# Understanding Breaking Changes in the Wild

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

Samuel Ou
University of Auckland
Auckland, New Zealand
sou323@aucklanduni.ac.nz

Kelly Blincoe
University of Auckland
Auckland, New Zealand
k.blincoe@auckland.ac.nz

## ABSTRACT

Modern software applications rely heavily on the usage of libraries, which provide reusable functionality, to accelerate the development process. As libraries evolve and release new versions, the software systems that depend on those libraries (the clients) should update their dependencies to use these new versions as the new release could, for example, include critical fixes for security vulnerabilities. However, updating is not always a smooth process, as it can result in software failures in the clients if the new version includes breaking changes. Yet, there is little research on how these breaking changes impact the client projects in the wild. To identify if changes between two library versions cause breaking changes at the client end, we perform an empirical study on Java projects built using Maven. For the analysis, we used 18,415 Maven artifacts, which declared 142,355 direct dependencies, of which 71.60% were not up-to-date. We updated these dependencies and found that 11.58% of the dependency updates contain breaking changes that impact the client. We further analyzed these changes in the library which impact the client projects and examine if libraries have adhered to the semantic versioning scheme when introducing breaking changes in their releases. Our results show that changes in transitive dependencies were a major factor in introducing breaking changes during dependency updates and almost half of the detected client impacting breaking changes violate the semantic versioning scheme by introducing breaking changes in non-Major updates.

## CCS CONCEPTS

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

## KEYWORDS

software libraries, software dependency, breaking changes, software evolution

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
*ISSTA '23, July 17–21, 2023, Seattle, WA, United States*
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598147

**ACM Reference Format:**
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 '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598147

## 1 INTRODUCTION

Software libraries can be an invaluable asset to client projects, providing reliable source code that saves development costs [8, 18, 28]. Like any other software, these libraries also evolve, releasing new versions [36]. It is crucial for the client projects to update their dependencies to the latest versions to access improved features and avoid potential risks associated with outdated software dependencies [5, 32].

However, updating a dependency to the latest version could also cause software failures in the client projects if the new version includes backward incompatible changes, also known as Breaking Changes (BCs). For example, deleting or renaming an artifact, like a method or class, being used by the client would cause BCs. BCs are often categorized as source, binary, and behavioral [13]. Source BCs cause compilation errors, binary BCs cause linkage errors, and behavioral BCs cause the software to behave differently at run time.

Semantic versioning [37] is the most common approach followed by library developers to inform dependent clients whether a new version is backward compatible with the previous version [14, 32, 39, 48]. This versioning scheme uses a three-digit number format of `Major.Minor.Patch`, and requires the `Minor` and `Patch` releases to be backward compatible. However, research has found that libraries often violate the semantic versioning scheme and introduce BCs under `Minor` or `Patch` releases [18, 32, 39].

Prior research analyzed BCs introduced between library versions [15, 26, 29, 39]. However, it is also important to examine the impact of BCs on the client projects, since not all BCs will cause failures in the clients, for example, if the change is in part of the code not used by the client. Raemaekers et al. [39] and Ochoa et al. [32] examined the impact of BCs on client projects, but their analysis was limited to binary BCs, so did not include a comprehensive impact of both source and binary breaking changes. These studies also focused on BCs that occur in adjacent library versions, which may underestimate the impact on real clients who are likely to be more than one release behind [27, 41, 44].

In this study, we examine the impact of both source and binary BCs on client projects, and we also examine the BCs that clients

would be faced with "*in the wild*" by examining the set of BCs they would face if they chose to update to the latest available stable version from the version they currently depend on. We perform a large-scale empirical analysis of Java projects which use Maven as their build tool. We analyzed 18,415 Maven artifacts, which declare 142,355 dependencies related to 7,454 Maven libraries. We were guided by the following four research questions:

**RQ1: To what degree are the dependencies in open-source repositories up-to-date?** We examined how many clients have kept their dependencies up-to-date based on the latest versions available. We also classify the updates available as `Major`, `Minor`, and `Patch` updates to understand the level of update required.

**RQ2: How often do client-impacting BCs occur in the wild?** For the clients with outdated dependencies, we updated the declared dependencies to the latest stable version available. After the update, we verified if the update introduced BCs to the clients and determined the percentage of clients affected by BCs. We investigated the exact library version in which the client-impacting BCs were introduced.

**RQ3: What are the common types of client-impacting source BCs?** For a sample of the client-impacting BCs, we analyze in depth the reason for the BC to understand the types of BCs that impact the clients.

**RQ4: Are client-impacting source BCs introduced in non-`Major` library releases?** We classify the versions which introduced BCs in client projects according to the semantic versioning level and report the results also based on the type of BC for some of the common types of client-impacting BCs.

To the best of our knowledge, this is the first large-scale study to examine the impact of both source and binary BCs on client projects.

Our results show that 11.58% clients would encounter BCs if updating their dependencies to the latest stable version of a library. We find that transitive dependency changes are a leading cause for client-impacting BCs, indicating better support is needed for helping project maintainers to become aware of the impact of changes in their own dependencies on their client projects. In addition, it was observed that 41.58% of the BCs that affected clients occurred during non-`Major` dependency updates. This highlights the need for library developers to exercise greater caution in their versioning practices and selecting the changes introduced in each library release.

## 2 BACKGROUND

This section gives the background of this work, which aims at studying the impact of source and binary BCs on client projects.

**Libraries** consist of logically grouped functionality exposed as APIs, to be utilized by client projects. Like any other software, libraries also evolve and release new versions. Each version might, for example, add or modify features, fix bugs, or improve the performance and security of the library. A **dependency** specifies the version or range of versions of a library that the client code relies on.

Libraries can also be clients of other libraries. As such, a client can have two types of dependencies: direct and transitive. **Direct dependencies** are declared in the client's build configuration and are required for the project to build because the client code "directly" invokes the library's API. **Transitive dependencies** (also called indirect) are not declared in the client's build configuration but are required for its direct dependencies to build and run [25]. Since transitive dependencies will also have their own direct dependencies, this creates a dependency tree, creating many levels of transitive dependencies.

**Breaking changes (BCs)** are changes made to a software library that might introduce incompatibilities in clients that were built using an earlier version of the library. Whether a BC will create incompatibilities with a particular client depends on whether the client's code is using the APIs that are affected by the change. BCs can be broadly defined into two categories syntactic and behavioral (semantic) BCs. Behavioral BCs lead to behavioral incompatibilities, which change the behaviour of the client code when running with the newer library version. Behavioral incompatibilities can only be detected at runtime and require extensive test suites. In this study, we focus on syntactic BCs only.

**Syntactic BCs** can be further divided into: source and binary BCs. **Source BCs** lead to source incompatibilities, which are exposed at compile time. The client source code must be modified to use the new version of the library. **Binary BCs** lead to binary incompatibilities, which are exposed at load time when linking the application and library binaries [20]. Both source and binary incompatibilities are related to syntactical changes applied to the API, which can typically be detected through static analysis. For example, source and binary BCs include changes made to an API signature, adding or removing super types of a class, or deleting a method, class, or an entire package.

