Detecting Exception-Related Behavioural Breaking Changes with UnCheckGuard

Vinayak Sharma*, Patrick Lam†
*University of Waterloo; *vinayak.sharma1@uwaterloo.ca*
†University of Waterloo; *patrick.lam@uwaterloo.ca*

***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.

***Index Terms*—client/library interactions, behavioural breaking changes, exceptions, static analysis**

I. Introduction

The use of libraries developed by others is ubiquitous in modern software development [1, 2]. Libraries enable developers to include functionality in their own client software without having to implement it themselves. However, libraries developed by others are also updated by others, on schedules that are not controlled by the client developers.

Especially when one is developing software that is exposed to the Internet, one has a responsibility to incorporate security updates for the libraries that one is using as a client [3], or else risk vulnerabilities being exposed in one's software [4, 5, 6]. The obligation to update libraries is a form of technical debt that accrues automatically with the passage of time.

However, upgrading libraries is not painless [7, 8, 9]: new versions of libraries may include breaking Application Programming Interface (API) changes [10], requiring developers to verify that their own client code continues working with the new library versions. This is inconvenient at best and can require nontrivial amounts of software development at worst, often without the reward of useful new features for the client software—reacting to upgrades just allows the client software to continue working, in a hopefully less-vulnerable state.

Compilers and simple static checkers (including japicmp[1] and Revapi[2] for Java as well as [11, 12]) can verify the absence of syntactic breaking changes in libraries, e.g. changes to signatures of public methods, retractions of formerly-existing methods, or even syntactic changes to library method implementations. 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.

In this work, we contribute a novel way to detect one type of behavioural breaking change in a library. Our work enables client developers to inspect relevant changes to the set of exceptions that may be thrown by a Java library, particularly by the APIs that are actually used by specific client code. A new exception thrown by a library constitutes a breaking change; uncaught exceptions can cause the client to crash or to exhibit unexpected behaviour.

Although developers tend to ignore even checked exceptions [13], we contend that incrementally informing developers only about relevant newly-added exceptions is more likely to result in developer action, consistent with the design principles of Google's Tricorder tool [14]. Thus, we leverage taint analysis to reduce the number of irrelevant reports that we report to client developers. We aim to show only changed library APIs that may realistically throw new exceptions in updated versions of client code, minimizing the number of false positives [15, 16]. We hope that our reports enable client developers to 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, including exceptions that are 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 [17] 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

II. Background

We define some concepts that underpin our approach to detecting behavioural breaking changes caused by newly added exceptions.

Taint Analysis. Taint analysis is a program analysis technique which can be implemented statically [18] or dynamically [19]. 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.

SootUp. SootUp [20] is a respin of the Soot [21] 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 [22] 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 [21] 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.

III. Motivating Example

We continue with a motivating example drawn from the DUETS collection [17] of client/library pairs.

We start with our client, `HttpAsyncClientUtils`, which is one of the clients in DUETS. This client declares a dependency on version 4.4.6 of the `httpcore` library[3]. Since the release of the version of `HttpAsyncClientUtils` that we are using, the `httpcore` developers have released a number of new versions, and at the time of writing, the latest version of `httpcore` is 4.4.16[4].

A revision of the `httpcore` library between 4.4.6 and 4.4.16 adds a check for an error condition. If the condition evaluates to true, the library method will explicitly throw an `IllegalArgumentException`. The client, `HttpAsyncClientUtils`[5], calls the relevant part of the library, and thus may be affected by the new exception. We

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

[4]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.

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

explain how UnCheckGuard finds this exception (and how it avoids some false positives).

a) Library: All constructors for the `org.apache.http.HttpHost` class transitively call the static method `Args.containsNoBlanks()`. Between version 4.4.6 and version 4.4.16, the `httpcore` developers added the following lines of code to `containsNoBlanks()`:

```
if (argument.length() == 0) {
   throw new IllegalArgumentException
                (name + "␣may␣not␣be␣empty");
}
```

Specifically, all `HttpHost` constructors take a `hostname` parameter and call `containsNoBlanks()` with that parameter (to check that it contains no blanks). It is therefore possible to trigger this newly-thrown exception in a client by attempting to instantiate a new `HttpHost` object and passing it an empty `hostname`.

