Understanding the Impact of APIs Behavioral Breaking Changes on Client Applications

DHANUSHKA JAYASURIYA, University of Auckland, New Zealand
VALERIO TERRAGNI, University of Auckland, New Zealand
JENS DIETRICH, Victoria University of Wellington, New Zealand
KELLY BLINCOE, University of Auckland, New Zealand

Libraries play a significant role in software development as they provide reusable functionality, which helps expedite the development process. As libraries evolve, they release new versions with optimisations like new functionality, bug fixes, and patches for known security vulnerabilities. To obtain these optimisations, the client applications that depend on these libraries must update to use the latest version. However, this can cause software failures in the clients if the update includes breaking changes. These breaking changes can be divided into syntactic and semantic (behavioral) breaking changes. While there has been considerable research on syntactic breaking changes introduced between library updates and their impact on client projects, there is a notable lack of research regarding behavioral breaking changes introduced during library updates and their impacts on clients. We conducted an empirical analysis to identify the impact behavioral breaking changes have on clients by examining the impact of dependency updates on client test suites. We examined a set of java projects built using Maven, which included 30,548 dependencies under 8,086 Maven artifacts. We automatically updated out-of-date dependencies and ran the client test suites. We found that 2.30% of these updates had behavioral breaking changes that impacted client tests. Our results show that most breaking changes were introduced during a non-`Major` dependency update, violating the semantic versioning scheme. We further analyzed the effects these behavioral breaking changes have on client tests. We present a taxonomy of effects related to these changes, which we broadly categorize as `Test Failures` and `Test Errors`. Our results further indicate that the library developers did not adequately document the exceptions thrown due to precondition violations.

CCS Concepts: • **Software and its engineering → Reusability**; **Software libraries and repositories**; **Maintaining software**; **Software evolution**.

Additional Key Words and Phrases: software libraries, software dependency, breaking changes, software evolution, behavioral breaking changes

ACM Reference Format:
Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. 2024. Understanding the Impact of APIs Behavioral Breaking Changes on Client Applications. *Proc. ACM Softw. Eng.* 1, FSE, Article 56 (July 2024), 23 pages. https://doi.org/10.1145/3643782

1 INTRODUCTION

Software libraries evolve rapidly, releasing new versions comprising new functionality, improvements, or bug fixes [9, 45]. These functionalities are exposed as Application Programming Interfaces (APIs) to be used by clients [64]. The API declares a contract established between the library and the client, that needs to be managed by both parties [6]. When the libraries release new versions, clients that depend on these libraries must keep the dependencies up-to-date to benefit

Authors' addresses: Dhanushka Jayasuriya, University of Auckland, Auckland, New Zealand, djay392@aucklanduni.ac.nz; Valerio Terragni, University of Auckland, Auckland, New Zealand, v.terragni@auckland.ac.nz; Jens Dietrich, Victoria University of Wellington, Wellington, New Zealand, jens.dietrich@vuw.ac.nz; Kelly Blincoe, University of Auckland, Auckland, New Zealand, k.blincoe@auckland.ac.nz.

from the changes in these new releases. However, updating dependencies can be difficult if the update breaks the API contract and contains backward incompatible changes known as Breaking Changes (BCs) [25, 52]. Previous research has indicated that the cost associated with addressing potential breaking changes can lead to client developers being hesitant to update their project dependencies [18, 33, 39, 40, 43, 61, 65]. BCs related to source and binary incompatibilities, known as syntactic BCs, are visible immediately if the application encounters compilation errors or issues arise while loading client code and library binaries. Behavioral BCs (BBCs), also known as semantic BCs, can only become apparent if the application compiles successfully but behaves differently than expected, which could lead to incorrect results or unsatisfactory performance [7, 19, 45]. Therefore, it is helpful for developers of the client projects if the libraries indicated the potential contract violations between two versions which introduce BBCs.

The practice of semantic versioning [50] is commonly employed by developers of software libraries to indicate whether a version includes backward-compatible changes compared to its previous version. Semantic versioning involves three parts: major, minor, and patch versions. Typically, minor and patch version releases are expected to maintain compatibility with previous versions. However, prior research has revealed instances where open-source libraries fail to comply with this versioning scheme, introducing BCs even in minor and patch releases [15, 18, 20, 25, 37]. Therefore, it is crucial to understand the potential impact BCs have on clients that utilize them.

Some studies investigated the impact of Syntactic BCs on client projects when updating their dependencies [5, 29, 30, 45, 51, 64]. Other researchers have primarily focused on identifying BCs arising from Syntactic BCs between two software library versions [22, 32, 41, 51]. There is limited research explicitly detecting the Behavioral BCs between two library versions [12, 42, 66]. Some prior research has focused on detecting Syntactic and Behavioral BCs impacting clients on JavaScript projects [43, 60]. Two techniques SENSOR [63] and CompCheck [67] were introduced to detect BBCs that manifest as `Test Failures`, which could potentially impact Java Clients. These studies primarily focus on the detection of BBCs. However, it is important to know how BBCs impact clients and how these impacts manifest. Previous research has not explored in-depth the effects of BBC manifestation on client tests and the origin of these issues, focusing on all dependencies connected to a client project.

Hence, our research aims to address this gap by identifying the impact of Behavioral BCs on client projects by examining how dependency updates affect client tests, how BBCs manifest, and their origin. Focusing on client tests might not provide the entire impact of BBCs on clients if tests do not cover all library functionalities invoked. However, it will be a starting point to understand the degree of impact of BBCs on clients and how better test coverage will help identify BBCs during dependency updates.

We did not limit our study to examining only adjacent library updates like most other research conducted to analyze the impact of BCs on clients [33, 54, 62] as clients are most likely more than one version behind the latest stable version. Furthermore, research shows that clients with outdated dependencies would probably update to the latest Major version available, indicating that versions in between are often skipped [57]. Hence, we followed a more comprehensive approach and updated the client's dependencies to the latest stable versions, allowing us to create a realistic environment that more closely mirrors the actual dependency update practices followed by client projects. We conduct an empirical analysis on 8,086 Maven built Java client projects that contain tests suites. The research questions we addressed through the work are as follows:

RQ1: How often are client-impacting BBCs captured through client tests? Using the data set we have prepared, we analyzed how often BBCs impact client projects by detecting test failures and errors introduced during client dependency updates. This will give us a primary understanding of how BBCs impact client projects.

RQ2: What types of effects do BBCs have on client tests? We identify how BBCs can impact client tests differently. This enables us to build a taxonomy of effect based on how the changes impact the client tests. This taxonomy serves as a resource for the client developers to understand the different types of failures and errors a BBC can have on client tests.

RQ3: Where does the BBC-related impact originate? Based on the identified failing tests, we analyzed if the `Test Errors` originated within the library-related code, if the changed result or issue passed down to the client code, or if the it originated from other dependencies of the client or Standard Java code. The origin of `Test Errors` will help client developers decide if they could handle the incompatibility raised due to the BBC.

RQ4: Do runtime exception messages effectively guide developers in handling BBCs? Finally, we analyzed if the precondition-related exceptions thrown during a Major dependency update guide the client developers on how they can overcome issues related to BBCs.

To our knowledge, this is the first comprehensive analysis conducted to study the impact and manifestation of BBCs during dependency updates and where the BBC-related issues are raised.

Among the 30,548 dependency updates applied for client projects, we identified that 2.30% resulted in failing client tests. However, this number is likely underestimating the true impact of BBCs because we cannot guarantee that the client's test cases cover all the dependency functionalities the client utilizes. Nevertheless, it is noteworthy that client tests capture the BBCs that actually impact clients, implying that no effective false positive results are generated from this approach [53].

The analysis of these dependency updates led to four main findings:

1) Most of the detected BBCs (48.35%) are introduced during minor updates of dependencies and 18.27% during patch updates. This confirm that the semantic versioning scheme is often violated by library developers.

2) Among the analyzed failing tests, the majority (59.35%) trigger `Test Errors` (e.g., runtime or checked exceptions) and 40.65% trigger `Test Failures` (e.g., assertion violations). This means that 59.35% of failing tests can expose the BBCs without executing the assertion oracles of the client test. Therefore, many BBCs (triggered as `Test Errors`) could be effectively exposed using test generator tools [26, 47] and would not require assertion oracles. Indeed, current test generators are very effective at increasing code coverage but not at generating assertion oracles that verify the intended program behavior [3].

3) A non-negligible amount (18.78%) of the `Test Errors` originated from other dependencies connected to a client or the updated library, making it difficult for the clients to handle these BBCs introduced during dependency updates effectively.

4) Library developers have not provided adequate documentation in the runtime exceptions of the BBCs introduced since almost all precondition exceptions were not raised at the API surface, and half of these exceptions did not contain enough information to handle them.

2 BACKGROUND

This section provides the context for our research, which is focused on investigating how Behavioral Breaking Changes (BBCs) affect client projects.

