Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central

An external and differentiated replication study

Lina Ochoa · Thomas Degueule · Jean-Rémy Falleri · Jurgen Vinju

Received: date / Accepted: date

Abstract Just like any software, libraries evolve to incorporate new features, bug fixes, security patches, and refactorings. However, when a library evolves, it may break the contract previously established with its clients by introducing Breaking Changes (BCs) in its API. These changes might trigger compile-time, link-time, or run-time errors in client code. As a result, clients may hesitate to upgrade their dependencies, raising security concerns and making future upgrades even more difficult.

Understanding how libraries evolve helps client developers to know which changes to expect and where to expect them, and library developers to understand how they might impact their clients. In the most extensive study to date, Raemaekers *et al.* investigate to what extent developers of Java libraries hosted on the Maven Central Repository (MCR) follow semantic versioning conventions to signal the introduction of BCs and how these changes impact client projects. Their results suggest that BCs are widespread without regard for semantic versioning, with a significant impact on clients.

In this paper, we conduct an external and differentiated replication study of their work. We identify and address some limitations of the original protocol

Lina Ochoa
E-mail: l.m.ochoa.venegas@tue.nl
Eindhoven University of Technology, Eindhoven, The Netherlands

Thomas Degueule
E-mail: thomas.degueule@labri.fr
Jean-Rémy Falleri
E-mail: falleri@labri.fr
Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI, Bordeaux, France

Jurgen Vinju
E-mail: jurgen.vinju@cwi.nl
Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
Eindhoven University of Technology, Eindhoven, The Netherlands

and expand the analysis to a new corpus spanning seven more years of the MCR. We also present a novel static analysis tool for Java bytecode, `Maracas`, which provides us with: (i) the set of all BCs between two versions of a library; and (ii) the set of locations in client code impacted by individual BCs.

Our key findings, derived from the analysis of 119,879 library upgrades and 293,817 clients, contrast with the original study and show that 83.4% of these upgrades do comply with semantic versioning. Furthermore, we observe that the tendency to comply with semantic versioning has significantly increased over time. Finally, we find that most BCs affect code that is not used by any client, and that only 7.9% of all clients are affected by BCs. These findings should help (i) library developers to understand and anticipate the impact of their changes; (ii) library users to estimate library upgrading effort and to pick libraries that are less likely to break; and (iii) researchers to better understand the dynamics of library-client co-evolution in Java.

Keywords software evolution, API evolution, breaking changes, backwards compatibility, Maven Central

1 Introduction

Just like any software, libraries evolve to incorporate new features, bug fixes, security patches, and refactorings. It is critical for clients to stay up to date with the libraries they use to benefit from these improvements and to avoid technical lag [18, 49]. When a library evolves, however, it may break the contract previously established with its clients by introducing Breaking Changes (BCs) in its public Application Programming Interface (API), resulting in compilation-time, link-time, or run-time errors. These errors burden client developers given the sudden urgency to fix issues without intrinsic motivation. As a result, clients may hesitate to upgrade their dependencies, raising security concerns and making future upgrades even more difficult [26, 32].

BCs are language-specific: they vary with the syntax and semantics of a particular programming language. In Java, seemingly innocuous changes such as altering the visibility or abstractness modifier of a type declaration, or simply inserting a new method into an abstract type can, under certain conditions, break client code [19]. Most refactoring operations, though essential to maintain and evolve libraries, also induce BCs. Thus, it does not come as a surprise that BCs are widespread in Java libraries [46]. It is, however, essential to realize that not all BCs are intrinsically harmful. Nonetheless, they should not come unannounced and take clients by surprise: it should be clear for clients what consequences upgrading their dependencies will have on their own software, so they can make an informed decision beforehand.

To this end, Java library developers can leverage various mechanisms to communicate with their clients on the stability of their APIs and the effort required to upgrade to a newer version. These mechanisms enable them to specify *when* and *where* BCs are to be expected. On the one hand, semantic versioning (`semver`) enables developers to use well-defined versioning conventions to classify new library releases as *major* releases (which may introduce BCs), *minor* releases (which may introduce new backward-compatible features but should not introduce any BC), *patch* releases (which should not affect the public API whatsoever), and *initial development* releases (which may break anything at any time) [34]. On the other hand, annotations directly placed on source code elements (*e.g.*, Google's `@Beta` and Apache's `@Internal`) and naming conventions (such as *internal* and *experimental* packages) can be used to indicate that certain parts of the public API are exempt from compatibility guarantees and subject to sudden changes.

Clients who upgrade towards a new major release of a library or early adopters who rely on beta-stage APIs are well-aware of the consequences. It is thus crucial to distinguish between libraries that evolve gracefully by introducing BCs only when and where appropriate, and those that "break bad" by introducing BCs in minor and patch releases or in allegedly stable APIs.

In the most extensive study to date, Raemaekers et al. dissect backwards compatibility issues in the Maven Central Repository[1] (MCR) with respect to semantic versioning [39]. The study uses the tool `clirr` to infer the list of BCs between two versions of a Java library and measures their impact on client code using the Java compiler itself. The empirical evaluation is carried on a complete snapshot of MCR, up to the year 2011. To name but a few of their findings, the study concludes that: (i) BCs are widespread without regard for versioning conventions; (ii) the adherence to semantic versioning principles has increased over time; and (iii) BCs have a significant impact on clients. The relevance and quality of this study for understanding the API-client co-evolution problem motivates us to replicate and expand its protocol and corpus.

In this paper, we conduct an external and differentiated replication study [31] of the study by Raemaekers et al. [39], which from now on we will refer to as the *original study*. After reviewing the original protocol, we introduce major changes to alleviate some of its limitations and address key threats to its validity. The main differences between our study and the original study are as follows:

– We refine the original protocol by introducing new filters and sanity checks to avoid analysing Maven artefacts that are not used as libraries and versions that are not meant to be used by clients—only 12% of all artefacts in our replication corpus are indeed used as libraries;
– We implement a new tool built atop `japicmp`,[2] `Maracas`, more accurate than `clirr`, which we use to analyse Java bytecode and compute the set of BCs between two versions of a library, as well as to compute how client projects are impacted by individual changes;
– We re-analyse the original corpus to assess the impact of our new protocol and tool, and expand the analysis to a new corpus spanning seven more years of the MCR (from 144K Maven artefacts to 2.4M).

[1] `https://search.maven.org/`
[2] `https://siom79.github.io/japicmp/`

We focus on a subset of three of the research questions investigated in the original study which are central to the API-client co-evolution problem, eluding other less relevant questions related to deprecation tags and characteristics of libraries that break more, among others. Our research questions are as follows:

Q1 How are semantic versioning principles applied in the Maven Central repository in terms of BCs?
Q2 To what extent has the adherence to semantic versioning principles increased over time?
Q3 What is the impact of BCs on clients?

Our results show that, overall, library and client projects on Maven Central Repository are *not* "breaking bad". First, 83.4% of all library upgrades comply with `semver` principles, introducing BCs only when they are expected. However, 20.1% of non-major releases are breaking, being a potential threat to their clients. Second, the tendency to comply with `semver` practices has significantly increased over time. In particular, the ratio of non-major releases introducing BCs has gradually decreased from 67.7% in 2005 to 16.0% in 2018. Third, only 7.9% of clients are actually impacted by BCs introduced in library releases. In most cases, clients do not use the breaking declarations (*i.e.*, the library declarations affected by BCs)—but when they do, they are very likely to break. These results should help library developers to understand and anticipate the impact of their changes; library users to estimate library upgrading effort and to pick libraries that are less likely to break; and researchers to better understand the dynamics of client-library co-evolution in Java and prioritize research in the future.

The remainder of this paper is organized as follows. We first introduce background notions on Maven, `semver`, and backwards compatibility in Java in Section 2. We then briefly present the original study in Section 3. In Section 4, we detail the key differences in the protocol and datasets for our replication study. We discuss our new results in Section 5 and then present related work in Section 6. We then discuss the key findings and implications of our study in Section 7 and finally conclude the paper and discuss future work in Section 8.

2 Background

In this section, we first introduce some background notions on Apache Maven, APIs, and backwards compatibility in Java. We also discuss the mechanisms available to developers to communicate on the stability of their libraries through versioning conventions and source code annotations.

2.1 Apache Maven

Apache Maven (simply referred to as Maven hereafter) is a build automation tool particularly popular in the Java ecosystem. Maven follows a plugin-oriented architecture that enables developers to specify the dependencies of a particular piece of software and how to build it. When used to build Java projects, it enables developers to convert Java source code to Java bytecode (`.class` files) typically bundled as Java ARchives (JARs), which potentially depend on other JARs. These artefacts can be deployed to and retrieved from remote Maven repositories. The most popular Maven repository is the Maven Central Repository (MCR) which, as of May 2021, hosts $6,723,367$ artefacts.

The cornerstone file defining a Maven project is the Project Object Model (POM) file. Typically, the POM file is an XML file that contains metadata about the current project, its dependencies, and additional configurations required to build it. Listing 1 illustrates the typical structure and tags defined within a POM file, using the Spring TestContext Framework as an example. The **`modelVersion`** tag specifies the POM version of the file; the **`groupId`** tag identifies the organization or group that develops the project (`org.springframework`); the **`artifactId`** tag identifies the project itself (`spring-test`); the **`version`** tag specifies the current version of the project (*e.g.*, `4.2.5.RELEASE`); and the **`packaging`** tag specifies how the project is packaged (*e.g.*, `jar`). Together, the group, artefact, and version (also known as *project coordinates* and denoted **`groupId`**`:`**`artifactId`**`:` **`version`**) uniquely identify a Maven artefact.

Dependencies of a project are declared within the **`dependencies`** tag. Each **`dependency`** points to a unique Maven artefact (using its project coordinates), possibly supplemented with additional metadata. In particular, the **`scope`** tag specifies when the dependency is needed and thus in which classpath(s) it is included (*e.g.*, compile-time, test-time, or run-time dependencies). One can automatically determine which libraries a Maven artefact depends on by parsing its POM file. In Listing 1, the Spring TestContext Framework declares a compile-time dependency towards the JavaServlet library version `3.0.1`. Note that dependencies may employ version constraints (*e.g.*, `[1.0, 2.0)`), letting the dependency resolver find a suitable version within this range.

Listing 1: Excerpt of the POM file of the Spring TestContext Framework project version `4.2.5.RELEASE`.

```xml
<project>
   <modelVersion>4.0.0</modelVersion>
   <groupId>org.springframework</groupId>
   <artifactId>spring-test</artifactId>
   <version>4.2.5.RELEASE</version>
   <packaging>jar</packaging>
   <dependencies>
      <dependency>
         <groupId>javax.servlet</groupId>
         <artifactId>javax.servlet-api</artifactId>
         <version>3.0.1</version>
         <scope>compile</scope>
      </dependency>
   </dependencies>
```

```java
public interface HttpServletRequest
  extends ServletRequest {
    public String getAuthType();
    public String getMethod();

    [...]
}
```

(a) JavaServlet version `3.0.1`.

```java
public interface HttpServletRequest
  extends ServletRequest {
    public String getAuthType();
    public String getMethod();
    public String changeSessionId();
    [...]
}
```

(b) JavaServlet version `3.1.0`.

```java
public class MockHttpServletRequest implements HttpServletRequest {
  @Override public String getAuthType() {
    return this.authType;
  }
  @Override public String getMethod() {
    return this.method;
  }
  // MockHttpServletRequest must implement method HttpServletRequest.changeSessionId()
}
```

(c) Spring TestContext Framework version `4.2.5.RELEASE`.

Fig. 2.1: Breaking change example: Adding a new abstract method in the `HttpServletRequest` interface will break the client type `MockHttpServletRequest`.

```
</project>
```

2.2 API Evolution & Backwards Compatibility

An Application Programming Interface (API) is an interface that exposes the set of services from a library that can be invoked by client projects. In Java and other object-oriented languages, this interface consists of programming constructs such as *packages*, *types*, *methods*, and *fields*. To delimit this interface, library developers use visibility modifiers and other dedicated constructs provided by the host language [15].

As an environment changes, software used in such environment face the need to change accordingly. This is what Lehman [29] coined as *software evolution*, later formalized as a set of eight Lehman's laws that synthesizes observations about software evolution [17, 30]. Consequently, APIs—being software themselves—undergo continual and progressive change over time. The motivation behind this evolution is to provide more value to users by patching security issues, adding new features, simplifying the current API, fixing bugs, and improving maintainability [13, 27].

API evolution comes with the introduction of changes that can be classified according to how they affect client projects [15] and specifically whether they ensure *backwards compatibility.* In Java, backwards compatibility is defined at the source, binary, and behavioural levels [13]. *Source compatibility* is checked by the compiler when recompiling a client project with the new version of an API. *Binary compatibility* is checked by the Java Virtual Machine (JVM) during the linking process, as described in Chapter 13 of the Java Language Specification (JLS) [12, 19]. Lastly, *behavioural compatibility* can only be verified at run time to check whether the program exhibits a behaviour that is different from its previous version, without triggering compilation or linkage errors [13].

There are two types of API changes, namely breaking and non-breaking changes. On the one hand, *breaking changes* (BCs) are not backwards compatible: client projects using an API entity affected by a BC might break when migrating to a more recent version of the API [15]. On the other hand, *non-breaking changes* (NBC) are backwards compatible, meaning that they do not trigger any source, binary, or behavioural incompatibility. If an API only introduces backwards compatible changes, it is said to be *stable.* It is important to note that some BCs break several kinds of compatibility (*e.g.*, removing a public method is both source and binary incompatible), but none is a super set of the other [23]. In this paper, to align with the original study, we only consider *binary compatibility* and the associated set of BCs.