Our UnCheckGuard tool analyzes the change in `httpcore` and finds that, in version 4.4.16, all of the `HttpHost` constructors may now throw an `IllegalArgumentException` via the `containsNoBlanks()` method. This exception was not thrown in 4.4.6.

To detect this change, UnCheckGuard processes JAR files for both `httpcore-4.4.6` and `httpcore-4.4.16`. It uses SootUp [20] to construct a call graph using Class Hierarchy Analysis (CHA) starting from the public `<init>(String, int)`[6] constructor on `HttpHost` and identifies the set of all methods transitively reachable by the client (which we will discuss below). UnCheckGuard then collects all unchecked exceptions thrown within this set of reachable methods, for both library versions.

b) Client: A newly-added exception is only relevant to a particular client if that client may potentially trigger that exception. We found that, often, a client will call a library, and the library code really does contain a newly-added exception, but there is no way for the *client* to cause the library to reach that exception. But, in this case, it turns out that our `HttpAsyncClientUtils` client has reachable code from its public `createAsyncClient(boolean)`[7] method that creates an `HttpHost` with an empty `host`. This method takes a `proxy` parameter and contains 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 {
  // ...
}
```

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

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

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

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

where `host` is a private field initialized to the empty string. Thus, calling `createAsyncClient(true)` triggers an exception when executed with `httpcore` version 4.4.16 but not with 4.4.6. Conversely, one can imagine a library design where `HttpHost` objects are always initialized with a `hostname` of `"localhost"`, such that the newly-added exception would not be triggerable.

To detect that our `HttpAsyncClientUtils` client calls a method from `httpcore-4.4.6` which, upon upgrading to `httpcore-4.4.16`, may throw a new unchecked exception, UnCheckGuard begins by identifying all external library methods invoked anywhere in the client. It then analyzes both the current and the latest versions of the library to determine whether any newly introduced unchecked exceptions are reachable from the client's code. Here, reachability means that the client can trigger the exception in the library on some execution of the program, using values it passes to the library as parameters.

To check if the client-supplied values can reach the exception-throwing site, we use FlowDroid [22]'s taint analysis. Taint analysis is essential in this scenario because the existence of a control-flow path from the client callsite to an exception-throwing statement is not sufficient to conclude that the exception is actually triggerable by the client. As alluded to above, we found that many such paths may exist in a library, but the path conditions leading to the exception might depend entirely on internal library values, rather than on client-supplied inputs; it is impossible for our client to cause the execution of any path that triggers the exception. In our experience, taint analysis helps distinguish actual behavioural breaking changes from false positives.

Specifically, we use taint analysis to track whether any client-supplied method parameters to library calls (source) can propagate to the exception object's constructor (sink). If taint analysis determines that no client-supplied input flows into the exception-triggering logic, then we can conclude that the newly added exception will not cause a behavioural breaking change, and we do not report that exception.

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, 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");
}
```

IV. Data Collection

This section describes the systematic approach we used to construct the dataset for our study on behavioural incompatibilities caused by newly added unchecked exceptions in upgraded Java libraries.

To begin our analysis, we first collected suitable client projects. We used the DUETS dataset [17], 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[8], 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.

V. Methodology

The previous section, Section IV, described how we collected the clients. 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 Section IV and collecting all the clients that have to be analyzed, we next analyze all the external method calls made by the client, in both the current (pre-upgrade) version and the latest version. This allows us to compare their behaviour across versions. 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.

A. Analysis Setup

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

1) Library Version Resolution: Our tool relies on analyzing both the version of the library currently used by the client and the latest available version (as stored in the Maven Central Repository). To collect the current version, we run the Maven command `mvn dependency:copy-dependencies`, which downloads all the dependencies declared in the client's build configuration.