Libraries are an important part of the modern software development as they provide reusable functionalities that help accelerate the development process. These libraries also evolve like any other software application and release new versions of the libraries that will bring new features, feature enhancements, and issue and vulnerability fixes. When a client utilizes the library functionality, it will depend on a specific version or range of versions, defined as a **dependency** on the client application. Therefore, as the libraries evolve, it is essential for clients to frequently update their dependencies to benefit from them and maintain the security of the overall ecosystem of the project [13, 48, 57].

```
878  904          private static void appendWhitespaceIfBr(Element element, StringBuilder accum) {
                  @@ -1074,8 +1100,13 @@ void outerHtmlHead(StringBuilder accum, int depth, Document.OutputSettings out)
1074 1100                .append(tagName());
1075 1101            attributes.html(accum, out);
1076 1102
1077       -          if (childNodes.isEmpty() && tag.isSelfClosing())
1078       -              accum.append(" />");
     1103 +          // selfclosing includes unknown tags, isEmpty defines tags that are always empty
     1104 +          if (childNodes.isEmpty() && tag.isSelfClosing()) {
     1105 +              if (out.syntax() == Document.OutputSettings.Syntax.html && tag.isEmpty())
     1106 +                  accum.append('>');
     1107 +              else
     1108 +                  accum.append(" />"); // <img> in html, <img /> in xml
     1109 +          }
1079 1110          else
```

Fig. 1. Behavioral BC under org.jsoup:jsoup library when updating from version 1.7.3 to version 1.8.1

There are two ways in which dependencies are associated with a client application: directly or transitively (indirectly). Dependencies explicitly declared in the project and necessary for the application's build process are referred to as **Direct Dependencies**. If a direct dependency of the client itself has its own direct dependency that it requires, that dependency is a **Transitive Dependency** for the client [31]. Since the client can access the functionalities of the transitive dependency (as it is bundled with the direct dependency), some clients may directly utilize this functionality [29].

Breaking changes (BCs) are the code changes made to a new library version that can potentially introduce compatibility issues for client projects developed using a previous library version as the new library version violates contracts. Not all BCs will impact a client, and whether or not a BC results in compatibility issues for a specific client depends on whether that client's code directly or indirectly calls the affected APIs or code during its execution. BCs can be broadly classified into two categories: syntactic and behavioral (semantic) BCs. Syntactic BCs, as the name implies, involve changes to the syntax of the library code, leading to compatibility issues on the client side. These syntactic changes can be further divided into Source and Binary BCs. Source BCs are identified during compilation, while Binary BCs are detected at load time when linking the client code with the library binaries. Syntactic changes can be identified with great accuracy using static analysis tools.

Behavioral Breaking Changes (BBCs) are the code changes made to a new library version that alter the expected behavior of the API of the library in a way that may cause existing client applications to break, malfunction, or behave unexpectedly. These changes are API contract violations related to either the preconditions, postconditions, or invariants that are bound to the methods syntactically or semantically [6]. Figure 1 shows a BBC in `org.jsoup:jsoup` when updating between versions 1.7.3 and 1.8.1. The change introduces an extra conditional statement that can change the content of the `StringBuilder` object `accum`, which will be passed on to other internal library methods that would finally affect the end result of the library API.

These behavioral changes are hard to capture using static analysis techniques as they rely on more approximation and abstraction. Exposing the BBCs dynamically by running test cases is precise. However, the dynamic analysis might undermine some of the issues since it might not cover

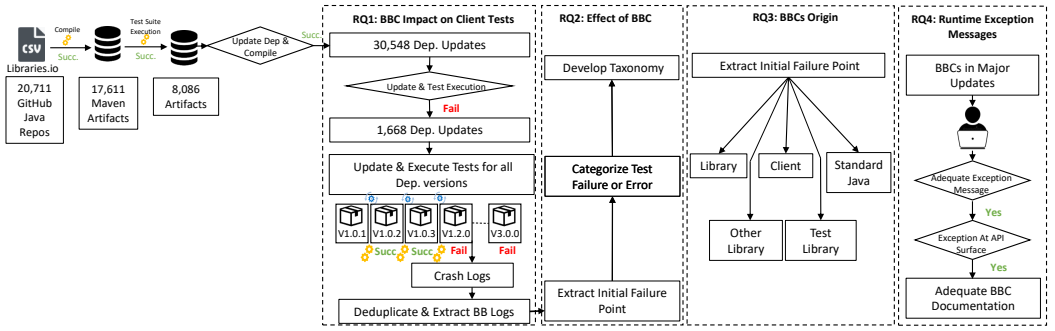

Fig. 2. Overview of the study

all possible input and environment conditions the program executes under, which would accurately represent the program's behavior [24]. Therefore, library developers must communicate to clients using their library the nature of the changes (breaking or compatible) included in their release. For this purpose, the library developers follow the **Semantic Versioning**[1] scheme to convey to their clients the nature of changes in the release. According to this scheme, BCs can only be introduced during a `Major` update while `Minor` and `Patch` updates must maintain backward compatibility.

3 STUDY DESIGN AND RESULTS

This section describes the design, analysis, and results of this study. Figure 2 provides an overview of the study, which includes the step-by-step processes followed to gather answers for each research question. We considered the Maven artifacts used in this experiment as the clients for the analysis. We executed the test suites of these clients with the dependency declared by the project, which we will refer to as the current dependency. Then, we updated the dependency to the latest stable version to determine the impact of Behavioral Breaking Changes (BBCs) on client tests. Using the crash logs of the failed test cases, we analyzed the different effects a BBC will have on tests, leading us to a taxonomy of effect. Then, we studied the origin of the failing tests to understand where the issues were initiated. Finally, we evaluated whether library developers adhered to proper documentation practices when releasing BBCs to understand if this documentation is sufficient for client developers to handle the issues caused by BBCs at the client end. The scripts, data, and results obtained for the study are available in the replication package [1].

3.1 Experiment Setup

We selected Java projects as the focus of our study for two main reasons. Firstly, Java ranks within the top five most popular programming languages (as of February 2024 [55]). Secondly, due to its static typing, we can unambiguously separate syntactic and behavioral breaking changes, with the behavioral ones being the focus of our study. Similarly to other research conducted in this area [2, 16, 27, 29], we selected subjects from the Libraries.io data set [59], which monitors over nine million open-source libraries available under 32 package managers. For the repositories available in the Libraries.io dataset, we applied the first filter of selecting Java projects, which are the client repositories for our research. Further, using the GitHub API[2] we filtered the repositories to contain a minimum of five stars and that are not fork repositories, to maintain the quality of the repositories selected for the research as done by previous research in the field [2, 16, 27, 29]. This provided

[1]https://semver.org/
[2]https://docs.github.com/en/rest

us with 20,711 Java repositories, in which we considered client projects that rely on third-party libraries for their dependencies. From these repositories we again filtered the ones using Maven as their build automation tool as their dependencies can be easily retrieved from the `pom.xml` file, which contains the necessary configuration. We decided not to include Gradle-built projects in our analysis because the repositories we used were created prior to 2020, a period when Gradle was still in the process of gaining popularity [49].

Next, we selected projects that compile successfully and contain passing test cases. Therefore, we first selected any repositories that compiled successfully with either Java 8 or Java 11. There were 3,524 and 325 projects that compiled in Java 8 and Java 11 respectively (3,839 in total). We chose these two Java versions because, between 2014 and 2021, the period the repositories had commit activities, they were the only versions supported by Oracle's Long-Term Support service. A Maven project can follow a parent-child structure, where

A Maven project may adhere to a parent-child structure arrangement, where a parent project can contain multiple child projects. Each Maven project is uniquely identified using the **GAV** coordinates, which are the **GroupId (G)**, denoting the organization or group responsible for creating the dependency; the **ArtifactId (A)**, denoting the artifact within the group; and the **Version (V)**, indicating the specific version of the artifact. Therefore, from these Maven repositories, we identified 20,169 unique Maven artifacts. However, not all parent Maven artifacts contained a folder with source code in them to evaluate tests, and therefore, we excluded them from the analysis, resulting in 17,611 clients for analysis.

To identify the impact of BBCs for these projects, we did not use static analysis techniques since they tend not to be precise [24], and precision is essential to many engineers [23, 53]. In order to achieve higher precision in static analysis techniques, complexity often increases, and the analysis quickly becomes unscalable [28, 35]. Further, static analysis can also be unsound [36, 58]. Static analysis can scale much better for syntactic BCs because it only has to look at the type system (mainly method signatures in declarations and call sites). Since running test cases to detect BBC is precise, we relied on client tests for BBC detection. Testing, however, under-approximates the issues, i.e. we will miss some and underreport incompatible changes. But all changes reported are actual changes a client may encounter, whereas in the static analysis, we would detect incompatibilities not corresponding to actual program behavior.