To illustrate how backwards incompatible changes might impact client projects, we refer to the Spring TestContext Framework example. The JavaServlet library releases version `3.1.0` in April 2013. This happens almost two years after its latest major release (*i.e.*, `3.0.1`) in July 2011. This new version introduces backwards incompatible changes that might break client code. Some of those changes include adding new abstract methods to classes and interfaces. These changes can potentially impact the Spring TestContext Framework in its `4.2.5.RELEASE` version. In some cases, stating that a BC affects client code is straightforward. For instance, removing a type, method, or field that is used by a client will obviously break this client. However, client code may also break when inserting new declarations in the library, for instance when inserting a new abstract method in an interface. This change will break client code if it extends this interface, as illustrated in Figure 2.1. In this case, the `changeSessionId()` method is added to the `HttpServletRequest` class within the JavaServlet library. Given that the `MockHttpServletRequest` class in the Spring TestContext Framework implements such interface, it will be forced to implement the new abstract method resulting in broken code. The literature often overlooks the BCs induced by uses of a library in an Inversion of Control (IoC) style (*i.e.*, where the client extends types exposed in the library, following the Hollywood principle "*don't call us, we'll call you!*") [6, 46]. In contrast, we include all of those in our analyses. An exhaustive list of the 31 BCs we consider in this paper is available on the companion webpage.[3]

[3] `https://crossminer.github.io/maracas/detections/`

2.3 API Stability Conventions

It is critical for clients to be able to pinpoint which versions and which parts of an API introduce changes that might break their code. *Semantic versioning*, also known as `semver`, is a popular convention to announce the introduction of BCs, and its use is encouraged in many software ecosystems (*e.g.*, npm, RubyGems, Cargo, Maven Central) [11]. This versioning scheme is used to label library versions according to compatibility guarantees. Each version number is specified in the form `<major>.<minor>.<patch>`, where `major`, `minor`, and `patch` are non-negative integers. A change in the `major` version signals the possible introduction of backward-incompatible changes. Changes in the `minor` or `patch` versions signal the introduction of new features or bug fixes in a backward-compatible fashion [34]. Initial development releases, which use zero as `major` version, should also be considered unstable ("*[m]ajor version zero (`0.Y.Z`) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable*" [34]). Finally, version numbers may be suffixed with hyphen-separated qualifiers specifying pre-releases or build metadata (*e.g.*, `2.1.1-beta2`).

At the code level, library developers may use annotations such as Google's `@Beta` and Apache's `@Internal` to signal unstable declarations. For instance, Guava developers state that "*APIs marked with the `@Beta` annotation at the class or method level are subject to change. They can be modified in any way, or even removed, at any time*,"[4] and Apache POI developers state that "*Program elements annotated `@Internal` are intended for [...] internal use only. Such elements are not public by design and likely to be removed, have their signature change, or have their access level decreased [...] without notice.*"[5] Naming conventions on packages (*e.g.*, *internal* and *experimental* packages) are sometimes used for the same purpose [9]. For instance, the following comment is attached to the class `Finalizer` contained in the package `com.google.common.base.internal` of Guava: "*While this class is public, we consider it to be *internal* and not part of our published API. It is public so we can access it reflectively across class loaders in secure environments*".[6] This comment highlights the lack of mechanisms for developers to fine-tune the boundaries of their APIs in languages such as Java. Some elements are made public because of technical constraints and not because of the desire to expose these elements in the API; developers must therefore rely on band-aid solutions such as naming conventions. When used in relation with `semver`, these code-level mechanisms enable library developers to delimit a portion of their API that escapes the strict rules regarding backwards compatibility. That is, BCs can be introduced in declarations labelled with these mechanisms without regard for `semver`.

[4] `https://guava.dev/#important-warnings`

[5] `https://poi.apache.org/apidocs/dev/org/apache/poi/util/Internal.html`

[6] `https://guava.dev/releases/9.0/api/docs/com/google/common/base/internal/Finalizer.html`

3 Original Study

In this section, we briefly introduce the original study object of this replication. We present its goal, main findings, and the protocol used to answer its research questions.

The original study by Raemaekers, van Deursen, and Visser, entitled "*Semantic versioning and impact of breaking changes in the Maven repository*" and published in *The Journal of Systems and Software* in 2017, investigates whether API developers use versioning practices to signal backwards incompatibility, and how unstable interfaces impact client projects in terms of compilation errors [39]. Although the original study is organized around seven research questions, we decide to focus our effort on three of them that are specifically aimed at understanding the API-client co-evolution problem. In particular, they address the relation between BCs and versioning conventions, and the impact of BCs on client code. The main findings of the original study are summarized in the following statements. Each of these answers one of the research questions we selected: statement H_i corresponds to question **Qi**. In this paper, we reuse these results as new hypotheses, which we aim to test under different conditions for replication purposes.

Q1 How are semantic versioning principles applied in the Maven Central repository in terms of BCs?
H_1 *BCs are widespread without regard for semantic versioning principles.*
Q2 To what extent has the adherence to semantic versioning principles increased over time?
H_2 *The adherence to semantic versioning principles has increased over time.*
Q3 What is the impact of BCs on clients?
H_3 *BCs have a significant impact in terms of compilation errors in client systems.*

On the one hand, the study relies on `clirr` [28] to study backwards compatibility. This tool is used to compute the list of changes between two versions of a Java library. However, the development of `clirr` has stopped in 2005, and Jezek and Dietrich [22] later showed that it is the least sound of a list of 9 tools for BCs detection in Java. On the other hand, to identify the impact of BCs on client code, the original study uses a novel approach that isolates individual changes on the newer version of the API, and injects them one by one in the older version. Then, clients are recompiled against each variant of the old version. The number of compilation errors raised by the Maven compiler is used as a proxy to measure the impact of BCs. As main corpus, the study uses a snapshot of MCR dated July 2011, consisting of 148,253 JARs and named the Maven Dependency Dataset. In the next section, we dive deeper into the design of our replication study to highlight how it differs from the original study in terms of protocol and corpora.

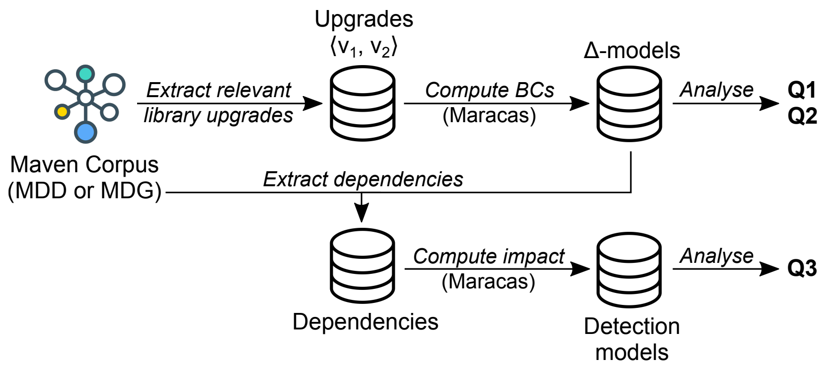


Fig. 4.1: Overview of the analysis protocol.

4 Design of the Replication Study

In this section, we present the protocol of our replication study, summarized in Figure 4.1. The source material of our study is extracted from two different corpora: the Maven Dependency Dataset (MDD) [37], which is used in the original study, and the Maven Dependency Graph (MDG) [3]. These two corpora are snapshots of MCR containing metadata information about artefacts, versions, and dependencies between artefacts. The MDD includes all artefacts from the MCR up to 2011, while the MDG spans seven more years up to 2018. However, due to subtle differences in the methodology used to build these snapshots, the MDD is not strictly a subset of the MDG. In this study, we run the very same analysis protocol on both corpora. On the one hand, re-analysing the MDD enables us to assess the impact of our updated protocol on the results obtained in the original study. On the other hand, analysing the MDG enables us to broaden the scope of analysed artefacts and strengthen our conclusions.

First, to answer **Q1** and **Q2**, we extract relevant upgrades for all libraries in the corpora, *i.e.*, pairs of adjacent releases (*e.g.*, JavaServlet versions 4.0.0 and 4.0.1) that conform to the selection criteria presented in Section 4.1.2. The outputs of this task are the *upgrades datasets* $\mathcal{D}_u^o$ (for the MDD) and $\mathcal{D}_u^r$ (for the MDG). Then, we use our tool `Maracas` to compute delta models (Δ-*models*) that store the list of BCs introduced in a particular upgrade between two versions of a library. We analyse the resulting Δ-models in Section 5 to answer our first two research questions.

Second, to uncover the impact of BCs on client code and answer **Q3**, we build the dependencies datasets $\mathcal{D}_d^o$ and $\mathcal{D}_d^r$ which consist of all clients in the corresponding corpus that might be impacted by the changes identified in a Δ-model (*i.e.*, all artefacts declaring a dependency towards a library upgrade extracted previously). We again use `Maracas` to identify locations in these clients that are impacted by BCs. The output is stored in a set of *detection models*, where BCs are linked to affected locations in client code. We analyse the resulting models in Section 5 to answer our last research question.

The remainder of this section is structured as follows. Section 4.1 describes the data extraction process of the protocol. Then, Section 4.2 gives an overview of our static analysis tool, `Maracas`. Finally, in Section 4.3, we highlight the key differences between our protocol and the original study's protocol in terms of data selection, filtering, and treatment.

4.1 Data Extraction

In this section, we introduce the two corpora used in this study, together with the datasets derived from them to answer our research questions.

4.1.1 Corpora

Our study relies on two corpora: the Maven Dependency Dataset (MDD) and the Maven Dependency Graph (MDG). The former is used to verify whether the main findings of the original study hold when following a different protocol, while keeping the same base data. The latter is included to assess whether the conclusions of the original study remain valid on a larger population, and whether the phenomenon under study (BCs and `semver` in MCR) has evolved between 2011 and 2018.

The Maven Dependency Dataset (MDD). The MDD is a publicly available snapshot of the MCR dated July 30, 2011 [35]. The corpus contains 148,253 JARs plus additional metadata stored in three different database formats: MySQL, Berkeley DB, and Neo4j [37]. For our purpose, we rely on the metadata stored in the MySQL database. More specifically, we consider the `files` table which stores information about the **`groupId`**, the **`artifactId`**, and the **`version`** of each JAR in the corpus. There is a minor difference in the number of JARs reported in the original study [39] and the dataset paper [37]. We consider the information presented in the latter after manually validating the data exposed in the MySQL database.

The Maven Dependency Graph (MDG). The MDG is a graph-based snapshot of all artefacts on MCR as of September 6, 2018 [3, 4]. It is available as a Neo4j graph database where nodes are Maven artefacts and edges are either dependency relations between two artefacts (denoted `:DEPENDS`) or upgrade relations between two artefacts of the same library (denoted `:NEXT`). The MDG contains 2,4M libraries, 9,7M dependency relations, and 2,1M upgrade relations. We rely on the MDG to extract libraries metadata (*e.g.*, versions, clients), and to identify dependency and upgrade relations from which we derive the datasets required for our analyses.

4.1.2 Derived datasets

In what follows, we present the datasets that are derived from each of the two corpora: the *upgrade datasets* $\mathcal{D}_u^o$ and $\mathcal{D}_u^r$, and the *dependencies dataset* $\mathcal{D}_d^o$ and $\mathcal{D}_d^r$.

Upgrades dataset. To answer **Q1** and **Q2**, we derive datasets from our corpora consisting of a set of library upgrades $\langle v1 \rightarrow v2 \rangle$. For our analysis to be accurate and relevant, these library upgrades must fulfil a set of criteria.

As a first filter, we only consider library upgrades $\langle v1 \rightarrow v2 \rangle$ such that v_1 and v_2 are two versions of the same library (uniquely identified by its **`groupId`** and **`artifactId`**) which comply with the `semver` scheme. More precisely, these versions must be of the form `X.Y[.Z]`, where `X, Y, Z` $\in \mathbb{N}$, `X` is the major version, `Y` the minor version, and `Z` the (optional) patch version. Versions suffixed with an additional hyphen-separated qualifier, often used to tag release candidates, beta versions, or particular build metadata (*e.g.*, `-b01`, `-rc1`, `-beta`, `-issue101`) are discarded, as they are not meant to be used by the general public. In the MDG, for instance, we find 328,448 suffixed versions for 8,251 different suffixes. The top five most frequent suffixes which we have discarded correspond to release candidates and milestones, namely: `-rc1` (20,007 occurrences, 6.1%), `-rc2` (12,373 occurrences, 3.8%), `-M1` (10,614 occurrences, 3.2%), `-rc3` (7,829 occurrences, 2.4%), and `-M2` (7,780 occurrences, 2.4%). Looking closely at the data, we also noticed that a number of versions, even though they technically comply with `semver`, use dates as versions numbers (*e.g.*, `2.5.20110712`). We decide to discard them, as they do not convey the meaning originally intended by `semver`.

Second, v_1 and v_2 must either be adjacent versions (v_2 was released immediately after v_1) or separated with non-`semver`-compliant versions only (all intermediate versions connecting v_1 and v_2 through upgrade relations do not match our criteria). For instance, considering the three versions $\langle$`3.0.1` $\rightarrow$ `3.1-b01` $\rightarrow$ `3.1.0`$\rangle$, only $\langle$`3.0.1` $\rightarrow$ `3.1.0`$\rangle$ would be included.

Third, we only consider upgrades where v_1 has at least one external client in the dependency graph (either MDD or MDG). An *external client* c of a library version v is a Maven artefact such that c depends on v and belongs to a different **`groupId`**. This way, we confirm that the artefacts we analyse are indeed used as libraries in practice, and that there are real clients potentially affected by the changes between v_1 and v_2. In the MDG, 56% of all artefacts do not have any client ($1,356,413$ out of $2,407,395$), and only 12% of all artefacts ($293,152$) have at least one external client. In the MDD, 61% of all artefacts do not have any client ($89,772$ out of $148,253$), and only 17% of all artefacts ($24,522$) have at least one external client.

Fourth, as we are only interested in the Java language, we discard all JARs that contain code written in any other JVM-based programming language (*e.g.*, Scala, Clojure, Kotlin, Groovy), also hosted on MCR. Our heuristic reads the `source` attribute of `.class` files, set by most bytecode compilers, to retrieve the

$$\begin{tikzcd}
\cdots \arrow[r, dotted, dash] & 3.0.1 \arrow[r, dotted, dash] \arrow[rr, bend left, "\mathit{\Delta}_{\textsc{Minor}}"] & \text{3.1-b01} \arrow[r, dotted, dash] & 3.1.0 \arrow[r, dotted, dash] \arrow[rrr, bend left, "\mathit{\Delta}_{\textsc{Major}}"] & \text{4.0.0-b01} \arrow[r, dotted, dash] & \text{4.0.0-b02} \arrow[r, dotted, dash] & 4.0.0 \arrow[r, dotted, dash] \arrow[r, bend left, "\mathit{\Delta}_{\textsc{Patch}}"] & 4.0.1 \arrow[r, dotted, dash] & \cdots
\end{tikzcd}$$

Fig. 4.2: Extracting relevant upgrades from the JavaServlet project (`javax.servlet :javax.servlet-api`) between versions `3.0.1` and `4.0.1`. Dotted lines denote upgrade relationships between Maven artefacts. Only major, minor, and patch upgrades are analysed: release candidates, alpha and beta versions, and other qualified versions are discarded. Here, $\mathit{\Delta}$-models are computed for the upgrades $\langle 3.0.1 \rightarrow 3.1.0\rangle$, $\langle 3.1.0 \rightarrow 4.0.0\rangle$, and $\langle 4.0.0 \rightarrow 4.0.1\rangle$.

source file that was used to produce the bytecode and infer the base language. When languages other than Java are detected in a JAR, it is discarded.

