuring that everything from the previous version still works (backward compatibility).
- **PATCH** version number refers to bug fixes only.

However, in practice, developers often introduce breaking changes even in minor or patch versions [23]. This behaviour makes it especially important to create and use tools that check behavioural compatibility instead of relying solely on version numbers.

Static Analysis. Static analysis involves debugging the source code or bytecode of a program without executing it. Developers use it to analyze potential errors and security vulnerabilities, and they can also apply it to support compiler optimizations. It is particularly useful when dynamic test cases are not available, incomplete, or insufficient, and allows developers to get prior information about possible errors and issues that may occur during actual execution [41].

Static analysis is particularly useful in analyzing large programs, especially those that incorporate multiple libraries. By providing early feedback to developers, static analysis reduces the likelihood of errors, thereby helping to maintain the reliability and performance of the software. Despite its benefits, static analysis also presents several drawbacks: (i) it often produces false positives, especially in complex codebases with dynamic behaviour such as Java reflection, dynamic class loading, or external dependencies; and (ii) since it analyzes code without executing it, it cannot always determine the actual execution path, which limits its precision.

Dynamic Analysis. Dynamic analysis runs the actual program to observe its behaviour during execution. It can potentially detect runtime issue caught during execution, but it relies on test coverage and can miss particular cases when a test case is not available for that case [28]. Dynamic analysis is helpful in analysing software behaviour. It does so by analysing program operations during execution.

Further, dynamic analysis has following advantages: (i) it minimizes false positives by reporting only those faults that actually occur during execution, (ii) It proves particularly useful in detecting security-related issues, including buffer overflows, improper input handling, and unauthorized access, provided that a suitable test case is already present. Despite these advantages, dynamic analysis has certain limitations also and same are as follows: (i) Its effectiveness mainly depends on the test cases used; (ii) part of code that is not executed during testing will not be analyzed for potential errors, and (iii) It is also

sometimes hard to actually run a large software system. Static analysis is sometimes more viable for these systems, even if it has to approximate the behaviour.

Taint Analysis. Taint analysis is a program analysis technique which can be implemented statically [34] or dynamically [38]. It relies on declarations of sources (for example, client input) and sinks (for example, critical operations or exceptions). Given these inputs, the analysis tracks whether the sources can reach the sinks.

The following example demonstrates how taint flows from a source to a sink:

```
public class FlowDroidExampleCode {
  public static int source() { return 1337; }

  public void exampleTaintAnalysis() {
    int temp = source(); int[] arr = new int[2];
    arr[0] = temp; arr[1] = 19;
    if (arr[0] == 1337) {
      throw new RuntimeException("hello"); }
    }
}
```

In this example, the method `source()` acts as the taint source. The statement `throw new RuntimeException("hello")` is the sink. The tainted value flows into the array `arr`, and later influences the conditional that triggers the exception. Although the exception is hardcoded, the fact that its execution depends on a tainted value makes this a valid taint flow from the source to the sink.

We apply taint analysis to detect whether newly added exceptions in a library are reachable from client-supplied values. This helps us detect behavioural breaking changes where a newly added unchecked exception is only triggered under specific conditions influenced by the client.

Call Graph Analysis. In static analysis, call graphs identify the function calls that the program will make. Nodes represent methods, and edges represent calls from one method to another. This technique plays a fundamental role in data flow analysis, control flow analysis, dead code elimination, and taint tracking [26]. For example, consider the following Java program:

```
public class Example {
    public static void main(String[] args) {
        int n = 10;
        int d = 0;
        doWork(n,d);
}

public static void doWork(int n, int d) {
    if ( d == 0){
        throw new IllegalArgumentException("denominator cannot
            be zero");
```

```
11      }
12      calculate(n,d);
13  }
14
15  public static void calculate(int n, int d) {
16      int result = Math.floorDiv(n,d);
17  }
```

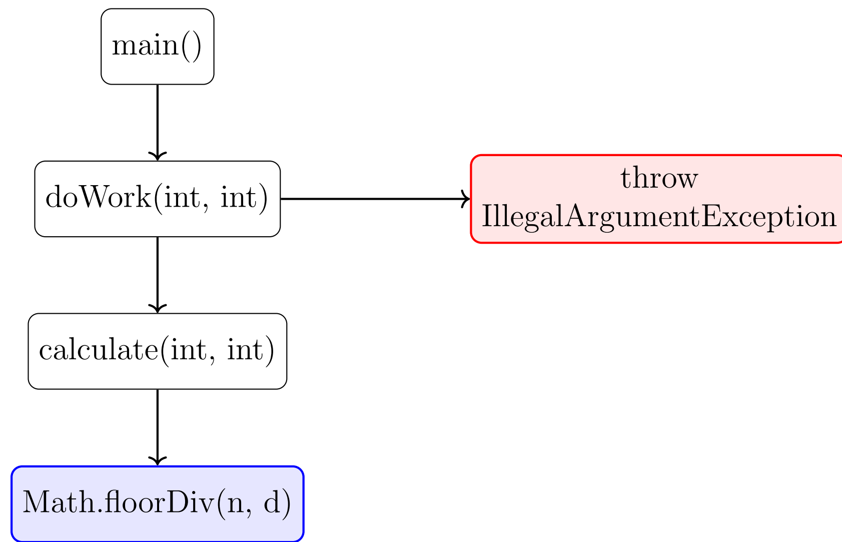

Figure 2.1: Call graph for the `Example` program showing method calls and exception sources.

Figure 2.1 illustrates the call graph for the given Java program. The graph shows the sequence of method calls starting from the `main()` method, through `doWork()`, and into `calculate()`. Inside `doWork()`, a critical operations occur: a user-defined unchecked exception (`IllegalArgumentException`) is explicitly thrown when the denominator is zero.

This call graph is useful in tracing the flow of potential exceptions-both user-defined and library-sourced-through the client code. If a library version introduces a new unchecked exception at any point in this call chain (e.g., inside `Math.floorDiv()`), the static call graph can help identify all the client-side methods that may now be affected by that change. This is essential for detecting BBCs.

DUETS Dataset. The DUETS dataset [13] provides a list of real-world Java client-library pairs. Each client in the DUETS dataset has over 5 stars on GitHub. For our evaluation, we selected 1011 clients from DUETS in a systematic manner, rather than using the entire dataset of 147,991 clients. The DUETS dataset provides a list of Java-based clients that are more actively maintained and show better community engagement, as all the projects have over 5 stars. It represents real-world library usage in Java-based projects.

SootUp. SootUp [25] is a respin of the Soot [44] framework that supports static analysis of Java bytecode.

SootUp transforms JVM bytecode into the intermediate representation Jimple, which simplifies analysis by converting low-level bytecode instructions into a higher-level format that makes method bodies, variable assignments, exception handling blocks, and method invocations accesible. SootUp also provides call graph generation with various algorithms and precision levels. When a library method throws a new unchecked exception, we use SootUp to determine whether client methods transitively call that library method by traversing the (Class Hierarchy Analysis) call graph. We also use the Jimple intermediate representation to inspect methods that may throw an exception, by examining throw statements and method calls within their bodies.

FlowDroid. FlowDroid [2] is a static taint analysis framework. It tracks data flow from declared sources to sinks within the application's code. It is built on top of the Soot [44] static analysis framework. FlowDroid models the complete Android lifecycle and callback structure—irrelevant for our purposes—but, relevant to us, enables flow-sensitive, field-sensitive, context-sensitive, and object-sensitive analysis of both Android and normal Java Virtual Machine programs.

It checks whether data from a source will taint a sink by computing possible paths along which the data can flow. In our tool, we use taint analysis to check the approximate reachability of newly added unchecked exceptions from client code.

Chapter 3

Motivating Example

We present a motivating example to showcase how a newly added unchecked exception can cause a BBC in the client code. We select the client/library pair from the DUETS dataset [13]. We also demonstrate how we wrote a test case to verify the behavioural breaking change in the client caused by the updated library.

For this motivating example, we use `HttpAsyncClientUtils`[1] as our client. Which is one of the clients in DUETS. It declares version 4.4.6 of the `httpcore` library[2] as one of its dependencies. Since the release of version 4.4.6, the developers of the `httpcore` library have released several newer versions. The latest available version is 4.4.16.[3].

The update from version 4.4.6 to 4.4.16 of the `httpcore` library introduces a new check in the `Args.containsNoBlanks()`[4] static method. In this update, the library checks the length of the argument that the client provides to the constructor `<init>(String, int)` of the `org.apache.http.HttpHost` class. Version 4.4.16 verifies whether the argument length is equal to 0, and if so, throws a new `IllegalArgumentException`. In the specific case of `HttpAsyncClientUtils` as the client and `httpcore` as the library, the client calls the `<init>(String, int)` constructor of the `org.apache.http.HttpHost` class, which in turn calls the `Args.containsNoBlanks()` static method containing the newly added exception. We now demonstrate how UnCheckGuard identifies that the client is using a method from the library, detects the newly added exception, and verifies that the exception is actually triggerable by the client to avoid false positives.

3.1 Detecting Invocation of Library Methods in Client

A newly added unchecked exception in a library is only relevant to a client if the client actually uses the library method that introduces the exception. UnCheckGuard eliminates anal-

[1] https://github.com/iiweniiang/HttpAsyncClientUtils

[2] https://hc.apache.org/index.html

[3] While `httpcore` 5.2.4 is in fact the latest version of this library, the library developers have released `httpcore5` as a distinct Maven component from `httpcore4`, and labelled `httpcore`(4) as end-of-life.

