The Journal of Systems & Software 203 (2023) 111738

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

On the relation of method popularity to breaking changes in the Maven ecosystem[☆]

Mehdi Keshani [*], Simcha Vos, Sebastian Proksch

Delft University of Technology, Netherlands

ARTICLE INFO

Article history:
Received 17 August 2022
Received in revised form 24 March 2023
Accepted 27 April 2023
Available online 3 May 2023

Keywords:
Maven Central
Method popularity
Compatibility
Semantic versioning
API evolution
Breaking changes

ABSTRACT

Software reuse is a common practice in modern software engineering to save time and energy while accelerating software delivery. Dependency managers like MAVEN offer a large ecosystem of reusable libraries that build the backbone of software reuse. Breaking changes, i.e., when an update to a library introduces incompatible changes that break existing client programs, are troublesome barriers to this library reuse. *Semantic Versioning* has been proposed as a practice to make it easier for the users to find *safe* updates by encoding the change impact in the version number. While this practice is widely studied from the framework perspective, no detailed insights exist yet into the ecosystem perspective. In this work, we study violations of semantic versioning in the MAVEN ecosystem for 13,876 versions of 384 artifacts to better understand the impact these violations have on the 7,190 dependent *versioned packages*. We found that 67% of the artifacts introduce at least one type of semantic versioning violation, either a breaking change or an illegal API extension in their history. An impact analysis on breaking methods that (direct or transitive) dependents reference, revealed strong centralization: 87% of publicly accessible methods are never used by dependents and among methods with at least one usage, half of the unique calls from dependents concentrate on only 35% of the defined methods. We also studied method popularity and could not find an indication that popularity affects stability: even popular methods break frequently. Overall, we confirm the previous result that *Semantic Versioning* is violated repeatedly in practice. Our results suggest that the frequency of breaking changes might be a sign of insufficient change-impact awareness on the ecosystem and we believe that developers require more adequate information, like method popularity, to improve their update strategies.

1. Introduction

Software reuse is a pillar of modern software engineering. The availability of mature and powerful open-source libraries has boosted the productivity of developers both in open-source and industry. For easy release, discovery, and distribution, package managers like MAVEN, NPM, or PYPI usually rely on centralized repositories. These repositories contain vast numbers of inter-dependent packages that build self-contained, ever-growing software ecosystems (Decan et al., 2019). The use of dependencies has already been the focus of a rich body of previous works that studied, for example, the evolution of dependency networks (Benelallam et al., 2019; Soto-Valero et al., 2019), bloated dependencies (Soto-Valero et al., 2021, 2020), and vulnerable dependencies (Zapata et al., 2018).

Depending on a third-party library means that a project accepts a coupling to the API of said library. However, libraries might introduce *breaking changes* in subsequent versions that change the API in an incompatible way, for example, by removing a method or by changing its arguments. Updates are often necessary for consumers to fix bugs or vulnerabilities, so such a breaking change to the API is often an unwelcome surprise for the developers of the depending project, which forces them to adapt their code. *Semantic versioning* has emerged as a best practice to signal compatibility of a change to the previous version, but it is voluntary and not enforced in the ecosystem.

Numerous studies already investigated various aspects such as usage (Qiu et al., 2016; Wang et al., 2020), evolution (Bavota et al., 2015; Hora et al., 2014; Koçi et al., 2019; Lamothe et al., 2021), and stability (Raemaekers et al., 2012) of APIs. However, the closest related work is presented by Raemaekers et al. (2017) who studied the relation of *semantic versioning* and *breaking changes* in Maven. The paper finds evidence that many libraries break versioning rules and introduce breaking changes even in

[☆] Editor: Laurence Duchien.
[*] Corresponding author.
E-mail addresses: m.keshani@tudelft.nl (M. Keshani), a.s.j.vos@student.tudelft.nl (S. Vos), s.proksch@tudelft.nl (S. Proksch).

https://doi.org/10.1016/j.jss.2023.111738

non-major upgrades. However, they did not provide any details about the extent of the method-level impact on the ecosystem which is the main goal of our study.

In this paper, we advance the perspective on breaking changes from the library creator to the ecosystem. Instead of treating a breaking change to a library method as a single incident, we consider the methods' popularity in the ecosystem to weigh the severity of a breaking change. This is achieved by identifying all dependents of a library in question through a global dependency graph (Soto-Valero et al., 2019). This global dependency graph includes a large part of the recent releases in the ecosystem. It is important to note that we can query this graph for all possible dependents of any given library that is released recently. All dependents start using a version after its release date. Therefore, if we select a library that was released x months ago, all of its dependents are at least x months old hence our dependent graph includes them. After generating the call graphs (CGs) for all dependents, we can identify all references to method definitions in the original library (through calls or inheritance). A breaking change can then be identified by comparing the list of extracted references to the available methods in the next version of the library. This methodology not only allows us to replicate previous work and identify breaking changes in the library but also helps us to reason about the severity of a breaking change for the ecosystem. We expect that widely-used methods are more damaging to break because they affect more users. In this study, we aim to answer three research questions: (1) How often do semantic versioning violations occur? (2) How is popularity distributed among library methods? and (3) Is there a relation between popularity and breaking changes?

We have created a sample of 384 artifacts and 19,639 unique versions of them. From each of these artifacts, we picked one version. For the picked versions we identified a total of 7190 unique dependents on Maven Central. We used these dependents to infer method popularity. Our results confirm previous work by showing that 67% of artifacts violate semantic versioning. To better understand the effects that these violations have on the ecosystem, we have further investigated the methods' popularity. We found that 87% of publicly defined methods are never called by another library. From the remaining 13% only 35% receive half of all calls to the respective library.

In this study, we show that maintainers introduce breaking changes in popular methods as often as less popular methods. One possible explanation for this is that library maintainers may not always be aware of the popularity of their methods. It is important to note that the adoption of a library is typically in the best interest of its maintainers, and the lack of upgradability may sometimes hinder this goal. Providing sufficient information to library maintainers about the popularity of their methods has the potential to help them enhance the upgradability of their library. While some breaking changes may be inevitable, there are cases where the severity of certain breaking changes may be underestimated. In situations where a breaking change affects a widely used method, maintainers may decide to notify users with a major release.

The contributions of this study are as follows:

- A quantitative analysis of API method extensions and contractions that violate semantic versioning.
- An empirical study of the popularity distribution in typical APIs of Maven libraries.
- An investigation of the extent of user breakage that semantic versioning incompatibilities cause on Maven.

2. Related work

We found three areas of research to be closely related to this paper: software ecosystems, dependency management, and APIs and breaking changes.

Software ecosystems. Multiple studies investigated the software ecosystems from different aspects. Decan et al. (2019) found that dependency networks are growing over time by analyzing the evolution of different package managers. Moreover, they realized that a minority of packages are used by a majority of the network. Several studies (Kula et al., 2017b; Benelallam et al., 2019; Soto-Valero et al., 2019) modeled software ecosystems using *graphs* with package versions as *nodes*. They used these graphs to study Maven Central and the CRAN. Their findings show that Maven ecosystem has a more conservative approach to updating dependencies than CRAN. They also studied *activity*, *popularity*, and *timeliness* of more than 1*M* artifacts in the Maven Central. However, these studies only consider package-level relations. Raemaekers et al. (2013) presented the dataset of Maven containing information about 148K Java libraries and their CGs. The authors only provided a dataset of metrics and CGs but contrary to our work they do conduct any ecosystem analysis using this data. Hejderup et al. (2018) proposed a fine-grained dependency network that uses CGs to model function interactions in the ecosystem. The authors present a methodology to construct and analyze this network. This study also does not provide insights into the method-level relations of the ecosystems.