While often syntactic BCs are both source and binary incompatible, there are subtle differences. For instance, Figure 1 shows a source BC made in the `ch.qos.logback:logback-core` library between version `1.1.0` and `1.1.1`, which throws a new exception. For this change, clients must either catch or rethrow this newly thrown exception. Unless the client is doing this already (for instance, by catching Exception), this will result in a compilation error. However, since throws clauses are not part of the method descriptor used during linking, this update remains binary compatible (though it can cause a crash at runtime if unhandled exceptions are thrown). Figure 2 shows a change which specializes the return type of a method in the `net.sourceforge.owlapi:owlapi-distribution` library between version `4.3.1` and `5.0.0`. This change is strengthening a postcondition, and is therefore source compatible. Callers are expecting a Collection to be returned, and as List is a subtype of Collection, the contract is upheld and the code will compile. However, in bytecode, the return type is part of the method descriptor used to identify a method during linking, and therefore the change is binary incompatible. More precisely, clients compiled with the old version will encounter a NoSuchMethodError when being executed with the new version of the library without being recompiled first [34].

These libraries require a mechanism to indicate to the clients using their APIs what changes (e.g., BCs) are included in each new

```
50       -   public String transform() {
    58   +   public String transform() throws ScanException {
51  59           StringBuilder stringBuilder = new StringBuilder();
```

**Figure 1: Binary compatible but Source Breaking Change ch.qos.logback:logback-core when updating from version 1.1.0 to 1.1.1. (https://github.com/qos-ch/logback/compare/v_1.1.0...v_1.1.1)**

```
95       -   @Nonnull
96       -   public Collection<Clause> getClauses(String tag) {
97       -       Collection<Clause> cls = new ArrayList<>();
    119  +   public List<Clause> getClauses(@Nullable String tag) {
    120  +       List<Clause> cls = new ArrayList<>();
```

**Figure 2: Source compatible but Binary Breaking Change under net.sourceforge.owlapi:owlapi-distribution when updating from version 4.3.1 to 5.0.0. (https://github.com/owlcs/owlapi/compare/owlapi-parent-4.3.1...owlapi-parent-5.0.0)**

version. For this, library developers often follow the semantic versioning scheme [32]. **Semantic Versioning**[1] provides a standard way of communicating the types of changes in a new version of a library, which makes it easier for developers to understand how their software will be affected by a library update. The system is based on a three-part numbering scheme: `Major.Minor.Patch`. The numbers are assigned and incremented based on the following rules:

- **`Major` version:** Incremented if there are changes that are not backward compatible with previous versions.
- **`Minor` version:** Incremented if new features are added that are backward compatible with previous versions. This includes adding new functionality, improving existing features, or making non-BCs to the API.
- **`Patch` version:** Incremented if backward-compatible bug fixes or minor changes are introduced.

This study analyses Java projects that use **Apache Maven** [19] as their build automation tool. **Project Object Model (POM)** is the fundamental unit of work in Maven, allowing the tool to handle the project's build and documentation from a central location. The `pom.xml` file handles this task and contains the project's metadata, dependencies, and additional configurations. One of its core features is dependency management, which will automatically download the required software artifacts and any of its transitively dependent artifacts from a remote repository and store them in a local repository. Maven Central [2] is the most popular remote repository, which maintains millions of software artifacts.

A Maven artifact can follow a parent-child structure where a parent can contain multiple child artifacts. Each software artifact is uniquely identified by **GAV** coordinates, which are the `groupid (G)`, representing the organization or group that created the dependency, the `artifactid (A)` representing the identifier for the artifact within the group, and the `version (V)`, which is the version of the artifact. This coordinate could also include a classifier that distinguishes the artifacts built using the same `pom.xml` but varying in terms of the functionality provided by the underlying code. The coordinates can also include a scope attribute that will determine at which life-cycle phase (build, runtime, test) the dependency is added to a project's classpath. The dependencies are declared in the `<dependencies>` section in the client's `pom.xml` file. Listing 1 illustrates how a dependency is defined within the `pom.xml` file in a client.

```xml
<dependency>
    <groupId>org.yaml</groupId>
    <artifactId>snakeyaml</artifactId>
    <scope>compile</scope>
    <classifier>android</classifier>
    <version>1.30</version>
</dependency>
```

**Listing 1: A Dependency Block in a pom.xml file**

## 3 STUDY DESIGN AND RESULTS

This section describes the study's design for data collection and analysis. Figure 3 provides an overview of the steps we followed in the study. We collected repositories containing Maven artifacts and analyzed their dependency usage. Next, we updated the outdated dependencies one at a time and compiled the artifacts. For each dependency that failed to compile, we extracted the compiler failure and mapped it with the change in the library which caused this incompatibility. We also used a static analysis tool to capture the client-impacting syntactic BCs. Finally, we analyzed how library updates at different semantic version levels introduce these impactful BCs. The study results, data, and scripts used to acquire and analyze the data are available in the replication package [23].

### 3.1 Experiment Setup

*3.1.1 Artifact Collection:* For the study, we examine Java repositories as Java is one of the popular statically typed programming languages and since there are many static analysis tools developed for Java that can detect both source and binary BCs.

We collected the repositories for the analysis using the Libraries.io dataset [43], following similar research done in this area [1, 11, 22]. We used these repositories as the client projects for our study. To ensure the quality of the repositories, we used the GitHub API[3] to select repositories with at least five stars and that are not a fork repository following similar research in this area [22, 30]. Next, we cloned the repositories selected to the local machine on June 2021, giving us 20,711 repositories. We then selected only the repositories that had configured Maven as their build tool by searching for projects inside the repositories that maintained a `pom.xml` file, resulting in 7,019 repositories for the study.

We consider only repositories that currently successfully compile so that we can identify BCs through new compilation errors when manually updating dependency versions in our analysis of RQ2. The compilation of the projects was time-consuming as it required resolving the dependencies to the local repository. We did not change the Maven local repository settings when resolving the

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

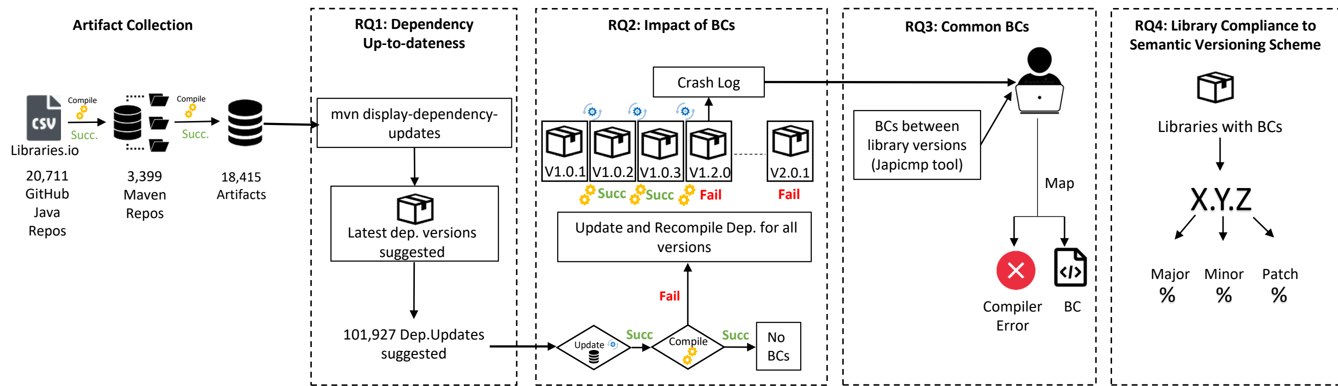


**Figure 3: Overview of the Study**

dependencies to prioritize the project-specific repository settings if configured in an individual project's `pom.xml` file. If local repository settings were configured for a project, these would take precedence over the project's `pom.xml` file, and Maven would download the dependencies from the specified repositories.