[4] Fully-qualified name: method `containsNoBlanks(java.lang.CharSequence,java.lang.String)` returning a `java.lang.CharSequence` on class `org.apache.http.util.Args`

ysis of unused library methods by first analyzing the client for external method invocations (which we define in Chapter 4). We perform this analysis by identifying all external method calls within the client code using SootUp [25]. In the case of `HttpAsyncClientUtils` as the client, it calls the `httpcore` library from its public `createAsyncClient(boolean)`[5] method, which creates an `HttpHost` with an empty `host`. This method accepts a `proxy` parameter and includes the following code:

```
if (proxy) {
 return HttpAsyncClients.custom()
  .setConnectionManager(conMgr)
  .setDefaultCredentialsProvider(credentialsProvider)
  .setDefaultAuthSchemeRegistry(authSchemeRegistry)
  .setProxy(new HttpHost(host, port))
  .setDefaultCookieStore(new BasicCookieStore())
  .setDefaultRequestConfig(requestConfig).build();
} else {
  // ...
```

The `HttpAsyncClientUtils` client declares the two variables (host and port) required for `HttpHost` (line 6 in the above code) in the following way:

```
   private String host = "";
   private int port = 0;
  // ...
```

The client initializes `host`, a private field of type `String`, with an empty string and sets `port` to 0. As a result, the client calls `<init>(String, int)`[6] with the host set to an empty string and the port set to 0.

After collecting all external method calls made by the client `HttpAsyncClientUtils`, UnCheckGuard begins comparing the version of the library currently used by the client with the latest available version. It stores all external method invocations made by the client–not just those to the `HttpHost` library—in a `JSON` file. If UnCheckGuard detects a newly added exception in any of these external method calls, it then uses taint analysis to check whether the exception is actually reachable from the client. For this client, UnCheckGuard identifies a call to `HttpHost`. The next step for UnCheckGuard is to detect the newly added exception and verify its reachability.

3.2 Detecting Newly Added Exceptions in Library

UnCheckGuard in the previous step found that the client `HttpAsyncClientUtils` makes a call to a constructor for the `org.apache.http.HttpHost` class. UnCheckGuard now analyzes the set of exceptions thrown by the constructor for both 4.4.6 and 4.4.16 version.

[5] Fully-qualified: method `createAsyncClient(boolean)` returning a `CloseableHttpAsyncClient` on class `Util.HttpClientUtil.HttpAsyncClient`.

[6] Specifically, constructor `<init>(String, int)` returning a `void` on class `org.apache.http.HttpHost`

To determine whether the newer version of the library introduces a newly added exception, UnCheckGuard analyzes the the implementations of `httpcore-4.4.6` and `httpcore-4.4.16`. It performs this analysis using SootUp [25]. UnCheckGuard constructs a call graph using Class Hierarchy Analysis (CHA), starting from the public `<init>(String, int)` constructor in `HttpHost`. With the help of this call graph, UnCheckGuard identifies the set of all methods transitively reachable by the client (discussed below). It then collects the list of exceptions present within those methods. UnCheckGuard performs this process for both the currently used version of the library and the newer version. During this step, it stores the list of exceptions found in different methods of a particular API (API is the set of public methods of the library) in a `JSON` file, as this information is needed later for comparison.

Now, based on the call graph constructed for the public `<init>(String, int)` constructor on `HttpHost`, UnCheckGuard finds that the constructor directly calls the static method `Args.containsNoBlanks()`[7] in the version 4.4.16. However, in certain cases, the client may invoke exception-throwing methods transitively. The exception-throwing method may reside within a Java library that the client's library depends on. These types of transitive calls significantly complicate the task of determining whether the client will be affected by a newly added unchecked exception, as the analysis must resolve method calls beyond the library under direct analysis.

Upon a manual inspection of the static method `Args.containsNoBlanks()`, we find that the developers have added the following lines of code to the 4.4.16 version:

```
1  public static <T extends CharSequence> T containsNoBlanks(
       final T argument, final String name) {
2          ...
3          if (argument.length() == 0) {
4              throw new IllegalArgumentException(name + " may
                   not be empty");
5          }
6          ...
7          return argument;
8      }
```

Specifically, all `HttpHost` constructors take a `hostname` parameter and call `containsNoBlanks()` with that parameter (to check that it contains no blanks). In the case when the length of the argument (`host`) is equal to zero it throws a unchecked exception `IllegalArgumentException`. A client can therefore trigger this newly added exception in a client by attempting to instantiate a new `HttpHost` object and passing it an empty `hostname`.

Our UnCheckGuard tool analyzes the changes in `httpcore` and finds that, in version 4.4.16, all of the `HttpHost` constructors can now throw an `IllegalArgumentException` through the `containsNoBlanks()` method. Version 4.4.6 does not throw this exception.

[7] Fully-qualified name: method `containsNoBlanks(java.lang.CharSequence,java.lang.String)` returning a `java.lang.CharSequence` on class `org.apache.http.util.Args`

To further verify whether the client can actually trigger this exception, we apply taint analysis, since a control-flow path from the client to the exception-throwing site does not guarantee that the exception is triggerable. The condition for triggering the exception might depend on internal values of the library rather than on client-supplied input.

3.3 Verifying If the Client Can Trigger the Exception

After UnCheckGuard collects the methods that introduce newly added exceptions in the latest version of the library, It verifies whether the client-supplied values can trigger the exception by checking if they reach the exception-throwing site. To perform this verification, UnCheckGuard applies FlowDroid [2]'s taint analysis. Taint analysis is necessary in this situation because the presence of a control-flow path between the client callsite and the exception-throwing statement is not sufficient to conclude that the exception is actually triggered by the client. In many scenarios, such a path may exist in the library, but the path condition leading to the exception may depend entirely on internal library values rather than on client-supplied input; in these cases, the client cannot cause the execution of any path that throws the exception. In our experience, taint analysis proves beneficial in distinguishing actual behavioural breaking changes from false positives. In Chapter 5, we provide an example of a library that introduces a newly added exception, but the exception is not triggerable with the client-supplied values.

Taint analysis tracks whether any client-supplied values to the external method used by the client can reach the exception object's constructor. UnCheckGuard marks the client-supplied values as `SOURCE` and the exception object's constructor as `SINK`. If taint analysis determines that the `SOURCE` cannot reach the `SINK`, or that the `SOURCE` cannot flow into the exception-triggering logic, then UnCheckGuard does not report the case to the client, as the newly added exception will not result in a behavioural breaking change.

UnCheckGuard marks the following as `SOURCE`:

- `hostname`
- `port`

UnCheckGuard marks the following as the `SINK`:

- `IllegalArgumentException` found in `Args.containsNoBlanks()`

In version 4.4.6, UnCheckGuard finds two sites throwing `IllegalArgumentException`, while in 4.4.16, it detects three—each of which the client can potentially trigger using the values it chooses to pass to the library as parameters.

Based on FlowDroid's confirmation of the reachability of the new exception's constructor, we report that the library-client pair `HttpAsyncClientUtils` and `httpcore` exhibits a behavioural breaking change.

Given this report from UnCheckGuard, it is straightforward to write a test case that calls the client's `createAsyncClient()` method and triggers the exception after an upgrade:

```
@Test
void testCreateAsyncClientThrowsExceptionForEmptyProxyHost() {
  HttpAsyncClient client = new HttpAsyncClient();

  IllegalArgumentException exception =
    assertThrows(IllegalArgumentException.class, () -> {
        client.createAsyncClient(true);
    });

    assertTrue(exception.getMessage()
    .contains("may not be empty"),
      "Expected exception due to empty hostname "+
      "after upgrading to httpcore-4.4.16");
}
```

The above test case passes for version 4.4.16 of `httpcore`, as the constructor method of `org.apache.http.HttpHost` throws an `IllegalArgumentException` when called with an empty `hostname`. The same test case fails for version 4.4.6 of `httpcore`, as this version does not contain the exception.

This test case[8] demonstrates a `PATCH` version upgrade (from 4.4.6 to 4.4.16) that introduces a behavioural breaking change due to a newly added exception. Further, it also demonstrates how UnCheckGuard operates.

[8]Link for the test case: https://github.com/vinayaksh42/HttpAsyncClientUtils/blob/new/src/test/java/Util/HttpClientUtil/HttpAsyncClientTest.java

Chapter 4

Data Collection

This chapter describes the approach we use to construct the dataset for our study on behavioural incompatibilities caused by newly added unchecked exceptions in upgraded Java libraries. The data we collect creates the foundation for our analysis in subsequent chapters. Specifically, it enables UnCheckGuard to detect methods that introduce newly added unchecked exceptions, identify affected client call sites (Chapter 5).

To begin our analysis, we first collected suitable client projects. We used the DUETS dataset [13], which provides a curated list of Java-based clients hosted on GitHub, each with at least five stars. DUETS also pairs libraries with the clients, but we ignore the DUETS library declarations and instead consider all libraries declared as dependencies by each client.