Dependency management. Various studies studied different aspects of dependency management such as their updates, trends, and adoption. Several other studies (Soto-Valero et al., 2021, 2020) studied the use of *bloated* (i.e., unused) dependencies. They showed that once a package becomes bloated in a project it is likely to stay bloated. They also found that bloated dependencies are mostly the result of transitive dependencies. Zapata et al. (2018) studied developer reactions to known vulnerable dependencies. This study shows that 73.3% of clients using vulnerable dependencies are not running vulnerable code. Hence they confirm that analysis at the library level is an overestimation and function-level analysis is needed. Alrubaye and Mkaouer (2018) automated library migration to save time and reduce the knowledge requirements for engineers. They use method-level changes in programs that already migrated and automate the procedure for others that are interested in migration. Mileva et al. (2009) presented an approach to support developers in their decision to upgrade a dependency using *wisdom of the crowds*. Macho et al. (2021) analyzed trends of changes in Maven build files. Kula et al. (2017a) presented a way to decide when a library needs to be updated based on its usage level. Kula et al. (2018) conducted an empirical study that covers over 4600 GitHub software projects and 2700 library dependencies. The findings of this study show that 81.5% of the studied systems keep outdated dependencies. Kula et al. (2015) studied the adoption habits and trust of maintainers towards new releases of an existing library. The study concludes that maintainers are becoming more trusting of new releases and becoming inclined to update their existing systems to the latest release.

The aforementioned studies inspect dependency-related topics. However, none of them empirically studies the API usages of the Maven ecosystem. Most of them only considered package-level relationships between packages. In this study, we will consider API usage and empirically study method-level networks. We leverage the *CGs* of libraries to provide insights about the *APIs* of the Maven with method-level precision.

APIs and breaking changes. Some studies specifically targeted the library APIs and investigated their patterns, stability, usage, etc. Qiu et al. (2016) studied API usage of 5k open-source projects. Their findings show that the API usage obeys a Zipf distribution and deprecated APIs are still widely used. Bavota et al. (2015) studied the evolution of a set of projects. They show that when releases contain major changes a large amount of them are bug fixes. They also show that developers are more reluctant to upgrade when APIs are removed or altered. Wang et al. (2020) empirically investigated the usage and updates of packages in Java projects. They also provide a prototype of a tool that alerts developers about updating packages. Xavier et al. (2017) studied the frequency of *breaking changes*, the behavior of these changes over time, their impact on users, and the characteristics of libraries with a high frequency of breaking changes. Lima and Hora (2020) categorized APIs into *popular*, *ordinary*, and *unpopular*. They found that popular APIs often have more public methods, more lines of code, a higher complexity, higher relative complexity per method, change more frequently, and have more contributors. However, they also found that there is no change in the relative line of code, method name length, or the number of parameters of popular APIs and they are often used early in the development cycle and are often more unstable. Harrand et al. (2022) performed a large scale study on Maven. They discovered that with sufficient users, all APIs seem to be used, but there is a concentration of usage on a small set of APIs. Meaning developers could focus on a smaller portion of APIs and save time. Hora et al. (2014) proposed a tool to extract rules by monitoring API invocation changes in the code history. This then can be used to keep track of the evolution of an API. Kim et al. (2011) performed a large analysis of API refactorings and bug fixes. Their findings revealed that the number of bug fixes increased after refactorings, while the time required to address these bugs decreased. Koçi et al. (2019) investigated changes that happen to APIs and classified them to gain a bigger picture of API evolution. Nguyen et al. (2010) presented LibSync, a tool for developers who want to upgrade their dependencies. It suggests to users a way of adapting their API usage by learning from clients that have already migrated to a new library version. Hora et al. (2018) performed an exploratory study to observe API evolution and its impact on the Pharo software ecosystem. Lamothe et al. (2021) reviewed the literature on APIs and API evolution. They conclude that the main challenges are identifying factors that drive API changes, creating a uniform benchmark for research evaluation, and the impact of API evolution on various programming languages. Raemaekers et al. (2012) proposed four different metrics for the stability of the libraries. Raemaekers et al. (2017) studied how new releases of a library impact the client libraries and their semantic versioning. They found that on average one in three releases introduce breaking changes that produce compilation errors that need to be addressed. The above-mentioned studies investigated APIs from different aspects. However, none of them focuses on an analysis of public APIs from the consumer perspective at the ecosystem level.

3. Experimental setup

Our approach contains multiple components that enable us to perform the desired analyses. In this section, we first provide a brief overview of these components and the overall methodology. Then we elaborate on different components for example the Sampler component that is used for the data selection.

3.1. Overview

Within Maven Central, all packages are uniquely identified by a triplet consisting of `groupId:artifactId:version`. In this paper, we refer to such an identifier as a *Versioned Package* (VP). We also use the term *artifact* to refer to a package but not a particular version, i.e. `groupId:artifactId`.

Fig. 1 illustrates an overview of the methodology of the paper. We resolve dependents of all *versioned packages* released in a particular time frame on Maven. For this, we use our Dependent resolver component which is described in Section 3.2. This component internally uses Dependency Resolver (red arrow in the overview figure). The next step is selecting a subset of *versioned packages* to analyze. We select 384 *versioned packages* for the analysis using the Sampler component. This component is described in detail in Section 3.3. For the *dependents* of any selected *versioned package*, we resolve their *dependencies*. We describe our dependency resolution in Section 3.4. Note that, the list of dependencies of each dependent contains the original *versioned package* that was selected as a target for the analysis.

In the rest of the paper, we call these *versioned packages* target *versioned packages* and their corresponding artifacts target artifacts.

In the next step, all dependency sets are transmitted to the Static Analyzer component (see Section 3.5). This component performs two crucial tasks. Firstly, it generates a Call Graph (CG) for each dependency set and transmits it to the CG Joiner. Secondly, it forwards the *method definitions* of the target artifacts and their respective versions to the SemVer analyzer (explained in Section 3.6). The CG Joiner (refer to Section 3.5) combines these Call Graphs into a single joined CG for each target *versioned package.*

These CGs contain all method calls from dependent *versioned packages* to the target *versioned packages*. All other edges such as internal calls of these libraries are filtered. Finally, we analyze method declarations to find violations of semantic versioning in SemVer Analyzer resulting in *violation information*. Popularity Analyzer then uses this information together with joined CGs to find if the violations affect the most popular methods.

Fig. 1 also shows the data that goes through this pipeline alongside an example. Consider $g : a : v_1$ as a *versioned package* that is released within our target time frame. $g : a : v_x$ is one of its dependents resolved by Dependent Resolver. In the second step Sampler selects $g : a : v_1$ as a target *versioned package*. Afterward, Dependency Resolver includes $g : a : v_1$ among the dependencies of $g : a : v_x$. In the fourth step, Static Analyzer generates a CG for all dependency sets including $g : a : v_x$'s. This component also uses Maven to find all the versions of selected artifacts and extracts their method definitions such as method `n()` in $g : a : v_1$.

After acquiring all the versions of the selected artifacts, the SemVer Analyzer computes any breaking changes and illegal API extensions, such as the removal of the method `n()` in $g : a : v_{1i}$, which is the subsequent minor release after $g : a : v_1$. Meanwhile, the CG Joiner joins the CGs it receives, resulting in a unified CG that contains an edge from method `m()`, defined in $g : a : v_x$, to method `n()`, defined in $g : a : v_1$. Using all the calls to the methods defined in the target *versioned packages*, the Popularity Analyzer calculates the popularity values for methods. For instance, the popularity value of `n()` is greater than zero because it is utilized at least once by $g : a : v_x$.

By removal of `n()` method `m()` would break should developers of $g : a : v_x$ decide to update to $g : a : v_{1i}$. We use popularity values, breaking changes, and illegal API extension information for the reports and figures of this study.

3.2. Dependent resolver

Dependent resolution is the process of identifying *versioned packages* that refer to a particular target *versioned package*, either directly or transitively. One needs to first perform dependency resolution for all *versioned packages* on the ecosystem. Using this information, one can create a so-called dependency graph. This graph determines which *versioned packages* are dependent on a given *versioned package* by retrieving the incoming edges of the given node. We replicated the approach presented by Benelallam et al. (2019) to create this graph.