To obtain the latest versions of these dependencies, we run the following Maven command:

```
mvn org.codehaus.mojo:versions-maven-plugin
    :2.18.0:use-latest-versions
```

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.

After acquiring relevant versions of the libraries on which the client depends, we extract all external method invocations made by the client.

2) Mapping External Method Invocations to Library Methods: We use SootUp [20] 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

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

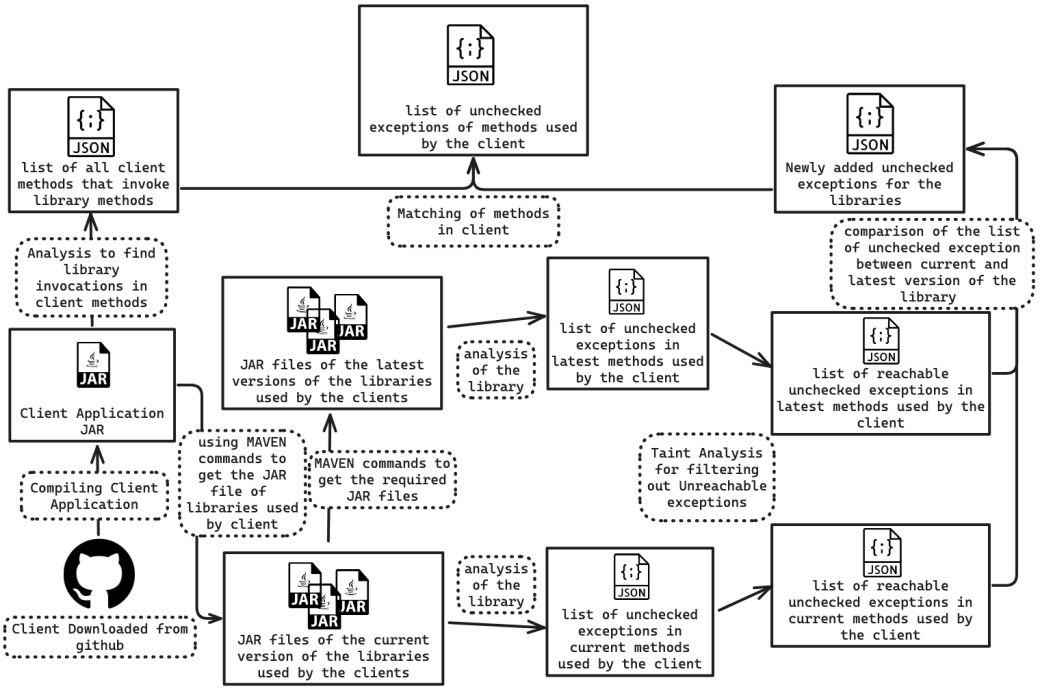

Fig. 1: Pipeline of UnCheckGuard for detecting behavioural breaking changes due to newly added unchecked exceptions.

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 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.

B. 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.

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 [23] answer possible, absent reflection and other dynamic features. Thus, it tends to over-approximate and report unreachable method calls. For example, in one case, CHA identified a path from the public `getString(String)`[9] 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.

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 [22], a static taint analysis framework.

Consider the following case from our corpus. The client `4ntoine/ServiceDiscovery-java`[10] uses the method `copyFromUtf8(String)` from the library

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

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

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:

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

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 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.