We compiled the projects using Java 8 and reran the failing projects using Java 11. We selected these Java versions because as of June 2021, when the repositories were cloned, these two Java releases and Java 7 were the only releases that received Long-Term Support from Oracle. Since Java 7 was released in 2011, and most of the projects we analyzed had the last commit to the project after 2011, we assumed that most of the projects would use Java 8 or 11. After compiling the Maven projects using the Maven command, 3,211 and 188 repositories (3,399 repositories in total) were successfully built on Java 8 and Java 11, respectively. Even though the number of projects built on Java 11 was few, it increased the total number of successfully built projects by 9%. Following the same procedure, we then compiled each Maven artifact with a unique GAV coordinate contained within these 3,399 repositories, resulting in 18,415 Maven artifacts that successfully compiled. These Maven artifacts will be the clients that we use for analysis. Hence, for the purpose of this research, we will examine the dependencies specified in these clients and assess the individual impact of updating each dependency, following a similar approach as conducted in a prior study [21].

## 3.2 RQ1: Dependency Up-To-Dateness

In this section, we answer **RQ1:** "To what degree are the dependencies in open-source repositories up-to-date?"

*3.2.1 Method:* For the 18,415 clients, we extracted 142,355 direct dependencies using Maven commands. To identify if these direct dependencies are up-to-date, we checked if new versions were available for them using the display-dependency-update Maven command. This command provides the latest versions for the dependencies that are outdated. These latest versions that are suggested could be either a `Major`, `Minor`, or `Patch` update. Some of the dependencies suggested through this command included 'alpha', 'beta', 'SNAPSHOT', and 'RC' versions, which are not stable releases according to the Maven Repository site and previous studies [32, 33].

**Table 1: Dependency Updates Suggested Based on the Semantic Versioning Level**

| Update level | Version | Number of updates | Percentage of updates |
|---|---|---|---|
| `Major` | | 31,464 | 22.10% |
| `Minor` | | 46,376 | 32.58% |
| `Patch` | | 22,818 | 16.02% |

Therefore, we omitted these unstable versions when considering the latest versions for the libraries. For this study, we did not take into account the analysis of Java as a dependency and the potential impact of version changes on client projects, as Java versions are widely recognized as being backward compatible [10]. Therefore, for the research, we maintain the Java version as a constant based on in which it was successfully compiled on.

For the outdated dependencies, to understand the degree of outdatedness, we developed scripts to determine whether the suggested dependency version would require a `Major`, `Minor`, or `Patch` update.

*3.2.2 Results:* Considering the current versions and the latest versions suggested for each dependency, we found that 71.60% of the dependencies in these clients were not up-to-date. This is closer to the findings of Salza et al. [41], who reported that 63% of the external libraries are never updated in mobile applications.

Table 1 gives a summary of the updates suggested at each semantic versioning level. Considering the total dependencies extracted from the clients, Maven suggested a `Major` update for 22.10% of the dependencies, a `Minor` update for 32.58%, and a `Patch` update for 16.02% of the dependencies.

Another 537 updates recommended could not be programmatically identified as to what semantic versioning level it could be categorized because they did not follow the correct semantic versioning scheme. Examples of some of the unclassified version updates are 9+181-r4173-1 -> 9-dev-r4023-3, 1.r.69-SNAPSHOT -> 1.r.69.20210929, 2.0.B1 -> 2.0.M1. Based on these results, dependency updates were suggested for 11,744 artifacts which shows that 43.79% artifacts had at least one outdated dependency. This is similar to the study by Wang et al. [44], who found that 54.9% of projects do not update half of their dependencies.

**Answering RQ1:** *To what degree are the dependencies in open-source repositories up-to-date?* According to our analysis, 71.60% of the dependencies in open-source projects were not up-to-date. For 22.10% of the dependencies, a `Major` update was available, and for 32.58% and 16.02%, respectively, `Minor` and `Patch` updates were suggested. Based on the artifacts used for the analysis, 43.79% had at least one outdated dependency.

## 3.3 RQ2: Impact of BCs

In this section, we answer **RQ2:** "How often do client-impacting BCs occur in the wild?"

*3.3.1 Method:* The next step in the research was to identify the BCs introduced by APIs on client projects. We will consider all `Major`, `Minor`, and `Patch` updates suggested in the previous section because previous research concludes that BCs are introduced not only in `Major` but in `Minor` and `Patch` versions as well [5, 32, 38]. We first identified BCs introduced due to source incompatibilities in APIs. We updated the declared version of the dependencies in the `pom.xml` to the latest versions suggested by Maven and checked if the dependency update causes compilation errors. If the update does cause a compilation error, it can signify that the new version could contain a BC. We updated the dependencies to the latest available stable version to mimic real world dependency updating. If a project is updating an outdated dependency, it would more likely select the latest stable release and not to the next adjacent version.

We developed a script to update one outdated dependency at a time in the `pom.xml` file and compile the client. We did this process for each outdated dependency in each client. We could not automatically update 31,464 dependencies, which is 22.10% of the total dependencies, for reasons such as the dependency or its version not being defined in the `pom.xml` or the dependency version being declared as a constant value which affects the entire project version and its dependencies. When updating the dependencies for clients defined inside a parent client, the dependency could be declared either in the child `pom.xml` file or the parent `pom.xml` file. Therefore, based on where the dependency update was applied, the script compiled either the child client or both the parent and child clients to verify if the dependency update was causing compilation errors.

Using the execution log, we determined if the client compiled successfully or not for each dependency update. Based on the build status, if the build was successful, no source BCs were encountered during the dependency update. Therefore, we marked it as 'Successful', and if the build failed, we marked it as 'Fail'. However, all these build failures might not be related to source BCs introduced by the libraries. There could be scenarios were a group of dependencies are updated together since the dependency version is defined as a common property for them and if latest version is not available for all the libraries this would cause the build to fail. Therefore, to identify if the build failure was related to the latest version being unavailable and to find the exact version the build failure was introduced, we recompiled the failed clients for the library versions between the current and latest. For each dependency update that encountered build failures, we wanted to identify the first version that introduced this compilation error to identify when the BC was first introduced and at which semantic versioning level. For this, we collected all version numbers released by a library between the current and the suggested breaking version using Maven Central [33] repositories. Starting from the version after the current version, we updated the declared dependency version and recompiled, one by one, up to the latest version until the version that introduced the BC was encountered.