Fifth, to ensure that `Maracas` can process the JARs accurately, we only select library upgrades for which v_1 and v_2 are packaged as JAR files and are compiled with a Java version up to 8 included, as the list and semantics of BCs differ in later versions with the introduction of new language constructs. This differs from the original study, given that Java 8 was released in 2014 and the original snapshot dates from 2011. Thus, we expect to report new types of BCs that were not considered for previous Java versions (*e.g.*, insertion of a new **`default`** method). The choice of Java 8 is motivated by its popularity: looking at the data, we noticed that it was still by far the most popular Java version on MCR.

Lastly, when looking for clients of libraries, we discard all dependency relations that are not in the `compile` scope or `test` scope since they are either not reliably resolvable (*e.g.* `provided` and `system` dependencies are not hosted on MCR) or are not included in the compile-time and link-time classpaths of the client and thus cannot impact binary compatibility (*e.g.* `runtime` dependencies). Only dependencies in the `compile` and `test` scopes are considered.

As an illustration of the selection process, Figure 4.2 depicts how interesting upgrades are picked up between versions `3.0.1` and `4.0.1` of the JavaServlet library, and how $\mathit{\Delta}$-models are classified as major, minor, or patch.

From the original corpus (MDD), we obtain the upgrades dataset $\mathcal{D}_u^o$ consisting of $11,384$ upgrade pairs, along with the associated $\mathit{\Delta}$-models computed using `Maracas`. This dataset differs from the one presented in the original study which contains $126,070$ pairs [39]. This difference is explained by the additional filters employed in our protocol: most upgrades are discarded because they do not have any external client; 848 because they contain bytecode generated from other JVM-based languages (2 in Clojure, 71 in Groovy, 76 in Scala, 699 a mix of these or other languages); 641 because of an invalid Java version; 306 because the JARs could not be retrieved from MCR; 2 because `Maracas` raised an exception when processing the JARs; 31 because they use dates as versions; 309 because the metadata states that v_1 of the library was released after v_2; and 27 that have an erroneous release date strictly greater than 2011.

From the replication corpus (MDG), we obtain the upgrades dataset $\mathcal{D}_u^r$ consisting of $119,879$ upgrade pairs. Most upgrades are discarded because

Table 4.1: Descriptive statistics of the datasets $\mathcal{D}_u^o$ and $\mathcal{D}_u^r$.

Dimension	Min.	Q1	Median	Mean	Q3	Max.
			$\mathcal{D}_u^o$			
External Clients	1	1	3	24.31	8	6,153
# of releases	1	1	1	1.23	1	55
Age (in months)	1	1	2	5.55	5	146
Releases / month	0.01	0.25	0.5	0.53	1	18
# of decls.	1	71	280	2,076	1,276	218,274
# of API decls.	1	49	200.5	1,515.6	891.2	159,478
			$\mathcal{D}_u^r$			
External Clients	1	1	2	25.31	7	36,186
# of releases	1	7	20	39.4	49	587
Age (in months)	1	10	24	34.38	49	158
Releases / month	0.01	0.4	0.85	1.96	1.79	320
# of decls.	1	79	329	2,519	1,346	586,172
# of API decls.	0	52	220	1,850	974	561,465

they do not have any external client; 39,986 because they are written in other JVM-based languages (20 in Clojure, 1,137 in Groovy, 1,359 in Kotlin, 14,402 in Scala, 23,068 a mix of these or other languages); 852 because of an invalid Java version; 10,588 because the JARs could not be retrieved from Maven Central; 271 because `Maracas` raised an exception when processing the JARs; 115 because they use dates as versions; and 2,929 because the metadata states that v_1 of the library was released after v_2.

Table 4.1 and Figure 4.3 summarize some descriptive statistics of both datasets. As most distributions are strongly skewed and difficult to visualize (number of clients, size, *etc.*), Table 4.1 lists their minimum, maximum, median, mean, and quartile values. As an illustration, the top five most popular libraries in $\mathcal{D}_u^r$ are `commons-io 2.4` (36,186 clients), `slf4j-api 1.7.21` (33,582 clients), `commons-codec 1.10` (32,990 clients), `slf4j-api 1.7.12` (25,317 clients), and `slf4j-api 1.7.7` (22,939 clients). We refer the reader to the companion webpage to access and interact with the entire datasets.

Dependencies dataset. To answer **Q3**, for each upgrade in $\mathcal{D}_u^o$ and $\mathcal{D}_u^r$, we compute the list of all clients potentially impacted. That is, all clients that declare a compile-time or test-time dependency towards the library $v1$ in a $\langle v1 \rightarrow v2 \rangle$ upgrade pair, which are potentially affected by the Δ-model between $v1$ and $v2$. We observe that, often, different versions of the same client (*e.g.*, c_{v_1}, c_{v_2}, and c_{v_3}) all depend on the same library version $v1$. In such a case, it is unlikely that c_{v_1} would migrate to v_2 as it is superseded by c_{v_3}. Thus, we only include c_{v_3} in the resulting datasets to avoid counting the impact of the Δ-model between $v1$ and $v2$ on c multiple times. The resulting datasets for $\mathcal{D}_u^o$ and $\mathcal{D}_u^r$ are $\mathcal{D}_d^o$ and $\mathcal{D}_d^r$ which contain, respectively, 35,539 and 293,817 clients.

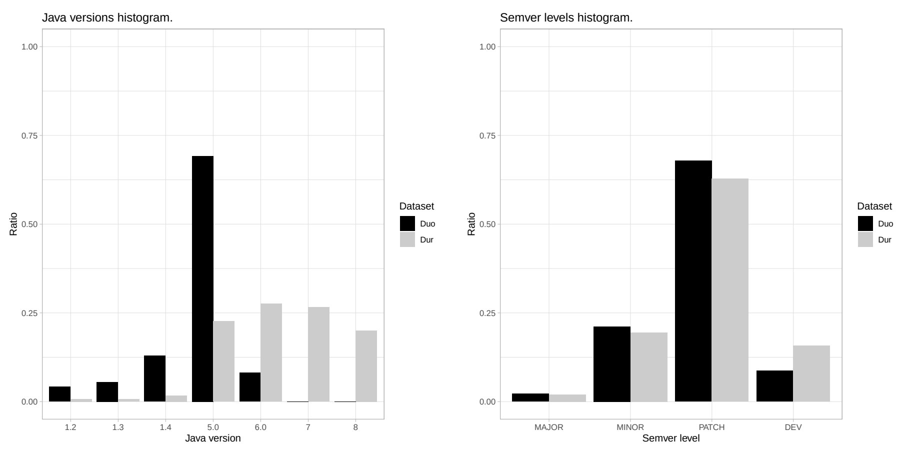


Fig. 4.3: Java versions and `semver` levels histograms.

4.2 Maracas

`Maracas` is a new static analysis tool written in Rascal [25] and Java, which allows us to (i) automatically compute a Δ-model between two binary versions of a library and (ii) detect locations in a client binary that are affected by the BCs listed in a Δ-model.

4.2.1 Δ-models.

A Δ-model is a model that stores the list of BCs between two versions $\langle v_1 \rightarrow v_2 \rangle$ of a library. To compute the Δ-model between two versions of a library, `Maracas` internally relies on `japicmp`.[7] `japicmp` is a tool that compares two JAR files and generates a list of BCs between these two JARs. It is able to identify 31 binary incompatible BCs as specified in the JLS 8th Edition [19]. Example of BCs include removals (*e.g.*, `fieldRemoved`, `methodRemoved`), changes in modifiers (*e.g.*, `methodNowAbstract`, `classNowFinal`) and visibilities (*e.g.*, `fieldLessAccessible`), type changes (*e.g.*, `methodReturnTypeChanged`), to name a few.[8] As we shall see later in Section 5.3, some are more critical than others in terms of impact on clients. A Δ-model in `japicmp` follows a tree structure and consists of a list of modified types (classes, interfaces) that recursively contain all modified child elements (*e.g.*, methods, fields, modifiers). Modified elements themselves are labelled with a kind of BC (*e.g.*, `classRemoved`, `fieldNowFinal`). `Maracas` transforms `japicmp`'s tree-structured models into a value in Rascal conforming to a Δ-model Algebraic Data Type (ADT), which we use for further analysis. The choice of

[7] `https://siom79.github.io/japicmp/.`

[8] A complete list and description of these BCs is available on the companion webpage: `https://crossminer.github.io/maracas/`

japicmp is motivated by its high popularity and accuracy [22], and by its active community.

Atop japicmp, we implement in Maracas a mechanism to classify declarations in a library as *stable* or *unstable*. Unstable declarations are code elements (*e.g.*, classes, methods, fields) that are explicitly marked with a specific annotation or that are contained in a package not meant to be used by clients. For instance, Google libraries use the annotation @Beta and Apache libraries use the annotations @Internal to denote code elements that are subject to sudden changes or that should not be used by clients. Similarly, Eclipse packages containing the word internal and JDK packages starting with sun.* should not be considered part of the API [9]. Because declarations that are explicitly marked as unstable by developers escape the rules of semver, it is important to classify stable and unstable declarations. To come up with a list of annotations to consider, we automatically extracted all annotations used in the 100 most popular libraries on MCR and manually reviewed their documentation to state whether they are used to delimit unstable APIs. This way, we extracted 185 unique annotations and a list of keywords that typically appear in their name (*api, alpha, beta, internal, protected, private, restricted, experimental, dev, access*). Then, we conducted a keyword-based search on the annotations used in the top 1,000 most popular libraries on MCR using as input the keywords extracted manually. This way, we extracted $1,258$ annotations of which 48 matched a keyword. The five most common API annotations encountered in these 1,000 libraries are as follows: @Beta (1,451 occurrences), @InterfaceAudience (1,819 occurrences), @InternalApi (1,414 occurrences), @Internal (716 occurrences), and @SdkInternalApi (607 occurrences). In Maracas, we use the list of extracted keywords to extract package and annotation names used to delimit unstable APIs. Then, we classify each BC in the Δ-models according to whether they affect a stable or unstable declaration of the library.

4.2.2 Detection models.

Prior work uses two main techniques to assess the impact of BCs on client projects: either by tracing library types that are imported in client code (through import statements in Java) [2, 46] or by measuring the ripple effect of changes on clients that have already been migrated manually by developers [40]. The former approach largely over-estimates the impact of BCs (a client may not use the broken declaration in the imported type) and the latter requires the availability of migrated clients. The original study employs a novel technique which consists in isolating and injecting each individual BC in the old library's source code. Then, every client is compiled against every ad-hoc version of the library where a single BC is inserted to measure its impact in terms of compilation errors [39]. To the best of our knowledge, however, it is rarely possible to inject individual changes in a library without having to refactor other parts of the API. Removing or renaming a method, for instance, triggers a ripple effect within the library itself, which results in multiple changes being inserted and impacting the clients. Therefore, we hypothesize that this technique overestimates the impact of BCs on clients. Moreover, while measuring the impact of *source incompatible* BCs by counting compilation errors is a valid approach, it is not appropriate to measure the impact of *binary incompatible* BCs which are instead checked by the JVM linker. This motivates the need of having a dedicated tool for identifying BCs impact using static analysis.

`Maracas` leverages the Δ-models and Rascal M^3 models to link BCs to affected client declarations using static analysis of binary code. An M^3 model is an ADT that models relations between Java elements, extracted from a JAR file, in immutable binary relations (*e.g.*, containment relations among classes and method declarations, invocation relations among method declarations) [1]. Internally, M^3 relies on the ASM framework[9] to parse Java bytecode and populate the relations. By combining information about breaking declarations in Δ-models and uses of these declarations in client code using M^3 model, `Maracas` is able to mark affected client declarations. This detection algorithm is based on the JLS specification [19], and its implementation in `Maracas` for each kind of BC is detailed on the companion webpage. The output of this task is a set of *detections* that point to the affected client element, the modified API element, the way it is being used (*e.g.*, `methodInvocation`, `fieldAccess`, `implements`), and the type of BC. In the example of Figure 2.1, the affected client element is `MockHttpServletRequest` which uses the modified API element `changeSessionId()` through an `implements` relation due to a `methodAddedToInterface` change.

4.2.3 Limitations

Overridden methods. M^3 models generated from Java binaries do not have information related to overridden methods. This means that `Maracas` cannot detect BCs impact on code that uses the API through method overriding. For instance, the *method now final* BC breaks clients that override the now-final method, but has no impact on clients that do not override this method. In such a case, due to the lack of information, we follow a pessimistic approach and report a detection in all cases.

Exceptions handling. Information related to thrown and caught exceptions is not part of the M^3 models. `Maracas` has no information related to the types of exceptions handled in the **`try-catch`** statements of clients. Thus, if a method in a library throws a new kind of checked exception, `Maracas` is not able to state whether the client will be impacted. In this case, we follow a pessimistic approach and always report a detection.

Inheritance hierarchy. Changing the type of a field, method, parameter, or any other member, or casting might turn out to be a generalization or specialization of the associated type. A type is generalized when it is changed to a supertype and a type is specialized when it is changed to a subtype. In `Maracas` we only have access to the client and to the analysed API binaries. Other APIs used

[9] `https://asm.ow2.io/`

by the client are not part of the analysis. Therefore, when a type is changed in a library, we cannot build the whole inheritance hierarchy to state whether this type change corresponds to a generalization or specialization. Without this information `Maracas` might report false positive detections, following a pessimistic approach.

The super keyword. The **`super`** keyword in Java gets a special treatment when detecting errors caused by BCs at the client level. When the visibility of a constructor goes from public to protected, and the constructor is invoked through the use of the **`super`** keyword in the subtype constructor, no error should be reported. However, if the constructor is invoked without using the **`super`** keyword, an error should be reported. `Maracas` is not able to differentiate between these two types of invocations, and thus follows a pessimistic approach to always report a detection.

The strictfp and native modifiers. `japicmp` does not report BCs related to the **`strictfp`** and **`native`** modifiers. Therefore, `Maracas` is unable to detect client code affected by changes related to these modifiers, directly affecting the recall of the tool.