Therefore, for the projects that compiled successfully, we checked if they contained test suites by running the `mvn test` command and identifying if it generated surefire-reports, which are the reports generated for the executed unit tests under the respective project's target folder. We selected 8,086 clients that contained tests and passed all test cases. It is worth mentioning that out of the total number of considered clients, 43.25% (7,618) did not contain a test suite. The remaining 10.83% (1,907) clients did have test suites, but they contained at least one failing test case. Consequently, we did not include them in the analysis.

We used the `mvn versions:display-dependency-updates` command on the selected clients to identify potential outdated dependencies. While extracting outdated dependencies, we implemented a rule to exclude releases labeled as 'alpha,' 'beta,' 'SNAPSHOT,' and 'RC,' as these versions are considered unstable, according to the Maven Central Repository site and prior studies [45, 46].

We updated the outdated dependencies one at a time using a script to the latest stable version available. If compilation errors occurred during the update process, we excluded them, considering them as dependency updates that cause syntactic BCs. Therefore, for the next steps, we considered the dependencies of the clients that successfully compiled when updated to the latest stable version.

3.2 RQ1: BBC impact on clients

In this section, we answer **RQ1:** "How often are client-impacting BBCs captured through client tests?"

3.2.1 Method: In the following steps, we use the clients unaffected by Syntactic BCs during a dependency update.

Extract Failing Tests: After updating the dependencies on the client, we executed client tests to assess whether they were affected by the dependency update. For the test suites that failed when a dependency was updated, we ran the test suite ten times to filter out tests that are potentially flaky [34, 38]. After filtering the flaky test failures, we considered the remaining failing tests as being directly impacted by the BBC in the updated dependency. Multiple versions could exist between the current and the latest versions, to retrieve the exact version in which a BBC was introduced we examined the versions between the current declared dependency and the latest version. For this purpose, we collected all library versions released between the current and latest versions by querying the Maven Central Repositories. [3] For the clients with test suite failures, when updated to the latest dependency version, we incrementally updated the dependency version starting from the version immediately after the current one until the version that first caused the test suite to fail was detected. We used the version that first introduced the BBC to analyze the impact of the BBC on client projects.

Deduplicate Tests and Extract Tests for BBCs: We observed that some failing tests were related to linkage-related issues arising from binary incompatibilities. Therefore, to calculate the impact of BBCs, we excluded the failures related to binary incompatibilities and selected dependency updates that contained at least one failing test related to a BBC. We further excluded failing tests related to Maven build and test engine issues. To determine if the test failure was related to a binary incompatibility, Maven build error, or a test engine issue, we extracted the individual test failures with their crash logs to make this decision. For this, we used the surefire reports generated for each unit test class under the project target folder when tests are executed for Maven projects. These reports contain all test passing and failing related details. Using these reports, we extracted all failing tests, along with their test name and the crash log generated for each failing test case.

For some clients, when the dependency was updated, it resulted in one failing test, but for some clients, multiple tests failed. Therefore, we deduplicated the multiple failing tests to determine if each failure was due to the same or different reasons by using the crash logs. As some crash logs contained a few lines while others contained multiple lines, which also consisted of multiple `Caused by` statements, we needed to detect the initial point at which the failure occurred to deduplicate the failing tests. According to the Java documentation, the low-level exception will occur at the last `Caused by` statement in the crash log.[4] Following this guideline, the initial point of the crash log was determined. Next, we excluded the client-related code from the crash logs to make it generic since crash statements would contain each test-specific method invocation. We also excluded the line numbers for the remainder of the crash logs statements. Next, the first five statements of the crash logs were compared with the crash logs of the tests that failed in the same client during the same dependency update. The idea of using the first five lines of a crash log is similar to the equivalence functions used in research on crash reproduction such as EvoCrash [56] and Botsing [17]. We used the normalized Levenshtein distance technique to measure the text similarity in the crash logs. If the crash logs had a difference threshold of less than 5%, we considered them related to the same BBC, following Bartz et al. [4], which reported that normalized edit distance

[3] https://mvnrepository.com/repos

[4] https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html

Table 1. BBC introduced at each Semantic Versioning Level

Update Version level	Total number of updates	Updates impacted by BBC	Distribution of BBC impact across the semantic levels	% of impacted updates based on total updates of that level
Major	6,584	232	33.38%	3.52%
Minor	14,856	336	48.35%	2.26%
Patch	7,945	127	18.27%	1.60%

between 0-20% provides better crash log similarity. Therefore, if the difference was greater than the threshold, we considered them related to different BBCs.

After deduplicating the crash logs, we created a script to divide them into `Test Failures` and `Test Errors`. `Test failures` are signaled by assertion errors and can be extracted from the crash logs by identifying if the first line contains keywords such as 'Assertion', 'AssertionFailedError', 'Assert' or 'ComparisonFailure'. For the `Test Failures` we extracted the Exception or Errors raised, by detecting the initial point of the crash log.

3.2.2 Results: To determine the number of dependency updates containing BBC, we used 30,548 dependencies declared under 8,086 client projects.

Impact of BBCs: We updated these dependencies to the latest stable versions and observed that 1,668 which is 5.46% of these updates contained a failing test(s). However, these failures were related to both BBC and Binary incompatibilities. Therefore, after excluding these Binary incompatibility-related impacts, the total BBC-related impacts was 704, which is 2.30% of the total dependency updates.

Semantic Version Compliance: By examining the version numbers that introduced the BBCs and comparing them with the current version that is defined in the client project, we assessed if the library versions aligned with the principles of the semantic versioning scheme. Table 1 summarizes the BBCs introduced during different semantic version levels. Our findings indicate that most BBCs emerged during `Minor` updates, accounting for 48.35% of the instances of Behavioral Breaking (BB) versions. When considering both `Minor` and `Patch` BB versions, it becomes evident that non-`Major` updates played a substantial role, contributing to around two-thirds of the breaking versions. However, when extracting these statistics, we should also consider the total number of dependency updates applied at each semantic level. The total number of `Minor` updates was higher than the updates at the other semantic levels, and this could be a reason for `Minor` updates having the majority of the BB versions. When considering the total updates impacted by a BBC at each semantic level, `Major` updates reported 3.52% and was higher than the BB updates reported for the non-`Major` semantic levels. There were another eight dependency updates that introduced BBC under the same semantic version, which means the `Major`, `Minor`, or `Patch` numbers did not change. However, the version's build number changed which is an identifier for a particular build following the pattern `Major:Minor:Patch:BuildNumber`. For example, 2.0-M3 -> 2.0-M5 of library `com.thoughtworks.qdox:qdox`, 5.8.0-M1 -> 5.8.0 of library `org.junit.jupiter:junit-jupiter-engine`.

These BB versions were extracted based on the first version to introduce the BBC. However, multiple versions might have been released between the current version and the BB version. Therefore, we looked at how many `Major`, `Minor`, and `Patch` versions did not introduce any BBCs between the current and the BBC version. When counting the successful releases, we looked at `Major`, `Minor`, and `Patch` BB updates separately as Minor and Patch would not contain `Major` updates, and furthermore `Patch` would not contain any `Minor` updates.

Figure 3 displays all successful update releases when the BBCs were introduced under `Major`, `Minor`, and `Patch` updates separately. Among the Major updates that included BBCs, 70.69% did

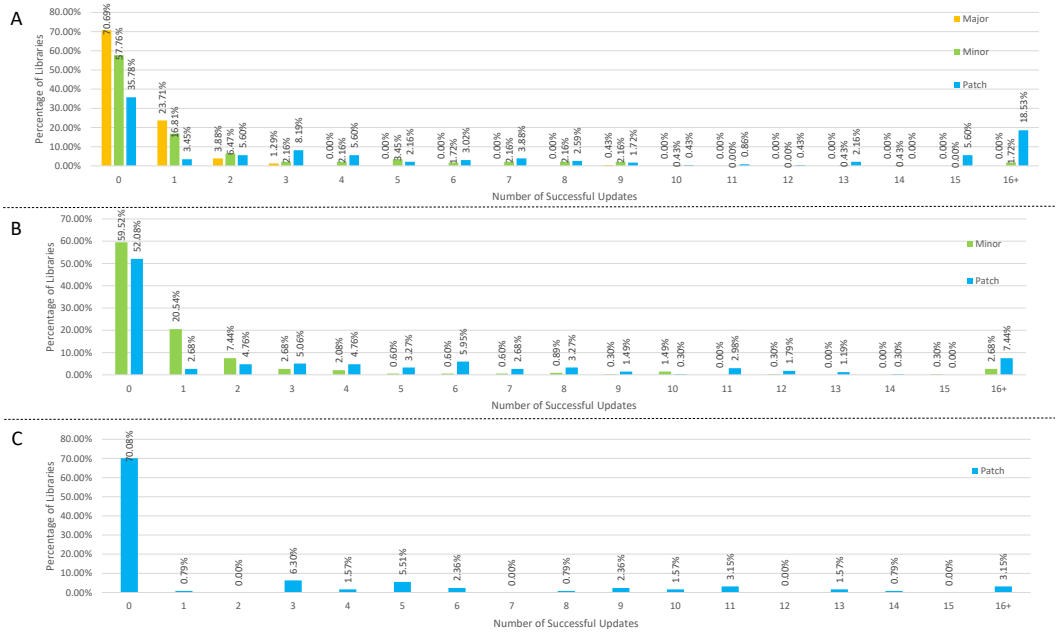

Fig. 3. A. Successful Releases between the Current and Breaking versions when the BBC was introduced in a Major Update B. Successful Releases between the Current and Breaking versions when the BBC was introduced in a Minor Update C. Successful Releases between the Current and Breaking versions when the BBC was introduced in a Patch Update

not have any `Major` updates between the current and the BB versions, which means the remaining 29.31% had at least one or more `Major` updates. Of these instances, 23.71% had just one `Major` update between the current and the BB versions. Additionally, 57.76% of these updates did not contain successful `Minor` updates, and 35.78% did not contain successful Patch updates between the current and the BB versions. The highest number of successful `Major` updates between the current and BB versions was nine, which occurred once, contributing to 0.43%, while the highest number of `Minor` and `Patch` updates were 27 and 219, respectively.