To verify whether the BCs could also be identified through a static analysis tool, we used the japicmp tool, which detects the syntactic BCs introduced between two library versions. We selected this tool since it has been used in prior research [31, 32] and the tool's GitHub repository[4] is continuously maintained, having its last commit in March 2023 (as of May 2023). This tool takes two versions of a jar file and runs a static analysis on the changes between the two versions. It then classifies the changes as compatible or incompatible and determines whether they are source and binary incompatible.

Since the japicmp tool requires the jar files of the libraries to identify the syntactic BCs, we created a script to download all the dependency jars from the Maven Central repositories using the version numbers extracted previously. Then for each library, we passed the adjacent library versions through the japicmp tool to calculate the total source and binary BCs introduced through each version update. Next, we counted all the source and binary BCs to identify the total BCs that the versions introduced but did not impact the clients.

To automatically detect if the BCs extracted from the japicmp tool impact the clients, we identified the library functionality used by these clients and checked if those functionalities contain BCs. To identify the library calls used in the clients, we used the ASM Framework [7]. ASM analyzed the bytecode generated for the clients and extracted all the functional calls. Then we determined the functional calls external to the clients while excluding the inbuilt Java functionality. We mapped the external functional calls with the relevant dependency by identifying all dependencies connected to the clients using the Maven dependency tree. The dependency tree provided both direct and transitive dependencies used by the clients and at which level the transitive dependencies are connected to the client. If the same library existed in different versions at different levels of the dependency tree, we used the dependency mediation [19] technique to map the external call with the nearest dependency to the client. This approach mapped all external calls to the dependencies connected with the client.

We then matched the external calls of a library with all the reports generated by the japicmp tool for each adjacent library version to detect if BCs were detected for those calls. We only considered the library calls made to the direct dependencies for the analysis since this research only focuses on the impact of the BCs raised through direct dependencies.

*3.3.2 Results:* Table 2 presents the number of dependencies that could be automatically updated based on the total outdated dependencies extracted under RQ1. Out of all outdated dependencies 69.50% of the dependencies could be automatically updated. This contained both successful and failed dependency updates. Out of the total dependency updates that could be automatically applied, 56,003, which is 79.05% of the updates, were successful, while 14,837,

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

**Table 2: Dependency Update Results**

| Dependency Update | Occurrence | Percentage |
|---|---|---|
| Unable to update | 31,085 | 30.49% |
| Successful | 56,003 | 54.94% |
| Fail | 14,837 | 14.56% |

which is 20.97% of the updates, failed. Then we recompiled the dependency updates that failed, attempting to use all library versions between the current declared version and the latest available version to identify which is the first version to cause a failure. After recompiling the clients for versions between the current and the latest, the breaking version of the library was the latest version available for 12.58% dependency updates and was a version before the latest for 87.42% of the dependency updates.

After analyzing the crash logs of these build failures, only 8,207, which is 55.31% of the total build failures, contained a compilation error in the crash log. This means that based on the total dependency updates that failed and were successful, source BCs were encountered for 11.58% of the dependency updates. The amount of client-impacting source BCs is higher than found in previous research [32, 39] which used the adjacent library versions to identify impactful Binary BCs. It should be noted that this count includes all unique dependencies, but in some cases, dependencies will be bundled together (for example, a set of dependencies with a shared `groupid` that are released together and must be updated together). In some cases, a version parameter will be used in the `pom.xml` file to ensure these sets of clients are updated together. There were 1,611 of the compilation errors that could be considered repetitive due to counting each unique dependency rather than considering these as a set. If we remove these compilation errors from the total number of source BCs, we still find 9.31% dependency updates impact on client projects, which is higher than previous research [32, 39]. For the build failures which did not produce compilation errors, manual analysis of a subset of these failures found that they were related to unresolved dependency and maven configuration issues.

For the updates that contained compilation errors in the crash log, we counted all the successful compilations when updating all adjacent versions before encountering a build failure. Considering these successful dependency updates with the number of times compilation errors occurred during dependency updates, we find that source BCs resulted in 4.35% of the updates when considering all versions. This value represents the possibility an artifact will encounter a source BC when updated to the adjacent version. This is similar to the previous studies by Ochoa et al. [32] and Xavier et al. [46], who updated dependencies to the adjacent versions and reported results of 7.9% and 2.54% as client impacting binary BCs, respectively. However, these may not be realistic numbers since projects are more likely to update to the latest stable version when performing dependency updates.

Using the library calls made by the client and the JAPICMP tool output, we next checked if the version that the JAPICMP tool reports as containing source BCs is similar to the breaking version captured through extracting compilation errors. We did this to understand if the JAPICMP tool provides consistent results compared to the results we received through compiling the clients. 18.53% of the dependencies that reported source BCs due to compiler errors were identified by the JAPICMP tool as well. For 13.90% instances, the JAPICMP tool reported source BCs before the compilation error occurred, and for 4.81% instances after the compilation error occurred. Therefore, the results from the JAPICMP tool do not align with the results we received through identifying compilation errors. This is mainly because the tool analyses the BCs in isolation and not the entire artifact when making the decisions.

We next extracted source and binary BCs using the JAPICMP tool for all the dependencies that could be automatically updated. For the dependencies analyzed, the tool reported 9,221 binary BCs, which is 13.01% of the total dependency updates, and 11,444 source BCs which is 16.15% of the total dependency updates. The source BCs detected through the JAPICMP tool are higher than the source BCs detected by compiling the artifacts. This could be due to reasons like the JAPICMP tool reporting a BC if the Superclass of a class changed, even if the functionality inside the new Superclass is the same as the previous Superclass and does not impact the client. Another example is when a set of classes is removed from a library and added as a dependency to the library, this will be reported as a BC since the classes were removed. But in reality, those classes are still available for the client as a transitive dependency. Therefore some of the BCs reported by the JAPICMP tool are false positives.

**Answering RQ2:** *How often do client-impacting BCs occur in the wild?* After updating a dependency in an artifact and compiling, the build failed for 11.58% of the dependency updates. When we updated the dependencies incrementally to adjacent versions and accounted for all successful client compilations for versions, BCs were encountered for 4.35% of the dependency updates. The BCs reported by the JAPICMP tool were higher than the BCs detected through analyzing compilation errors, but some of the BCs reported through the JAPICMP tool are false positives because the tool does not evaluate the context of how the BC functionality is used in the client code and only considers the change in isolation when reporting BCs.

## 3.4 RQ3: Common BCs

In this section, we answer **RQ3:** "What are the common types of client-impacting source BCs?"

*3.4.1 Method:* For the client impacting syntactic BCs extracted in the previous RQ, we performed a manual analysis to identify the change in the library which caused the client to fail compilation.

When extracting compiler error messages, we disregarded the code-specific information such as the message location and reference information so that the same error messages could be classified together. The code-specific information was replaced with a '...' notation for tracking purposes. For example: 'package org.osgi.util.tracker does not exist' was extracted as 'package ... does not exist'. Following this approach, we extracted 105 types of compiler error messages. In one crash log, there were scenarios where multiple compiler error messages were present, and the same compilation error message was present in different locations in the client code.