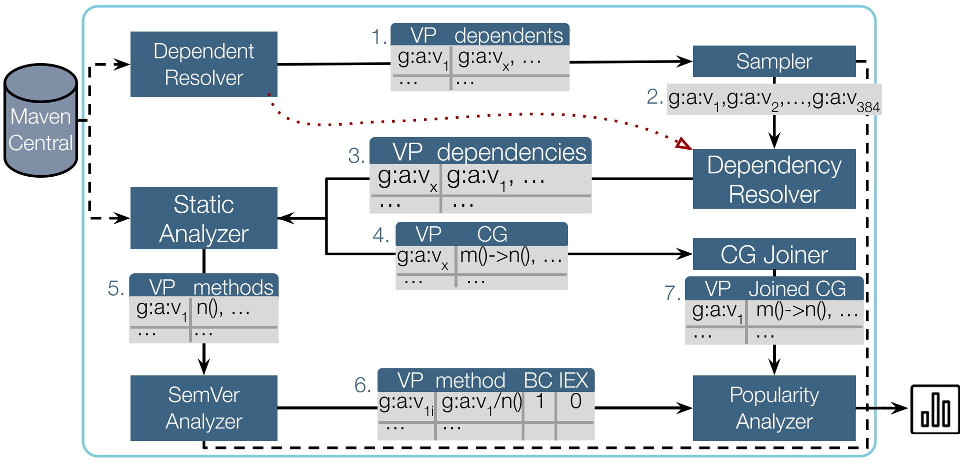


Fig. 1. Overview of the methodology. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

We created this graph by including releases from 1st of October 2021 to 31st of March 2022. This contains 537K *versioned packages* that are released in this time frame and we add them as initial nodes of this graph.

Maven index (mavenIndex, 2022) lists all *versioned packages* that are being released. We use this index to include releases of our target time frame. We also include all dependencies of each *versioned package* by analyzing their *POM* files. We do not resolve the exact versions of these dependencies in this phase and only store the information available in the *POM* file. Most dependency definitions specify the exact versions. For these cases, we simply add a dependency edge between two *versioned packages*. However, some *POMs* define version range dependencies. This is less common than exact versions dependency definitions. For these cases, we also store the range information separately. Consider *versioned package* `g:a:0` and `g:a:1` that both define the dependency `g:d:[1,5]` in their *POM* files. This dependency refers to all releases between version `1` and `5`. We use this example in the rest of this paragraph to show how Dependent Resolver functions. While adding `g:d` as a dependency we store the range information `[1,5]`. We resolve the exact versions only on demand once we receive queries. For each query about the dependents of a given *versioned package*, we first return all potential dependents that exist in the whole graph. This includes both direct and transitive dependents. For example, when `A` depends on `B` and `B` depends on `C` we also count `A` as a dependent of `C`. Next, we use the dependency resolver (see Section 3.4) to resolve the dependencies of each potential dependent at the current moment. After that, we check whether or not the given *versioned package* is present in their dependency sets. For example, when we receive a query about the dependents of `g:d:1` we return both `g:a:0` and `g:a:1`. We then resolve dependencies of `g:a:0` and `g:a:1` to validate whether or not `g:d:1` is among their dependencies. Assuming that `g:a:0`'s dependency set does not contain `g:d:1` and `g:a:1`'s does, we keep `g:a:1` and eliminate `g:a:0` from the list of its dependents. We opted for analyzing a particular time frame because it enables us to find the complete set of dependents for any *versioned package* that is released within or after our target time frame. Users can depend on each *versioned package* only after its release, not before, so this approach provides a comprehensive view of the dependents existing on Maven.

3.3. Sampler

We have continuously collected data from Maven central and created a dataset that represents the current state of the ecosystem. This dataset contains all *versioned packages* that are released between the 1st of October 2021 and 31st of March 2022. We call this time frame our sampling frame. The sampler component is responsible for selecting a set of *versioned packages* that is representative of this dataset. This sampling is done because CG generation is a highly expensive task and not feasible to perform for all existing *versioned packages*. Fig. 2 shows an example of this sampling. In the remainder of the Section, we use this example to describe the steps we take in our data selection. In the sampling time frame, some artifacts may have only one release (`g2:a1`), some may have multiple releases (`junit:x, g3:a2`), and some may have no release (`g1:a2, g3:a1`). As the first step of our selection, we filter the artifacts without any release in the sampling frame (`g3:a1, g1:a2`). Maven Central repository contains approximately 9*M* indexed packages. However, the aforementioned 6 months time frame contains approximately 537*K* *versioned packages*. These *versioned packages* are released within the sampling frame. For example, in Fig. 2 there are 6 *versioned packages* within the sampling frame (gray area) including two `junit:x`, one `g2:a1`, and three `g3:a2` releases. Fig. 3 shows the number of *versioned packages* that are released within the sampling frame on Maven Central.

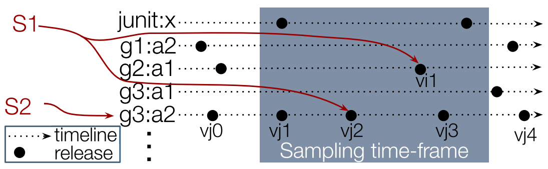


Fig. 2. Random selection example.

Before we sample, we apply two filters to the sampling frame; which allows us to create a more representative set of *versioned packages*. The first filter we apply is to remove testing-related *versioned packages* that contain the keywords `assertj`, `junit`, `mock` and `test` from the dataset. We do this because our purpose is to analyze the regular library API usage while testing-related libraries have different usage patterns. Also, the *versioned packages* uploaded to Maven Central usually do not contain sufficient bytecode information to analyze the testing-related part of the code. In the example, artifact `junit:x` will be filtered after this step which leads to four remaining *versioned packages* derived from two unique artifacts i.e. `g2:a1, g3:a2`. After applying this filter on Maven data, approximately 380K *versioned packages* remain. These *versioned packages* are derived from 10.6K unique *artifacts*. To avoid a bias towards artifacts with a high release frequency, we randomly select only one version from each of the

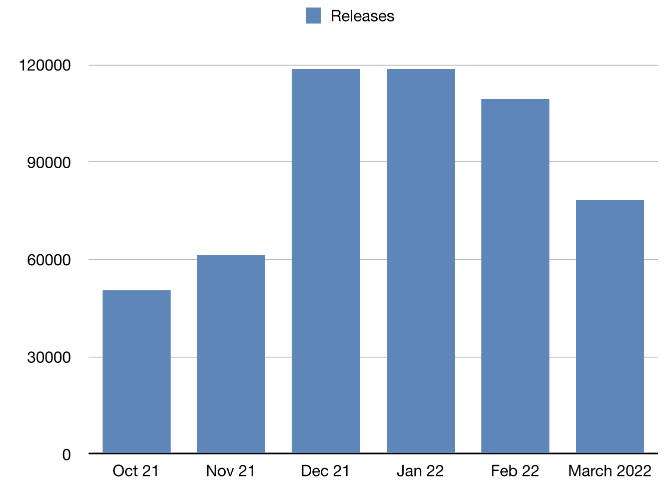


Fig. 3. Number of releases in the dataset per month.

10.6K unique artifacts that we could identify in the dataset. In the example, this step is specified with *S1*. I.e. from each remaining artifact (`g2:a1, g3:a2`) we randomly select one version. In case of `g2:a1` there is only one *versioned package* (`Vi1`) in sampling the frame, while `g3:a2` has three versions and we pick the second one (`Vj2`) randomly as shown in Fig. 2.