In our analysis, we mark the parameters of library methods that are invoked by the client as taint sources (in the example in Section III, 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("...");
 }
}
```

In this example, the parameter input is the taint source, and the new IllegalStateException() is the sink (to be precise, the exception's constructor). The public applicableTo(PCollection)[11] method is used by the 0xdecaf/beam-enrichment-patterns[12] 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 that are deemed reachable by taint analysis, and identify new exceptions. (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.

C. 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.

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.

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

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

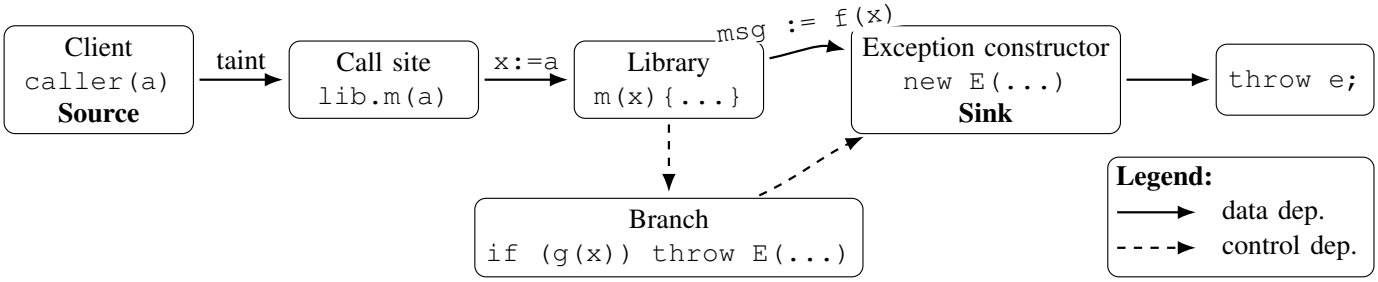

Fig. 2: Taint propagation from a client parameter (source) to an exception site (sink) occurs via either *data dependence* (solid lines, e.g., a value passed into the exception constructor arguments) or *control-flow dependence* (dashed lines, e.g., a value influencing a branch that leads to the `throw` statement).

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:

```
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.

VI. Results

As discussed in Section IV, we evaluated UnCheckGuard on 1011 Java-based clients from the DUETS dataset [17].

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 I 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 I: Exception Analysis Funnel

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

A. 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 III presents the top 15 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 ourself all-too-often unable to write a test case 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 Section V, 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 II: Distribution of reachable newly-added exceptions across version types

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

B. Newly-added Unchecked Exceptions in Java Libraries

Semantic versioning [24] 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 II 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.

TABLE III: Selected clients, libraries, versions, and counts of callsites reaching newly-added exceptions

Client	Current Version	Latest Version	Number of Callsites	Reachable Callsites
codes.brewing.flinkexamples-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.flinkexamples-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

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`).

C. 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 [25], 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.

VII. 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 some 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 [26] 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 [27] 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 [28] 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 [29, 30] investigate the prevalence of breaking changes in the wild. In principle, under semantic versioning [24], 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.

We have proposed a static approach to detecting breaking changes. Mujahid et al [31] 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 [32] 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 [33] 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.

VIII. 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.

In our evaluation of 352 library–client pairs from the DUETS dataset, we identified 1708 callsites affected by newly introduced unchecked exceptions. Notably, these issues arose not only in major library updates but also in a patch version update—highlighting the risk that developers may unknowingly introduce runtime failures even during seemingly safe updates.

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.

Acknowledgements. We thank the anonymous reviewers as well as Weiyi (Ian) Shang and Derek Rayside for helpful comments, particularly the incitement to increase the size of the evaluation, making the results significantly stronger than in the initial submission.

References

[1] K. Huang, B. Chen, C. Xu, Y. Wang, B. Shi, X. Peng, Y. Wu, and Y. Liu, "Characterizing usages, updates and risks of third-party libraries in Java projects," *Empirical Software Engineering*, vol. 27, no. 4, p. 90, 2022.

[2] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. 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)*. IEEE, 2020, pp. 35–45.

[3] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zou, and H. Jin, "Understanding the threats of upstream vulnerabilities to downstream projects in the Maven ecosystem," in *Proc. ICSE 2023*, 2023, pp. 1046–1058.