These compilation error messages were counted as different instances as they could relate to different library changes. The total number of compilation errors was 94,737.

Next, for the manual analysis, we sampled the compilation error messages with a confidence interval of 95% and an error rate of 5% [45] which provided a sample size of 380. We used stratified sampling to create a balanced sample set covering all types of compiler error messages for the analysis. For the sample selected, we created a data set including the following details: the crash log, the line number in the crash log, the compilation error message, the dependency with its versions, the output of the JAPICMP tool, the GitHub URL for the client project, the class in which the compiler error occurred, and the Git-Diff Web URL link for the particular library versions.

To conduct the manual analysis, we followed the thematic analysis approach, which helps to identify patterns in qualitative analysis [9]. We developed codes to use for labeling the data, which would represent different types of syntactic BCs in the libraries. We followed an integrated approach [9] in developing the codes, where we started the analysis with a set of predefined codes and derived other codes while conducting the analysis. We created the predefined codes based on the different types of syntactic BCs introduced between two library versions identified by previous researchers [5, 15, 46] and an article published by IBM [12]. We created 101 predefined codes, categorized under Package, Interface, and Class levels.

Two of the authors, both familiar with Java programming, conducted the manual analysis to detect the change causing the compilation error. As O'Connor et al [35] recommends in their study to apply multiple coding between 10-25% of the data, we decided to use 20% of the data previously sampled for manual analysis by both coders. This gave us a sample size of 68 records which we selected using stratified sampling technique so that different compilation error messages were included in the sample set.

First, to get familiar with the codes and check the agreement on the coding, we randomly picked four samples for coding from the initially sampled data which are not included in the 68 sampled for multi coding. Each coder labeled the samples independently and checked the agreement. If more than one sample was labeled differently, then the agreement would be less than 75%, which will be below acceptable [47]. After the first coding round, the agreement was below acceptable; therefore, we discussed the issues we encountered in coding and how we extracted the coding values. We then repeated the process again until there was an agreement of more than 75% and both coders were familiar with the coding process.

After getting familiar with the coding process, the coders did the multi-coding separately on the 68 records sampled. To calculate the inter-rater reliability on the coded values, we used Cohen's Kappa coefficient, a statistical measure for calculating the inter-rater reliability of qualitative data [17]. This algorithm considers the two coders' agreement and random agreement when calculating the Kappa score. For the multi-coded values, we received a Kappa score of 0.68, which was a substantial agreement based on the interpretation of the Kappa score value. Therefore, since it took around 5-10 minutes to analyze one record in the sample, only one of the authors coded the rest of the samples in the manual analysis set, and the results received are presented in the following section.

**Table 3: Top Ten Compilation Errors which had the most Occurrences**

| Compiler Error | Occurrence | Percentage |
|---|---|---|
| cannot find symbol | 54,194 | 57.20% |
| package ... does not exist | 20,405 | 21.53% |
| method does not override or implement a method from a supertype | 3,186 | 3.36% |
| incompatible types: ... cannot be converted to | 2,403 | 2.54% |
| cannot access | 2,333 | 2.46% |
| reference to ... is ambiguous | 1,722 | 1.82% |
| static import only from classes and interfaces | 1,587 | 1.68% |
| no suitable method found for | 1,075 | 1.13% |
| is not abstract and does not override abstract method | 890 | 0.94% |
| method ... cannot be applied to given types | 871 | 0.92% |

*3.4.2 Results:* The ten compilation error messages that occurred the most are listed in Table 3. This shows that some compilation error messages are more frequent than other types.

Through the manual analysis, we noticed that there were five types of compiler error messages related to Groovy classes and another five that were not reproducible. This reduced the Java-related compiler error messages to 96. The labels derived through the manual analysis gave us more insight into BCs in the libraries, which impact the artifacts during dependency updates. These changes included the library being compiled in a different Java version which is incompatible with the artifact, a dependency of the library being modified, or the deprecated annotation being introduced to an interface, class, or method. Another reason for encountering BCs was the incompatibility of another library when the dependency is updated individually. The type parameter changes (generics in Java) we encountered during the analysis were introduced in the article by IBM [12]. However, it was discussed as a limitation under the research by Ochoa et al. [32].

The most common reason for the compilation error was transitive dependency changes which occurred 20.36% of the time. This occurred when the artifacts under analysis depended on the dependencies of its direct dependencies. Therefore, when the dependencies of the direct dependencies were updated, the artifact using those transitive dependencies was not compatible with the change. An incompatibility with another library when a single dependency was updated, and an incompatibility with the used Java version occurred 3.89% and 3.45% of the time, respectively.

The top ten changes in the library which caused source incompatibilities and their occurrence percentage are mentioned in Table 4. According to the results, changing the result type of a method [5] in a class is a common change across libraries which introduced source incompatibilities to client artifacts. Changes such as deleting an API package or a class also significantly impacted the clients.

Based on the analysis, we also notice that the same BC impacts artifacts differently based on how it has been used. For example, the 'Change result type of method in class' BC can affect a client in different ways based on how the method's return value is used.

[5] The term 'changing the result type' was taken from [12] and can be also referred to as 'changing the return type' of a method.

**Table 4: Top ten changes in libraries which causes Source Incompatibility with its percentage of occurrence**

| Library Change | Occurrance % |
|---|---|
| Change result type of method in class | 5.68% |
| Delete API package | 5.09% |
| Delete class | 3.89% |
| Rename API package | 2.10% |
| Delete type parameters from class | 2.10% |
| Decrease access of constructor in class | 2.10% |
| Delete method from class | 1.79% |
| Delete interface method | 1.50% |
| Delete interface | 1.50% |
| Delete checked exceptions thrown from method in class | 1.50% |

It can be used in a conditional statement comparison, a loop iteration, a method invoked based on the value, or passed as another method's return value. In all these scenarios, if the type of the value changes, it will cause the client to be affected. Also, if this method was overriden by a class in the client it will be affected. Therefore, understanding the context of how the dependency is used is essential for the dependency update process.

During the analysis, we found it challenging to identify the reason for some of the records by only considering the syntactic changes which could occur. For 16 of the samples (4.97%) analyzed, which were related to code generation libraries, the incompatibility was not due to a syntactic BC introduced between the two versions of the library. However, when analyzing the source code changes between the two library versions, we understood it was due to changes in the underlying logic related to the code generation process within the client's code base during compilation. Therefore, changes in code generation libraries that lead to syntactic BCs in clients might differ from other library changes which cause syntactic BCs hence will not be detected as a BC when using static analysis tools. For another 22 samples(6.83%), we could not identify the reason which caused the incompatibility, and we assume it could be related to a transitive dependency change.

Since transitive dependency changes were a significant factor in introducing syntactic BCs in clients, we analyzed the transitive dependency usage in clients. Given in Figure 4 is a dependency tree generated for the client 'org.opennms.newts:newts-metrics-reporter'. According to the dependency tree, this client has two direct dependencies, five transitive dependencies at level one, and six more transitive dependencies at level two. One of the direct dependencies is 'org.opennms.newts:newts-api' version 2.0.1-SNAPSHOT. 'com.google.guava:guava' version 23.0 is a direct dependency of 'org.opennms.newts:newts-api', which in return becomes a transitive dependency to the client.