4.2.4 Validation.

`Maracas` is the cornerstone tool of our approach as it is used to both compute the Δ-models revealing BCs (using `japicmp` under the hood) and the detection models revealing their impact. To correctly interpret our results, it is essential to analyse the accuracy of `Maracas`.

When `Maracas` cannot accurately state whether a BC actually has impact due to the limitations listed above, the approach we follow is to always over-approximate the detections at the cost of sometimes reporting false positives, while avoiding any false negative. This means that the results we obtain regarding BCs and their impact might be slightly overestimated, but they are not underestimated: every BC and detection we report do exist.

Binary compatibility is checked by the JVM linker. To evaluate `Maracas`, we aim to compute its accuracy by comparing `Maracas` detections with the error messages thrown by the JVM linker itself when encountering code impacted by BCs. The JLS states that binary compatibility should be checked during the loading and linking phases of the JVM, but the choice of implementing lazy or eager initialization of classes is up to the implementors. In practice, the reference implementation (OpenJDK HotSpot) implements lazy loading and waits for class initialization to load and link a class. It follows that, to record the errors thrown by the linker, it is necessary to *execute* a Java program making use of breaking declarations. In addition, the Java linker throws an exception and stops processing the class after the first error is encountered, so the executed Java programs should contain only a single use of a breaking declaration to record all errors.

To evaluate the accuracy of `Maracas`, we thus reuse and extend the benchmark proposed by Jezek and Dietrich [22]. Their benchmark consists of a library $v1$, a library $v2$ that breaks $v1$ in all possible ways, and a client c that uses all declarations of $v1$ in various ways. In order to trigger linking errors, the client c consists of a set of `Main` files. In the original version of the benchmark, however, some `Main` files use several declarations of the library $v1$: if the first use fails, the others are not evaluated by the linker. Thus, we split the client c in more cases so that every case exercises a single declaration of $v1$.

Our final benchmark for detections consists of 345 cases, where each case consists of a Java entry point (`Main` file and method) that exercises one particular BC and one particular way of using it. The benchmark script first compiles $v1$, $v2$, and c in their binary form (JAR), and then attempts to run every single `Main` file in c, replacing $v1$ with $v2$ in its classpath. Whenever a linking error is encountered, it is written to disk. Then, we run `Maracas` giving it $v1$, $v2$, and c as inputs to get the list of detections. If a detection matches an error reported by the linker, it is a true positive, if it does not match any linker error it is a false positive, and if there is no detection for a particular linker error it is a false negative. To support future research, we have made our benchmark publicly available on the companion webpage.

Out of the 345 cases, the JVM linker reports 132 errors and `Maracas` reports 135 detections. Out of the 135 detections, 130 are true positives and 5 are false positives. There is 2 false negatives. In this benchmark, `Maracas` achieves a precision of 96.3% and a recall of 98.5%. The five false positives are due to the limitation listed in the section above. The two false negatives are due to limitation of `japicmp` which does not compute BCs related to the **`strictfp`** and **`native`** modifiers.

In addition to this benchmark, we developed throughout `Maracas` development a test suite consisting of 402 test cases. Using our own test cases, we highlighted a bug in `japicmp` which we fixed through a pull request accepted by the project maintainers.[10]

4.3 Analysis Approach

In this section, we compile some of the most relevant aspects of the analysis performed in this study, and we contrast them against the original study (*cf.*, Table 4.2). We refer the reader to Section 4 of the original work for further information.

Backwards compatibility. The original study computes binary incompatible changes with `clirr`. However, `clirr` is not able to report BCs related to exceptions and generics and misinterprets changes related to inheritance and other modifiers [22]. In this study, we use `japicmp` to compute both source and binary incompatible changes. The latter performs better than `clirr` according

[10] `https://github.com/siom79/japicmp/pull/251`

Table 4.2: Main commonalities and differences between the original study and the replication study protocols.

	Original Study	Replication Study
Corpus	MDD [37]	MDD+MDG [3]
Corpus date interval	2005–2011	2005–2018
Backwards compatibility type	Binary	Binary
Backwards compatibility tool	`clirr`	`japicmp`
Compared versions	Adjacent	Adjacent
Versioning scheme	`semver`	`semver`
Languages	JVM-based	Java
Clients per library	≥ 0	≥ 1
Client impact detection	Compilation errors	Static analysis
Code-level mechanisms	`@Deprecated`	Annotations, package naming

to Jezek and Dietrich [22]. Although `japicmp` is unable to identify changes related to generics—which, due to type erasure in Java, does not impact binary compatibility analysis—it accurately reports all changes related to exceptions and inheritance. In addition, it is more accurate than `clirr` when reporting on changes associated to modifiers. We also contributed to the tool by fixing a bug related to the detection of modifier changes. Other committers have also made some contributions to improve `japicmp` accuracy in recent times. With these changes, one new case within the Jezek and Dietrich [22]'s benchmark passes, the decrease of a nested interface access modifier from **`public`** to **`protected`**.

Library and version selection. As is the case in the original study, we only compute deltas between adjacent versions of an API, which strictly follow the `X.Y[.Z]` version convention. However, in contrast, we only consider artefacts that have at least one external client on the MCR. This way, we ensure that the artefacts we analyse are indeed used as libraries by clients. It is a significant difference with the original study, as only 17% of all artefacts in the MDD and 12% of all artefacts in the MDG have at least one external client. We also account for initial development releases (`0.Y[.Z]`) which are not discriminated in the original study.

Parallel branches and maintenance releases. The original study does not account for maintenance releases which happen in practice. For instance, suppose version `2.4` is released after `3.0`, as a maintenance release for the `2.X` branch. Using release dates to infer the order of versions, BCs for the upgrade ⟨`3.0` → `2.4`⟩ would be computed, even though this is not the expected behaviour. Instead, we employ the MDG which properly represents these upgrade relations regardless of release date, and accounts for maintenance releases.

Breaking changes impact. The original study detects client code affected by BCs by means of injecting changes in the source code of an API. After the code injection the client is compiled against the modified API and new compilation errors are recorded. There might be pre-existing compilation errors before changes are injected in the API. These errors are intentionally excluded from the analysis. However, this approach introduces a set of limitations that can affect the outcome of the study, some of them have already been identified by Raemaekers et al. [39]. We describe them as follows: (i) injecting changes in isolation might introduce compilation errors that must be fixed. In some cases, multiple changes should be injected at the same time in order to avoid introducing compilation errors; (ii) pre-existing errors might hide new errors related to the injected BC; (iii) reporting on compilation errors gives us an idea of how source compatibility is affected. However, binary compatibility is not equivalent to source compatibility. Compilation errors account for source incompatible changes and cases of binary incompatibility that are shared between both sets; (iv) we cannot guarantee that all expected compilation errors are reported by the compiler. For instance, if at least one imported package in a class cannot be found, the compiler will not reach subsequent errors [39]; (v) when injecting changes in the API, it is difficult to manage cases where one piece of code is related to multiple BCs. For instance, we inject a piece of code related to changes C_1 and C_2 in a given API. If we want to measure the impact of both changes, we will end up with the same number of compilation errors for both cases without discriminating their origin; and (vi) the compiler cannot tell the cause or the BC that produces a given error.

Overall, given the widespread use of the language features involved in the above possible causes of inaccuracy, and their relation to the research questions, we believe that developing and using a more accurate tool will have significant impact on the outcome. Thus, we use `Maracas` to detect affected code on the client side and report on its accuracy.

Deprecated and unstable interfaces. The original study uses `@Deprecated` annotations to identify unstable interfaces. Occurrences of this annotation are computed, except for nested cases. This means that the analysis will not detect declarations within a deprecated class, where explicit annotations have not been used [39]. These cases are considered in the present work. Moreover, there are also other mechanisms to signal unstable interfaces. We argue that other annotations, such as Google's `@Beta` and Apache's `@Internal` annotations, are also used to signal instability in an API. In addition, naming conventions on packages are also used for the same purpose. We then include the detection of these cases to perform a deeper analysis of the derived datasets.

5 Results & Analysis

In this section, we analyse the data extracted using the protocol described in Section 4. Each subsection describes the method, results, and analysis of a particular research question.

5.1 **Q1**: How are semantic versioning principles applied in the Maven repository in terms of BCs?

5.1.1 Method

With **Q1**, we analyze when and where BCs happen and with which frequency. We attempt to distinguish expected and unexpected BCs according to the `semver` principles and the use of code-level mechanisms to signal unstable APIs. In a first step, we seek to highlight the impact of the updated protocol described in Section 4 on the results reported in the original study. To do so, we compute the Δ-models between every $\langle v_1 \rightarrow v_2 \rangle \in \mathcal{D}_u^o$, while distinguishing between major, minor, patch, and initial development releases. In a second step, to assess whether the results hold on the larger dataset $\mathcal{D}_u^r$ comprising seven more years of Maven Central, we run the same analysis for every $\langle v_1 \rightarrow v_2 \rangle \in \mathcal{D}_u^r$. The Δ-models distinguish between BCs that are introduced in stable and unstable parts of the APIs, according to code-level annotations (such as `@Beta` or `@Internal`, *cf.*, Section 4), as well as naming conventions (such as *internal*). As BCs in unstable parts of an API are to be expected, only BCs introduced in the stable parts are included.

To know where to expect BCs or in which type of upgrades, we compare the percentage of breaking upgrades per `semver` level. Alongside `semver` categories (*i.e.*, major, minor, patch, and initial development), we also consider the group of non-major releases as a whole (*i.e.* minor and patch releases combined). Then, to know how many BCs are usually introduced in each `semver` level, we consider the distribution of BCs over all groups.

We wrap up the analysis of **Q1** by studying the frequency of each type of BC. From these results, we identify the most common BCs in our datasets and compare these results against the ones presented in the original study.

5.1.2 Results

Breaking upgrades. Table 5.1 highlights the main results obtained for **Q1**. The first block lists the results reported in the original study [39], the second block the results obtained for $\mathcal{D}_u^o$ (for replication purposes), and the third block the results obtained for $\mathcal{D}_u^r$.

First of all, we compare the results reported in the original study against those obtained for $\mathcal{D}_u^o$. While we report a similar number of breaking upgrades overall (*cf.*, *Total* row: 32.2% in $\mathcal{D}_u^o$ vs. 30.0% in the original study), we observe that the difference in ratio of breaking upgrades per `semver` level is stronger (*cf.*, *Major*, *Minor*, *Patch*, and *Initial development* rows). As expected, most major upgrades in $\mathcal{D}_u^o$ introduce BCs (72.7%), which contrasts with the results obtained in the original study (35.9%). While the original study reports that there are as many breaking major upgrades as breaking minor upgrades, we observe a sharper difference between these two levels in the same corpus: 72.7% of major upgrades (vs. 35.9% in the original study) and 50.1% minor upgrades (vs. 35.7% in the original study) break in $\mathcal{D}_u^o$. With regards to patch upgrades,

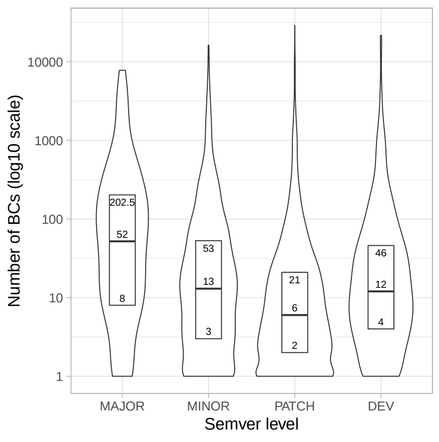
Table 5.1: Total and breaking upgrades in the original study, $\mathcal{D}_u^o$, and $\mathcal{D}_u^r$ datasets.

	Total		Breaking	
Level	Count	%	Count	%
Original study				
Major	11,892	14.8	4,268	35.9
Minor	29,957	37.2	10,690	35.7
Patch	38,740	48.1	9,239	23.8
Dev	n/a	n/a	n/a	n/a
Non-major	68,697	85.3	19,929	29.0
Total	80,589	100	24,197	30.0
$\mathcal{D}_u^o$				
Major	253	2.2	184	72.7
Minor	2,413	21.2	1,228	50.1
Patch	7,728	67.9	1,870	24.2
Dev	990	8.7	389	39.3
Non-major	10,141	89.1	3,098	30.5
Total	11,384	100	3,671	32.2
$\mathcal{D}_u^r$				
Major	2,431	2.0	1,503	61.8
Minor	23,309	19.4	8,837	37.9
Patch	75,282	62.8	11,031	14.6
Dev	18,857	15.7	5,036	26.7
Non-major	98,591	82.2	19,868	20.1
Total	119,879	100	26,407	22.0

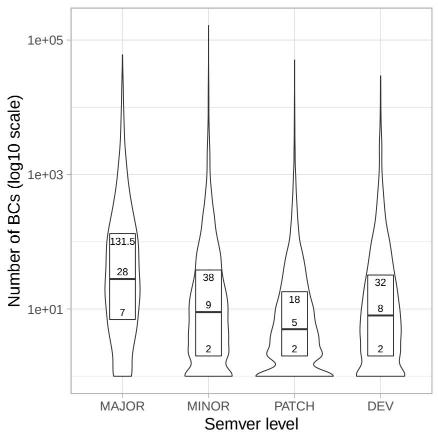
we observe a similar percentage of breaking cases (24.2% in $\mathcal{D}_u^o$ vs. 23.8% in the original study). 39.3% of the initial development upgrades, which are not considered in the original study, are breaking. Overall, we report that 30.5% of non-major releases do not conform to `semver`, which matches the results obtained in the original study (29.0%). Upgrades that comply with the scheme principles (*i.e.*, major upgrades, initial development upgrades, and non-breaking minor and patch upgrades) represent 72.8% of all upgrades in $\mathcal{D}_u^o$.

For $\mathcal{D}_u^r$, which spans seven more years of the MCR and comprises ten times more upgrades, we observe that the tendency to comply with `semver` improves. The ratio of breaking upgrades is lower overall (22.0% in $\mathcal{D}_u^r$ vs. 32.2% in $\mathcal{D}_u^o$), and for each individual level: 61.8% for major upgrades, 37.9% for minor upgrades, 14.6% for patch upgrades, and 26.7% for initial development upgrades. This amounts to 83.4% of all upgrades conforming to `semver` principles. However, 20.1% of non-major upgrades are still breaking and thus do not comply with the versioning conventions. The difference in results between $\mathcal{D}_u^o$ and $\mathcal{D}_u^r$ suggests that the adherence to semantic versioning may have increased over time. This intuition is investigated further in the next research question **Q2**.