Popular *versioned packages* with many dependents influence the overall ecosystem more, thus we perform weighted random sampling based on the number of dependents (direct and transitive) that *versioned packages* have. In example, assume that `g3:a2:vj2` has 20 dependents and `g2:a1:vi1` has 10. Consequently, `g3:a2:vj2` is twice more likely than `g2:a1:vi1` to be selected in this step. Sampler uses Dependent resolver (see Section 3.2) to get the number of dependents that each *versioned package* has. Table 1 shows the number of dependents per *versioned package*. More than 7.5K artifacts do not have any dependents, which using weighted random sampling cannot be selected. To verify the correctness of this, we randomly picked 10 of these cases and manually checked their dependent number with two other sources: Libraries.io[1] and the usage tab of Maven. They all had no dependents. We observed that some of them are very new releases. Therefore, they did not have the chance yet to attract users. The rest are unpopular *versioned packages* due to different reasons such as being from a very unpopular package. We believe these cases happen because the majority of the *versioned packages* are barely used, especially in the early stages of their lives while a minority are highly used. Previous research (Soto-Valero et al., 2019; Harrand et al., 2022; Decan et al., 2019) as well as our findings show very similar patterns in library usage within the ecosystem. To achieve generalizable results, we made sure to select a representative subset of libraries. The dependent distribution for *versioned packages* follows an inverse logarithmic distribution. We selected 384 *versioned packages* from the 3.1K *versioned packages* with 10.7K non-unique dependents (7190 unique), which gives our results a confidence level of 95% and max the margin of error of 5%.

Our popularity analysis requires CG generation for all dependents of selected *versioned packages*. Thus we use the described 384 *versioned packages* in *Popularity Analyzer* component (Section 3.8). However, Static Analyzer (Section 3.5) and SemVer Analyzer (Section 3.6) use all versions of the selected artifacts that are available on Maven. In Fig. 2 we show this by *S2* as the final

Table 1
Number of dependents.

Dependents	*versioned package*
0	7533
1	2091
2–9	844
10–24	88
25–49	33
50	27

step of our selection process. Using weighted random sampling we pick `g3:a2:vj2`. This *versioned package* is used in *Popularity Analyzer* component for the impact analysis. However, Static and SemVer Analyzers inspect all versions of the corresponding artifact `g3:a2` including `vj0, vj1, vj2, vj3`, and `vj4`.

3.4. Dependency resolver

Dependency resolution is resolving what dependencies a *versioned package* needs to compile, build, test or run. These dependencies are (directly or transitively) specified in the *pom.xml* file. For transitive dependency resolution, one needs to recursively get the dependencies of all dependencies and consider the Maven-specific resolution rules for solving conflicting definitions within the dependency set.

Dependency resolution in Maven is not deterministic because of version ranges (VersionRanges, 2022). New releases on Maven may change the outcome of dependency resolution for existing projects, even if the specified dependencies in *pom.xml* are stable. One such scenario arises when there are conflicting versions in transitive dependencies. Suppose we have two dependencies, D_1 and D_2, that rely on different versions of the library *L*, for instance, $D_1 \rightarrow L_{v1}$ and $D_2 \rightarrow L_{v2}$, and project *X* depends directly on both D_1 and D_2. This leads to a dependency conflict because *X* cannot include both L_{v1} and L_{v2} in its dependency set simultaneously. Different versions of the same package may have varying APIs and behaviors, so Maven permits only one version from each package to be present in the dependency set after resolution.

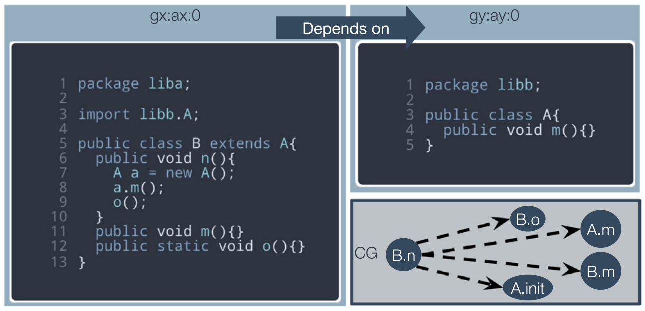
When a conflict occurs, Maven addresses it by conducting a breadth-first search and choosing the closest version of the conflicting dependency to the root. For instance, if *X* defines D_1 before D_2, the closest version of *L* to the root (*X*) is L_{v1}. As D_2 defines L_{v2} in the dependency tree of *X*, L_{v2} appears after L_{v1}. In this example, the dependency set may alter from the perspective of D_2 when *X* relies on D_1, compared to when it does not. Maven has numerous similar cases of dependency resolution, making it excessively complicated. We do not implement this feature ourselves, but instead, we use a re-implementation of Maven's built-in dependency resolution from a Java library (artifact, 2022). This tool enables us to include all dependencies, including transitive ones, when resolving dependencies. As a result, we handle transitive dependencies similarly to direct dependencies.

3.5. Static analyzer

The static analyzer component extends an existing framework OPAL (Helm et al., 2020) to perform two types of analyses. This analyzer both generates CGs for *versioned packages* and their dependencies and extracts the method definitions in versions of a given artifact. In the following, we explain each of these analyses.

Generation. After receiving a dependent and its dependency set we perform *class hierarchy analysis (CHA)* (Dean et al., 1995) to

1 https://libraries.io/.



```
package liba;

import libb.A;

public class B extends A{
  public void n(){
    A a = new A();
    a.m();
    o();
  }
  public void m(){}
  public static void o(){}
}
```

```
package libb;

public class A{
  public void m(){}
}
```

Fig. 4. Example dependency set and its CG.

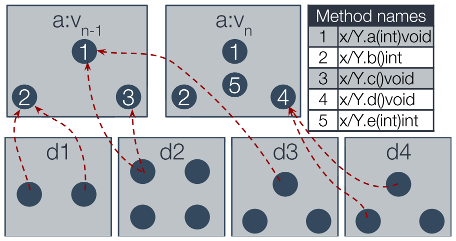
ensure all possible method calls are contained within the CG. A class hierarchy analysis determines a program's class inheritance graph and the set of methods defined in each class. Using these two pieces of information we add all possible invocations of any method to the CG.

Fig. 4 illustrates a minimal dependency set and its corresponding CG, which we will use as an example to elaborate on the steps we take in our analyses. As previously explained, CHA analysis overapproximates and draws edges to all possible implementations of a target method. For example, in the running example, method `n` in class `B` calls method `m` using object `a`. However, since the CHA algorithm does not reason about the control flow of the program, the exact type of `a` is unknown. Therefore, `B.n` is connected to both `A.m` and `B.m`, as illustrated in Fig. 4. This becomes more complicated when object `a` is not defined in the same scope, such as when it is passed as an argument to method `n`. It is worth noting that `A` could also be an interface, and `m` could be an abstract method. In such cases, the algorithm would perform similarly, except it would not draw an edge to `A.m` because an abstract method is not executable. Nonetheless, the algorithm considers `A` in the type hierarchy and draws edges to its subtypes.

It is important to note that the imprecision caused by this algorithm only occurs for virtual dispatch calls, which are calls that cannot be statically resolved. Despite this imprecision, we believe CHA is a suitable trade-off for our analysis, balancing soundness, scalability, and precision. More precise analyses, such as CFA CG generation, lack scalability. Dynamic CG generation is also not practical for our use case due to a lack of scalability and coverage.

In this part, we also generate unique *identifiers* for each method within the ecosystem. We call these identifiers *Global IDs* (GID). These *GIDs* help us join the generated CGs in CG Joiner 3.7.

Method definition extraction. For each target artifact, we first query Maven for all versions. Having all versions we extract all *public method definitions* that they have and assign them a *GID*. It is important to note that we only consider method definitions (methods with a body) because CGs only resolve edges to defined methods since declared methods (without a body) cannot be executed. In the next steps of the study, we limit the scope to methods with body i.e. method definitions to be able to connect semantic versioning violations to method popularity. Moreover, since public visibility is the most common access modifier for library usage we consider such methods as API endpoints that we analyze. Hence, we use the term *publicly defined method* to refer to such methods.



Method names	
1	x/Y.a(int)void
2	x/Y.b()int
3	x/Y.c()void
4	x/Y.d()void
5	x/Y.e(int)int

Fig. 5. Two releases of an example artifact and their dependents.

3.6. SemVer analyzer

This component analyzes *semantic versioning* violations in a given *versioned package*. We categorize these violations into two categories. The first category is *breaking changes*, which are changes that break compatibility within a major version. The second category is *illegal API extensions*, which is considered an extension of the API in patch versions.