For the `Minor` updates that contained BBCs, 59.52% did not have any successful `Minor` updates between the current and the BB versions, and thus the remaining 40.48% contained at least one successful `Minor` update. Of these instances, 20.54% contained one successful `Minor` update, while the highest number of successful `Minor` updates was 51, which occurred only for one instance. 52.08% of these `Minor` updates, which introduced a BBC, did not contain any successful `Patch` updates between the current and BB versions. For the `Patch` updates that contained BBCs, 70.08% did not have any successful `Patch` updates between the current and the BB versions. The highest number of successful `Patch` updates between the current version and the breaking update was 27. At the same time, three was the most common number of successful `Patch` updates between the current and the BBC-introducing update, contributing to 6.30%.

Scope of BBCs: From these libraries that caused BBCs in the client application, we analyzed in which scope the library is connected to the client. Suppose the library is used for the compilation or runtime of the client for its functional usage. In that case, it will be defined either as a compiler, a runtime, or a provided scope dependency in the Maven configuration. Suppose the dependency is only needed for the compilation or runtime of the tests in the client project. In that case, they

```
java.lang.AssertionError: expected:<200> but was:<404>
        at org.testng.Assert.fail(Assert.java:89)
        at org.testng.Assert.failNotEquals(Assert.java:480)
        at org.testng.Assert.assertEquals(Assert.java:118)
        at org.testng.Assert.assertEquals(Assert.java:365)
        at org.testng.Assert.assertEquals(Assert.java:375)
        at org.sonatype.restsimple.sitebricks.test.addressBook.AddressBookSitebricksTest.testPost(AddressBookSitebricksTest.java:106)
```

Fig. 4. Test Assertion failure in client org.sonatype.restsimple:restsimple-sitebricks when com.google.inject.extensions:guice-servlet dependency was updated from version 3.0 to 4.0

will be listed as a test scope dependency in the Maven configuration. We categorized them as test and non-test-related dependencies. Based on the scope of the BBC introducing dependencies, 399 (56.67%) were non-test related dependencies, and 305 (43.32%) were test-related dependencies. Therefore, the data set we used for the analysis consisted of a similar amount of both test scope and non-test scope dependencies.

> **Answering RQ1:** *How often are client-impacting BBCs captured through client tests?* We observed that 2.30% of the dependency updates contained failing tests introduced due to BBCs. Moreover, our findings indicate that a significant portion of these BBCs were introduced through Non-Major updates.

3.3 RQ2: Effects of BBC

In this section, we answer **RQ2:** " What types of effects do BBCs have on client tests?"

3.3.1 Method: For this research question, we looked at how BBCs can be detected by analyzing client tests, which will provide insights into how BBCs affect clients. For this analysis, we used all failing tests related to a BBC extracted in RQ1. Under RQ1, we categorized the Failing tests as `Test Failures` and `Test Errors`, and under this section, we further categorize them to build a Taxonomy of Effect.

Categorizing Failing Tests: The `Test Failures` were divided into `Test Assertions` and `Program Assertions`. Test assertions occur when the actual outcome of a method under test is not the same as the expected outcome after the dependency is updated. If the library methods invoked through the test execution process were updated and changed the output, it would contribute to changing the final output of the client code. A sample crash log of a Test Assertion failure is provided in Figure 4. Under this example for the `org.sonatype.restsimple:restsimple-sitebricks` client, when the `com.google.inject.extensions:guice-servlet` dependency is updated from version 3.0 to 4.0, the test expects the method to return 200, while the result after the dependency update returns 404.

Developers use program assertions as conditional checks inside methods to verify that the execution flow of the program is correct. These are different from test-related assertions, as test assertions would evaluate the result of an execution flow, while code-related assertions verify the program conditions that are needed for execution during the execution process. Program assertions can be introduced either as an `Assert condition` or `explicitly throwing an Assertion Error`. Figure 5 displays a program assert extracted from the `org.needle4j:needle4j` client code. The client code raised this assertion error when the `com.h2database:h2 dependency` was updated from version 1.3.170 to 1.4.178. Due to the changes in the library, the client code could not clean the database tables successfully.

We wrote a script to determine whether the assertion was test-related or program-related. First, we marked the assertions thrown or originated from any of the classes in the following

```
java.lang.AssertionError: unable to clean tables [NEEDLE_TEST_ADDRESS]
        at org.needle4j.db.operation.hsql.HSQLDeleteOperation.deleteContent(HSQLDeleteOperation.java:142)
        at org.needle4j.db.operation.h2.H2DeleteOperationTest$H2DeleteOperationForTest.deleteContent(H2DeleteOperationTest.java:162)
        at org.needle4j.db.operation.hsql.HSQLDeleteOperation.tearDownOperation(HSQLDeleteOperation.java:53)
        at org.needle4j.db.DatabaseTestcase.after(DatabaseTestcase.java:152)
        .
        .
        .
        at org.apache.maven.surefire.booter.ForkedBooter.runSuitesInProcess(ForkedBooter.java:153)
        at org.apache.maven.surefire.booter.ForkedBooter.main(ForkedBooter.java:103)
```

Fig. 5. Program Assert in client org.needle4j:needle4j when com.h2database:h2 dependency was updated from version 1.3.170 to 1.4.178

```
javax.cache.CacheException: Error opening URI [hazelcast]
        at com.hazelcast.cache.impl.AbstractHazelcastCachingProvider.getCacheManager(AbstractHazelcastCachingProvider.java:134)
        at com.hazelcast.cache.impl.AbstractHazelcastCachingProvider.getCacheManager(AbstractHazelcastCachingProvider.java:163) .
        .
        .
        at org.apache.maven.surefire.booter.ForkedBooter.main(ForkedBooter.java:418)
Caused by: java.lang.IllegalStateException: Unable to connect to any cluster.  --> Exception Type
        at com.hazelcast.client.impl.connection.nio.ClientConnectionManagerImpl.doConnectToCluster(ClientConnectionManagerImpl.java:397) --> Exception Origin
        at com.hazelcast.client.impl.connection.nio.ClientConnectionManagerImpl.connectToCluster(ClientConnectionManagerImpl.java:346)
```

Fig. 6. IllegalStateException thrown in client org.apache.dubbo:dubbo-filter-cache during com.hazelcast:hazelcast dependency update from 3.11.1 to 4.0

testing libraries JUnit[5], Hamcrest[6], TestNG[7], Mockito[8], EqualsVerifier[9] as `Test Assertions`. Next, if the class generating the assertion contained the Keyword 'test', we assumed that the assertion was raised under a Test-related class and hence would be part of the `Test assertions`. For the remaining assertions, one author manually verified if they were valid `Program Assertions` and discussed them with the other authors to ensure impartiality.

Categorizing Test Errors: We categorized the rest of the crash logs that did not include assertions under the `Test Errors`. To determine the different errors generated, we had to extract the error or the exceptions thrown at the initial point of the crash log. Figure 6 shows a IllegalStateException thrown when client `org.apache.dubbo:dubbo-filter-cache` updates its dependency `com.hazelcast:hazelcast` from 3.11.1 to 4.0. We categorized the exceptions extracted based on the Java Throwable subclass Exceptions and Errors, and the Exceptions we further divided into Checked Exceptions and Runtime Exceptions.

The Exceptions extracted from the crash log also included custom exceptions defined by the client or a library. Therefore, we needed to extract their superclass details to determine if these custom exceptions were Checked or Runtime exceptions. If the superclass belonged to `java.lang.Exception` or any of its subclasses not extending `java.lang.RuntimeException`, we classified it as a Checked Exception. [10] If the Exception inherits from `java.lang.RuntimeException`, we categorized it as a Runtime Exception.