The DUETS dataset contains a total of 147,991 Java projects with more than 5 stars on GitHub. DUETS further filters these projects to retain only single-module Maven projects, executes the test suite within each project, and ultimately leaves 34,280 projects. We then filter this dataset further by increasing the minimum number of GitHub stars for a Java project to 10 or more. Due to the newly added rate limiting by Maven Central[1], running all 19,290 Java projects collected after the filtering based on 10 or more GitHub stars has become difficult. We sample from DUETS by systematically selecting clients at an interval of 19 from the list. This approach yields a dataset of 1,011 clients, small enough to avoid the rate limiting, that represents real-world Java usage.

We attempted to download each client repository and discarded any client that failed to download. Next, we checked whether the project included a `pom.xml` file, which indicates that it is a Maven-based project. This step was essential, as our analysis depends on running Maven commands. We compiled each client to produce a JAR file and kept only those clients that compiled successfully for further analysis.

[1] https://www.sonatype.com/blog/maven-central-and-the-tragedy-of-the-commons

Chapter 5

Methodology

The last chapter, Chapter 4, described how we collected the clients and libraries, as well as the necessary information to perform our analysis. In this section, we describe our methodology for detecting and verifying newly added unchecked exceptions in a library when it is updated from an older version to a newer one. Our focus is on identifying the impact of such changes on client code. Specifically, we analyze client programs to detect usage of library methods that were updated to throw previously non-existent unchecked exceptions. Java distinguishes between checked exceptions, which appear as part of method signatures, and unchecked exceptions, which do not. Unchecked exceptions may therefore introduce a class of breaking changes that method signature-based syntactic approaches for Java cannot detect.

After carrying out the data collection steps in Chapter 4 and extracting all external library methods invoked by the client, we next analyze their implementations, in both the current version and the latest version. This allows us to compare their behaviour across versions. Our analysis is specifically targeted at detecting one particular class of behavioral breaking changes: the introduction of new unchecked exceptions in updated library methods that the client can trigger with the client-supplied values. If we find, through our analysis, that a method now throws a newly added unchecked exception in the latest version, and that the exception can be triggered from the client code, we flag a potential behavioural breaking change. To verify whether this change is in fact breaking, we currently manually write test cases to verify that the client may be affected by the newly introduced exception. We found that this was easy to do given the information that UnCheckGuard reports.

5.1 Problem Definition

Let P be a Java client program that depends on a library L at version v_{old}, and suppose that L is later updated to version v_{new}. We wish to determine whether any methods in L now throw *new unchecked exceptions* in v_{new} that did not occur in v_{old}, and whether such exceptions can be triggered by valid usage patterns in P.

Formally, given:

- A client program P, compiled as a JAR file;
- A library L in two versions: L_{old} and L_{new};
- A set of method invocations from P to L, denoted $\texttt{Calls}(P \rightarrow L)$;

Our task is to identify a set of pairs (m, e) where:

- m is a method in L that is invoked by P;
- e is a subclass of `java.lang.RuntimeException` or `java.lang.Error`;
- e is not thrown (transitively) by m in L_{old}, but is thrown in L_{new};
- There exists a path from a client-controlled input in P to the exception site where e is instantiated.

We refer to such pairs as *client-reachable newly added unchecked exceptions*, and consider them potential behavioural breaking changes.

Note that we focus exclusively on **unchecked** exceptions, which do not appear in Java method signatures. Checked exceptions are excluded from this study as they are already caught by signature-diffing tools such as japicmp.

5.2 Analysis Setup

We divide analysis setup into two steps: 1) library version resolution; and 2) mapping external method invocations to library methods. Figure 5.1 shows the full pipeline.

As described in Chapter 4, we begin by selecting Java-based clients from the DUETS dataset [13]. We retain only those clients that are Maven-compatible and successfully compile to a JAR file, ensuring compatibility with our analysis tooling.

After selecting the valid clients and compiling them into JAR files, we proceed to extract relevant method-level knowledge from both the client and the current version of the libraries it depends on.

5.2.1 Library Version Resolution

UnCheckGuard depends on analyzing both the current version of the library used by the client and the latest available version of the library, as stored in the Maven Central Repository. To collect both versions, we run a set of Maven commands. To retrieve the current version of the library, we run the following set of Maven commands:

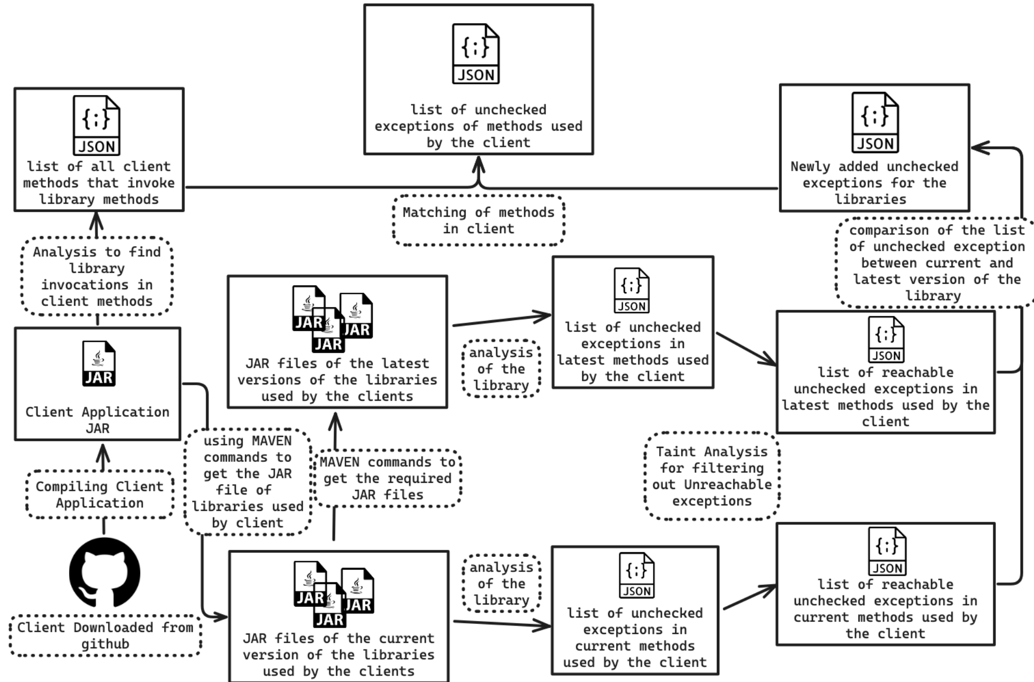

Figure 5.1: Pipeline of UnCheckGuard for detecting behavioural breaking changes due to newly added unchecked exceptions.

```
mvn clean package -DskipTests -fn
mvn dependency:copy-dependencies
```

This command allows us to store all the dependencies currently used by the client.

To obtain the latest version of the libraries, we run the following Maven command:

```
mvn org.codehaus.mojo:versions-maven-plugin:2.18.0:use-latest-
   versions
mvn clean package -DskipTests -fn
mvn dependency:copy-dependencies
```

This command updates the `pom.xml` file with the most recent versions of all declared dependencies. We then re-run `mvn dependency:copy-dependencies` to download the updated set of libraries.

This process fetches both the current and the latest versions of each library used by the client, enabling us to perform a comparative analysis of behavioural changes across library versions. Out of the 1011 clients that we collected, we found that these clients collectively used 302 libraries.

5.2.2 Mapping External Method Invocations to Library Methods

We use SootUp [25] to analyze the client JAR and identify all external method invocations. By external methods, we mean methods that are not defined within the client's own codebase—methods whose definitions reside in third-party libraries. UnCheckGuard performs this analysis by traversing the Jimple intermediate representation of each client method and checking whether any statement contains an `InvokeExpr`, which represents a method invocation. For each invocation, we retrieve the declaring class type of the target method. We then check whether this class type is part of the client's SootUp view—essentially, whether it was declared in the client JAR file or the Java standard library. If the class type is not found in the view, we mark the method as external. This process allows us to filter out internal method calls and consider only invocations to external library methods.

In parallel, we analyze the current version of each library used by the client. We extract all method signatures defined in the library JAR. We then match each external method call made by the client to the corresponding method in the library, by comparing their fully qualified method signatures. For the matching process, for simplicity, we perform an exact match between the declaring type of the method invoked by the client and that in the library to create a client/library mapping. This approach may miss some valid matches in the presence of dynamic dispatch—the declared receiver object type may differ from the actual receiver object type that the client uses at the call site—so the current version of UnCheckGuard may underreport some breaking changes.

Our client/library mapping identifies library methods and links them to where they can be invoked by the client. This mapping serves as a foundation for later stages in our analysis, where we detect behavioural changes in the latest versions of libraries and traces their potential impact on client call sites.

5.3 Finding Newly Added Unchecked Exceptions

Our primary goal is to detect whether upgrading a library introduces new unchecked exceptions that could affect client behaviour. To achieve this, we divide the process into two stages: first, identifying newly added unchecked exceptions using a call graph; and second, verifying their reachability from client input using a taint analysis. We then compute differences between versions in the sets of reachable exceptions.

5.3.1 Exception Discovery

To detect newly added unchecked exceptions in the latest library versions, we first construct a call graph using SootUp's Class Hierarchy Analysis implementation. CHA includes all methods with the correct signature defined in subclasses and interface implementations.