We define breaking changes as changes that alter or remove a method signature. In addition to the name, our definition of method signature also includes return type and all arguments. Such changes should be introduced in major version upgrades only, *Semantic Versioning* is violated when these changes occur in minor or patch releases. This means that, when a new minor or patch version is released, a public method signature has been altered or removed which breaks the compatibility of an API. This leads to issues in case this method is called by some dependents. Therefore, changes that remove method signatures are not allowed, unless they are part of a major version upgrade.

To detect such violations we analyze the evolution of method signatures in the release history within the sampling frame. We look at the set of method signatures publicly available in a given *versioned package* v_n and compare it to the previous version's set of method signatures, the goal is to identify cases, in which signatures that existed before are removed in v_n (See Eq. (1)).

$$BC(v_n) = \begin{cases} Sig(v_{n-1}) - Sig(v_n) & \exists\{(v_n, v_{n-1}) \in Major(v_n)\} \\ \emptyset & otherwise \end{cases} \tag{1}$$

where:

$BC(v_n)$	= Set of breaking change signatures introduced in v_n
$Major(v_n)$	= Set of patch and minor in the same major version as v_n
$Sig(v_n)$	= Set of public method signatures defined in v_n

Fig. 5 shows two consecutive releases of artifact a. We use this figure as a running example to explain the next steps of our approach. For conciseness, we use a number to refer to each method. This figure also shows fully qualified method names next to their corresponding number. To calculate the breaking changes of *versioned package* $a : v_n$, assuming that v_n and v_{n-1} are within the same major version, we need to subtract the set of methods in $a : v_n$ from $a : v_{n-1}$. This means $\{1, 2, 3\} - \{1, 2, 4, 5\}$ resulting in $\{3\}$ which can also be illustrated as $BC(a : v_n) = a : v_{n-1}/x/Y.c()void$.

The second category of violations is the extension of the API in a patch version. We use a similar approach as before with a small adaptation. We iterate over the different patch versions and detect if a new method signature is added. This means there has been an extension of the API within a patch version. We find illegal API extensions of *versioned package* v_n by finding the difference between the previous patch version's set of method signatures and the set of method signatures in v_n, the goal is to identify cases, in which a signature did not exist before but was added to the v_n (See Eq. (2)).

$$IEX(v_n) = \begin{cases} Sig(v_n) - Sig(v_{n-1}) - Up(v_n) & \exists\{(v_{n-1}, v_n) \in Minor(v_n)\} \\ \emptyset & otherwise \end{cases} \tag{2}$$

where:

$IEX(v_n)$	= Set of signatures illegally added to v_n
$Up(v_n)$	= Set of updated signatures in v_n
$Minor(v_n)$	= Set of patch releases within the same minor version as v_n
$Sig(v_n)$	= Set of public method signatures defined in v_n

Our approach shares similarities with a conventional diff calculation function in that it treats updated parts as a removal and an addition. Consequently, the updated methods belong to both the 'BC' and 'IEX' categories if we do not exclude them. While these updated methods qualify as 'BC' because they may impact users, they should not be considered as 'IEX' because they are related to previously existing methods and not independently added. To identify the methods that truly belong to the 'IEX' category, we subtract the updated methods ($Up(v_n)$) in Eq. (2). To compute the set of potential updated methods, we use a heuristic approach that only considers sets of fully qualified signatures in releases. The heuristic approach, outlined in Algorithm 1, identifies five categories of changes in a release, including package name, class name, method name, parameters, and return types refactoring. This heuristic searches for pairs of removals and additions that contain only one renamed piece of fully qualified method names. For instance, if an added method has a similar package name, class name, method name, and parameters to a removed method and only differs in the return type, we consider it as one potential update. We do not consider other cases of updates in this heuristic such as when two or more elements of the signature are updated. Although this approach may not capture all types of updates that can occur in a release, it provides a reasonable approximation for our study.

$Up(v_n)$ is the only source of unsoundness in our calculation of 'IEX' and can be replaced with more accurate approaches if necessary. Notably, achieving accuracy in this context would require calculating the differences between the complete binary files of consecutive releases, which is resource-intensive and impractical in terms of scalability. Therefore, this approach falls outside the scope of our study.

Algorithm 1 Find Updated Methods

```
Require: added: list of method signatures added to $v_n$
Require: removed: list of method signatures removed from $v_n$
Ensure: result: list of updated signatures in $v_n$
1:  result ← {}
2:  for $a \in added$ do
3:      for $r \in removed$ do
4:          $r.sig \leftarrow [r.pkg, r.class, r.name, r.params, r.return]$
5:          $a.sig \leftarrow [a.pkg, a.class, a.name, a.params, a.return]$
6:          if r.sig differs from a.sig in only one element then
7:              $result \leftarrow result \cup \{a\}$
8:          end if
9:      end for
10: end for
11: return result
```

Returning to the example in Fig. 5, we illustrate the process of calculating illegal API extensions for *versioned package* $a : v_n$. First, we subtract the set of methods in $a : v_{n-1}$ from v_n. That is, $1, 2, 4, 5 - 1, 2, 3$, which results in $4, 5$. Alternatively, we can express this as $IEX(v_n)+Up(v_n) = x/Y.d()void, x/Y.e(int)int$. Next, using Algorithm 1, we compare all pairs of additions and removals in v_n. The set of additions in v_n is $4, 5$ and the set of removals is 3. This means that in this algorithm, we compare $x/Y.c()void$ to $x/Y.d()void$ and $x/Y.e(int)int$. The only two cases with a single element difference (method names) are 3 and 4. Thus, we count 4 as an updated version of 3. This implies that $Up(v_n) = x/Y.d()void$ and $IEX(v_n) = x/Y.e(int)int$.

3.7. CG joiner

Using the list of dependents, we need to determine how every dependent interacts with target *versioned package*. Within Static Analyser 3.5 we generated a CG for every dependent and its dependencies. Note that the dependencies of each dependent contain a target *versioned package*. Initially, we build one unique CG per dependent, but as every node has a unique identifier, we can join these individual CGs to get a joint CG for one target *versioned package*. Every node within our CGs has a unique identifier. While analyzing each *versioned package* for the first time we use a combination of Maven coordinates of the *versioned package* and fully qualified names of methods (including java package and return types) to uniquely identify each method within the ecosystem. Consequently, we can join CGs that are related to a target *versioned package* into a single CG. See Fig. 6 for an example of CG joining. In this Figure, a node P is common between two graphs. When we join these graphs the result shows what other nodes call this node from both graphs. See Eq. (3) for the mathematical formula behind joining CGs.

$$G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(a, b) :\in V_1, b \in V_2\}) \tag{3}$$

where:

G_x	= Graph x
V_x	= Vertices of graph x
E_x	= Edges of graph x

After joining the CGs of a target *versioned package* we have a representation of the interactions between *versioned package* and its dependents, in an individual CG. However, this joined CG contains many edges that are irrelevant to our study. Every possible edge that happens in the context of each dependent is present in this joined graph, such as internal calls of the dependents. In the next step, we filter all irrelevant edges. Suppose we are analyzing target *versioned package* x, we reduce this joint CG to the edges that have a source outside of x and a target inside of x. Consider the running example in Fig. 4. Suppose `gy:ay:0` is one of our target *versioned packages*. In the filtering step, we iterate over all four CG edges shown in this figure. For each of them, we check the two aforementioned conditions. All of these edges pass the first check which is whether or not their source method is defined outside of `gy:ay:0`. This is because `B.n` is defined in `gx:ax:0` and not in our target *versioned package* (`gy:ay:0`). In the second condition check, however, two of the edges are identified as filterable edges. Since `B.o` and `B.m` are defined within `gx:ax:0` we filter `B.n->B.o`, `B.n->B.m`. Similarly, we process any existing edge in the joined CG. This procedure also filters indirect calls to *versioned packages*. For example, a call from another method to `B.n` would be filtered since `B.n` is not defined within target *versioned package*. In this study, our focus is the intentional usage of library methods and indirect calls in the CG do not capture them. The remaining CG only includes the method-level interactions between the library and its dependents. This