3.3.2 Results: For the 704 dependency updates in clients, we captured 1,043 tests after the deduplication applied in RQ1. We divided these tests based on the assertions and the exceptions.

[5]https://junit.org/junit5/
[6]https://hamcrest.org/JavaHamcrest/
[7]https://testng.org/doc/
[8]https://site.mockito.org/
[9]https://jqno.nl/equalsverifier/
[10]https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

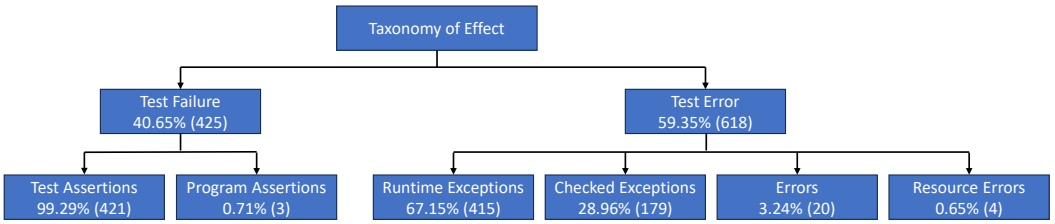

Fig. 7. Taxonomy of Effect on client Tests due to BBCs

Table 2. Most Common Runtime Exception effects due to BBCs on client Tests

Runtime Exceptions	Number of Occurrence	Percentage Out of All Runtime Exceptions
java.lang.NullPointerException	128	30.84%
Custom Runtime Exceptions	107	25.78%
java.lang.IllegalStateException	69	16.63%
java.lang.IllegalArgumentException	51	12.29%
java.lang.ClassCastException	17	4.10%

A total of 425 BBC effects were caused by `Test Failures`, contributing to 40.65% of the tests. 618 BBC effects were caused by `Test Errors`, which contributed to 59.35% of the tests. This outcome indicates that more than half of the failing tests could be identified using test generation tools wit hout the need for assertion oracles. Despite being highly proficient at enhancing code coverage, test generators lack efficacy in generating assertion oracles for verifying the intended program behavior. Based on these `Test Failures` and `Test Errors`, we built a Taxonomy of Effects for client tests caused by BBCs introduced between library versions. Figure 7 displays the contribution of the `Test Errors` and `Test Failures` and the different types of Errors and Failures that occurred.

Test Failures: From the `Test Failures`, almost all assertions were `Test Assertions` except for three, which were related `Program Assertions`. Therefore, `Test Assertions` covered 99.29% of the Failing tests while `Program Assertions` covered 0.71% of the Failing tests.

Test Errors: We categorized the Test error effects as `Runtime Exceptions`, `Checked Exceptions`, `Errors`, and `Resource Errors`.

Runtime Exceptions: `Runtime Exceptions` were the most common category of `Test Errors`, constituting 67.15% of the total Test Errors. Among these, the most frequently encountered `Runtime Exception` was NullPointerException, accounting for 30.84% of all `Runtime Exceptions`. The second most common `Runtime Exception` category was Custom Exceptions, defined by clients or libraries. We observed 15 types of `Runtime Exceptions`, and Table 2 provides details of the five most commonly observed `Runtime Exceptions`. Based on the top `Runtime Exceptions` observed, Null Pointer, Illegal State, and Illegal Argument Exceptions, we can conclude that most of these test errors occurred due to precondition violations introduced by the BBCs.

Checked Exceptions: According to our observations, the second common category of `Test Errors` consists of `Checked Exceptions`, accounting to 28.96% of the total Test Errors. We observed 18 distinct types of `Checked exceptions`, and Table 3 provides the most common `Checked Exceptions` observed. The most common `Checked Exception` was the Custom Exceptions defined under the clients of the libraries, which contributed to 51.40% of the total `Checked Exceptions`. Notably, IOException and FileNotFoundException occur among the top five `Checked Exceptions`, indicating that issues linked to input and output streams due to BBCs in libraries had an evident impact in generating `Checked Exceptions`.

Table 3. Most Common Checked Exception effects due to BBCs on client Tests

Checked Exceptions	Number of Occurrence	Percentage Out of All Checked Exceptions
Custom Exceptions	92	51.40%
java.io.IOException	16	8.94%
java.sql.SQLException	12	6.70%
java.io.FileNotFoundException	7	3.91%
java.net.SocketException	7	3.91%

Table 4. Error related Effects due to BBCs

Error Type	Number of Occurrence	Percentage Out of All Errors
java.util.ServiceConfigurationError	8	40.00%
Custom Error	8	40.00%
java.lang.VerifyError	4	20.00%

Errors: A total of 20 Errors affected the client tests, accounting for 3.24% of all Test Errors. Table 4 provides the three types of Errors that were reported. The most frequently encountered Errors were the java.util.ServiceConfigurationError, and the custom errors declared by either clients or libraries. We investigated the changes that occurred between the library versions to determine the cause of the ServiceConfigurationError and found that they were related to the changes introduced under the META-INF service directory, which consists of the service configuration details of the Maven projects.
Resource Errors: The final category of effects of BBCs under Test Errors is related to issues raised due to insufficient resources for the test execution, which we categorized as Resource Errors. These types of errors were not quite common. However, there were three different types of Exceptions raised, which are all related to the expected result not being returned within the anticipated timeframe. org.junit.runners.model.TestTimedOutException was raised twice for different clients, and this contributed to 50% of the total resource-related errors raised since the total errors in this category were four errors. java.net.SocketTimeoutException exception was raised in one instance, and the other exception was a custom exception com.jayway.awaitility.core.-ConditionTimeoutException raised by a library-related code. Even though resource-related issues raised due to BBCs during library update were uncommon, it is important to highlight that they exist and can potentially introduce performance issues in client projects.

> **Answering RQ2:** *What types of effects do BBCs have on client tests?* Our Analysis concludes that BBCs can broadly have two effects on client tests: Test Failures and Test Errors. Most of the effects were caused by Test Errors, accounting for 59.35% of the effects, and most of these errors were raised as Runtime Exceptions. The other categories of Test Error effects are Checked Exceptions, Errors, and Resource Errors. Test Failures were categorized into Test Assertions and Program Assertions, with the vast majority of assertions being related to test-related assertions.

3.4 RQ3: Origin of BBC impact

In this section, we answer **RQ3:** "Where does the BBC-related impact originate?"

3.4.1 Method: Based on the results we have obtained for the effect of BBCs, we conducted an analysis to determine the source of these effects. Specifically, we wanted to determine whether the origin of these BBC effects could be traced back to the client's code or if they were in code related

to libraries. If the origin exists in either of these two locations, it would assist clients in handling the BBCs.

We created a script to read the crash log and extract the initial point of the `Test Failure` or `Test Error`. If there were any `Caused by` statements, we extracted the block of crash log statements after the last `Caused by` statement, or if it did not contain any `Caused by` statements, then we used the entire crash log for the next steps. Then, we extracted the first line containing class details and the method of where the failure originated. As previously provided in Figure 6, the exception that was thrown during the test execution was a `IllegalStateException`, and the exception originated from the `ClientConnectionManagerImpl` class under the `doConnectToCluster` method in line number 397. Similarly, we extracted the first line containing the class details and the method of where an assertion occurred for assertion-related Test failures.

Next, we followed the subsequent steps to identify the origin of these Exceptions and Errors. We extracted all Java classes under the client project and then verified if the method that originated the test failure was included in the client-related classes. If there was a match, the failure originated at the client's end. Otherwise, to determine if it was related to the updated library or any other direct or transitive dependency of the client, we extracted the jar files of the libraries. Using `mvn dependency:copy-dependencies` command, we downloaded all jar files of the libraries for a project. By extracting the classes in a jar file, we could detect if the class that originated the test failure was included under the jar file. We first did this by matching the class with the classes under the updated library. If it was not a match, we matched it with classes of the other libraries connected with the project. Then, based on the client's Maven dependency tree, we detected if the library was a direct or transitive dependency to the client. If it was a transitive dependency, we further checked if the dependency was connected to the client through the updated library or a different direct dependency. If the class was included under a library that was not the updated library, we looked at the scope of the library to determine if it was used as a test-related library for the project. If the test failure originated class was not included within the client's codebase or any associated library classes, we checked whether the class was part of any Java built-in classes.

When identifying the origin of the test failure for exceptions, we looked at whether the exception thrown was explicitly defined in the code or if it was an implicit exception. For this step, if the exception was thrown from the client code, we created a script to read the particular source file of the class and detect if the line the exception is thrown from contained a throws statement with the particular exception reported. If the exception was reported under a library code, we had to process the binary code of the jar file to determine if any library-related class files explicitly defined the exception. For this purpose, we used the ASM Framework [11], a bytecode analysis and manipulation framework commonly used even within the OpenJDK ecosystem. ASM analyzed the bytecode of the particular class in the jar file, and we checked if the method under which the exception was thrown contained an instantiation of the exception class. If the method contained an instantiation of the exception, then we considered that the exception was explicitly defined.