In Figure 5, we have captured a code snippet in 'org.opennms.-newts:newts-api' client, which is invoking the com.google.common.-collect.Lists. newArrayList() method defined in the 'com.google.-guava:guava' version 23.0. This is a use of transitive dependency functionality in a client source code. The usage of transitive dependencies can cause syntactic BCs for different reasons. For instance, BCs would occur if this transitive dependency was removed by the

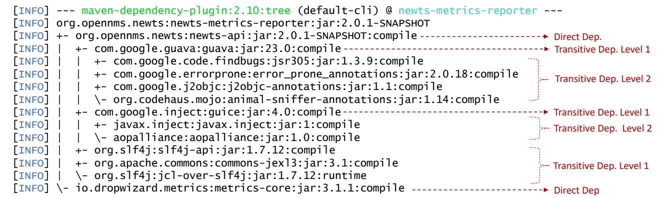


**Figure 4: Dependency Tree of 'org.opennms.newts:newts-metrics-reporter' MAVEN artifact**

```
public void report(SortedMap<String, Gauge> gauges, SortedMap<String, Counter> counters,
                   SortedMap<String, Histogram> histograms, SortedMap<String, Meter> meters,
                   SortedMap<String, Timer> timers) {
    Timestamp timestamp = Timestamp.fromEpochMillis(clock.getTime());

    List<Sample> samples = Lists.newArrayList(); ——> com.google.common.collect.Lists.newArrayList()

    for (Map.Entry<String, Gauge> entry : gauges.entrySet()) {
        reportGauge(samples, timestamp, entry.getKey(), entry.getValue());
    }

    for (Map.Entry<String, Counter> entry : counters.entrySet()) {
        reportCounter(samples, timestamp, entry.getKey(), entry.getValue());
    }

}
```

**Figure 5: 'org.opennms.newts:newts-api' artifact using functionality of 'com.google.guava:guava' by invoking com.google.common.collect.Lists.newArrayList() method**

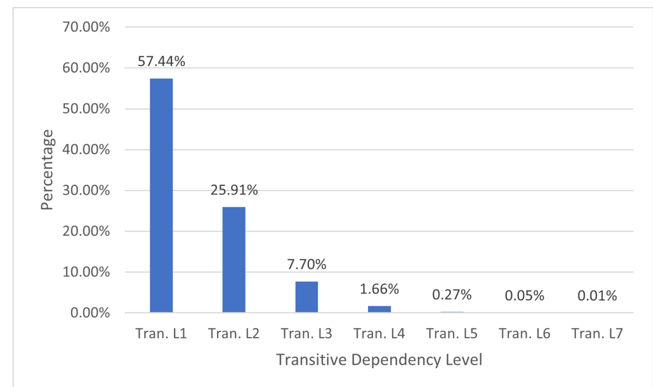


**Figure 6: Bar Chart representing maximum level of Transitive Dependency usage in client projects**

dependency declaring it or a new version of the transitive dependency, which contains BCs, is updated by the dependency declaring it.

Therefore, to understand to what extent the artifacts use transitive dependencies' functionality directly, we conducted an analysis to determine if it is common for artifacts to use transitive dependencies and to which level of transitive dependencies are used by the clients. For this we used the process explained under section 3.3.1 to extract the direct usage of transitive dependencies.

According to the data we received for transitive dependency usage conducted on the clients, 61.24% of all clients directly used at least one transitive dependency. Figure 6 displays a bar chart with the percentage of artifacts using transitive dependencies at

different levels. According to this chart, 57.44% of the artifacts used transitive dependencies at level one. It was also recorded that 2.95% of artifacts that used transitive dependencies at level one utilized more than ten unique transitive dependencies at that level. Even though level two and level three transitive dependencies were not as frequently used as level one, they reported 25.91% and 7.70% of usage, respectively.

We tracked the transitive dependency usage up to level ten, but none of the clients used transitive dependencies beyond level seven. Two clients utilized transitive dependencies at level seven, contributing to 0.01% of the total clients analyzed. We found that an client that uses a transitive dependency at level three does not always use transitive dependencies at the prior levels. Also, some clients that did not use direct dependencies' functionality used transitive dependencies' functionality. These results on transitive dependency usage in clients help us understand why most incompatibilities during dependency updates were linked to transitive dependency changes. Kikas et al. also confirm that transitive dependency usage is popular in software ecosystems such as Javascript [25].

**Answering RQ3:** *What are the common types of client-impacting source BCs?* The manual analysis showed that changes in transitive dependencies were the most common reason for syntactic BCs during dependency updates. The experiments conducted to verify the transitive dependency usage in clients showed that 61.24% of the clients had used transitive dependencies in their code. The top two changes in a library that introduces syntactic BCs are changing the result type of a method in a class and deleting an API package. The other library changes, which raised syntactic BCs, could not be derived as having a significant contribution than the others.

## 3.5 RQ4: Library Compliance to the Semantic Versioning Scheme

In this section, we answer **RQ4:** "Are client-impacting source BCs introduced in non-`Major` library releases?"

*3.5.1 Method:* Based on the library versions which introduced syntactic BCs, we ran an analysis to check if the libraries have introduced BCs under non-`Major` updates in their releases and analyzed the common syntactic BCs introducted at each semantic versioning level. We also looked at how BCs introduced by direct and transitive dependencies are spread across different versions under the semantic version levels.

*3.5.2 Results:* Table 5 summarizes the number of BCs that impact the clients at each semantic level. 58.41% of the BCs were introduced during a `Major` update. 33.49% and 8.09% of the BCs were introduced during a `Minor` and `Patch` update, respectively. Therefore non-`Major` updates contributed to almost half of the observed BCs.

From the sampled data that was manually analyzed, 147 records were `Major` updates, 138 were `Minor` updates, and 45 were `Patch` updates. During `Major` updates that introduced incompatibilities changing the result type of a method in a class, deleting a class, renaming an API package, and deleting an API package were common

**Table 5: Update level in libraries and the number of impacted artifacts**

| Update Level | Number of Impacting Artifacts | Percentage |
|---|---|---|
| `Major` | 4,742 | 58.41% |
| `Minor` | 2,719 | 33.49% |
| `Patch` | 657 | 8.09% |

**Table 6: Distribution of BC introduced by Direct and Transitive Dependencies at each Semantic Versioning Level**

| Source of BC | Semantic Version Level | | |
|---|---|---|---|
| | `Major` | `Minor` | `Patch` |
| Direct Dependency | 48.68% | 37.72% | 13.60% |
| Transitive Dependency | 38.57% | 54.28% | 7.14% |

BCs impacting clients. For a `Minor` update causing incompatibilities, the most significant BCs were deleting an API package, changing a static method to a non-static method in a class, and changing the result type of a class method. No BC contributed significantly to the incompatibilities during `Patch` updates.

We further analyzed how the BCs introduced by direct and transitive dependencies were spread across different semantic version levels. Table 6 shows that `Major` updates contain BCs introduced by direct dependencies more commonly than non-`Major` updates. While BCs introduced by transitive dependencies are common under non-`Major` updates. Although the semantic version scheme provides rules for introducing BCs at `Major` releases for libraries, many developers will be unaware of the BCs introduced by transitive dependencies.