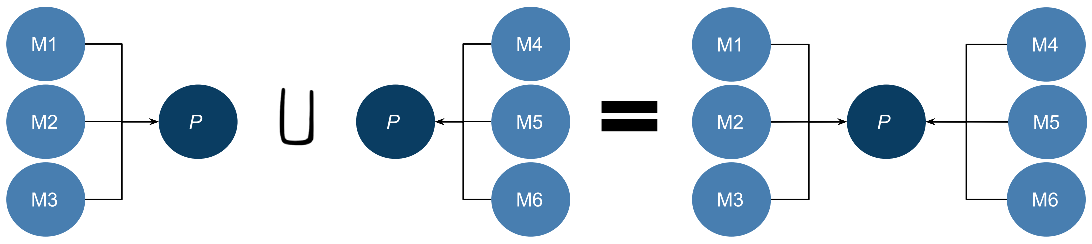


Fig. 6. Example of CG joining.

allows us to determine popularity scores and influence ratings based on all interactions within a given *versioned package*.

3.8. Popularity analyzer

This component calculates two types of popularity values. Firstly, it calculates the popularity of the target *versioned packages*. And secondly, it calculates the popularity of methods within them. We are inspired by Raemaekers et al. (2017) to use the term popularity in this study. For the library popularity, we divide the number of dependents of *versioned package* v by the number of all unique dependents (7190) to calculate the popularity of v (see Eq. (4)). The value reflects the popularity of a *versioned package* among dependents.

$$P(v) = \frac{|Dependents(v)|}{\left|\bigcup_{t\in T} Dependents(t)\right|} \tag{4}$$

where:

$P(v)$	= popularity of *versioned package* v
$Dependents(v)$	= dependent set of *versioned package* v
T	= set of target *versioned packages*

Consider Fig. 5. To calculate the popularity of $a : v_n$ we should divide the number of dependents that $a : v_n$ has by the number of all dependents. Assuming that our dataset only includes four dependent *versioned packages* ($d1, d2, d3, d4$) we should divide one by four. $d4$ is the only dependent that uses $a : v_n$ thus $P(v_n) = 1/4$. This value is $3/4$ for $P(a : v_{n-1})$.

Having the joined CGs we count the distinct dependents that call each method of the target *versioned package*. We then divide this by the number of all dependents that the target *versioned package* has to understand the relative popularity of a method among its dependents. We devise a simple metric called *distinct dependents usage*. Eq. (5) shows this metric for a given method M.

$$DR(m) = \frac{1}{|Dependents(p)|} \sum_{d\in Dependents(p)} \begin{cases} 1 & \exists\{d, m\} \in CG_P \\ 0 & otherwise \end{cases} \tag{5}$$

where:

$m \in p$	= *versioned package* p defines method m
$DR(m)$	= ratio of dependents that call the method m
$Dependents(p)$	= dependents set of the target *versioned package* p
$\{d, m\}$	= edge between a dependent d and a method m
CG_P	= joined CG of p

Consider Fig. 5 once again. The popularity value for $a : v_{n-1}/x/Y.a(int)void$ is 2/3 since out of three dependents of $a : v_{n-1}$ ($d1, d2, d3$) two of them ($d2, d3$) call this method. Moreover, $DR(a : v_{n-1}/x/Y.b()int) = 1/3$ since only $d1$ calls this method. $DR(a : v_{n-1}/x/Y.c()void)$ equals 1/3 because only $d2$ calls it. Finally, $DR(a : v_n/x/Y.d()void) = 1/1$ as $d4$ is the only dependent of $a : v_n$ and the only caller of this method.

Table 2
Type of release per artifact for the selected artifacts.

	Mean	Median	Std	Sum
Major	1.4	1.0	2.1	509
Minor	11.1	5.0	16.6	3 894
Patch	26.8	11.0	55.2	9 473
Total	39.3	23.0	60.1	13 876

4. RQ1: How often do semantic versioning violations occur?

The first category of violations of semantic versioning is through breaking changes, which are changes that break compatibility. We expect that the evolution of an API sometimes leads to incompatibility, due to developers removing or changing the existing set of method signatures of an artifact within a minor or patch release. The second category of violations of semantic versioning is the extension of an API through patch versions. Hence, we investigate the extent of semantic versioning violations in the first research question.

As a first step, we filter the *versioned packages* that do not comply with the semantic versioning structure. I.e we filter *versioned packages* with a qualifier such as pre-releases, snapshots, betas, etc. We do this because semantic versioning does not provide any rules for them. Selected artifacts have 5763 releases with one type of such qualifiers. However, they have 13,876 releases that adhere to the default structure of the semantic versioning and we use them in our study.

We use *SemVer Analyzer* as described in Section 3.6 to retrieve the set of breaking change methods in all remaining target *versioned packages* and their newer versions. We compare the retrieved set of breaking change methods with the set of total methods defined in respective artifacts and calculate the percentage of methods that break in each artifact. Using this data, we can show the extent of violations within the ecosystem at the library level. We then compare the number of methods that artifacts define and the number of breaking changes they introduce. This information helps us discover any method-level trends between the number of methods and the number of violations if they exist. We conduct similar analyses for illegal API extensions as well.

To understand the extent of semantic versioning violations we analyze the selected artifacts. Table 2 shows the number of releases per artifact. As this table shows, patch releases (9473) happen more often than minor releases (3894). This is expected because multiple patches are usually released between two minor releases. The same explanation is also valid when comparing the number of minor (3894) and major (509) releases. Overall, we analyze all releases (13,876) of 384 selected artifacts. In the rest of this section, we investigate how often breaking changes occur and, if they occur, what portions of the artifacts are affected. Firstly, we found that among selected artifacts 244(63%) introduce at least one breaking change in their history. Secondly, as Fig. 7 shows the percentage of methods involved in breaking changes differs among artifacts that have at least one breaking

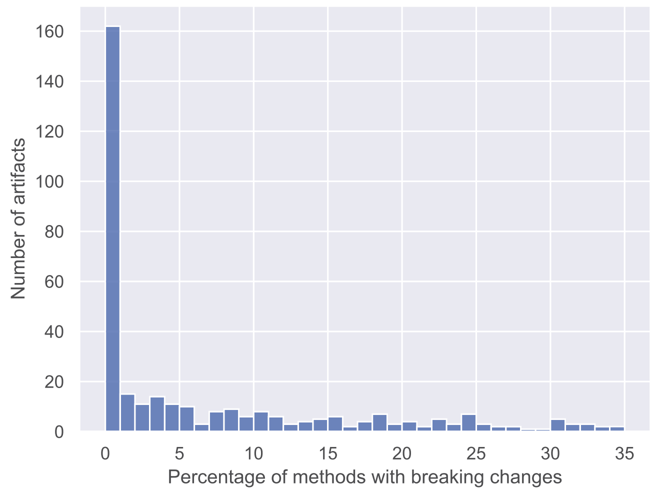


Fig. 7. Ratio of methods involved in breaking changes.