By definition, CHA reports the most conservative soundy [31] answer possible, absent reflection and other dynamic features. Thus, it tends to over-approximate, reporting

method calls that are unreachable in practice. For example, in one case, CHA identified a path from the public `getString(String)`[1] method, reporting an exception thrown in the `JSONObject` constructor as reachable. However, manual inspection revealed that this path was spurious—the method `getString` never reaches the constructor in question because, in the specific program under analysis, no code instantiates a `JSONObject`. Our next step, taint analysis, filters out some such unreachable methods.

We traverse our callgraph and collect all instantiated exceptions that are subclasses of either `java.lang.RuntimeException` or `java.lang.Error`. Per the definition of the Java programming language, such exceptions represent the complete set of unchecked exceptions that the client might be newly exposed to due to the library upgrade.

5.3.2 Exception Filtering with Taint Analysis

Once we collect the list of unchecked exceptions, we need to determine which of them can actually be triggered by client inputs. This is necessary because many exceptions that show up during call graph analysis are not reachable in practice—they rely on internal values rather than any parameters the client supplies (see below for an example). To filter out such cases, we use FlowDroid [2], a static taint analysis framework.

Consider the following case from our corpus. The client `4ntoine/ServiceDiscovery-java`[2] uses the method `copyFromUtf8(String)` from the library `protobuf-java-2.6.1`. This method, in turn, reaches the internal method `newInstance` in its call graph. When the library is upgraded to `protobuf-java-4.30.1`, the implementation of `newInstance` introduces a new unchecked exception—an `IllegalArgumentException`. Our tool initially flags this as a behavioural breaking change because the exception is newly introduced, and an interprocedural control-flow path exists from the client code to the exception site.

However, a closer inspection shows that this exception cannot be triggered by any value passed from the client. The internal method that throws the exception looks like this:

```
1  static CodedInputStream newInstance(
2      final byte[] buf, final int off, final int len, final boolean bufferIsImmutable) {
3    ArrayDecoder result = new ArrayDecoder(buf, off, len, bufferIsImmutable);
4    try {
5      result.pushLimit(len);
6    } catch (InvalidProtocolBufferException ex) {
7      // The only reason pushLimit() might throw an exception
8      // here is if len is negative. Normally pushLimit()'s
9      // parameter comes directly off the wire, so it's
10     // important to catch exceptions in case of corrupt or
11     // malicious data. However, in this case, we expect
12     // that len is not a user-supplied value, so we can
13     // assume that it being negative indicates a
14     // programming error. Therefore, throwing an unchecked
15     // exception is appropriate.
16     throw new IllegalArgumentException(ex);
17   }
18   return result;
19 }
```

[1]Fully-qualified name: method `getString(String)` returning a `String` on class `com.alibaba.fastjson.JSONObject`

[2]https://github.com/4ntoine/ServiceDiscovery-java

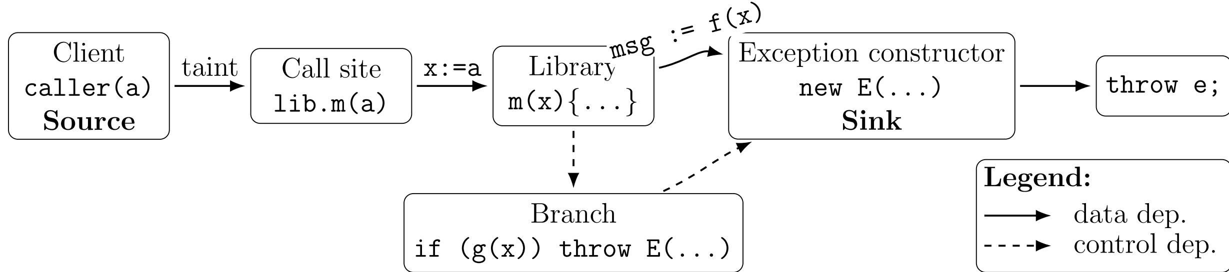

Figure 5.2: Taint propagation from a client parameter (source) to an exception site (sink) occurs via either *data dependence* (solid lines, e.g., a value passed into the exception constructor arguments) or *control-flow dependence* (dashed lines, e.g., a value influencing a branch that leads to the `throw` statement).

The library developer's comment states that this exception will never be thrown by this non-public method, essentially because `len` cannot be directly supplied by a client. Clients can only reach this `newInstance` method through methods that are part of `protobuf`'s public API. Our taint analysis confirms that no client-supplied value (source) flows into the `IllegalArgumentException` constructor (sink). We choose exception constructors as sinks because taintedness of the exception constructor means that the client-controlled value can affect the reachability of the exception, i.e. whether the exception might be thrown or not. Hence, taint analysis helps reason about whether the newly-added exception can actually cause a behavioural breaking change in the client.

We mark an exception site as *reachable* if either the client-supplied value is used as an argument for the exception constructor (explicit) or the client-supplied value influences the control flow reaching the `throw` statement (implicit). This approach ensures that we correctly identify both explicit data dependence and implicit control-flow dependence on the client-supplied value. Figure 5.2 illustrates these two ways that client-supplied values can reach an exception site.

For technical reasons related to FlowDroid, we automatically generate a *driver stub* for each value that the client supplies to the library. FlowDroid does not allow method parameters to be marked directly as taint sources. To work around this, we wrap each parameter in a synthetic method and mark its return value as a source.

This approach allows us to simulate tainted inputs and track their flow through the library method. Stub generation, implemented using SootUp, handles a variety of cases, including:

- Constructor methods (`<init>` using `new ClassName(...)`)
- Static and instance methods
- Void and non-void return types
- Primitive parameters (e.g., `int` → `0`)
- Object parameters (defaulted to `null`)

- Nested classes (converting $ to .)
- Multiple parameters (sources named SourceN(), where N is the parameter index)
- Overloaded methods (only one version retained)

An example of a driver stub follows.

```
public class DriverStub {
    public static java.lang.String source0() {
        return null;
    }

    public static void run() {
        int result = new com.alibaba.fastjson.JSONObject().
           getIntValue(source0());
    }
}
```

In the above example of the driver stub, we generate a stub for library `fastjson-1.2.58`'s public `getIntValue(String)` method[3]. This method has a `java.lang.String` parameter. We therefore declare a method named `source0` in the driver stub, and declare that method to be a source; FlowDroid then marks its return value as a source. We generate the way `getIntValue` is called based on the type of method it is, and obtain the properties of the method using SootUp.

In our analysis, we mark the parameters of library methods that are invoked by the client as taint sources (in the example in Chapter 3, the `HttpHost` constructor parameters), since these are the only values under the client's control. We also mark each exception identified in the Analysis Setup step as a potential taint sink. We use the taint analysis to estimate whether the client-supplied parameter values can trigger newly introduced exceptions. If they cannot, then the exception is effectively unreachable from the client, and thus does not constitute a behavioural breaking change.

Consider the following method from the `beam-sdks-java-core` library:

```
public static void applicableTo(PCollection<?> input) {
 WindowingStrategy<?, ?> ws = input.getWindowingStrategy();
 if (ws.getWindowFn() instanceof GlobalWindows
      && ws.getTrigger() instanceof DefaultTrigger
      && input.isBounded() != IsBounded.BOUNDED) {
  throw new IllegalStateException("...");
 }
}
```

[3]Fully-qualified name: method `getIntValue(String)`, returning a `int` on class `com.alibaba.fastjson.JSONObject`

In this example, the parameter `input` is the taint source, and the `newIllegalStateException()` is the sink (to be precise, the exception's constructor). The public `applicableTo(PCollection)`[4] method is used by the `0xdecaf/beam-enrichment-patterns`[5] client.

In terms of our methodology, for methods that appear in both the current and latest versions of the library, we compare the sets of unchecked exceptions that they throw, filter using taint analysis, and then diff to identify new exceptions (in the next stage). So, if the current version has unreachable exception E which becomes reachable in the latest version, but E was present in both versions, then we report E. (If a *method* exists in the current library version but is missing from the latest version, we exclude it from our analysis. Its removal may indicate a method signature-based breaking change, but those are handled by existing tools and lie outside the scope of our detection. Our work only detects changes to the set of exceptions that are thrown.)

We compare exceptions using both the exception type (e.g., `java.lang.IllegalArgumentException`) and the fully-qualified signature of the method in which the exception occurs. If, after removing all exceptions common to both versions, the method in the latest version still contains additional unchecked exceptions, we classify it as a method with a newly-added unchecked exception. Otherwise, we discard it from consideration; our technique sees no new exception-related behavioural breaking changes for this method.

After collecting the unchecked exceptions from both versions of the library, we store the list of exceptions present in each method in a `JSON` file. We use the following format to store this information:

```
[
    ...
    {"org.apache.http.HttpHost": [{
        "methodSignature": "<org.apache.http.HttpHost: void <init>(java.lang.String,int)>",
        "unchecked_exceptions": [
            "java.lang.IllegalArgumentException <org.apache.http.util.Args: java.lang.CharSequence containsNoBlanks(java.lang.CharSequence,java.lang.String)>",
            "java.lang.IllegalArgumentException <org.apache.http.util.Args: java.lang.CharSequence containsNoBlanks(java.lang.CharSequence,java.lang.String)>"
        ]
    }]},
    ...
]
```

[4] Fully-qualified name: method `applicableTo(PCollection)` returning a `void` on class `org.apache.beam.sdk.transforms.GroupByKe`

[5] https://github.com/0xdecaf/beam-enrichment-patterns

This `JSON` file contains a list of classes in the library. For each class, it includes a list of methods, and for each method, it stores two fields: `methodSignature`, which holds the method's signature, and `unchecked_exceptions`, which lists all unchecked exceptions associated with the method. The tool uses this `JSON` file to compare the current and latest versions of the library.