**Answering RQ4:** *Are client-impacting source BCs introduced in non-`Major` library releases?* `Major`-level updates contained the highest number of BCs when updating dependencies in clients. However, non-`Major` updates contributed to 41.58% of the BCs, violating the semantic versioning principle. When considering the BCs introduced by transitive dependencies, non-`Major` versions contained the most BCs.

# 4 DISCUSSION

In this section, we discuss the findings of the study based on the observations of our results.

In the analysis for RQ1, we found that most of the dependencies used in the open-source repositories in our dataset were not up-to-date. Therefore, the client projects using these dependencies do not fully utilize the libraries' features and might also contain vulnerable code fixed by libraries in their later versions. This aligns with prior research and illustrates that better tools are needed to support project maintainers in keeping their dependencies up-to-date, particularly when BCs exist in the new versions.

While many of the BCs did not affect the client projects, we found that a significant number (12%) of the BCs would impact client projects according to RQ2. However, many of the BCs detected by the static analysis tools would not impact the client projects, and relying only on the output of static analysis tools is not reliable

for understanding how the dependency updates will impact the clients. One reason is because the static analysis tools we used did not consider the impact of transitive dependencies, which were the leading cause of syntactic BCs in our analysis under RQ3. Tools also need to consider the context of how the dependency is used when suggesting potential BCs. The current tool suggested false positives which are BCs that actually do not impact the clients. Previous research [3, 16, 40] has highlighted this problem with static analysis tools, leading to a decrease in developers' trust and confidence in their effectiveness. Thus, better support is needed for project maintainers to understand the impact of changes in all of their dependencies, including the transitive dependencies.

RQ3 further showed that deleting an entire API package was the second most common BC between library versions. However, current static analysis tools provide a lower-level of granularity when reporting changes in the libraries, indicating if a field, method, or class were removed for example. These tools could be improved to report the BCs at the appropriate level of granularity to help project maintainers better understand and resolve BCs.

While we saw under RQ4, that BCs that impact clients were more common in `Major` releases, which would be anticipated by developers based on the semantic versioning scheme, we saw transitive dependencies were more likely to introduce client-impacting BCs in non-`Major` releases, where client projects would not expect BCs. This further illustrates the need for better awareness of potential BCs in transitive dependencies.

We can draw several practical implications from our study. Library developers should be vigilant when introducing changes during non-`Major` updates, as BCs should not be introduced during these release versions. However, our study reports that many BCs exist in non-`Major` updates.

Client developers should think carefully if they want to directly use transitive dependencies, as using them leads to many BCs during dependency updates. Further, we also found that a BC will affect clients differently based on how the breaking functionality has been used in their code. Therefore, when fixing incompatibilities related to BCs, the client developers must consider the context in which the functionality is used. When applying dependency updates in isolation, they must consider whether that dependency depends on another dependency or vice-versa. Because if a dependency is updated in isolation, it might be incompatible with other dependencies in the artifact.

For researchers, investigating techniques to capture clients' transitive dependency usage and how they contribute towards BCs during dependency updates will be a much-needed research area, as our results show that changes in transitive dependencies were a significant factor in introducing BCs during dependency updates.

## 5 THREATS TO VALIDITY

One threat to construct validity of the research is how we selected successfully compiled artifacts for the analysis. When compiling the artifacts, we could not use the MAVEN compile command as some artifacts relied on the binaries of other artifacts in the same repository. Hence, we had to use the MAVEN install command when building the repositories. Therefore, initially, when selecting projects for our analysis, the projects that compiled but could not be packaged into an executable had to be omitted. We, therefore, could have missed some types of BCs that occurred in these projects.

Another threat to construct validity is that some transitive dependency calls might be missed through our analysis. For example, this could occur, when a client class (A) defines a superclass (B) from its direct dependencies, and that superclass (B) inherits features from another superclass (C) of one of its direct dependencies (transitive to the client). In such cases, we have not accounted for the transitive dependency calls at that level of granularity. If the client's direct dependency removed its direct dependency or if class C in the above scenario was removed from the dependency this would also cause an incompatibility to the client using the library. Thus, we could have underestimated the impact of transitive dependencies.

A threat to internal validity is the number of BCs reported due to an incompatibility with another library. This category of BCs could increase due to updating one dependency at a time. If a group of dependencies had to be updated together since they depend on one another, the approach we followed to update one dependency at a time would cause incompatibilities among the dependencies. However, some artifacts used one variable to define the dependency version for these dependent dependencies and referred to that variable when declaring each dependency version in the `pom.xml`, so the version was updated together for these groups.

An external threat to validity for this study is generalizing the results for other languages. We studied Java projects using MAVEN as its build tool. We used more than 18,000 MAVEN artifacts filtered based on quality factors. These artifacts used more than 9,000 unique Java libraries which we used for the analysis. Nevertheless, as with other empirical research in this area [21, 32, 39], the findings of this research cannot be generalized to other languages. Our study was also limited to analyzing projects that could successfully compile in Java 8 or 11, and our study did not capture specific incompatibilities for other Java versions. However, only 3.45% of the incompatibilities were related to an incompatible Java version, so it is unlikely that using additional Java versions would have changed the results significantly.

## 6 RELATED WORK

To the best of our knowledge, this is the first large-scale study to examine the impact of both source and binary BCs on client projects. We now discuss the closest related work on library evolution and BCs.

**Library evolution studies** Library evolution and stability have been widely studied in the literature due to their importance in software development [15, 26, 29, 46]. Changes introduced during library evolution were categorised by Dig et al. as Breaking and non-Breaking Changes, and the BCs were further categorized as syntactic and semantic BCs [15]. They found that 80% of all BCs are introduced through syntactic BCs [15]. In another study, Dietrich et al. analysed a set of libraries and all their adjacent versions and found that 75% of the versions contain BCs [13]. Research has found that the frequency of BCs between adjacent library versions has increased over time [46]. These studies illustrate the prevalence of BCs in libraries and also inspired us to focus on syntactic BCs.

Since BCs can cause failures in clients, it is important for clients to be made aware when BCs are introduced. BCs should ideally be documented in changelogs and release notes [4, 6], but prior

research has found that less than 50% of BCs are documented [26]. Since a lot of BCs are not documented, clients need to know when it is safe to upgrade. The semantic versioning scheme provides another way for library developers to indicate when new versions contain BCs. However, studies have found that libraries do not strictly follow the semantic versioning principles, introducing breaking changes also in non-`Major` releases [14, 24, 38, 39]. Our research confirms these findings by providing further evidence that BCs are introduced in `Minor` and `Patch` releases. However, these studies all focused on the BCs introduced between library versions without considering the impact of these BCs on their clients. Differently, our study analyses the impact of BCs on client projects.