3.4.2 Results: Based on the method explained, we created scripts and extracted the origin of the BBCs impact on Client tests. The results of the `Test Failures` showed that 55.53% of the time, the origin resided in a testing library, which is expected as they are responsible for condition violation checks. The remaining origins were distributed between the client (12.94%), Standard Java library(12.94%), or the updated library(8.47%), which could mainly result from unexpected Exceptions and Errors originating from these code locations.

Similar to extracting the origin of the Test Failures, we extracted the origin of the Test errors following the same approach. Table 5 shows the results received for the Test Error origin. Unlike the Test Failures, most Test Errors originated from the Updated Library and Client source code, which

Table 5. Origin of Test Errors

Origin	Number of Occurrence	Percentage
Library Updated	146	23.66%
Client	144	23.34%
Standard Java Library	92	14.91%
Library used for Testing	86	13.94%
Other Direct Dependency	79	12.80%
Transitive Dependency	33	5.33%
Dependency of the Updated Library	4	0.65%

occurred 23.66% and 23.34%, respectively. Next, the Test errors originated either in the Standard Java Library source code or a Testing library source code. Based on this data, we can conclude that most Test Failures originated from either the Updated Library or the Client Source Code. We could not extract the Test error origin for some Test error instances as the Crash Log statements were incomplete and did not provide the class details in which the error originated. Also, there were errors originating from Scala, Groovy [11] or Kotlin [12] source code, which could not be extracted as we did not write scripts to handle the binaries of these languages.

We conducted a detailed analysis of the Test Errors to determine whether the exceptions or errors were explicitly defined in the code or thrown implicitly during program execution. Among the Test Errors for which we detected their origin, 296 cases, making up 50.60% of the total, were found to have exceptions or errors that were explicitly defined in the source code. For the remaining 289 cases, accounting for 49.40%, the exceptions or errors were implicitly thrown during program execution. Therefore, this shows that almost half of the exceptions and errors thrown were explicitly defined in the source code.

> **Answering RQ3:** *Where does the BBC-related impact originate?* A similar percentage of `Test Errors` (~23%) originate (e.g., the exception is thrown) from either the updated library or the client's code. A non negligible amount of `Test Errors` originate from other dependencies of the client or the updated library. These BBCs will be generally more difficult for the client developers to understand and handle.

3.5 RQ4: Runtime Exception Messages

In this section, we answer **RQ4:** "Do runtime exception messages effectively guide developers in handling BBCs?"

3.5.1 Method: We evaluated if library developers follow best practices and guide their client developers on handling or adapting to the BBCs they encounter during dependency updates. We only used `Major` updates that cause BBCs for this analysis, as these BCs are anticipated during `Major` dependency updates. We selected the `Runtime Exceptions` that indicate potential precondition violations. Following the precondition-related exceptions defined by Dietrich et al. [21] we looked at the `IllegalArgumentException`, `IllegalStateException`, `NullPointerException`, `Custom Runtime exceptions` and, `ArrayIndexOutOfBoundsException` which were the exceptions we received for the `Test Error` effects. Then, based on the exception origin, we filtered the 'Test Errors' that originated at the updated library or another direct or transitive dependency.

[11] https://groovy-lang.org/
[12] https://kotlinlang.org/

For these exceptions, we looked at whether the library developers have provided a customized exception message that would potentially help guide the user into how to avoid this BBC. Two authors of the paper conducted a manual analysis to determine if the crash log contained an exception message and if it was meaningful to help guide the clients to handle the exception. Further, we detected if the `Runtime Exception` was thrown at the Library API surface, which is the part of the API accessible to others, by analyzing the crash log. We determined this by examining if the first line where the exception occurred was in library-related code, followed by a line related to the client code. If the API surface threw the `Runtime Exception`, it would likely indicate that the client violated the precondition by passing an invalid parameter(s).

3.5.2 Result: We extracted 43 `Test Errors`, resulting in a precondition-violating exception that originated from either the updated library or a direct or transitive dependency of the client. From these precondition-violating exceptions, 53.49% provided an exception message that would help guide the client to what is the cause of the BBC. Listing 1 displays a well-documented exception message when a precondition violation occurs. However, only one of these precondition-violating exceptions was thrown from the library API surface. This indicates that almost all of these exceptions were not adequately handled by the library developer since they were not thrown at the API surface or within any precondition handling library or methods.

Listing 1. A well document exception message

```
java.lang.IllegalStateException: Failed to serialize object of the class 'org.
    apache.ignite.tests.MyPojo'
```

> **Answering RQ4:** *Do runtime exception messages effectively guide developers in handling BBCs?* Our results indicate that almost half of the precondition-violation-related exceptions did not provide proper exception messages into the cause of the BBC, and almost all the exceptions were not thrown from the API surface. Hence, we can conclude that library developers do not adequately document the causes of BBC.

4 DISCUSSION

In this section, we discuss the key findings of the research and their implications.

According to our results, we concluded that Behavioral Breaking Changes between library updates impact 2.30% of the client test cases. This implies that these BBCs could also impact the client applications. We observed that two-thirds of these BBC-introducing versions were Non-`Major` updates, which shows that open-source libraries have not correctly followed the semantic versioning scheme during their library releases.

Since BBCs affect client tests differently, we introduced a taxonomy of effects, which will help client developers during the dependency update process understand the nature of effects that can be observed when there are BBCs in the updating version. We broadly categorized the effects as `Test Failures` and `Test Errors`, and our results indicate that approximately half of the effects caused by BBCs manifested as `Test Failures`. This suggests that roughly half of these BBCs can be identified by implementing comprehensive test cases or utilizing tools capable of generating tests that cover the invoked library functionalities, as test failures attributed to ~40% of the failing tests. This taxonomy will help guide the creation of patch templates for fixing the issues caused by BBCs. For example, runtime exceptions often represent precondition violations, which indicates that the call site needs to be changed to comply with the new library requirements. Similarly, if

a test assertion fails, it represents a violated invariant or a postcondition, which would require a different response.

While we did not find evidence of tests not being executed after the dependency update in our dataset, another potential effect could be that client tests are skipped due to BBCs. This scenario can occur when test assumptions are used. If the conditional statement does not evaluate to true, it would avoid the execution of the test case. Since the test statement was not executed, it will be marked as a test that was not executed. These tests will be tracked as skipped tests under the Maven test report. This category of BBC effects can be captured through monitoring if the number of tests skipped before a dependency is updated increases after a dependency is updated. Future work can further investigate whether skipped tests are indicative of BBCs.

Under resource-related errors in the taxonomy of effect, our results set only captured the Timeout Exception due to an expected result not being returned within a specific time limit. Other than this, `OutOfMemoryError` and `StackOverflowError` can also be included in this category. `OutOfMemoryError` will occur if the heap memory in the JVM runs out due to too many objects being created due to a change in the behavior of the program execution. `StackOverflowError` can occur if too many methods reside in the stack without being cleared, which causes the stack to run out of space. A common scenario in which this could occur is when a method is being called recursively without a proper termination condition. If the termination condition was decided based on an output of a library and the library does not return the same value due to a behavior change, this scenario could occur. The result set used for the analysis did not include results for either `OutOfMemoryError` or `StackOverflowError` during the dependency updates. However, it is important to note that these could also be potential effects on client tests due to dependency updates.

Through our analysis of the origin of the BBC, which impacts client tests, we observed that the origin of the BBC could reside in different locations like the updated library, the client, different direct libraries, transitive dependencies, or the Java standard library itself. Therefore, the behavioral change can be in the updated library, and the error could originate from the library itself or descend to other functional calls in different locations. Around half of the `Test Errors` originated from the updated library or client, which could potentially mean that the client could handle these BBCs. However, a non-negligible amount of `Test Errors` originated from other client dependencies, which would be hard for the clients to handle during the dependency updates.

Our analysis on whether library developers provide adequate documentation to handle runtime exceptions raised due to BBCs showed that proper documentation was not provided, especially for the `Major` updates, which they anticipated BBCs.

4.1 Implications of the Study

The practical implications we could draw through this study are outlined below.

Client developers should write comprehensive test suites covering the functionalities of both the application functionalities and the external API functionalities used. Our results indicated that 43.25% of the clients used for the analysis did not contain any test cases in their projects, which is not a good software practice. Writing test cases is a proactive approach to ensure the application's functionality works as expected after any change. Therefore, in the event of a dependency update introducing a BBC, having comprehensive test cases would help detect the BBC. Otherwise, all functionality would have to be manually verified by a quality assurance engineer of the project.