This output enables our tool to highlight call sites in the client application that may exhibit behavioural breaking changes, helping developers assess the impact of upgrading their dependencies.

5.4 Filtering Untriggerable Unchecked Exceptions

Based on the information collected about newly added unchecked exceptions, we use the previously generated client-to-library method mapping to determine which client methods invoke a library method that now throws a new unchecked exception. This step allows us to identify specific call sites in the client that may be affected by behavioural breaking changes introduced in the upgraded library version. The client-to-library method mapping is stored in `JSON` format in the following way:

```
1 [
2     {
3         "client_method": "<waditu.tushare.common.HTTParty:
             java.lang.String get(java.lang.String,java.lang.
             String)>",
4         "external_call": "<org.apache.http.util.EntityUtils:
             java.lang.String toString(org.apache.http.
             HttpEntity,java.lang.String)>"
5     }
6 ]
```

To validate the practical impact of these changes, we manually write test cases to assess whether the client can actually trigger the exception. Our goal is to write a test case that uses client code to trigger the exception.

We construct client-focussed test cases as follows. To understand the exception, we start from the client call site identified in the mapping and examine the library method that UnCheckGuard had flagged as containing a newly added unchecked exception. This information is available in the JSON output produced by our tool, which includes the exception type and the method signature in which it occurs. Given the exception type and method signature, we can easily find the exact exception-throwing line in the library. This enables a detailed inspection of how the exception is triggered.

To craft a test case, we start on the library side by first triggering the exception by directly calling the relevant library method, in the library context, with crafted parameters. If this call triggers the exception (as we would expect), we then proceed to construct a full test case that invokes the client method, propagating the same parameter values.

In some scenarios, we are unable to trigger the exception through the client due to certain code structures:

- The client may pass a hardcoded constant value to the library which does not trigger the exception.
- The client may apply explicit guards or checks before calling the affected library method.

There are thus at least two ways to fail in creating a test suite: (1) the client that we have will not trigger the exception because of how it uses the library; (2) no client can trigger the exception through the library's public API. Case (2) would be less common than case (1), since the library developers usually add an exception for a reason.

The potential failure to write a test case is similar in spirit to, for instance, security tools which report a number of potential vulnerabilities; the onus remains on the tool user to go from a potential vulnerability to proof-of-concept code.

In cases where no current client-based test case could possibly trigger the exception, this is still a situation where future library code changes (e.g., modifying a hardcoded value or removing a check) could make the call site vulnerable to the newly introduced exception. Ideally, the library developer ought to have added a description of this exception, and the circumstances under which it could be thrown, to the library method's documentation, as we have seen in the unreachable `InvalidProtocolBufferException` above.

We present an example of an untriggerable case which still passes the taint analysis. The client project `github.com/4ntoine/ServiceDiscovery-java` contains the following code:

```
if (serviceInfo.getPayload() != null)
 builder.setPayload(
   ByteString.copyFrom(serviceInfo.getPayload()));
```

In this case, the library method with the newly added unchecked exception is:

```
com.google.protobuf.ByteString.copyFrom(byte[])
```

The client uses version 2.6.1 of `protobuf-java`, while the latest version is 4.31.0. The newly added exception, `java.lang.NullPointerException`, is thrown in the latest library if a null value is passed to `copyFrom`. The relevant transitively-called code from the library is:

```
LiteralByteString(byte[] bytes) {
    if (bytes == null) {
        throw new NullPointerException();
    }
    this.bytes = bytes;
}
```

Although the latest version introduces a new unchecked exception, the client had already placed a guard condition, which was the first line above:

```
1  if (serviceInfo.getPayload() != null)
```

The guard condition does prevent the exception from being triggered by calling client methods. Therefore, we cannot generate a client-centric test case for this call site. However, we still report it, and we claim that it is potentially relevant. The reason that we report it is that we actually only analyze the clients to extract calls to the library, so that the client-side guard is not interesting to our analysis. Our taint analysis starts on the library side of the client/library interface.

In contrast, for cases where the client does not enforce such conditions and passes input parameters that can trigger the new exception, our experience has shown that we can generate a test case to demonstrate the behavioural breaking change. In these situations, the change is not merely hypothetical—it represents an actual, runtime-breaking behaviour that occurs when the latest library is used. These tests offer actionable insights to developers by highlighting call sites could possibly trigger newly added exceptions in new library versions.

Chapter 6

Results

As discussed in Chapter 4, we evaluated UnCheckGuard on 1011 Java-based clients from the DUETS dataset [13].

The goal of our tool is to detect whether a client calls a library method that, upon upgrading the library to a newer version, introduces a previously non-existent unchecked exception—potentially resulting in a behavioural breaking change.

We explore the following research questions:

RQ1: Do library clients call methods with new added exceptions, and is it possible for the clients to trigger these exceptions? Furthermore, is it possible to write client-focussed test cases that trigger the exceptions?

RQ2: For library changes that introduce triggerable new unchecked exceptions, under what circumstances do such exceptions occur (i.e. major/minor/patch versions)?

Table 6.1 summarizes our empirical findings about the prevalence of newly-added exceptions in our corpus and how their number changes as we perform more analysis stages.

Table 6.1: Exception Analysis Funnel

Stage	**Count**
Client invocations of external methods	15678
Exceptions passing taint analysis	1708

6.1 Client Calls to Newly-added Exceptions

Our evaluation includes 1011 client applications, which depend on 302 distinct libraries. Across these, we formed 352 client-library pairs in which the library had an available upgrade, each corresponding to a combination of a specific client and one of the libraries that

it depends on. Table 6.2 presents the client-library pairs, ordered in descending number of callsites that pass the taint analysis reachability filter; for each pair, it also presents the number of client callsites invoking library methods with newly-added exceptions.

UnCheckGuard detected 15678 callsites across these 352 pairs where the upgraded version of the library could throw a new unchecked exception. We initially tried to write test cases for these callsites but found ourselves unable to write test cases that could trigger the newly added unchecked exception. In most of these cases, we observed that the parameters responsible for triggering the exceptions were not the ones passed by the client to the library method. Hence, it was not possible to trigger all of these exceptions using the client's methods, even with a free choice of parameters to pass to the client code.

We therefore applied a taint-based reachability analysis to filter out cases that definitely could not result in actual runtime failures. After this filtering step, we identified 1708 callsites in total—spanning 120 distinct libraries—that appeared to potentially be affected by a newly added unchecked exception. As with the `protobuf` case in Chapter 5, which added a new-but-untriggerable unchecked exception, taint analysis played a crucial role in reducing the number of false positives.

To assess the real-world consequences of these remaining 1708 callsites, we manually constructed test cases. For 3 of the clients (out of 21 attempts), we were able to provide inputs that trigger the newly added exceptions, confirming that they represent real behavioural breaking changes.

In other cases, the exception was still untriggerable because the client passed hardcoded values or had safeguards like null checks. In these cases, it is possible that modifying the client might trigger the exception, but we considered that out of scope: we wanted it to be possible for client code, as written, to trigger the exception.

Answer RQ1: Yes, client applications do call methods with newly added unchecked exceptions. Out of 352 client-library pairs in our corpus, we identified 1708 callsites that reached newly-added exceptions, distributed across 136 of our 1011 clients. We were able to construct test cases that trigger the exception in 3 cases.

Table 6.2: Selected clients, libraries, versions, and counts of callsites reaching newly-added exceptions

Client	**Current Version**	**Latest Version**	**Calls**	**Reachable**
codes.brewing.flink				

Continued on next page

Client	Current Version	Latest Version	Calls	Reachable
examples-1.0-SNAPSHOT	commons-logging-1.1.1	commons-logging-1.1.3	3479	436
api-2.0.2	gson-2.3	gson-2.13.1	453	209
cosyan-0.0.1-SNAPSHOT	json-20180130	json-20250517	365	112
TopicModelingTool	junit-4.11	junit-4.13.2	308	78
android-facebook-1.6	android-1.6_r2	android-4.1.1.4	154	77
indextank-engine-1.0.0	commons-cli-1.2	commons-cli-1.10.0	328	51
commons-pipeline-1.0-SNAPSHOT	commons-digester-1.7	commons-digester-2.1	76	48
codes.brewing.flink examples-1.0-SNAPSHOT	commons-codec-1.3	commons-codec-1.4	47	38
mrdpatterns-1.0-SNAPSHOT	hadoop-core-1.1.1	hadoop-core-1.2.1	466	36
rehttp	xembly-0.31.1	xembly-0.32.2	61	36
indextank-engine-1.0.0	log4j-1.2.16	log4j-1.2.17	33	33
Timeline-2.0.0	tablestore-4.11.2	tablestore-5.17.6	479	32
HospitalAction-1.0	poi-5.2.2	poi-5.4.1	42	28
MavenProject-0.0.1-SNAPSHOT	selenium-api-3.141.59	selenium-api-4.35.0	120	24
amazon-kinesis-aggregators-.9.2.9	commons-logging-1.1.1	commons-logging-1.3.5	105	23
commons-pipeline-1.0-SNAPSHOT	commons-logging-1.0.4	commons-logging-1.3.5	186	21
v20-3.0.25	gson-2.8.2	gson-2.13.1	129	20
shadowsocks-netty-server-0.0.1	bcprov-jdk15on-1.52	bcprov-jdk15on-1.70	27	19
donors	commons-logging-1.1.3	commons-logging-1.2	43	16