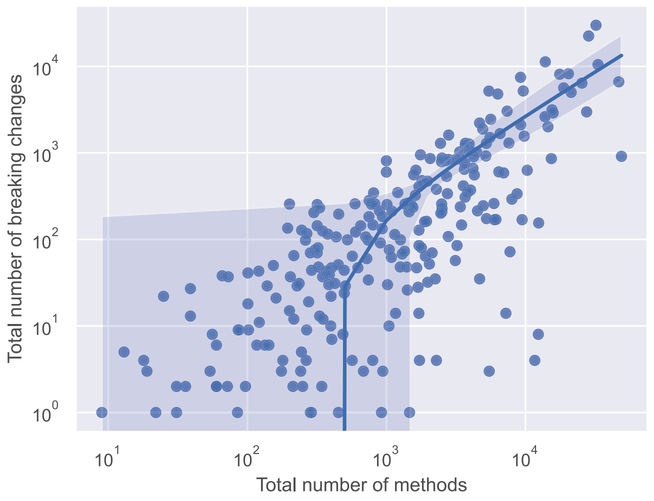


Fig. 8. Increase of breaking changes in relation to the total number of methods.

change. This figure shows the percentage of public methods, defined in an artifact, that are involved in breaking changes. In 22(9%) of artifacts, less than 1% of methods are involved in breaking changes. These percentages are calculated for only artifacts with breaking changes. On average, 19% of methods of the artifacts with breaking changes are affected by the breakage. In addition, we can notice that for higher ratios, only a small number of artifacts have such a high ratio. Only 22(9%) artifacts have more than 50% of their methods involved in breaking changes. Fifty-five percent of artifacts feature a ratio that is not bigger than 15%. However, several outliers exist that feature a noticeably higher ratio, indicating that these artifacts contain many breaking changes.

Now we realize different artifacts break to different extents. That is the number of methods involved in breaking changes highly differ between artifacts. We suspect that some libraries may break more methods because they have more methods in total. To this end, we investigate whether or not the total number of methods that these artifacts define is related to the number of breaking change methods. Fig. 8 shows the total number of public methods that artifacts have against the total number of breaking changes that occur in them. The axes in this figure are logarithmically scaled because the number of methods and breaking change methods highly differ between artifacts. Therefore, it is not practical to show them in a linear plot. As can be seen in this figure there is a tendency for artifacts with more methods to involve more methods in breaking changes. We fit a linear regression model with a confidence interval of 95% to better show this trend. The blue line in the figure shows this regression model. We conclude that indeed a larger number of methods leads to a larger number of breaking changes. However, we observe that the number of breaking changes does not grow as rapidly as the number of methods. This is in line with the findings of Raemaekers et al. (2017). They found a correlation between the number of methods in a library and the number of breaking changes and showed that bigger libraries introduce more breaking changes. They also reported that 30% of all releases contain at least one breaking change. Unlike them, in this RQ we conduct artifact-level analysis. To further understand how our results compare to theirs we calculated the number of minor and patch releases that contain breaking changes. We found that 14% of minor and patch releases contain breaking changes. We did not include the major releases here because major releases are allowed to have backward incompatible changes according to semantic versioning. The difference can be explained by another finding of Raemaekers et al. (2017) that says adherence to semantic versioning principles increases over time. For example, they reported a decrease of breaking changes in non-major releases from 28.4% in 2006 to 23.7% in 2011. This is a positive observation about the ecosystem and shows significant improvement in practices over time.

We showed that there exist some artifacts that introduce the first category of semantic versioning violation, breaking changes, and they do this to different extents. Now, we investigate the second type of semantic versioning violation, Illegal API extensions. Therefore, we inspect the number of methods that are added in patch releases. In Fig. 9, the number of artifacts is plotted against the percentage of methods of artifacts that are added via illegal API extensions. One may notice the distribution is similar to breaking changes (Fig. 7), however, the number of artifacts with API extensions is overall a bit lower than breaking changes. More specifically, 207(54%) of the artifacts introduce at least one illegal API extension in their history. Note that, the set of artifacts is not equal to the set of artifacts that feature breaking changes. Overall, 257(67%) of artifacts introduce at least one type of semantic versioning violation. It is worth mentioning that artifacts with illegal extensions on average add 14% to their set of existing methods through the illegal API extension. Thirty-eight (18%) of these artifacts illegally extend their API methods up to 1% and only 12(5%) of them add 50% or more methods via illegal extensions.

Finally, after elaborating on illegal extensions, we realize that similar to breaking changes the extent of the effect is different between libraries. Therefore we aim to understand whether or not this extent is related to the overall size of the artifacts. One could expect that bigger libraries may prevent adding yet more methods in general not to mention adding via illegal extensions. However, surprisingly Fig. 10 shows that similar to breaking changes the number of illegal API extensions also grows along with the number of methods.

Sixty-three percent of the artifacts introduce breaking changes, and 67% of the artifacts feature at least one type of semantic versioning violation. Furthermore, the more public methods artifacts define the more likely it is that they introduce semantic versioning violations.

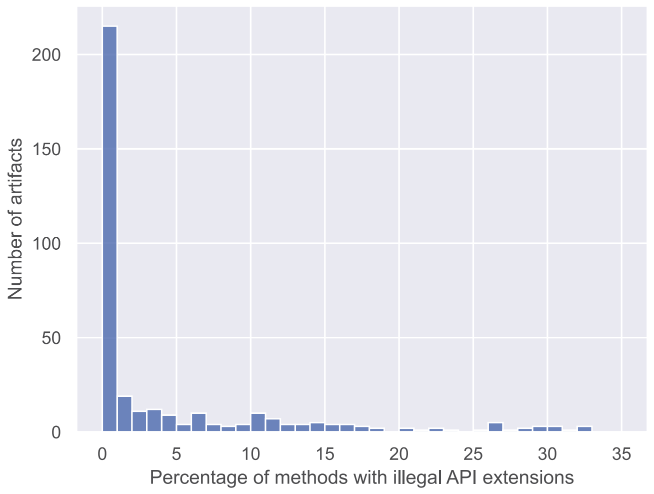


Fig. 9. Ratio of methods involved in illegal API extensions.

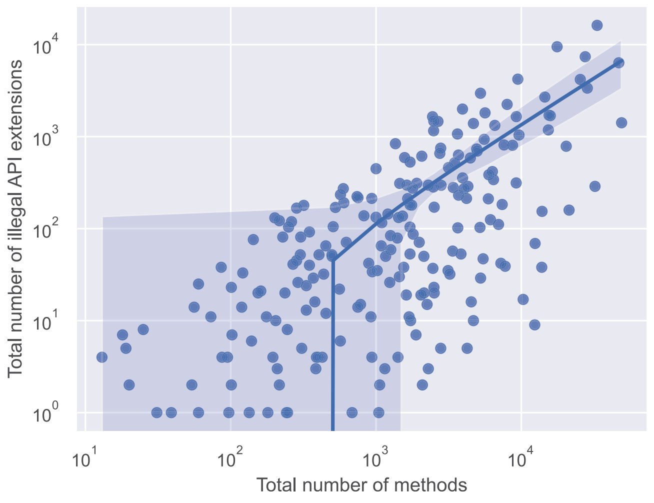


Fig. 10. Increase in number of API extensions in relation to the total number of methods.

5. RQ2: How is popularity distributed among library methods?

Now that we have answered the first research question, we have an insight into how often violations of semantic versioning occur in general. However, understanding the importance of the methods that break from the perspective of users within the ecosystem is also critical. Hence, as the next step, we investigate the popularity of the methods. Despite any relation that popularity and breaking changes may have, all attributes of popular methods propagate more within the ecosystem. Therefore, it becomes important to understand what portion of libraries are generally popular and widely used. Maintainers could for example pay extra attention to the important parts of their libraries and be more careful not to break them.

To answer this research question, we first investigate the popularity of target *versioned packages* at the library level. We use the output of *Sampler* component (see Section 3.3) to show the difference between the number of dependents that target *versioned packages* have. After this, we analyze the method-level popularity. We use the *Popularity Analyzer* as described in

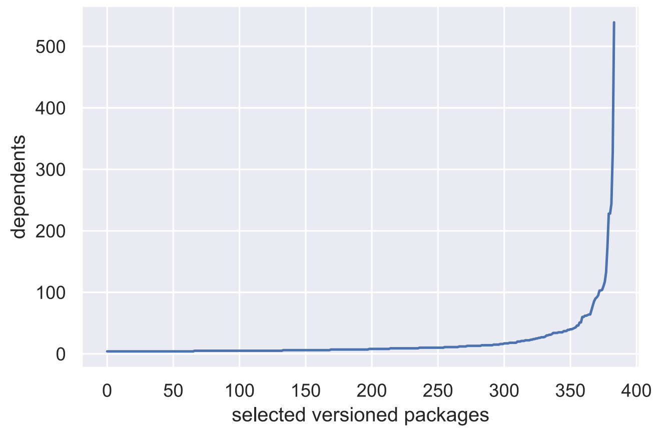


Fig. 11. Distribution of number of dependents for selected *versioned packages.*

Section 3.8 to calculate a popularity score for every method within target *versioned packages*. We inspect these popularity scores in two groups. Firstly, the popularity scores for the majority of the public methods indicate zero usage by other libraries. Hence, we first look into the ratio of unused methods compared to the total number of public methods that *versioned packages* define. Secondly, we analyze the methods that are used by other libraries at least once and show how their popularity scores are distributed.