Number of BCs. To study the frequency of BCs introduction, we first look at the number of BCs introduced in breaking releases, *i.e.*, releases that contain



(a) $\mathcal{D}_u^o$



(b) $\mathcal{D}_u^r$

Fig. 5.1: Violin plots of the number of BCs in breaking upgrades per `semver` level. Inside each violin plot, we display the first quartile, the median and the third quartile.

at least one BC. Figure 5.1 shows the distribution in a logarithmic scale of BCs per `semver` level for breaking releases in $\mathcal{D}_u^o$ and $\mathcal{D}_u^r$. Looking at the median values, we notice that the number of BCs is higher in major upgrades (52 in $\mathcal{D}_u^o$ and 28 in $\mathcal{D}_u^r$). Minor and initial development upgrades tend to have similar number of BCs in both datasets (13 and 12 in $\mathcal{D}_u^o$, and 9 and 8 in $\mathcal{D}_u^r$, respectively). Patch upgrades introduce the least number of BCs (6 in $\mathcal{D}_u^o$ and 5 in $\mathcal{D}_u^r$). This suggests that non-major development releases not only do break less often, they also tend to introduce fewer BCs when they do.

BC types. Figure 5.2 presents the ratio of BC types (*e.g.*, *method removed*, *field removed*) using a bar plot, for both $\mathcal{D}_u^o$ and $\mathcal{D}_u^r$. BCs are discriminated by `semver` levels and ordered from the most to the least frequent. We notice that the ratio of BC types is consistent across `semver` levels (except perhaps for the *method return type changed* and *field type changed* in the original dataset). In both datasets, the 10 most common BC types and their associated ratios remain mostly unchanged: *method removed*, *field removed*, *interface removed*, *constructor removed*, *superclass removed*, *class removed*, *interface added*, *method added to interface*, *method return type changed*, and *field type changed*. Similarly, in both datasets, the BC kinds ranked after *method return type changed* are very rare. Interestingly, the five most frequent BC kinds are all related to the removal of API entities. We cannot directly compare these results with the original study given that not all reported BC types are identified in the same way between the underlying tools (*i.e.*, `clirr` and `japicmp`). However, our observations align with the results reported in the original study where method, class, and field removal headed the list. Naturally, the BCs *methodNewDefault* and *methodAbstractNowDefault* do not occur in the $\mathcal{D}_u^o$ dataset, as they relate to the `default` operator which was only introduced in Java 8 (2014), while the most recent artefacts in $\mathcal{D}_u^r$ date back to 2011.

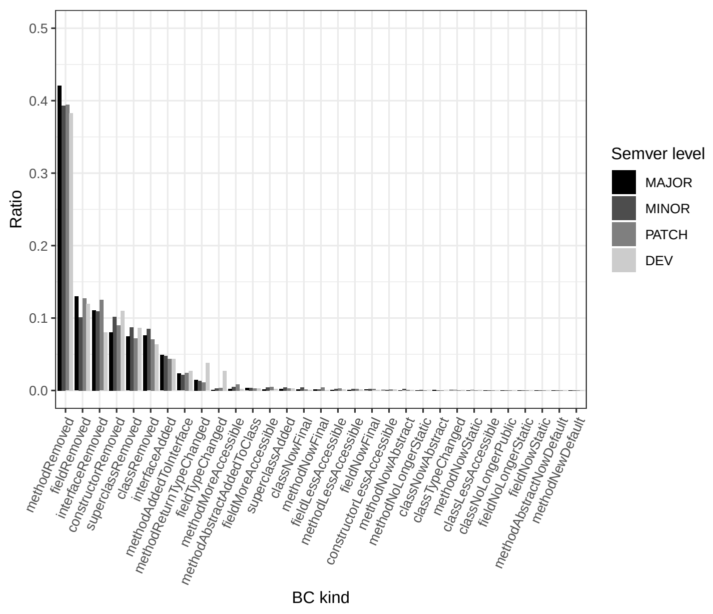
5.1.3 Analysis

Artefacts on MCR do not strictly follow `semver`, as an important ratio of non-major upgrades are breaking (20.1% in $\mathcal{D}_u^r$), confirming the main result from the original study. However, we observe a sharp difference in the ratio of breaking upgrades per `semver` level with our protocol (61.8% of major upgrades, 37.9% of minor upgrades, 14.6% of patch upgrades, and 26.7% of initial development upgrades break). This contrasts with the original study, which reports similar ratio of breaking upgrades for major and minor cases, with patch upgrades only slightly more stable. In general, differences between the results reported in the original study and $\mathcal{D}_u^o$ are explained by the additional filters considered in our protocol, the increased accuracy of `Maracas` and `japicmp` in detecting BCs, and the consideration of APIs annotated as unstable at the source code level (*cf.* Section 4.3). Differences between $\mathcal{D}_u^o$ and $\mathcal{D}_u^r$ are mainly due to the increased timespan and the number of artefacts. This rationale applies to the forthcoming analyses. Our results suggest that `semver` principles are followed to some extent in practice, as 83.4% of the library upgrades we analyse do comply with the backwards compatibility requirements of the versioning scheme.

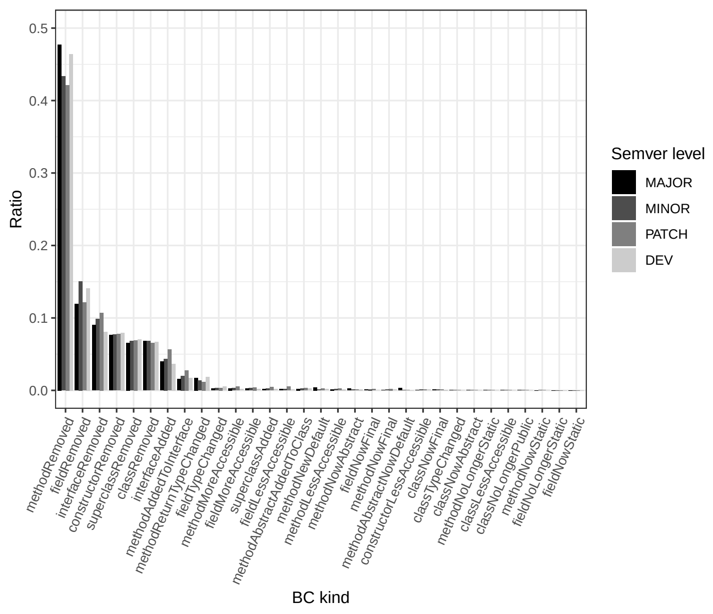
We also notice that major upgrades not only result in a higher number of breaking cases but also tend to introduce more BCs per breaking upgrade. Patch upgrades are the ones introducing the least number of BCs. This suggests that, even when a non-major release is breaking, the amount of work ending on the clients' shoulders is not as high as for a major release. Finally, the most common BCs are aligned with results presented in the original study: removal of API members is the most common type of BC occurring in libraries.

Q1: How are semantic versioning principles applied in the MCR?

$\mathbf{H_1}$ asserts that "*BCs are widespread without regard for versioning principles.*" From our analysis, we conclude that although `semver` principles are not always strictly applied (20.1% of non-major releases are breaking), they are largely followed: 83.4% of all upgrades comply with `semver` regarding backwards compatibility guarantees, and the differences between `semver` levels are notable. Not only do minor and patch releases break less often than major releases, they also introduce fewer BCs. This leads us to reject $\mathbf{H_1}$.



(a) $\mathcal{D}_u^o$



(b) $\mathcal{D}_u^r$

Fig. 5.2: BC types frequency per `semver` level.

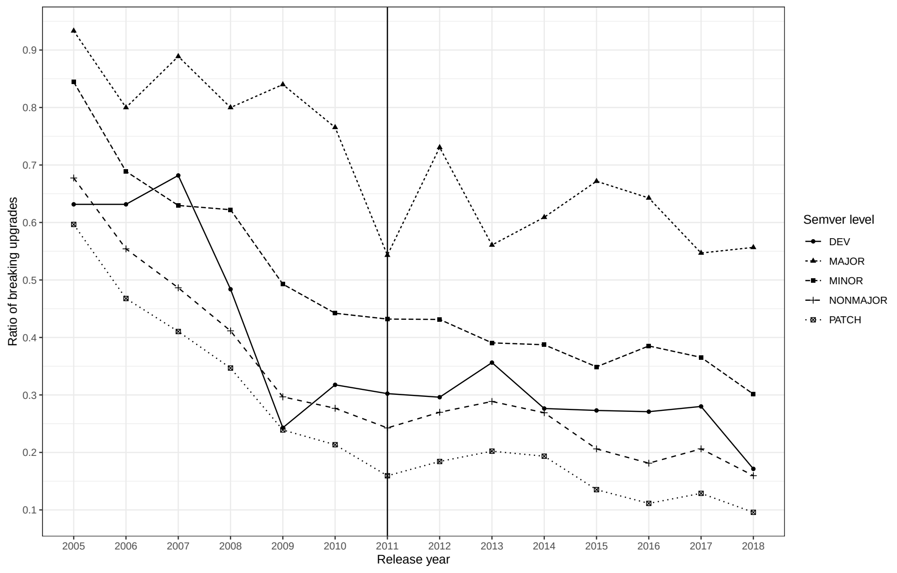


Fig. 5.3: **Q2**: Evolution of the ratio of breaking upgrades per `semver` level in $\mathcal{D}_u^r$. Each data point aggregates the number of breaking upgrades of the given type for an entire year. A vertical line delimitates the periods of the original and updated datasets.

5.2 **Q2**: To what extent has the adherence to semantic versioning principles increased over time?

5.2.1 Method

To answer **Q2**, we first study how the ratio of breaking upgrades for the various `semver` levels has evolved over time, aggregated per year. The ratio of breaking upgrades corresponds to the number of upgrades containing at least one BC over the total number of upgrades per `semver` level. We still consider the four different `semver` levels plus the analysis of non-major upgrades as a whole. Reported results are based on the data extracted from the $\mathcal{D}_u^o$ and $\mathcal{D}_u^r$ datasets. The latter spans fourteen years of Maven artefacts from MCR (2005 to 2018 included). We then contrast these results against the ones reported in the original study. Studying the evolution of the adherence to semantic versioning principles is especially relevant as the `semver` specifications are fairly recent in the history of MCR: `semver` 1.0.0 was released in 2009 and `semver` 2.0.0 in 2013. It is thus likely that the principles of `semver` did not percolate yet in the dataset used in the original study.

5.2.2 Results

Figure 5.3 depicts how the ratio of breaking upgrades has evolved for major, minor, patch, initial development, and non-major levels. Overall, the ratio of breaking upgrades, regardless of the `semver` level, tends to decrease. As expected, the ratio of breaking upgrades of major and initial development levels are more chaotic since BCs are allowed in these releases. Nevertheless, even major and initial development releases contain fewer BCs in 2018 than in 2005. One possible hypothesis is that clients' tolerance for BCs has decreased over time and that libraries are more and more avoiding them, even when allowed. Most notably, over a period of 14 years, the ratio of breaking minor upgrades has decreased almost by a factor of three (from 84.4% to 30.1%) and the ratio of breaking patch upgrades has decreased by a factor of six (from 59.7% to 9.6%). This is to be contrasted with the results of the original study, which finds that, from 2005 to 2011, the number of non-major breaking upgrades has decreased from 28.4% to 23.7%. Conversely, we find that non-major breaking upgrades have decreased from 67.7% in 2005 to 16.0% in 2018.

5.2.3 Analysis

As `Maracas` and `japicmp` are able to detect more types of BCs, the percentages we report are higher than the ones reported in the original study, which makes the decrease of the ratio of breaking non-major upgrades much steeper than originally reported (a 44% reduction instead of a 5% decrease). However, the decrease on the extended period is less important: only a 9.2% decrease. Visually, 2011 appears as a turning point w.r.t. the decrease of breaking non-major upgrades, as the slope is less steep after this date. Nonetheless, we found no plausible explanation for this phenomenon. Overall, it confirms once more the statement that even though not all artefacts on MCR follow `semver` guidelines, there is an increasing tendency to comply with the versioning scheme principles.

Q2: To what extent has the adherence to semantic versioning principles increased over time?

$\mathbf{H_2}$ states that "*The adherence to versioning principles has increased over time.*" Our results confirm the results of the original study and complement them by showing that the improvement over time is much higher than initially reported for the 2005–2011 period. The tendency persists in the 2011-2018 period, although the slope is less steep. This leads us to accept $\mathbf{H_2}$.

5.3 **Q3**: What is the impact of BCs on clients?

5.3.1 Method

In **Q3**, we investigate to which extent BCs introduced in Java libraries impact their clients on MCR. More formally, for every $\varDelta$-model computed between versions $\langle v_1 \rightarrow v_2\rangle$ of a given library (*cf.*, **Q1**), we extract all the clients c declaring a compile-time or test-time dependency towards v_1 to uncover the impact $\varDelta\langle v_1, v_2\rangle$ would have on c if it was updated to $v2$. Concretely, we use the static analysis capabilities of `Maracas` to pinpoint which code locations in c are impacted by individual BCs of the corresponding $\varDelta$-model (*cf.*, Section 4.2).

The impact a BC has on client code varies according to if and how the client uses the declaration affected by the change (*cf.*, Figure 2.1). Hence, determining the impact of BCs requires a deep understanding of how clients and libraries interact. For every client, we classify the impact of each individual BC in one of three categories: (i) the declaration affected by the change is not used in client code (*unused*); (ii) the declaration affected by the change is used in a non-breaking way (*non-breaking*); and (iii) the declaration affected by the change is used in a breaking way (*breaking*).

As with the first two research questions, we report results for both the MDD and the MDG corpora. The datasets $\mathcal{D}_d^o$ and $\mathcal{D}_d^r$ contain, respectively, $35,539$ and $293,817$ clients which are potentially impacted by a $\varDelta$-model extracted in **Q1**. As it would be impractical to analyse these cases exhaustively, we resort to analyse a subset of them by performing a random sampling. The question we ask for each case is: does client c break when upgrading from version $v1$ to version $v2$ of a library it uses? To answer this question with a confidence level of 99% ($c = 0.99$), an error margin of 1% ($e = 0.01$), and an estimated proportion of the population $p = 0.5$ (the more conservative value yielding the largest sample size) of broken clients, we apply the standard Cochran's sample size formula to determine sample sizes for each kind of upgrade (*i.e.*, major, minor, patch, and initial development). Then, we draw upgrades at random, without replacement, from the set of all upgrades, all major upgrades, all minor upgrades, all patch upgrades, and all development upgrades, yielding the samples depicted in Table 5.2. For each tuple $\langle c, v_1, v_2\rangle$ in the corresponding samples, we use `Maracas` to compute their detection models and analyse the impact of $\varDelta(v_1, v_2)$ on client c, distinguishing between *unused* declarations, *non-breaking* uses, and *breaking* uses.