Continued on next page

Client	Current Version	Latest Version	Calls	Reachable
indextank-engine-1.0.0	antlr-runtime-3.4	antlr-runtime-3.5.3	139	14
ripme-1.5.0	commons-cli-1.2	commons-cli-1.10.0	35	13
feeyo-redisproxy-1.9	commons-lang3-3.7	commons-lang3-3.18.0	26	13
cassandra-tutorial-1.0-SNAPSHOT	hector-core-1.0-2	hector-core-1.0-5	270	12
HttpAsyncClientUtils-1.0-SNAPSHOT	httpcore-4.4.6	httpcore-4.4.16	48	12
ecir-compression-1.0-SNAPSHOT	fastutil-6.5.15	fastutil-8.5.16	24	12
vTools-1.0	opennlp-tools-1.5.3	opennlp-tools-2.5.5	36	9
codes.brewing.flink examples-1.0-SNAPSHOT	jackson-core-2.1.1	jackson-core-2.7.4	32	9
android-facebook-1.6	json-20080701	json-20250517	15	9
cosyan-0.0.1-SNAPSHOT	commons-io-2.5	commons-io-2.20.0	19	8
server-1.0	cglib-nodep-3.2.8	cglib-nodep-3.3.0	12	8
mvbench-0.1.0-SNAPSHOT	metrics-core-3.1.0	metrics-core-4.2.33	44	7
codes.brewing.flink examples-1.0-SNAPSHOT	jackson-databind-2.1.1	jackson-databind-2.7.4	24	7
shadowsocks-java-1.0-SNAPSHOT	bcprov-jdk15on-1.56	bcprov-jdk15on-1.70	16	7
paystackjava-1.1.0	json-20170516	json-20250517	7	7
util-0.0.1	poi-3.17	poi-5.4.1	33	6
qagile-qainfolabs-0.0.1-SNAPSHOT	selenium-api-2.19.0	selenium-api-4.35.0	27	6

Continued on next page

Client	Current Version	Latest Version	Calls	Reachable
cosyan-0.0.1-SNAPSHOT	commons-csv-1.4	commons-csv-1.14.1	16	6
activemq-monitor-1.0	log4j-1.2.15	log4j-1.2.17	6	6
fileServer-1.0-SNAPSHOT	netty-all-4.0.32.Final	netty-all-5.0.0.Alpha2	217	5
HttpAsyncClientUtils-1.0-SNAPSHOT	httpclient-4.5.1	httpclient-4.5.14	54	5
natcross2-2.3.2	fastjson-2.0.9	fastjson-2.0.58	35	5
riak-hadoop-0.2-SNAPSHOT	hadoop-core-0.20.203.0	hadoop-core-1.2.1	22	5
DiceRelevancy Feedback-1.0	lucene-analyzers-common-6.3.0	lucene-analyzers-common-8.11.4	9	5
fileServer-1.0-SNAPSHOT	commons-beanutils-1.9.3	commons-beanutils-1.11.0	5	5
DovakinMQ-1.0-SNAPSHOT	netty-all-4.1.13.Final	netty-all-5.0.0.Alpha2	144	4
shadowsocks-java-1.0-SNAPSHOT	netty-all-4.1.42.Final	netty-all-5.0.0.Alpha2	114	4
qrcode-core-1.1	thumbnailator-0.4.8	thumbnailator-0.4.20	24	4
HospitalAction-1.0	itextpdf-5.5.13.3	itextpdf-5.5.13.4	18	4
mafia-maven-plugin-2.1.12-SNAPSHOT	commons-io-2.4	commons-io-2.20.0	8	4
simple-kafka-producer-pool-1.0-SNAPSHOT	kafka_2.10-0.8.0	kafka_2.10-0.10.2.2	7	4
Explainer-0.0.1-SNAPSHOT	spark-core_2.10-1.5.2	spark-core_2.10-2.2.3	7	4
gardener-1.0-SNAPSHOT	args4j-2.0.12	args4j-2.37	6	4

Continued on next page

Client	Current Version	Latest Version	Calls	Reachable
easy-rpc-1.0.0-RELEASE	protostuff-runtime-1.6.2	protostuff-runtime-1.8.0	4	4
MavenProject-0.0.1-SNAPSHOT	testng-6.9.10	testng-7.11.0	4	4
sailthru-java-client-2.4.2-SNAPSHOT	gson-2.8.6	gson-2.13.1	66	3
netty-server-test-1.0-SNAPSHOT	netty-all-4.1.0.Final	netty-all-5.0.0.Alpha2	53	3
api-2.0.2	okhttp-2.6.0	okhttp-2.7.5	31	3
gardener-1.0-SNAPSHOT	mongo-java-driver-2.2	mongo-java-driver-3.12.14	11	3
untitled2-1.0-SNAPSHOT	json-20141113	json-20250517	10	3
zpush-0.0.1-SNAPSHOT	netty-handler-4.0.27.Final	netty-handler-5.0.0.Alpha2	8	3
cryptolite-2.0.0-SNAPSHOT	commons-codec-1.11	commons-codec-1.19.0	5	3
html2image-2.0-SNAPSHOT	xercesImpl-2.9.1	xercesImpl-2.11.0	5	3
paymill-java-5.1.7-SNAPSHOT	commons-beanutils-1.9.2	commons-beanutils-1.11.0	3	3
lyrebird-java-client-1.1.6	json-path-2.3.0	json-path-2.9.0	3	3
DotGraphics-0.0.1	log4j-1.2.15	log4j-1.2.17	3	3
sonar-gitlab-plugin-4.1.0-SNAPSHOT	sonar-plugin-api-7.0	sonar-plugin-api-5.1	463	2
shadowsocks-netty-server-0.0.1	netty-all-4.1.5.Final	netty-all-5.0.0.Alpha2	58	2
spark-intercooler-1.0-SNAPSHOT	spark-core-2.5	spark-core-2.9.4	45	2
check-point-0.1.2	poi-3.17	poi-5.4.1	43	2

Continued on next page

Client	Current Version	Latest Version	Calls	Reachable
thick-0.0.1	netty-codec-http-4.0.13.Final	netty-codec-http-5.0.0.Alpha2	33	2
rojo-1.1.0	jedis-2.9.0	jedis-6.1.0	32	2
streams-1.0-SNAPSHOT	kafka-clients-0.10.1.1	kafka-clients-4.0.0	30	2
mockroservices-1.2.0	gson-2.8.9	gson-2.13.1	28	2
DotGraphics-0.0.1	httpcore-4.3	httpcore-4.4.16	28	2
kafka-metrics-reporter-1.2.1	kafka_2.9.2-0.8.1.1	kafka_2.9.2-0.8.2.2	26	2
epiphany	netty-all-4.1.7.Final	netty-all-5.0.0.Alpha2	26	2
qrcode-core-1.1	core-3.3.0	core-3.5.3	20	2
indextank-engine-1.0.0	json-simple-1.1	json-simple-1.1.1	19	2
qrcode-core-1.1	javase-3.3.0	javase-3.5.3	18	2
simple-kafka-producer-pool-1.0-SNAPSHOT	slf4j-api-1.6.3	slf4j-api-1.6.4	8	2
v20-3.0.25	httpcore-4.4.6	httpcore-4.4.16	6	2
flume-ng-failover-appender-0.0.1-SNAPSHOT	log4j-1.2.16	log4j-1.2.17	5	2
pi-1.0-SNAPSHOT	slf4j-api-1.7.7	slf4j-api-2.0.12	4	2
DiceRelevancyFeedback-1.0	slf4j-api-1.7.7	slf4j-api-2.0.13	3	2
tiny-spring-1.0-SNAPSHOT	cglib-2.2.2	cglib-3.3.0	2	2
ripme-1.5.0	commons-configuration-1.7	commons-configuration-1.10	2	2
de.flapdoodle.embed.				