[4] S. A. Haryono, H. J. Kang, A. Sharma, A. Sharma, A. Santosa, A. M. Yi, and D. Lo, "Automated identification of libraries from vulnerability data: Can we do better?" in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, 2022, pp. 178–189.

[5] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. 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*, ser. ICSE '21. IEEE Press, 2021, p. 1695–1707. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00150

[6] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in Python packages," *Empirical Software Engineering*, vol. 28, no. 3, p. 59, 2023.

[7] R. Elizalde Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. 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)*, 2018, pp. 559–563.

[8] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. 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*, 2017, pp. 2187–2200.

[9] A. Dann, B. Hermann, and E. Bodden, "UpCy—safely updating outdated dependencies," in *ICSE '23: Proceedings of the 45th International Conference on Software Engineering*, 2023, pp. 233–244.

[10] J. Dietrich, K. Jezes, and P. 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)*, 2014, pp. 64–73.

[11] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "APIDiff: Detecting API breaking changes," in *25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*, 2018, pp. 507–511.

[12] D. Foo, H. Chua, J. Yeo, A. M. Yi, and A. Sharma, "Efficient static checking of library updates," in *ESEC/FSE '18*, 2018.

[13] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in Java projects: an empirical study," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 500–503. [Online]. Available: https://doi.org/10.1145/2901739.2903499

[14] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *International Conference on Software Engineering (ICSE)*, 2015.

[15] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1592–1609, 2020.

[16] ——, "Vulnerable open source dependencies: Counting those that matter," in *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, 2018, pp. 1–10.

[17] T. Durieux, C. Soto-Valero, and B. Baudry, "Duets: A dataset of reproducible pairs of Java library-clients," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 545–549.

[18] A. C. Myers, "JFlow: practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 228–241. [Online]. Available: https://doi.org/10.1145/292540.292561

[19] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *NDSS Symposium*, 2005.

[20] K. Karakaya, S. Schott, J. Klauke, E. Bodden, M. Schmidt, L. Luo, and D. He, "Sootup: A redesign of the soot static analysis framework," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 229–247.

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

[22] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. 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*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[23] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, no. 2, p. 44–46, jan 2015. [Online]. Available: https://doi.org/10.1145/2644805

[24] T. Preston-Werner, "Semantic versioning 2.0.0," https://semver.org, 2023.

[25] C. Zhu, M. Zhang, X. Wu, X. Xu, and Y. Li, "Client-specific upgrade compatibility checking via knowledge-guided discovery," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, May 2023. [Online]. Available: https://doi.org/10.1145/3582569

[26] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear, "Verification modulo versions: Towards usable verification," in *PLDI*, 2014.

[27] A. Møller, B. B. Nielsen, and M. T. Torp, "Detecting locations in JavaScript programs affected by breaking library changes," in *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, November 2020, pp. 1–25.

[28] P. Lam, J. Dietrich, and D. J. Pearce, "Putting the semantics into semantic versioning," in *Onward! Essays*, 2020.

[29] D. Jayasuriya, V. Terragni, J. Dietrich, S. Ou, and K. Blincoe, "Understanding breaking changes in the wild," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1433–1444. [Online]. Available: https://doi.org/10.1145/3597926.3598147

[30] D. Jayasuriya, S. Ou, S. Hegde, V. Terragni, J. Dietrich, and K. Blincoe, "An extended study of syntactic breaking changes in the wild," *Empirical Software Engineering*, vol. 30, no. 2, December 2024.

[31] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, "Using others' tests to identify breaking updates," in *17th International Conference on Mining Software Repositories (MSR '20)*, 2020, pp. 466–476.

[32] D. Jayasuriya, V. Terragni, J. Dietrich, and K. Blincoe,

"Understanding the impact of APIs behavioral breaking changes on client applications," *Proceedings of the ACM on Software Engineering*, vol. 1, July 2024.

[33] D. Kong, J. Liu, L. Bao, and D. Lo, "Toward better comprehension of breaking changes in the NPM ecosystem," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–23, 2025.