To uncover which kinds of upgrade break clients the most, we compare the percentage of overall broken clients per `semver` level. Afterwards, we consider the number of broken locations per client for each level.

5.3.2 Results

Broken clients. Table 5.2 depicts the size of each `semver` sample and the number and proportion of broken clients for both $\mathcal{D}_d^o$ and $\mathcal{D}_d^r$. We observe that 9.5% and

Table 5.2: Samples derived from the population of dependencies.

	All	Major	Minor	Patch	Dev
		$\mathcal{D}_d^o$			
Population size	35,539	2,861	13,444	17,425	1,809
Sample size	11,310	2,440	7,426	8,498	1,631
Broken clients	1,076	309	883	514	300
% broken clients	9.5%	12.7%	11.9%	6.0%	18.4%
		$\mathcal{D}_d^r$			
Population size	293,817	29,847	111,830	123,286	28,854
Sample size	15,701	10,663	14,445	14,621	10,533
Broken clients	1,237	1,250	1,130	735	1,772
% broken clients	7.9%	11.7%	7.8%	5.0%	16.8%

7.9% of all clients for $\mathcal{D}_d^o$ and $\mathcal{D}_d^r$, respectively, would break if they upgraded their dependency to the next release.

Taking into account the kind of upgrade yields interesting results: in both datasets, initial development upgrades lead to the highest percentage of broken clients (18.4% for $\mathcal{D}_d^o$ and 16.8% for $\mathcal{D}_d^r$), followed by major (12.7% for $\mathcal{D}_d^o$ and 11.7% for $\mathcal{D}_d^r$), minor (11.9% for $\mathcal{D}_d^o$ and 7.8% for $\mathcal{D}_d^r$), and finally, patch upgrades (6.0% for $\mathcal{D}_d^o$ and 5.0% for $\mathcal{D}_d^r$). This indicates that clients are more likely to break when upgrading to a version of a library that is potentially breaking according to `semver` conventions, with initial development releases being the most problematic. Conversely, clients that upgrade to minor and patch releases are less likely to be affected.

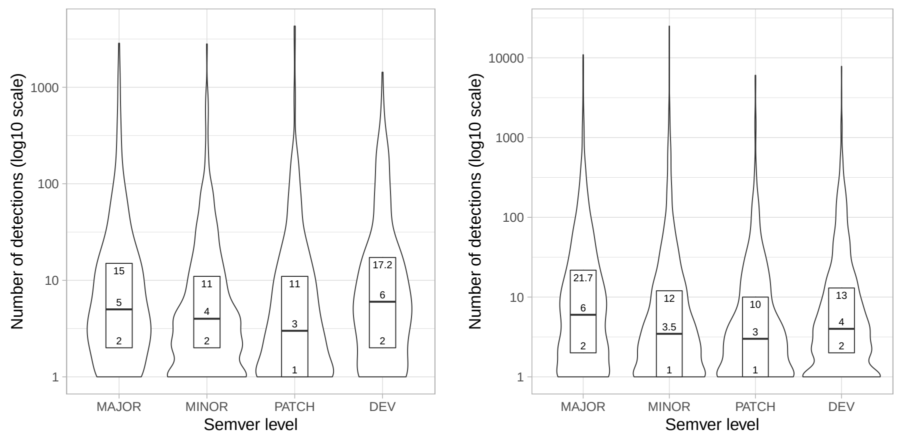
As we resorted to random sampling to estimate the proportion of broken clients, we use statistical inference to assess our raw results. For the sake of simplicity, we only perform the statistical analysis for the $\mathcal{D}_d^r$ dataset. We have the following null hypothesis: "*the proportion of broken clients is the same across each `semver` level of library upgrades*". Note that in the remainder of this section, we use * to label the significance of the p-values using the following scale: * indicates a $p < 0.1$, ** a $p < 0.05$ and *** a $p < 0.01$. We run a $\mathcal{X}^2$ (chi-squared) test on the table containing the amount of broken and non-broken clients for each level. This test yields a $p < 2.2 \times 10^{-16}$ ***, we therefore reject the null hypothesis and accept the alternative hypothesis "*the proportion of broken clients is different across each `semver` level of library upgrades*".

To assess the differences across `semver` levels, we conduct post-hoc analyses for each pair of groups using Fisher's exact test on the contingency tables. We adjust the resulting p-values using a Holm-Bonferroni correction. Finally, we assess the effect size using odds ratio. We obtain the results shown in Table 5.3.

The p-values are all significant considering a 0.01 threshold. If we look at the direction of the odds ratios, the results are as expected w.r.t. the differences among levels. The proportion of broken clients is higher for initial development upgrades, then major upgrades, then minor upgrades, and finally patch upgrades. Looking at the values of the odds ratios, we note that the difference in the odds of being broken depending on the `semver` level is perhaps

Table 5.3: p-values and odds ratios across all pairs of `semver` levels in $\mathcal{D}_d^r$.

`semver` level	p-value	Odds ratio
Major vs minor	7.45×10^{-25} ***	0.64
Major vs patch	3.02×10^{-83} ***	0.40
Major vs dev	6.34×10^{-26} ***	1.52
Minor vs patch	1.80×10^{-22} ***	0.62
Minor vs dev	2.13×10^{-104} ***	2.38
Patch vs dev	1.37×10^{-206} ***	3.82



(a) $\mathcal{D}_d^o$ (b) $\mathcal{D}_d^r$

Fig. 5.4: Number of detections per `semver` level.

not as high as one would expect. For instance, for major versus minor the odds of being broken in a minor upgrade is 0.6 times the odds of being broken in a major upgrade. An interesting finding is that, in the Maven ecosystem, initial development upgrades break a greater proportion of clients than major upgrades.

Number of detections. Figure 5.4 presents the distribution in logarithmic scale of the number of broken declarations (*i.e.*, detections) per client. Figures are presented for each `semver` sample in both $\mathcal{D}_d^o$ and $\mathcal{D}_d^r$. In these distributions we only consider broken clients, that is, clients that have at least one declaration affected by a BC. In both datasets $\mathcal{D}_d^o$ and $\mathcal{D}_d^r$, we observe a similar trend: major upgrades yield the highest number of broken declarations (medians of 5 and 6, respectively), followed by minor upgrades (medians of 4 and 3.5, respectively) and patch upgrades (medians of 3 and 3, respectively). In $\mathcal{D}_d^o$, initial development upgrades yield even more broken declarations than in major upgrades (median of 6), as opposed to what we observe in $\mathcal{D}_d^r$ (median of 4).

As we resort to random sampling to estimate the number of breaking declarations in broken clients, we use statistical inference to assess our raw

Table 5.4: p-values and Cliff's delta across all pairs of `semver` levels in $\mathcal{D}_d^r$.

`semver` levels	p-value	Cliff's delta
Major vs minor	4.89×10^{-11} ***	0.16 (small)
Major vs patch	2.79×10^{-13} ***	0.20 (small)
Major vs dev	2.04×10^{-8} ***	0.12 (negligible)
Minor vs patch	0.165	0.04 (negligible)
Minor vs dev	0.165	−0.04 (negligible)
Patch vs dev	0.004 ***	−0.08 (negligible)

results. For the sake of simplicity, we only perform the statistical analysis for the $\mathcal{D}_d^r$ dataset. We have the following null hypothesis: "*the number of broken declarations is the same across each `semver` level of library upgrades*". We run a Kruskal-Wallis rank sum test on the number of broken declarations for each `semver` level. This test yields a $p = 3.82 \times 10^{-16}$ ***, we therefore reject the null hypothesis and accept the alternative hypothesis "*the amount of broken declarations of broken clients different across each `semver` level of library upgrades*".

To make an in-depth assessment of the differences across `semver` levels, we conduct post-hoc analyses for each pair of groups, using a two-tailed Mann-Whitney test. We adjust the resulting p-value using a Holm-Bonferroni correction. In addition, we compute Cliff's delta to assess the effect size and report the interpretation of its value using Cohen's scale. We obtain the results shown in Table 5.4.

We note that two pairs are not significant (minor versus patch and minor versus dev), while the others are all significant at the 0.01 threshold. Looking at the direction of Cliff's deltas, the results are aligned with our expectations: the number of breaking declarations in major upgrades is greater than in minor, patch and initial development upgrades, and the number of breaking declarations in initial development upgrades is greater than in minor and patch upgrades. Looking at the values of Cliff's deltas, however, we note that the differences are very small across the groups. It indicates that, when a client is broken, the number of broken declarations it contains is similar whatever the `semver` level of the upgrade is.

BC types. Figure 5.5 shows the ratio of breaking and non-breaking uses of broken declarations for each BC type in $\mathcal{D}_d^r$. We note that most BCs result in breaking clients as soon as they use the broken declaration. Interestingly, we find several BCs that, in most cases, do not break clients even when the broken declaration is used in client code. On the other hand, apart from the *interface removed* and *interface added* BCs, all other popular BCs (as computed in Section 5.1.2) are prone to break clients. However, it should be noted that, for most types of BCs, there is not enough data to support a definitive conclusion. This prevents us from proceeding to a reliable statistical analysis.

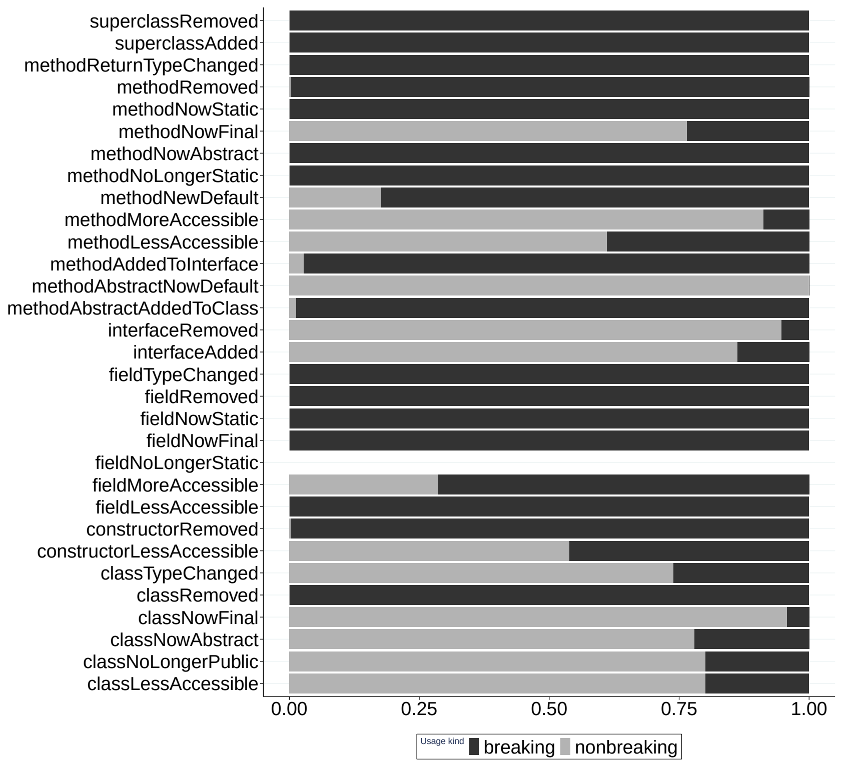


Fig. 5.5: Ratio of breaking and non-breaking uses of API elements w.r.t. the BC type in $\mathcal{D}_d^r$.

5.3.3 Analysis

Considering the results for the MDG corpus, we find that initial development and major releases tend to impact a higher number of clients (11.7% and 16.8%, respectively), as compared to minor and patch releases (7.8% and 5.0%, respectively). The same tendencies can be observed in the MDD corpus. Additionally, not only do clients break more often in major and initial development upgrades, but they also tend to break more. In general, clients are rarely impacted by breaking declarations in the libraries they use because they do not explicitly use the affected declaration. However, when a client uses a declaration that is affected by a BC, it is likely to break. These results are probably explained by the fact that library developers introduce BCs in parts of their API that are less likely to be used by their clients. This intuition has already been investigated in the literature [20], and our results are aligned with their observations.

Q3: What is the impact of breaking changes on clients?

$\mathbf{H_3}$ asserts that "*BCs have a significant impact in terms of compilation errors in client systems.*" Conversely, we observe that in most cases breaking declarations are not used by client projects, which instead yields a low number of broken clients (7.9% for all releases). The number is even lower in the case of minor and patch upgrades. However, when a breaking declaration is used by a client, there is a high chance that it will be impacted. These results contrast with those of the original study and lead us to reject $\mathbf{H_3}$.

5.4 Threats to Validity

In this section, we discuss the main threats to the validity of our replication study, following the structure recommended by Wohlin et al. [43].

Internal validity. As explained in Section 4.2, the tool we implemented and used to detect BCs and their impact in client code is not perfectly accurate, which impacts the metrics we compute. As a sanity check, we reused and extended a benchmark for Java evolution and compatibility to evaluate the accuracy of our tool, using the reference implementation of the Java linker itself as groundtruth. `Maracas` obtained a precision of 96.3% and a recall of 98.5%, making the impact of this threat very low. Moreover, our tool is designed to be pessimistic and to over-approximate the impact of BCs in case of uncertainty. Therefore, the impact of this threat is to slightly overestimate the number of broken clients. `Maracas` does not reach 100% recall because of two false negatives: these are due to limitations of the underlying tool `japicmp` which is not able to detect BCs related to the **`strictfp`** and **`native`** modifiers. However, we do not expect that BCs related to these modifiers are common in practice.

We identified unstable API declarations of the libraries using a pre-defined list of naming conventions and annotations that was extracted by semi-automatically analysing the top-1000 most popular libraries on MCR. However, this list is not exhaustive and cannot account for library-specific or organization-specific conventions. As a result, we have probably misidentified some API declarations as stable. Assuming that unstable declarations are more likely to break than stable ones, the impact of this threat is to overestimate the number and impact of BCs.

Since it is not possible to reliably state whether a particular Maven artefact is a library or not, we consider that an artefact from MCR is a library if it has as least one *external* client. As a result, we potentially misidentified some artefacts as libraries. Assuming that libraries are more likely to be careful about BCs than other kinds of projects, the impact of this threat is to overestimate the number and impact of BCs.