Continued on next page

Client	Current Version	Latest Version	Calls	Reachable
redis-1.11.5-SNAPSHOT	jedis-2.9.0	jedis-6.1.0	2	2
gettingstarted-1.0-SNAPSHOT	json-20180813	json-20250517	2	2
simple-kafka-producer-pool-1.0-SNAPSHOT	json-simple-1.1	json-simple-1.1.1	2	2
FastRegex-1.0	junit-4.11	junit-4.13.2	2	2
ecir-compression-1.0-SNAPSHOT	junit-4.8.2	junit-4.13.2	2	2
ripme-1.5.0	jsoup-1.8.1	jsoup-1.21.1	458	1
admin-persistence-1.2.1-SNAPSHOT	javaee-api-7.0	javaee-api-8.0.1	53	1
streams-1.0-SNAPSHOT	kafka-streams-0.10.1.1	kafka-streams-4.0.0	46	1
redis-game-transaction-1.8.010	jedis-2.9.0	jedis-6.1.0	36	1
amazon-kinesis-aggregators-.9.2.9	amazon-kinesis-client-1.7.0	amazon-kinesis-client-1.15.3	30	1
Explainer-0.0.1-SNAPSHOT	spark-sql_2.10-1.5.2	spark-sql_2.10-2.2.3	29	1
spring-social-naver-1.0.2	spring-social-core-1.1.0.RELEASE	spring-social-core-1.1.6.RELEASE	28	1
jbosscc-needle-2.3-SNAPSHOT	hibernate-jpa-2.0-api-1.0.0.Final	hibernate-jpa-2.0-api-1.0.1.Final	26	1
util-0.0.1	javax.servlet-api-3.0.1	javax.servlet-api-4.0.1	25	1
sonar-stash-plugin-1.7.0-SNAPSHOT	async-http-client-2.8.1	async-http-client-3.0.2	23	1
at.bestsolution.releng.distrobuilder-0.0.1-SNAPSHOT	ant-1.8.4	ant-1.10.15	20	1

Continued on next page

Client	Current Version	Latest Version	Calls	Reachable
depview-0.2-SNAPSHOT	config-1.3.1	config-1.4.4	18	1
mafia-maven-plugin-2.1.12-SNAPSHOT	vtd-xml-2.13	vtd-xml-2.13.4	12	1
ipaTest-1.0-SNAPSHOT	dd-plist-1.16	dd-plist-1.28	11	1
jbosscc-needle-2.3-SNAPSHOT	junit-4.11	junit-4.13.2	11	1
Explainer-0.0.1-SNAPSHOT	spark-mllib_2.10-1.5.2	spark-mllib_2.10-2.2.3	11	1
aws-sample-1.0-SNAPSHOT	aws-java-sdk-core-1.11.534	aws-java-sdk-core-1.12.788	9	1
cosyan-0.0.1-SNAPSHOT	commonmark-0.11.0	commonmark-0.17.0	9	1
cosyan-0.0.1-SNAPSHOT	jetty-server-9.4.11.v20180605	jetty-server-11.0.25	9	1
thick-0.0.1	netty-buffer-4.0.13.Final	netty-buffer-5.0.0.Alpha2	8	1
simple-kafka-producer-pool-1.0-SNAPSHOT	curator-framework-1.0.1	curator-framework-1.3.3	7	1
SimpleTomcat-3.0	javax.servlet-api-3.0.1	javax.servlet-api-4.0.1	7	1
flume-ng-failover-appender-0.0.1-SNAPSHOT	flume-ng-sdk-1.4.0	flume-ng-sdk-1.11.0	5	1
benchmark-thrift-0.0.1	jcommander-1.78	jcommander-1.82	4	1
lyrebird-java-client-1.1.6	junit-4.12	junit-4.13.2	4	1
cosyan-0.0.1-SNAPSHOT	jetty-servlet-9.4.11.v20180605	jetty-servlet-11.0.25	3	1

Continued on next page

Client	Current Version	Latest Version	Calls	Reachable
dynamic-rules-1.0-SNAPSHOT	drools-templates-6.2.0.Final	drools-templates-10.1.0	2	1
lyrebird-java-client-1.1.6	jackson-databind-2.9.4	jackson-databind-2.19.0	2	1
money-transfer-1.0.0-SNAPSHOT	jetty-server-9.4.41.v20210516	jetty-server-12.1.0.beta3	2	1
money-transfer-1.0.0-SNAPSHOT	jetty-servlet-9.2.3.v20140905	jetty-servlet-11.0.25	2	1
netty-rest-0.110-SNAPSHOT	paranamer-2.8	paranamer-2.8.3	2	1
util-0.0.1	snakeyaml-1.26	snakeyaml-2.4	2	1
lyrebird-java-client-1.1.6	socket.io-client-1.0.0	socket.io-client-2.1.2	2	1
alltv-1.2.0	bcprov-jdk15on-1.58	bcprov-jdk15on-1.70	1	1
commons-pipeline-1.0-SNAPSHOT	commons-beanutils-1.6	commons-beanutils-1.8.3	1	1
server-1.0	commons-beanutils-1.9.2	commons-beanutils-1.11.0	1	1
tiny-spring-1.0-SNAPSHOT	commons-beanutils-1.9.3	commons-beanutils-1.11.0	1	1
java-plist-1.1-SNAPSHOT	commons-codec-1.8	commons-codec-1.19.0	1	1
oisdos-1.0	commons-codec-1.9	commons-codec-1.19.0	1	1
complex-1.0-SNAPSHOT	commons-io-2.1	commons-io-2.20.0	1	1
java-plist-1.1-SNAPSHOT	commons-lang3-3.1	commons-lang3-3.18.0	1	1
lyrebird-java-client-1.1.6	converter-jackson-2.5.0	converter-jackson-3.0.0	1	1

Continued on next page

Client	Current Version	Latest Version	Calls	Reachable
dataflow-log-analytics-1.0-SNAPSHOT	google-http-client-1.20.0	google-http-client-2.0.0	1	1
gossip-0.0.1-SNAPSHOT	log4j-1.2.16	log4j-1.2.17	1	1
buspass-ws-1.0.0-SNAPSHOT	spring-boot-1.2.1.RELEASE	spring-boot-1.3.8.RELEASE	1	1

6.2 Newly-added Unchecked Exceptions in Java Libraries

Semantic versioning [40] proposes that version numbers have three parts, $x.y.z$. According to semantic versioning, library developers are to change the major version x when an upgrade is breaking—that is, a client may have to modify their code to use the new versioning. Minor version upgrades (indicated by changes to y) may include new features, while patch upgrades (changes to z) fix bugs. We sought to investigate how often behavioural breaking changes (at least, the ones we can detect) occur in each of these types of changes.

Table 6.3: Distribution of reachable newly-added exceptions across version types

Version Type	Libraries
Major Version Change	50
Minor Version Change	57
Patch Version Change	14

Table 6.3 shows the distribution of newly-added exceptions reachable from clients, across upgrade types. Notably, 50 out of these 120 libraries introduced new unchecked exceptions as part of a major version bump. However, we also observed 14 cases in a patch version upgrade. While we are not making any broader claims about how often behavioural breaking changes occur in general, our results indicate that minor and patch upgrades do introduce behavioural breaking changes via unchecked exceptions which may affect clients—something that developers may not anticipate.

Answer RQ2: Java libraries introduce newly added unchecked client-relevant exceptions across versions frequently enough to be relevant to clients. We found newly added unchecked exceptions in 120 out of 302 distinct libraries (39.7%). These changes in major version upgrades (50 times), minor version upgrades (57 times), and patch (14 times) version upgrades (e.g., `httpcore-4.4.6` → `httpcore-4.4.16`).

6.3 Discussion: Developer-Facing Implications

Behavioural breaking changes caused by unchecked exceptions during API evolution are particularly dangerous. Such changes do not show up at compile time, and they do not affect method signatures, which means that the existing tools that we are aware of cannot detect them. For instance, both japicmp and Revapi, widely used tools for detecting breaking changes, focus on syntactic differences in method signatures. While they can both flag checked exceptions—since they appear in method declarations—they do not analyze the method implementations, and thus have no way of identifying newly added unchecked exceptions. As a result, developers who rely solely on either japicmp or revapi could remain unaware of serious runtime-breaking issues.

Some tools have tried to tackle the challenge of behavioural breaking changes. CompCheck [49], for example, works by identifying test cases in some clients and reusing them for others with similar API usage. But this approach depends heavily on the presence of thorough test suites. Most clients that we have looked at do not have such comprehensive coverage, especially not for edge cases involving unchecked exceptions.

This is where UnCheckGuard steps in. Unlike existing work, it does not rely on existing test cases. Instead, it compares the old and new versions of a library using static analysis to detect newly added unchecked exceptions, and then runs taint analysis to filter out changes that do not affect the client. By avoiding the need for a test suite, it can reveal behavioural breaking changes that other tools overlook.

In doing so, UnCheckGuard addresses an important gap. It gives developers visibility into a class of breaking changes that are easy to miss but costly in practice—helping them catch potential failures early, before they reach production.

Chapter 7

Related Work

While much program analysis research considers a single version of a software artifact, some related work treats changes between versions, and we discuss the related work in that area. We also discuss empirical efforts to detect and empirically survey the prevalence of and reasons for breaking changes.

Logozzo et al [32] proposed the concept of verification modulo versions. Like us, verification modulo versions observes that program verification needs to recognize that software evolves over time and that verification tools must take this into account—in particular, a developer often wants to know about potential verification issues unique to new code, rather than re-triaging issues previously reported. A fundamental difference between their work and ours is that we put the interface between the client and the library at the centre of our approach, and ensure that changes in the library must be visible to the client before we report them, while the verification modulo versions approach aims to detect behavioural differences between two versions of some software.

Møller et al [35] propose a domain-specific language for JavaScript library developers to use to indicate to client developers what has changed in a new version of their library. Our work addresses a specific subset of the breaking changes problem but automatically deduces changes in the library that are relevant to a particular client. It does not require additional work on the part of the library developer. More generally, and at the same time, Lam et al [30] proposed the development of semantic version calculators, including the usage of both traditional and lightweight contracts for libraries, to allow library developers to declare, and client developers to understand, the impact of potential breaking changes in libraries.