**BCs impact on client projects** Some recent research has investigated how syntactic BCs introduced during library evolution impact their clients [2, 24, 46]. Xavier et al. [46] and Bavota et al. [2] reported that only 2.54% and 5% of the clients in their dataset were impacted due to BCs, respectively. However, these studies estimated the impact of BCs by considering the import statements of library functionality in the client code, which can overestimate the impact for certain BCs and overlook BCs that cannot be traced through direct imports. Jezek et al. studied binary BCs in client projects and found that only 12 clients were impacted due to binary BCs, indicating that client-impacting binary BCs are relatively rare [24]. Similarly, Ochoa et al. [32] reported that 7.9% of clients are potentially impacted by binary BCs. On the other hand, Raemaekers et al. [39] reported that BCs do have a significant impact in client environments, but they experimented by injecting each BC one by one into the older library version, which may overestimate the impact of real dependency updates. Similar to our work, these studies analysed the impact of BCs on client projects. However, Jezek et al. [24], Raemaekers et al. [39] and Ochoa et al. [32] focused on only binary BCs, while our study considered both source and binary BCs. In addition to also considering source BCs, we also did not only focus on updates to adjacent versions of the library like all previous research, which may not represent the version projects would realistically select when performing dependency updates. Therefore, our results revealed that syntactic BCs can have a more significant impact on clients than what previous studies have previously reported.

**BC detectors** Tools have been created to detect BCs between two library versions. RefDiff [42] and ApiDiff [5] both classify breaking and non-BCs in Java libraries based on the syntactic changes applied to its source code. However both these tools focus on 13 refactoring operations, which does not cover all syntactic BCs.

Static analysis tools such as Clirr [6], SigTest [7], Japicmp, JapiChecker, JapiTool [8], and Revapi [9] use both source and binary code analysis to identify incompatibilities between two API versions.

All these tools extract potential BCs between library versions from the libraries perspective, therefore they can provide many false positives for clients who likely use only a subset of the functionality provided by the library. Ochoa et al. [32] developed Maracas to automatically identify the breaking declarations that are used in the client code using a static analysis technique. This tool has limitations that impact its BC detection capability. For example, it is unable to detect inheritance hierarchies, overridden methods, and exception handling. Our approach reported BCs irrespective of whether static analysis tools identified backward compatibility (BC) issues or not. By doing so, our findings are not influenced by the detection limitations of existing tools, distinguishing our work from the previous study [32]. This approach yielded novel insights and results. Notably, we discovered that transitive dependencies are the primary contributors to BCs in clients. BC detectors are unable to identify BCs introduced by transitive dependencies since they typically analyze multiple versions of a single library.

[6]https://clirr.sourceforge.net/

[7]http://wiki.apidesign.org/wiki/SigTest

[8]https://packages.debian.org/stretch/devel/japitools

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

## 7 CONCLUSION

In this paper, we conduct an empirical analysis of BCs that are encountered when updating software dependencies using 142,355 direct dependencies declared in 18,415 clients (Maven artifacts).

Through this study, we looked at the outdated dependencies in clients and concluded that 71.60% of the dependencies used in the clients were not up-to-date with the latest version available for that library. These outdated dependencies were distributed among 43.79% of the clients. When updating these dependencies to the latest version available, 11.58% failed to update as they contained BCs that impacted the artifact. The most common BC in the library that caused incompatibilities in a client was changing the result type of a method in a class. The most common change that caused incompatibilities in a client was transitive dependency changes between library versions. Based on the dependency update failures at each semantic version level, non-`Major` updates contained 41.58% of BCs changes, violating the semantic versioning principle.

In our future work, we aim to broaden our research scope by analyzing Java projects that utilize Gradle as their build tool. Additionally, we intend to explore similar methodologies in other statically-typed programming languages to identify the impact of BCs introduced by their libraries. Our focus will also shift towards investigating transitive dependency usage and the BCs they introduce in artifacts. Another important future work is to study the impact of behavioral BCs on clients, which is an understudied problem.

## ACKNOWLEDGMENTS

This work was supported by the Marsden Fund Council from Government funding, administered by the Royal Society Te Apārangi. 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] 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

[2] 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 Computer Society, USA, 280–289. https://doi.org/10.1109/ICSM.2013.39

[3] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (feb 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[4] 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

[5] 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

[6] 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. https://doi.org/10.1109/SANER.2018.8330214

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

[8] 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. https://doi.org/10.1109/ICSE.2015.140

[9] Daniela S. Cruzes and Tore Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement(ESEM)*. IEEE, 275–284. https://doi.org/10.1109/ESEM.2011.36

[10] Joe Darcy. 2021. *Kinds of Compatibility*. https://wiki.openjdk.org/display/csr/Kinds+of+Compatibility

[11] 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

[12] Jim des Rivières. 2017. *Evolving Java-based APIs 2*. https://wiki.eclipse.org/Evolving_Java-based_APIs_2

[13] 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

[14] 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

[15] 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

[16] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (jul 2019), 62–70. https://doi.org/10.1145/3338112

[17] Khaled El Emam. 1999. Benchmarking Kappa: Interrater agreement in software process assessments. *Empirical Software Engineering* 4 (1999), 113–133.

[18] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient Static Checking of Library Updates. In *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

[19] The Apache Software Foundation. 2023. *Apache Maven Project*. https://maven.apache.org/

[20] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2021. *The Java language specification*. Oracle America, Inc.

[21] Nicolas Harrand, Amine Benelallam, César Soto-Valero, François Bettega, Olivier Barais, and Benoit Baudry. 2022. API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages. *Journal of Systems and Software* 184 (2022), 111134. https://doi.org/10.1016/j.jss.2021.111134

[22] 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

[23] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. 2023. *Replication Package for Understanding Breaking Changes in the Wild*. https://doi.org/10.5281/zenodo.7978507

[24] 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

[25] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and Evolution of Package Dependency Networks. In *14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17)*. IEEE Press, 102–112. https://doi.org/10.1109/MSR.2017.55

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

[27] 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

[28] 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

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

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

[31] Lina Ochoa, Thomas Degueule, and Jean-Rémy Falleri. 2022. BreakBot. In *ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. ACM. https://doi.org/10.1145/3510455.3512783

[32] 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

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

[34] Oracle. n.d.. *Java Virtual Machine Specification: Chapter 5. Loading, Linking, and Initializing*. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html

[35] Cliodhna O'Connor and Helene Joffe. 2020. Intercoder Reliability in Qualitative Research: Debates and Practical Guidelines. *International Journal of Qualitative Methods* 19 (2020), 1609406919899220. https://doi.org/10.1177/1609406919899220

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

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

[38] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2014. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 215–224. https://doi.org/10.1109/SCAM.2014.30

[39] 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

[40] 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 (mar 2018), 58–66. https://doi.org/10.1145/3188720

[41] 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

[42] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 269–279. https://doi.org/10.1109/MSR.2017.14

[43] Inc Tidelift. 2022. *Libraries.io - The Open Source Discovery Service*. https://libraries.io/data

[44] 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

[45] Thomas H. Wonnacott and Ronald J. Wonnacott. 1991. Introductory Statistics.

[46] 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

[47] Zach. 2021. *What is Inter-rater Reliability*. https://www.statology.org/inter-rater-reliability/

[48] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2022. Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556956


Received 2023-02-16; accepted 2023-05-03