Our protocol excludes every library version suffixed with a qualifier (*e.g.*, `-beta1`, `-rc2`) as they are not final and are not meant to be used by the general public. This complies with the Maven principle stating that every qualified version is anterior to the corresponding non-qualified version. However, we found some libraries that always tag their versions with specific qualifiers. For instance, the developers of the popular Google Guava library tag every version released since 2017 with a `-jre` or `-android` suffix: there are no unqualified versions. Google Guava versions released after 2017 are thus excluded from our datasets—even though they are legitimate, while anterior versions are included. The impact of this threat is however low, as most of the qualifiers we found in our datasets correspond to pre-releases (*cf.*, Section 4.1.2).

The corpora we used to extract our datasets (MDD and MDG) do not contain any information regarding version ranges and constraints. Concrete dependency versions have been resolved at the time the corpora were created, so a dependency with a version range (or no version at all) would be replaced with a concrete version picked by the Maven dependency resolver. As a result, even if some artefacts were using version ranges, we are not able to see and analyse them. We expect the impact of this threat to be low as version ranges are not popular in the Maven repository.

External validity. Our study targets Java libraries and clients inside the MCR ecosystem. Since the definition of BCs is specific to a particular programming language and since ecosystems have very different practices when it comes to BCs and versioning culture and habits [11], there is no guarantee that our results generalize to the ecosystems of other programming language or to other Java ecosystems.

6 Related Work

Prior research in the field of library evolution has focused on understanding *why* and *how* evolution happens [14]. Answering the *why* involves understanding the motives triggering the need to change a library and its API. In particular, researchers study the social factors motivating software change [2, 6, 5, 45]. Conversely, to understand *how* library evolution occurs, researchers analyse the API evolution process and the evolving software itself. In this section, we discuss a set of studies that aim at understanding *how* software libraries evolve over time. We consider studies that analyse the evolution of software ecosystems as a whole; the nature of change in terms of backwards compatibility; and the impact that API evolution stirs up on client projects.

6.1 Ecosystems Evolution

Several studies aim at understanding the evolution of a software ecosystem on its own (*e.g.*, Eclipse, Apache). This is done to catch a glimpse of the evolution practices and expectations within the ecosystem community [5]. As a direct consequence, researchers are able to create models and claims that support the development process within the studied ecosystem.

In the case of the Eclipse ecosystem, Businge et al. [7] evaluate the applicability of the Lehman's laws of software evolution to their corpus. In a later study, the same authors analyse to which extent client plug-ins rely on unstable and internal Eclipse APIs [8]. They claim that more than 40% of Eclipse plug-ins depend on this type of libraries. Likewise, Wu et al. [44] study API changes and usages in both Eclipse and Apache ecosystems. The Apache ecosystem is also studied by Bavota et al. [2] and Raemaekers et al. [36]. On the one hand, Bavota et al. [2] report on the evolution of dependencies among projects within the ecosystem. As main conclusion, they discover that both the size of the ecosystem and the number of dependencies grow exponentially over time. On the other hand, Raemaekers et al. [36] measure Apache APIs stability with a set metrics based on method removal, modification, and addition.

The Squeak and Pharo ecosystems have also been target of research. Robbes et al. [40] study the ripple effects caused by method and class deprecation in these ecosystems. They state that 14% of the deprecated methods and 7% of the deprecated classes impact at least one client project. Hora et al. [21] complement Robbes et al. [40] findings. They reach the conclusion that 61% of client projects are impacted by API changes, and more than half of those changes trigger a reaction in affected clients. In addition, Decan and Mens [11] perform an empirical study where they analyse the compliance to `semver` principles of projects hosted in four software packaging ecosystems (*i.e.*, Cargo, npm, Packagist, and RubyGems). They discover that certain ecosystems, such as RubyGems, do not adhere to `semver` principles when analysing dependency constraints.

The abovementioned studies give a good overview of the evolution of certain ecosystems. However, conclusions drawn by these studies do not hold outside the studied ecosystem [42]. Our study contributes to this body of knowledge by complementing the original results of Raemaekers et al. [39] regarding the adherence to `semver` and the impact of BCs in the Maven Central ecosystem.

6.2 Backwards Compatibility

The growing interest on BCs and NBCs is related to the need of analysing the stability of APIs and the impact these changes have on client projects. One of the main observations in the literature is that backwards incompatible changes are often introduced between two versions of an API. In fact, in a corpus of Java APIs, Dietrich et al. [13] find that 75% of library upgrades introduce breaking changes between adjacent versions. This study was later enhanced by Jezek et al. [23] who argue that 80% of API releases are backwards incompatible. Mostafa et al. [33] also report that 76.5% of the releases they analyse introduce behavioural incompatible changes. Nevertheless, there is still disagreement regarding these figures. For instance, Xavier et al. [46] claim that only 14.78% of the changes in their dataset are backwards incompatible, while Brito et al. [6] state that 39% of the introduced changes are classified as breaking. These differences are due to the diversity of libraries that are analysed and their characteristics, and the criteria used to select them (for instance, the most popular Java libraries hosted on GitHub). In this study, we detail a protocol that enables us to give a clear overview of the state of BCs in MCR over the past 13 years.

In addition, Raemaekers et al. [38, 39] conduct a study that relates `semver` with backwards incompatibility. The authors discover that `semver` is not strictly followed in practice. That is, BCs are also introduced in minor and patch releases [23, 38]. They also claim that minor releases introduce more changes than major releases [39]. Similarly, some studies relate the nature of the API with the tendency to introduce BCs. Xavier et al. [46] find that APIs with a higher frequency of BCs introduction tend to be more popular, larger, and active. They also argue that the frequency of BCs increases over time. Raemaekers et al. [39] partially confirm this claim: larger libraries tend to introduce more BCs. However, they also conclude that more mature APIs do not introduce more BCs, which seems counter-intuitive when contrasting the results against Xavier et al. [46] study. Decan and Mens [11] study adherence to `semver` principles at the dependency constraints level. They find out that newer ecosystems tend to follow `semver` guidelines, and that `semver` practices have become more popular as time passes. Our study complements these results by studying the adherence to `semver` in Maven Central.

It is also important to be aware of the type of changes that are usually introduced during API evolution. Cossette and Walker [10] analyse a set of binary BCs based on the affected entity type (*i.e.*, class, method, field) and its visibility (*i.e.*, protected, public). In more recent work, Wu et al. [44] undergo a study that analyses 23 types of changes related to API types and methods [12, 19]. They find that missing classes and methods are important types of BCs affecting client projects. Ketkar et al. [24] study type changes and required code adaptations. They find out that type changes are more common than renamings, and that they usually appear on public entities. Furthermore, a particular kind of NBC has drawn much attention from the community: API deprecation [6, 38, 40, 42]. While deprecating an API entity does not immediately break client code, it signals that BCs may be coming in the future—as the semantics of the annotation suggests. However, Raemaekers et al. [38, 39] notice that API developers tag deprecated API entities without ever removing them from their API. In other cases, they do quite the opposite: API developers remove declarations from the API without deprecating them first. Brito et al. [6] point out that this is to reduce the required maintenance effort by API developers.

In spite of the contributions and findings of the abovementioned studies, there is still a long way to go. First, some studies do not define a clear scope of the applicability of their conclusions. In essence, it is not clear if findings account for source, binary, or behavioural incompatibility. Moreover, how to detect and classify behavioural incompatibilities is still an open problem. Second, the selection and study of the subset of BCs seems arbitrary, incomplete, and in some cases incorrect. For instance, some studies concluding on backwards compatibility claim that adding a method to a class is a NBC. Although this change is indeed binary compatible, it will break source compatibility when the method is added to an abstract class that is extended by client code. Third, there is a lack of consensus in research findings across studies. This is the case when reporting on the percentage of incompatible API releases and the correlation between API properties and BCs frequency. This might be related to the underlying datasets: some studies analyse only popular projects [6, 33, 46], and others consider few libraries [27, 41].

6.3 Refactorings

Refactorings are changes aimed at improving the structure of a project without changing its observable behaviour [16]. One of the main inquiries concerning refactorings in API evolution is understanding to what extent API changes are actual refactorings. Dig and Johnson [14] discover that between 3% and 27% of changes on two common Java APIs are refactorings. Furthermore, they find that at least 81% of BCs in four Java APIs are due to refactorings. However, in more recent studies this number might be lower. For instance, Brito et al. [6] show that 47% confirmed BCs in their corpus are actual refactorings, and these are the most common types of BCs. Additionally, Kula et al. [27] state that refactorings break less than 37% of all clients of a given API. They find a tendency to find more BCs and refactorings in API internal entities [6, 27].

The main limitation of these studies is the level of abstraction at which the analysis is performed. That is, a refactoring might be composed of multiple BCs and, in some cases, by multiple refactorings. For instance, method renamed could be recorded in a delta as both a method removed and method added change. Analysing these changes require an additional effort on dissecting compound cases.

6.4 Impact of API Evolution

More recent studies attempt to understand how client projects are impacted by API evolution. In some of them [2, 13, 40, 46] we find the same claim: there is no massive impact of API changes on client code. In fact, Bavota et al. [2] state that only around 5% of the projects in their corpus are impacted by API evolution; Xavier et al. [46] discover that only 2.54% client projects in the dataset are impacted by API BCs; and Robbes et al. [40] show that 14% and 7% of class and method deprecations, respectively, impact client projects. From a different perspective, Kula et al. [27] state that BCs are more likely to appear in API entities that are not used by client code. Later, Raemaekers et al. [39] relate the number of compilation errors with the introduced BCs. They do so by individually inserting BCs in the API source code.

Regarding API-client co-evolution, there is a growing interest in understanding *why*, *when*, and *how* client projects upgrade to a newer version of an API. On the quest of answering the *why* and *when*, Raemaekers et al. [38]

show that API upgrading tends to be performed when major API updates are released. Bavota et al. [2] indirectly confirm this claim by stating that client projects upgrade to a newer version of an API only when substantial changes are introduced. Regarding the *how*, Bavota et al. [2] highlight that even though few client projects might be affected by API evolution, certain dependencies that offer cross-cutting services can strongly impact them. To support these first insights, Robbes et al. [40] find that resolving the first 25% ripple effects in the Squeak and Pharo ecosystem, requires more than 14 developers. In addition, several commits are registered to resolve the issue, which suggests the existence of non-trivial changes. Both Robbes et al. [40] and Sawant et al. [42] claim that finding systematic changes in affected client code is rare. There are many cases where impacted code is simply dropped, or an ad-hoc solution is provided. These findings are contrary to what developers postulate in Brito et al. [6] study: they argue that API migration results in minor and easy changes. In addition, Mostafa et al. [33] claim that 67% of bugs introduced by behavioural changes can be fixed by simple changes (*e.g.*, replacing arguments, converting return values).

Despite the new contributions in the field, few papers study both how APIs evolve and how this evolution impacts client projects. Moreover, when they analyse API usage there is a misalignment between the API change and usage types. For instance, Wu et al. [44] label *inheritance for IoC* as an API usage type. However, this type of usage can be split into other categories, such as *method overriding*, *class extension*, and *interface implementation*. With this differentiation it is possible to relate API changes at different levels (*i.e.*, class, method, and field levels) with atomic API usages; and accurately point to affected client members. Finally, as in the case of studies related to backwards compatibility, we perceive contradicting results that are most likely due to differences in the studied datasets.

7 Discussion

In this section we discuss the main implications of our findings for library developers, library clients, and researchers.

Implications for library developers. The introduction of BCs is inherent to software evolution and cannot always be avoided. Although libraries are encouraged to preserve backwards compatibility, the need to introduce new features and improve the quality of the library sometimes results in incompatible changes. Our claim is that introducing BCs is tolerable as long as they are properly announced in advance to not take clients by surprise. Versioning conventions or code-level mechanisms (*e.g.*, annotations, naming conventions) are regularly used for this purpose.

Introducing too many BCs may, however, hurt the reputation of a library. It is very hard for developers to manually detect BCs in source code, so we believe that the use of tools such as `japicmp` and `Maracas` is important to help library maintainers make the right decisions. The Apache Commons developers, for instance, use `japicmp` on every new release of their libraries to check for backwards compatibility.[11] These tools are also able to automatically generate reports that help clients to anticipate the changes.[12] The first step towards a more disciplined evolution of libraries is to detect and communicate on BCs, two aspects addressed by these tools. As we have shown in our analysis of **Q3**, some BCs appear to be more critical than others for client developers. We thus encourage library developers to interpret the output of these tools wisely and to account for the severity of different changes.

The ability of `Maracas` to infer the impact of BCs on client code is also beneficial for library developers. Developers may run `Maracas` as part of a continuous integration pipeline to check that a particular commit, pull request, or release does not significantly impact their clients, and reconsider the changes if the impact is too high.

Implications for client developers. The main implication of our results for client developers is that the situation is not as bad as reported in the literature. In MCR, most releases comply with `semver` requirements and avoid BCs in non-major releases. Besides, as we have shown in **Q2**, the situation has significantly improved over time.

Each ecosystem has its own policy regarding versioning conventions and the treatment of BCs. Maven, npm, Packagist and Cargo, for instance, do not apply semantic versioning in the same way [Mens]. Client developers should thus pay attention to ecosystem-specific guidelines and pick an ecosystem that advocates a strict policy to minimize the risk of being impacted by unwanted changes.

Tools such as `Maracas` should also be beneficial to client developers. When faced with the possibility of upgrading a dependency, developers may employ `Maracas` to evaluate the impact of different versions, and choose the one that addresses their requirements without causing too much disruption.

Implications for researchers. As we have seen, the use of code-level mechanisms to delimit unstable APIs relies on conventions that might vary from one organization to the other or from one library to the other (`@Beta` for Google, `@Internal` for Apache, `sun.*` packages in the JDK, *etc.*). Contrary to semantic versioning, there is no standardization of these mechanisms, which makes it hard for clients to understand which declarations should be considered stable or unstable. Besides, it is not clear how `semver` and code-level mechanisms interact: the `semver` specification only mentions the `@Deprecated` annotation. Should developers release a major revision when they introduce a BC in a beta-stage API? We believe that clarifying the role of code-level mechanisms and their relation with `semver` would be beneficial. Another interesting line of work would be to incorporate better mechanisms to delimit APIs directly

[11] https://garygregory.wordpress.com/2020/06/14/how-we-handle-binary-compatibility-at-apache-commons/

[12] https://commons.apache.org/proper/commons-lang/japicmp.html

in programming languages: developers are currently forced to make some declarations public only for technical reasons (*cf.*, Section 2) even though they are not part of the intended API.