Jayasuriya et al [24, 22] investigate the prevalence of breaking changes in the wild. In principle, under semantic versioning [40], library developers ought to indicate breaking changes by incrementing the major version number (i.e. the first number in the version triplet); however, Jayasuriya et al found that 41.58% of (syntactic) breaking changes were not identified as such (our comparable number is 57/120, or 47.5%), and that 11.58% of changes were breaking.

Chenguang et al. [49] propose a tool, CompCheck, for detecting behavioural breaking changes. It uses both static and dynamic techniques to identify such changes. The approach

begins by locating test cases that pass for the current version of the library used by the client but fail after the library is upgraded. CompCheck relies heavily on the presence of existing test cases to detect behavioural breaking changes. It then extracts the usage pattern of the API in the failing test case and uses this pattern to identify other clients who use the same library in a similar manner. Here, the original client refers to the client whose test case fails after the upgrade. CompCheck generates a new test case for each newly identified client using EvoSuite [18]. In contrast, our tool does not rely on existing test cases to discover behavioural breaking changes caused by newly added exceptions. It also does not depend on tools like EvoSuite for test generation. Instead, we use taint analysis to filter out newly added exceptions that the client cannot trigger.

We have proposed a static approach to detecting breaking changes. Mujahid et al [33] proposed a dynamic approach to this problem. Their goal is to answer the question of whether a new version includes breaking changes or not, and they combine tests from "the crowd" (a collection of other projects) to decide the question, finding that such tests found breaking changes 60% of the time. Our approach is much more specific to a particular library/client pair, and aims to detect if library X's upgrade may break client Y. More like us, Jayasuriya et al [23] also use a dynamic approach (compared to our static approach) on a client/library pair to detect behavioural breaking changes in the client using its tests, finding that 2.30% of library updates broke the client, as witnessed by a particular test.

In terms of better understanding why breaking changes exist, Kong et al [27] analyzed the reasons that library developers introduced breaking changes (reducing code redundancy, improving identifier names, and improving API design) and proposed a taxonomy of types of changes.

Chapter 8

Conclusion

In this work, we demonstrated the impact of behavioural breaking changes caused by newly added unchecked exceptions in client applications. These changes are particularly difficult to detect, as they evade Java's compile-time checks and are not reflected in API signatures.

We introduced UnCheckGuard, a static analysis tool designed to detect such exceptions and help client developers avoid behavioural breaking changes. By combining extracted information with taint analysis, UnCheckGuard filters out unreachable exceptions, focusing only on those that are actually triggerable by client inputs.

We evaluated 352 library–client pairs from the DUETS dataset. Our tool flagged 15678 client callsites that invoked external methods. After applying taint analysis, we reduced this to 1708 callsites that could potentially trigger an exception at runtime. We wrote manual test cases for these callsites and confirmed that 3 of them resulted in real behavioural breaking changes.

These 1708 callsites came from 120 distinct libraries. Of those 120 libraries, 50 introduced newly added unchecked exceptions during a major version upgrade, 57 introduced it during a minor version upgrade and the rest during patch version upgrade. While we are not making any claims about how frequently BBCs happen in general, but our results indicate that minor and patch upgrades do introduce BBCs by addition of unchecked exceptions.

UnCheckGuard addresses a concerning gap in existing tools by targeting behavioural breaking changes due to unchecked exceptions. By statically analyzing both the library and client, it provides an effective way to catch runtime issues early and improve software robustness.

References

[1] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical Analysis of Security Vulnerabilities in Python Packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 446–457, 2021.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery.

[3] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K. Roy, and Kevin A. Schneider. How Developers Use Exception Handling in Java? In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 516–519, 2016.

[4] Ethan Bommarito and Michael J. Bommarito II. An empirical analysis of the Python Package Index (PyPI). *CoRR*, abs/1907.11073, 2019.

[5] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. APIDiff: Detecting API breaking changes. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*, pages 507–511, 2018.

[6] Oracle Corporation. Java se development kit. https://www.oracle.com/java/technologies/javase-downloads.html, 2024. Version 21.

[7] Russ Cox. Surviving software dependencies. *Communications of the ACM*, 62(9):36–43, August 2019.

[8] Andreas Dann, Ben Hermann, and Eric Bodden. UpCy—safely updating outdated dependencies. In *ICSE '23: Proceedings of the 45th International Conference on Software Engineering*, pages 233–244, 2023.

[9] Guilherme B. de Pádua and Weiyi Shang. Revisiting exception handling practices with exception flow analysis. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 11–20, 2017.

[10] Alexandre Decan, Tom Mens, and Eleni Constantinou. An empirical comparison of dependency issues in OSS packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, 2019.

[11] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200, 2017.

[12] Jens Dietrich, Kamil Jezes, and Premek Brada. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE 14)*, pages 64–73, 2014.

[13] Thomas Durieux, César Soto-Valero, and Benoit Baudry. Duets: A dataset of reproducible pairs of Java library-clients. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 545–549, 2021.

[14] Ecma International. Ecmascript language specification. Technical Report ECMA-262, Ecma International, 2024.

[15] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 559–563, 2018.

[16] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 791–796, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Python Software Foundation. Python: A dynamic, open source programming language. https://www.python.org, 2023. Version 3.11.

[18] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[19] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java™ Language Specification*. Oracle America, Inc., java se 17 edition, 2021.

[20] Stefanus A. Haryono, Hong Jin Kang, Abhishek Sharma, Asankhaya Sharma, Andrew Santosa, Ang Ming Yi, and David Lo. Automated identification of libraries from vulnerability data: Can we do better? In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, pages 178–189, 2022.

[21] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empirical Software Engineering*, 27(4):90, 2022.

[22] Dhanushka Jayasuriya, Samuel Ou, Saakshi Hegde, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. An extended study of syntactic breaking changes in the wild. *Empirical Software Engineering*, 30(2), December 2024.

[23] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. Understanding the impact of APIs behavioral breaking changes on client applications. *Proceedings of the ACM on Software Engineering*, 1(FSE):1238–1261, July 2024.

[24] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. Understanding breaking changes in the wild. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1433–1444, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. Sootup: A redesign of the Soot Static Analysis Framework. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–247, Cham, 2024. Springer Nature Switzerland.

[26] Mehdi Keshani, Georgios Gousios, and Sebastian Proksch. Frankenstein: fast and lightweight call graph generation for software builds. *Empirical Software Engineering*, 29(1), 2024.

[27] Denzhen Kong, Jiakun Liu, Lingfeng Bao, and David Lo. Toward better comprehension of breaking changes in the npm ecosystem. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–23, 2025.

[28] V. V. Kuliamin. A survey of software dynamic analysis methods. *Programming and Computer Software*, 50(1):90–114, February 2024.

[29] Karim Lakhani and Eric Hippel. How open source software works: 'free' user-to-user assistance? *Research Policy*, 32:923–943, 05 2000.

[30] Patrick Lam, Jens Dietrich, and David J. Pearce. Putting the semantics into semantic versioning. In *Onward! Essays*, 2020.

[31] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, jan 2015.

[32] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. Verification modulo versions: Towards usable verification. In *PLDI*, 2014.

[33] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. Using others' tests to identify breaking updates. In *17th International Conference on Mining Software Repositories (MSR '20)*, pages 466–476, 2020.

[34] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 228–241, New York, NY, USA, 1999. Association for Computing Machinery.

[35] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. Detecting locations in JavaScript programs affected by breaking library changes. In *Proceedings of the ACM on Programming Languages*, volume 4, pages 1–25, November 2020.

[36] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of exception handling patterns in Java projects: an empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 500–503, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of exception handling patterns in Java projects: an empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 500–503, New York, NY, USA, 2016. Association for Computing Machinery.

[38] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS Symposium*, 2005.

[39] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering*, 48(5):1592–1609, 2020.

[40] Tom Preston-Werner. Semantic versioning 2.0.0. https://semver.org, 2023.

[41] Md Shahidur Rahaman, Agm Islam, Tomas Cerny, and Shaun Hutton. Static-analysis-based solutions to security challenges in cloud-native systems: Systematic mapping study. *Sensors*, 23(4), 2023.

[42] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soederberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*, 2015.

[43] Dêmora Sousa, Paulo Maia, Lincoln Rocha, and Windson Viana. Studying the evolution of exception handling anti-patterns in a long-lived large-scale project. *Journal of the Brazilian Computer Society*, 26, 12 2020.

[44] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[45] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in Java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020.

[46] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. Understanding the Threats of Upstream Vulnerabilities to Downstream Projects in the Maven Ecosystem. In *ICSE '23: Proceedings of the 45th International Conference on Software Engineering*, pages 1046–1058, 2023.

[47] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. Formal framework for managing software dependencies in open-source software. *Information and Software Technology*, 108:170–186, 2019.

[48] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. ATVHunter: Reliable version detection of third-party libraries for vulnerability identification in Android applications. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1695–1707. IEEE Press, 2021.

[49] Chenguang Zhu, Mengshi Zhang, Xiuheng Wu, Xiufeng Xu, and Yi Li. Client-specific upgrade compatibility checking via knowledge-guided discovery. *ACM Transactions on Software Engineering and Methodology*, 32(4), May 2023.