Library developers should be careful to correctly follow the semantic versioning scheme to indicate if BBCs are included in the releases. Our study indicates that most BBCs that impacted client tests were during a Non-Major update, which shows that the semantic scheme was violated during most of the updates that contained BBCs that impacted client tests. Further, library developers

should adequately document the BBCs so that the documentation would guide clients in handling the BBC.

Previous research indicates that approximately half of Syntactic BCs are introduced in Non-`Major` updates [29]. Our study strengthens this finding by revealing that two-thirds of BBCs are introduced in Non-Major updates. It is evident that BBCs are an inherent part of software evolution, as all software will inevitably change its existing functionalities. Consequently, library maintainers likely have difficulty knowing when they introduce BBC as they can be subtle changes that do not impact their current tests but will impact clients, so better support is needed to help them with this. Therefore, researchers should develop better tooling to indicate when potential BBCs are introduced between two library versions. This would also benefit clients to make informed decisions when updating dependencies.

Our findings indicated that more than half (59.35%) of the failing tests are exposed as `Test Errors`; therefore, automated test generation tools[26, 47] could efficiently identify these BBCs without requiring assertion oracles.

4.2 Threats to Validity

One threat to construct validity relates to how we determine the impact of BBCs on client tests during dependency updates. The impact can vary depending on the comprehensiveness of the tests, especially in terms of their ability to cover the library functionalities utilized in the client project. Also, if a client test does not pass the precise conditions or provide the necessary parameters to expose a BBC, it is possible that certain BBCs may not be detected as part of our study. Additionally, we excluded dependency updates that led to compilation errors when selecting projects for the analysis. If a dependency update introduced both Syntactic and Behavioral BCs, our study did consider them. Thus, our study's number of client tests impacted by BBCs may be underestimated.

Another threat to construct validity relates to the tests' flakiness; we examined only tests that exhibited failures during all ten times of the test execution. It is important to note that in cases of test flakiness, there might be situations where specific tests failed because a BBC was not consistently captured; it might have passed during some test runs due to the inherent unpredictability of these tests. Consequently, it is worth noting that the impact of BBCs on client tests could vary slightly under these circumstances.

One more threat to construct validity is the manual verification we had to conduct to determine the assertion categories and the exception message documentation. We mitigated the threat of bias by having more than one author engaged in the decision-making to make the final decision impartial.

A threat to external validity of our study is generalizing these results to other programming languages. Like other studies conducted to detect and quantify the impact of BBCs [43, 60, 67], we also focused on capturing the impact for a specific programming language under this study. The approach we used to identify the BBCs using tests will be appropriate for statically typed languages, as dynamic programming languages typically rely on test failures to detect syntactic and behavioral changes.

Another threat to external validity is that our experiments were conducted using open-source projects, so our findings depend on the nature of these projects. Consequently, we can only reasonably assume that closed-source projects might yield similar outcomes.

5 RELATED WORK

Library usage is convenient for clients with centralized package distribution systems such as Maven for Java, NPM for JavaScript, PyPI for Python, and Ruby for Ruby [25, 40]. Although these libraries reduce the development cost of application development [13, 14, 44], it adds extra effort

into maintaining them as libraries evolve and releases new versions. When updating a dependency, the project maintainers should consider the context in which the dependency is used [29]. Based on the context and the changes introduced in the releases, the maintainers could decide to update to the latest stable version or migrate to a different library. Updating to a new version can be time-consuming if the new version contains Breaking Changes (BCs) compared with its previous version, and is a factor for clients having outdated dependencies [33, 54]. This has led researchers to study more about the different types of BCs, and our study focuses on Behavioral Breaking Changes (BBCs) and its impact on client projects.

Breaking Changes in Library Evolution: In recent years much research has focused on detecting the syntactic (source and binary) BCs between two Java library versions [8, 10, 22, 32, 41, 45, 64]. However, the focus on analyzing BBCs between Java libraries has been relatively limited [12, 42, 66]. Sembid [66] is a static analysis tool, which detects BBCs between two library versions with a precision of 81.29%. However, this static technique has limitations, particularly in classifying benign changes and determining what APIs are exposed for client utilization. Mostafa et al. [42] used cross-version testing only to detect BBCs between two library versions and proposed a classification for the different BBCs as exceptions and crashes, return value change, and other effects. Similarly, DeBBI [12] is a cross-project testing approach that prioritizes client tests to detect BBCs faster. However, they relied on the confirmed BBCs by the library developers to detect their existence, which is not a comprehensive approach as not all BBCs are intentional. These approaches focused on detecting BBCs between library versions, while our study focused on identifying the impact of BBCs between library versions on clients that utilize them.

BCs Impact on Clients: The studies explained above all focus on detecting BBCs between library versions, but it is also important to study how these BBCs impact clients that use them. Research conducted on Java-related libraries and their clients has revealed that Syntactic BCs introduced between library versions impact clients that rely on these libraries during dependency updates [5, 30, 45, 51, 64]. Mezzetti et al. [37] conducted a study using client test suites to detect BCs introduced in Node.js libraries. They manually classified the test failures related to Syntactic and Behavioral BCs and used these Syntactic BCs to develop a type regression testing technique to detect Syntactic BCs, as they are more easily detectable than Behavioral ones. Building upon this research, Mujahid et al. [43] and Venturini et al. [60] utilized client tests to detect BBCs introduced by Node packages. Mujahid et al.'s study only focused on BC detection and concluded that better performance could be gained by utilizing more client test cases. Venturini et al. concluded that Syntactic and Behavioral BCs affected approximately 11.7% of the clients [60] and further categorized these errors manually based on their origin and reported that 13.9% of these BCs reside in the library. However, these studies did not perform an in-depth analysis specifically targeting BBCs, whereas our study led to novel findings and outcomes focused on the impact of BBCs on clients. Moreover, their investigations focused on the Javascript ecosystem, whereas ours was on the Java ecosystem.

SENSOR [63] and CompCheck [67] are both BBC detecting tools built for Java projects, which focus on identifying the existence of BBCs between two library versions by generating incompatibility revealing tests so they would assist client developers during the dependency update process. Both these tools can only detect BBCs which manifest as assertion errors as they utilize test cases to identify the BBCs. In alignment with these studies, our research follows a similar approach using client tests to identify how BBCs impact Java clients. However, these prior studies do not cover BBCs that manifest as Test Errors and do not determine the impact of these BBCs or how they manifest within client tests. Therefore, our study goes beyond detection and analyzes how BBCs influence dependency updates in clients, their impact on client tests, and the origins of these effects.

6 CONCLUSION

In this research, we conducted an empirical study to analyze the impact of BBCs on client test cases when updating 30,548 dependencies under 8,086 Maven clients. This study examined the impact of BBCs on client tests and observed that 2.30% of the dependency updates contained a BBC that impacted a client test. Most BBC impacts occurred during a Non-`Major` dependency update. Using the client tests that were affected due to a BBC, we looked at how these changes manifest in client tests by introducing a Taxonomy of Effect. We broadly categorized these effects as `Test Failures` and `Test Errors`. We further categorized the `Test Failures` as `Test Assertions` and `Program Assertions`. `Test Errors` were categorized as `Runtime Exceptions`, `Checked Exceptions`, `Errors` and `Resource Errors`. Almost all `Test Failures` were due to `Test Assertions`, and most `Test Errors` manifested as `Runtime Exceptions`. We observed that the origin of the `Test Failures` mostly resides in a testing library while the origin of `Test Errors` resides in either the updated library or the client's source code. We also concluded that the library developers do not provide adequate documentation to handle strengthened preconditions, resulting in runtime exceptions.

For our future research, we hope to examine the source code change of the library that contributed towards a BBC and categorize these changes. Based on the source code changes, we could determine if the client needs to adopt these changes, remain under the same version, or migrate to a different library. We would further intend to mine the dependency updates applied in open-source projects and extract changes contributing to resolving the BBCs. By identifying patterns of these fixes, we could create patch templates for client developers as fixes to resolve the BBCs during dependency updates. To further refine our study on the documentation of BBCs, we will extend our analysis to include data extracted from release notes of the libraries, as release notes would potentially provide detailed discussions and decisions about the BBCs, offering valuable insights into the rationale behind these changes and their implications.

DECLARATIONS

Acknowledgments. This work was supported by the Marsden Fund Council from Government funding, administered by the Royal Society Te Apārangi, New Zealand. The work of the third author was supported by a gift by Oracle Labs Australia. In addition, the authors wish to acknowledge the Centre for eResearch at the University of Auckland for their help in facilitating this research (http://www.eresearch.auckland.ac.nz).

REFERENCES

[1] 2024. *Replication Package for Understanding the Impact of APIs Behavioral Breaking Changes on Client Applications.* https://doi.org/10.5281/zenodo.10678853

[2] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2021. Empirical Analysis of Security Vulnerabilities in Python Packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* 446–457. https://doi.org/10.1109/SANER50967.2021.00048

[3] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[4] Kevin Bartz, Jack W. Stokes, John C. Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihle. 2008. Finding similar failures using callstack similarity *(SysML'08).* USENIX Association, USA, 1.