Researchers should strive to design and implement benchmarks to compare tools related to library evolution objectively. The benchmark of Jezek and Dietrich [22], which we reuse and extend to evaluate the accuracy of `Maracas`, is a first step in this direction and should be complemented with other benchmarks, for instance related to behavioural compatibility.

The design of a study protocol and the creation of the underlying datasets should be carefully performed. Sampling bias is a recurrent threat to validity that can hurt the interpretation of API evolution studies. For instance, selecting only the most popular libraries on a repository, or only the ones related to a particular ecosystem hurts the generalization of the study findings.

Finally, because BCs are not always avoidable, researchers should continue to develop tools and methods that assist client developers in automatically migrating their code [10, 47, 48].

8 Conclusion

In this paper, we conduct an external and differentiated replication study of the work presented by Raemaekers et al. [39]. The motivation behind this study is to better understand which kind of BCs happen in libraries hosted on the MCR, and what is their impact. We rely on `semver` principles to draw conclusions that are aligned with versioning conventions that signal API instability. Our protocol addresses some limitations of the original study and expand the analysis to a new dataset spanning seven more years of the MCR. We implement and use `Maracas` to compute BCs between adjacent versions of libraries, and to detect locations in client code that are affected by such BCs.

The main results of the study are as follows:

Q1: **How are semantic versioning principles applied in the MCR?** 83.4% of all upgrades on MCR do comply with `semver` principles. Still, 20.1% of non-major releases are breaking, threatening client projects.

Q2: **To what extent has the adherence to semantic versioning principles increased over time?** The tendency to comply with `semver` practices has significantly increased over time: the number of non-major breaking releases has decreased from 67.7% in 2005 to 16.0% in 2018.

Q3: **What is the impact of breaking changes on clients?** Only 7.9% of the clients we analyse are impacted by the BCs introduced in adjacent library releases. However, when breaking declarations are used by client projects, they are likely to break.

According to these results, we state that libraries and client projects on the Maven ecosystem are *not* “breaking bad”. To be precise, developers of Maven projects tend to follow `semver` principles and are for the most part disciplined when introducing BCs. While the situation has improved over time, there is still room for improvement. Although the impact of BCs on client projects is low, more research is needed to support clients that are impacted and need to migrate their code. Differences with results reported on the original study are explained by major changes introduced in the protocol and the extended timespan of the new corpus.

As future work, we first would like to perform qualitative analyses to complement our findings. In particular, we would like to explain the phenomenon we observed: what are the motivations behind inserting BCs in non-major releases, and why has the adherence to semantic versioning increased so significantly. These questions could be answered by interviewing library maintainers and clients. Second, we would like to study how the evolution of the Java language itself impacts the definition of BCs, and how this affects libraries and clients. As new constructs are made available in Java (*e.g.*, the **`default`** operator in Java 8 or the **`record`** data type in Java 15), new BCs appear. At the same time, these new constructs provide new strategies to deal with certain BCs (*e.g.*, **`default`** methods allow to gracefully evolve an interface without forcing changes in existing implementations). Third, we believe that the understanding of how client projects react to BCs is another step towards finding a way to support library-client co-evolution. Thus, we aspire to study how clients react in the wild and which patterns can be identified from these reactions. Finally, we also would like to study behavioral incompatible changes in Java libraries.

Declarations

Funding This work was partially supported by the EU's Horizon 2020 Project No. 732223 CROSSMINER.

Conflict of interest None.

Data and code availability In order to ease the replication of our results, we make available the datasets, the analysis scripts, and `Maracas` available in a single replication package hosted on Zenodo:[13]

`https://zenodo.org/record/XXXXXX`

The archive contains a Docker image that setups a Java 8 environment, clones and compiles the analysis scripts and their dependencies (`Maracas`, `japicmp`), fetches the MDD and MDG corpora, and runs the delta computation and detection algorithms on both datasets. It also contains the source code and manual of `Maracas`, the tests we defined to assess our algorithms, a set of Jupyter notebooks that we used to analyse the data, and instructions for running the Docker image and replicating our results.

[13] The final archive will only be made available when the paper is finalized. In the meantime, we refer the reviewers to the companion webpages.

References

1. Basten B, Hills M, Klint P, Landman D, Shahi A, Steindorfer MJ, Vinju JJ (2015) M3: A General Model for Code Analytics in Rascal. In: 1st International Workshop on Software Analytics, IEEE, Piscataway, pp 25–28, DOI 10.1109/SWAN.2015.7070485
2. Bavota G, Canfora G, Penta MD, Oliveto R, Panichella S (2013) The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In: International Conference on Software Maintenance, IEEE, Washington, pp 280–289, DOI 10.1109/ICSM.2013.39
3. Benelallam A, Harrand N, Valero CS, Baudry B, Barais O (2018) Maven Dentral Dependency Graph. DOI 10.5281/zenodo.1489120
4. Benelallam A, Harrand N, Soto-Valero C, Baudry B, Barais O (2019) The Maven Dependency Graph: A Temporal Graph-based Representation of Maven Central. In: 16th International Conference on Mining Software Repositories, IEEE, Piscataway, pp 344–348, DOI 10.1109/MSR.2019.00060
5. Bogart C, Kästner C, Herbsleb J, Thung F (2016) How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In: 24th International Symposium on Foundations of Software Engineering, ACM, New York, pp 109–120, DOI 10.1145/2950290.2950325
6. Brito A, Xavier L, Hora AC, Tulio Valente M (2018) Why and How Java Developers Break APIs. In: 25th International Conference on Software Analysis, Evolution and Reengineering, IEEE, pp 255–265, DOI 10.1109/SANER.2018.8330214
7. Businge J, Serebrenik A, van den Brand MGJ (2010) An Empirical Study of the Evolution of Eclipse Third-party Plug-ins. In: Workshop on Software Evolution and International Workshop on Principles of Software Evolution, ACM, New York, pp 63–72, DOI 10.1145/1862372.1862389
8. Businge J, Serebrenik A, van den Brand MGJ (2015) Eclipse API Usage: the Good and the Bad. Software Quality Journal 23(1):107–141, DOI 10.1007/s11219-013-9221-3
9. Businge J, Kawuma S, Openja M, Bainomugisha E, Serebrenik A (2019) How stable are eclipse application framework internal interfaces? In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 117–127
10. Cossette BE, Walker RJ (2012) Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries. In: 20th International Symposium on the Foundations of Software Engineering, ACM, New York, pp 55:1–55:11, DOI 10.1145/2393596.2393661
11. Decan A, Mens T (2019) What Do Package Dependencies Tell Us About Semantic Versioning? IEEE Transactions on Software Engineering pp 1–1, DOI 10.1109/TSE.2019.2918315
12. Des Rivières J (2007) Evolving Java-based APIs. `https://tinyurl.com/yyqguo34`, last access 26.07.2019
13. Dietrich J, Jezek K, Brada P (2014) Broken Promises: An Empirical Study Into Evolution Problems in Java Programs Caused by Library Upgrades.

In: Conference on Software Maintenance, Reengineering, and Reverse Engineering, IEEE, pp 64–73, DOI 10.1109/CSMR-WCRE.2014.6747226
14. Dig D, Johnson R (2005) The Role of Refactorings in API Evolution. In: 21st International Conference on Software Maintenance, IEEE, Washington, pp 389–398, DOI 10.1109/ICSM.2005.90
15. Dig D, Johnson R (2006) How Do APIs Evolve? A Story of Refactoring: Research Articles. J Softw Maint Evol 18(2):83–107, DOI 10.1002/smr.v18:2
16. Fowler M (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston
17. Godfrey MW, German DM (2014) On the Evolution of Lehman's Laws. J Softw Evol Process 26(7):613–619, DOI 10.1002/smr.1636
18. Gonzalez-Barahona JM, Sherwood P, Robles G, Izquierdo D (2017) Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is. In: Open Source Systems: Towards Robust Practices, Springer, Cham, pp 182–192, DOI 10.1007/978-3-319-57735-7_17
19. Gosling J, Joy B, Steele GL, Bracha G, Buckley A (2014) The Java Language Specification, Java SE 8 Edition, 1st edn. Addison-Wesley Professional
20. Harrand N, Benelallam A, Soto-Valero C, Barais O, Baudry B (2019) Analyzing 2.3 Million Maven Dependencies to Reveal an Essential Core in APIs. arXiv preprint arXiv:190809757
21. Hora A, Robbes R, Valente MT, Anquetil N, Etien A, Ducasse S (2018) How Do Developers React to API Evolution? A Large-scale Empirical Study. Software Quality Journal 26(1):161–191, DOI 10.1007/s11219-016-9344-4
22. Jezek K, Dietrich J (2017) API Evolution and Compatibility: A Data Corpus and Tool Evaluation. J Object Technol 16(4):2:1–23, DOI 10.5381/jot.2017.16.4.a2
23. Jezek K, Dietrich J, Brada P (2015) How Java APIs Break - An Empirical Study. Inf Softw Technol 65(C):129–146, DOI 10.1016/j.infsof.2015.02.014
24. Ketkar A, Tsantalis N, Dig D (2020) Understanding Type Changes in Java. In: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, New York, DOI 10.1145/3368089.3409725
25. Klint P, van der Storm T, Vinju JJ (2010) EASY Meta-Programming with Rascal. Leveraging the Extract-Analyze-SYnthesize Paradigm for Meta-Programming. In: 3rd International Summer School on Generative and Transformational Techniques in Software Engineering, Springer, LNCS
26. Kula RG, Germán DM, Ouni A, Ishio T, Inoue K (2018) Do Developers Update Their Library Dependencies? - An Empirical Study on the Impact of Security Advisories on Library Migration. Empirical Software Engineering 23(1):384–417, DOI 10.1007/s10664-017-9521-5
27. Kula RG, Ouni A, German DM, Inoue K (2018) An Empirical Study on the Impact of Refactoring Activities on Evolving Client-used APIs. Inf Softw Technol 93(C):186–199, DOI 10.1016/j.infsof.2017.09.007
28. Kühne L, Massol V, Kitching S (2003) The Clirr Maven Plugin. `https://tinyurl.com/y6az94l4`, last access 08.04.2020

29. Lehman MM (1978) Programs, Cities, Students— Limits to Growth?, Springer, New York, pp 42–69. DOI 10.1007/978-1-4612-6315-9_6
30. Lehman MM, Fernández-Ramil JC, Wernick PD, Perry DE, Turski WM (1997) Metrics and Laws of Software Evolution - The Nineties View. In: 4th International Symposium on Software Metrics, IEEE, Washington, pp 20–32
31. Lindsay Murray R, Ehrenberg ASC (1993) The Design of Replicated Studies. The American Statistician 47(3):217–228, DOI 10.2307/2684982
32. Mirhosseini S, Parnin C (2017) Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-date Dependencies? In: Rosu G, Penta MD, Nguyen TN (eds) 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE, pp 84–94, DOI 10.1109/ASE.2017.8115621
33. Mostafa S, Rodriguez R, Wang X (2017) Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries. In: 26th International Symposium on Software Testing and Analysis, ACM, New York, pp 215–225, DOI 10.1145/3092703.3092721
34. Preston-Werner T (2013) Semantic Versioning 2.0.0. https://tinyurl.com/y7t7g6t5, last access 30.07.2019
35. Raemaekers S (2013) The Maven Dependency Dataset. https://tinyurl.com/uveepue, last access 09.04.2020
36. Raemaekers S, Visser J, van Deursen A (2012) Measuring Software Library Stability Through Historical Version Analysis. In: International Conference on Software Maintenance, IEEE, Washington, pp 378–387, DOI 10.1109/ICSM.2012.6405296
37. Raemaekers S, Deursen Av, Visser J (2013) The Maven Repository Dataset of Metrics, Changes, and Dependencies. In: 10th Working Conference on Mining Software Repositories, IEEE, Piscataway, pp 221–224
38. Raemaekers S, van Deursen A, Visser J (2014) Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In: 14th International Working Conference on Source Code Analysis and Manipulation, IEEE, pp 215–224, DOI 10.1109/SCAM.2014.30
39. Raemaekers S, van Deursen A, Visser J (2017) Semantic Versioning and Impact of Breaking Changes in the Maven Repository. J Syst Softw 129(C):140–158, DOI 10.1016/j.jss.2016.04.008
40. Robbes R, Lungu M, Röthlisberger D (2012) How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem. In: 20th International Symposium on the Foundations of Software Engineering, ACM, New York, pp 56:1–56:11, DOI 10.1145/2393596.2393662
41. Sawant AA (2019) The Impact of API Evolution on API Consumers and How This Can Be Affected by API Producers and Language Designers. PhD thesis, Delft University of Technology
42. Sawant AA, Robbes R, Bacchelli A (2016) On the Reaction to Deprecation of 25,357 Clients of 4+1 Popular Java APIs. In: International Conference on Software Maintenance and Evolution, IEEE, pp 400–410, DOI 10.1109/ICSME.2016.64

43. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell
44. Wu W, Khomh F, Adams B, Guéhéneuc YG, Antoniol G (2016) An Exploratory Study of API Changes and Usages Based on Apache and Eclipse Ecosystems. Empirical Softw Engg 21(6):2366–2412, DOI 10.1007/s10664-015-9411-7
45. Xavier L, , Hora A, Valente MT (2017) Why Do We Break APIs? First Answers from Developers. In: 24th International Conference on Software Analysis, Evolution and Reengineering, IEEE, pp 392–396, DOI 10.1109/SANER.2017.7884640
46. Xavier L, Brito A, Hora A, Valente MT (2017) Historical and Impact Analysis of API Breaking Changes: A Large-scale Study. In: 24th International Conference on Software Analysis, Evolution and Reengineering, IEEE, pp 138–147, DOI 10.1109/SANER.2017.7884616
47. Xi Y, Shen L, Gui Y, Zhao W (2019) Migrating Deprecated API to Documented Replacement: Patterns and Tool. In: 11th Asia-Pacific Symposium on Internetware, ACM, New York, DOI 10.1145/3361242.3361246
48. Xu S, Dong Z, Meng N (2019) Meditor: Inference and Application of API Migration Edits. In: International Conference on Program Comprehension, IEEE, Piscataway, pp 335–346, DOI 10.1109/ICPC.2019.00052
49. Zerouali A, Mens T, González-Barahona JM, Decan A, Constantinou E, Robles G (2019) A formal framework for measuring technical lag in component repositories - and its application to npm. J Softw Evol Process 31(8), DOI 10.1002/smr.2157, URL `https://doi.org/10.1002/smr.2157`