[5] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The Evolution of Project Inter-Dependencies in a Software Ecosystem: The Case of Apache *(ICSM '13).* IEEE, 280–289. https://doi.org/10.1109/ICSM.2013.39

[6] Antoine Beugnard, J-M Jézéquel, Noël Plouzeau, and Damien Watkins. 1999. Making components contract aware. *Computer* 32, 7 (1999), 38–45.

[7] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 42 (2021), 56 pages. https://doi.org/10.1145/3447245

[8] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. 2020. You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering* 25, 2 (2020), 1458–1492. https://doi.org/10.1007/s10664-019-09756-z

[9] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 507–511. https://doi.org/10.1109/SANER.2018.8330249

[10] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 255–265.

[11] Eric Bruneton, Eugene Kuleshov, Andrei Loskutov, and Rémi Forax. 2022. *ASM*. https://asm.ow2.io/

[12] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming Behavioral Backward Incompatibilities via Cross-Project Testing and Analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 112–124. https://doi.org/10.1145/3377811.3380436

[13] Joel Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. 2015. Measuring Dependency Freshness in Software Systems. In *MOBILESoft 2015 : second ACM International Conference on Mobile Software Engineering and Systems (2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), Vol. 2)*. IEEE Press„ 109–118.

[14] Russ Cox. 2019. Surviving Software Dependencies. *Commun. ACM* 62, 9 (8 2019), 36–43. https://doi.org/10.1145/3347446

[15] Alexandre Decan and Tom Mens. 2021. Lost in zero space – An empirical comparison of 0.y.z releases in software package distributions. *Science of Computer Programming* 208 (2021), 102656. https://doi.org/10.1016/j.scico.2021.102656

[16] Alexandre Decan and Tom Mens. 2021. What Do Package Dependencies Tell Us about Semantic Versioning? *IEEE Transactions on Software Engineering* 47, 6 (6 2021), 1226–1240. https://doi.org/10.1109/TSE.2019.2918315

[17] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. 2021. Botsing, a search-based crash reproduction framework for Java. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1278–1282. https://doi.org/10.1145/3324884.3415299

[18] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. 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 (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2187–2200. https://doi.org/10.1145/3133956.3134059

[19] Jens Dietrich, Kamil Jezek, and Premek Brada. 2014. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 64–73. https://doi.org/10.1109/CSMR-WCRE.2014.6747226

[20] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency Versioning in the Wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 349–359. https://doi.org/10.1109/MSR.2019.00061

[21] Jens Dietrich, David J Pearce, Kamil Jezek, and Premek Brada. 2017. Contracts in the wild: A study of java programs. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[22] Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve? A Story of Refactoring: Research Articles. *Journal of software maintenance and evolution: Research and Practice* 18, 2 (3 2006), 83–107. https://doi.org/10.1002/smr.328

[23] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.

[24] Michael D Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*. 24–27.

[25] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. 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)*. Association for Computing Machinery, 791–796. https://doi.org/10.1145/3236024.3275535

[26] Gordon Fraser and Andrea Arcuri. 2011. 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*. 416–419.

[27] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. 2021. A Large-Scale Empirical Study on Java Library Migrations: Prevalence, Trends, and Rationales *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 478–490. https://doi.org/10.1145/3468264.3468571

[28] Nevin Heintze and David McAllester. 1997. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 342–351.

[29] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. 2023. Understanding Breaking Changes in the Wild. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*

(ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 1433–1444. https://doi.org/10.1145/3597926.3598147

[30] Kamil Jezek, Jens Dietrich, and Premek Brada. 2015. How Java APIs Break - An Empirical Study. 65, C (sep 2015), 129–146. https://doi.org/10.1016/j.infsof.2015.02.014

[31] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 102–112.

[32] Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. 2019. Classification of Changes in API Evolution. In *IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. 243–249. https://doi.org/10.1109/EDOC.2019.00037

[33] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do Developers Update Their Library Dependencies? *Empirical Software Engineering* 23, 1 (2 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[34] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020. Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 403–413. https://doi.org/10.1109/ISSRE5003.2020.00045

[35] Ondvrej Lhotak. 2007. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 37–42.

[36] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhotak, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[37] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries (Artifact). *Dagstuhl Artifacts Series* 4, 3 (2018), 8:1–8:2. https://doi.org/10.4230/DARTS.4.3.8

[38] J. Micco. 2017. *The state of continuous integration testing at Google*. https://bit.ly/2OohAip

[39] Samim Mirhosseini and Chris Parnin. 2017. Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-Date Dependencies?. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, 84–94.

[40] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting Locations in JavaScript Programs Affected by Breaking Library Changes. *Proc. ACM Program. Lang.* 4, OOPSLA (11 2020), 1–25. https://doi.org/10.1145/3428255

[41] Anders Møller and Martin Toldam Torp. 2019. Model-Based Testing of Breaking Changes in Node.Js Libraries. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 409–419. https://doi.org/10.1145/3338906.3338940

[42] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 215–225. https://doi.org/10.1145/3092703.3092721

[43] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. 2020. Using Others' Tests to Identify Breaking Updates. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. Association for Computing Machinery, New York, NY, USA, 466–476. https://doi.org/10.1145/3379597.3387476

[44] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A Graph-Based Approach to API Usage Adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 302–321. https://doi.org/10.1145/1869459.1869486

[45] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. 2022. Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central: An External and Differentiated Replication Study. *Empirical Softw. Engg.* 27, 3 (may 2022), 42 pages. https://doi.org/10.1007/s10664-021-10052-y

[46] Fernando Rodriguez Olivera. 2022. *MVN Repository: repository stats*. https://mvnrepository.com/repos

[47] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[48] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable Open Source Dependencies: Counting Those That Matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '18)*. Association for Computing Machinery, New York, NY, USA, Article 42, 10 pages. https://doi.org/10.1145/3239235.3268920

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

[50] Tom Preston-Werner. n.d. *Semantic Versioning 2.0.0*. https://semver.org/

[51] S. Raemaekers, A. van Deursen, and J. Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158. https://doi.org/10.1016/j.jss.2016.04.008
[52] Eric Ruiz, Shaikh Mostafa, and Xiaoyin Wang. 2015. Beyond api signatures: An empirical study on behavioral backward incompatibilities of java software libraries. *Department of Computer Science, University of Texas at San Antonio, Tech. Rep* (2015).
[53] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Commun. ACM* 61, 4 (2018), 58–66.
[54] Pasquale Salza, Fabio Palomba, Dario Di Nucci, Cosmo D'Uva, Andrea De Lucia, and Filomena Ferrucci. 2018. Do Developers Update Third-Party Libraries in Mobile Apps? *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 255–265. https://doi.org/10.1145/3196321.3196341
[55] TIOBE Software. 2023. *TIOBE Index*. https://www.tiobe.com/tiobe-index/
[56] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. 2018. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1294–1317.
[57] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. 2020. Technical Lag of Dependencies in Major Package Managers. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. 228–237. https://doi.org/10.1109/APSEC51365.2020.00031
[58] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the recall of static call graph construction in practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1049–1060.
[59] Inc Tidelift. 2022. *Libraries.io - The Open Source Discovery Service*. https://libraries.io/data
[60] Daniel Venturini, Filipe Roseiro Cogo, Ivanilton Polato, Marco A. Gerosa, and Igor Scaliante Wiese. 2023. I Depended on You and You Broke Me: An Empirical Study of Manifesting Breaking Changes in Client Packages. 32, 4, Article 94 (2023), 26 pages. https://doi.org/10.1145/3576037
[61] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. 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)*. 35–45. https://doi.org/10.1109/ICSME46990.2020.00014
[62] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. 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)*. 35–45. https://doi.org/10.1109/ICSME46990.2020.00014
[63] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. 2022. Will Dependency Conflicts Affect My Program's Semantics? *IEEE Transactions on Software Engineering* 48, 7 (2022), 2295–2316. https://doi.org/10.1109/TSE.2021.3057767
[64] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 138–147. https://doi.org/10.1109/SANER.2017.7884616
[65] Ahmed Zerouali, Tom Mens, Jesus Gonzalez-Barahona, Alexandre Decan, Eleni Constantinou, and Gregorio Robles. 2019. A formal framework for measuring technical lag in component repositories—and its application to npm. *Journal of Software: Evolution and Process* 31, 8 (2019).
[66] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2023. Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing *(ASE '22)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556956
[67] Chenguang Zhu, Mengshi Zhang, Xiuheng Wu, Xiufeng Xu, and Yi Li. 2023. Client-Specific Upgrade Compatibility Checking via Knowledge-Guided Discovery. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 98 (2023), 31 pages. https://doi.org/10.1145/3582569

Received 2023-09-29; accepted 2024-01-23