To fully comprehend the distribution of library-method popularity, one must recognize that not all libraries have the same number of users. Fig. 11 shows the number of dependents that selected *versioned packages* have. As shown in this figure a limited number of *versioned packages* are used by many dependents but most of the *versioned packages* have less than 100 dependents. More specifically, only 12(3%) of the selected *versioned packages* have more than 100 dependents while 310(80%) of them have less than 20 dependents. The distribution follows the *Pareto principle*, which states that roughly 80% of consequences come from 20% of the causes (Dunford et al., 2014).

There also exist public methods within libraries that are not used by others. To this end, we first look into the extent of public methods that are not used by other libraries. Fig. 12 shows that the majority of publicly defined methods are not used by any other library. On average *versioned packages* that we selected for this study define 1688 public methods. However, the median of public methods for this *versioned packages* is 386 which indicates that some outlier projects skew the average. That is, we have a *versioned package* with 32k publicly defined methods as an outlier that affects the average. Additionally, there are some *versioned packages* that primarily concentrate on interfaces and define only a few public methods that can be called externally. In fact, we have 13 *versioned packages* with less than 10 publicly defined methods. Moreover, the second violin plot shows the distribution of publicly defined methods that are not called by any other library. On average 1.4k unused public methods are defined by selected *versioned packages*. However, similar to the number of methods this average is skewed by the outliers such as the project with the maximum number of unused methods with 31.6k unused methods. Therefore, the median of the unused methods is 386. There are also 14 *versioned packages* with less than 10 unused public methods. Finally, on average, the ratio of public methods that are not used by other libraries is 87% (median of 92%). The minimum ratio is 0.39 and the maximum ratio is 1. That is, (1) maximum coverage of public methods is 39% among

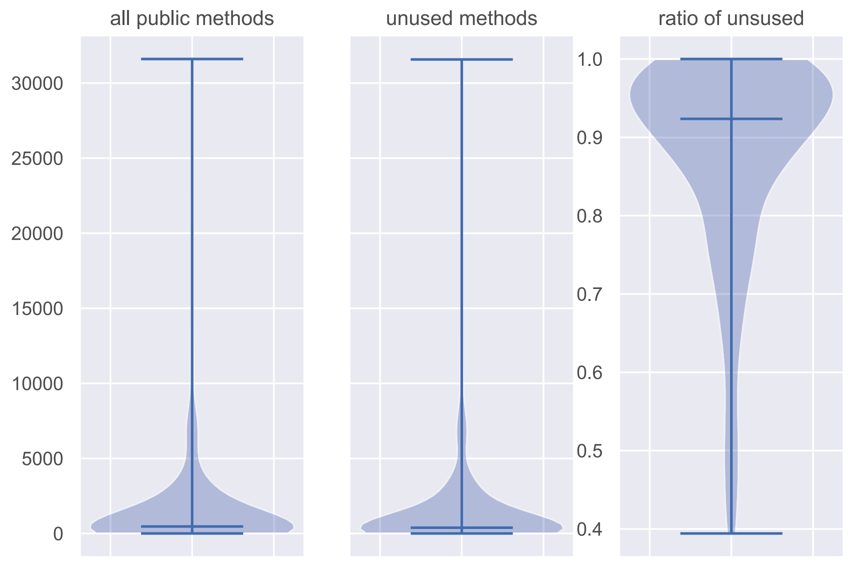


Fig. 12. Unused public methods of selected *versioned packages*.

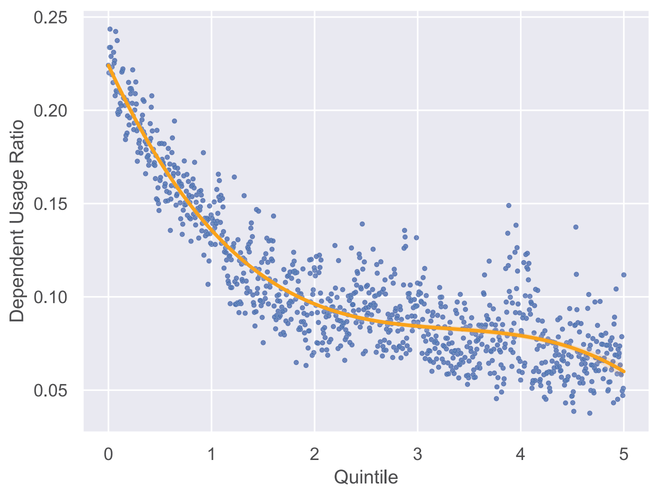


Fig. 13. Popularity distribution for dependent usage percentage.

all analyzed libraries, and (2) there exist libraries within selected packages that non of their publicly defined methods are used by others, which is because of their interface-based nature as our manual inspection revealed.

To look more closely at the used parts of these libraries we calculate the dependent usage ratio metric (see Section 3.8) for each method in the selected libraries that is used at least once by others. As seen in Fig. 13, the dependent ratio follows a logarithmic distribution. The *x*-axis of this plot shows the public methods of libraries. Since libraries have a different number of methods we normalize this axis by using the Quintiles when sorted by the value of the *y*-axis. The *y*-axis is the ratio of dependents calling the method. Each dot in this plot shows the mean of all selected *versioned packages* in that particular *x* value. The orange line shows the trend that the mean of all selected *versioned packages* follow. This line is calculated by a regression model that we fit on the aforementioned dots. Hence, this figure shows that on the current state of Maven on average there are only a limited number of methods in each library that are widely used.

The statement regarding the Pareto Principle also relates to the conclusions made by Harrand et al. (2022), which notes that most clients depend on a small fraction of an API. From our research, we can come to a similar conclusion; popularity skews towards the most used methods, but comparatively, more methods exist with low popularity values. More specifically, the area under the orange curve in Fig. 13 is 0.53. A coordinate with x = 1.76 and y = 0.1 is a point where the left area and right area under the curve are almost equal. This means that on average 35% of the methods in Maven libraries are receiving 50% of all calls from dependents. The remaining majority (65% of the public methods) receive the other half of the unique dependent calls. Note that these are only about the 13% of the methods that are at least called once by another library.

The majority of public methods that libraries define are not used by others. On average, 87% of publicly defined methods are never used by other libraries, and 35% of the remaining 13%, cover half of the dependent calls.

6. RQ3: Is there a relation between popularity and breaking changes?

In the next step, we want to investigate the extent of the problem from the users' point of view by connecting popularity information and breaking changes. Deletion of an unpopular method does not nearly have as many consequences as the deletion of a popular method. We aim to understand the relationship between popularity and breaking changes, this would reveal if the maintainers of the libraries are already aware of their users and try not to break them.

We investigate this research question in two levels similar to previous questions, i.e., library and method levels. For the library-level investigation, we first get the library popularity scores for the target *versioned packages* from *Popularity Analyzer* as described in Section 3.8. We want to inspect if there is any relation between these popularity scores and semantic versioning violations. So we also retrieve the violations of the target *versioned packages* from *SemVer Analyzer* as described in Section 3.6. We then calculate the ratio of the number of packages with violations to their corresponding popularity scores. By fitting a second-order regression model on the popularity scores and their corresponding violation ratio, we inspect whether or not a trend exists. Next, we use the *Popularity Analyzer* to obtain method-level popularity scores for target *versioned packages*. We investigate the difference

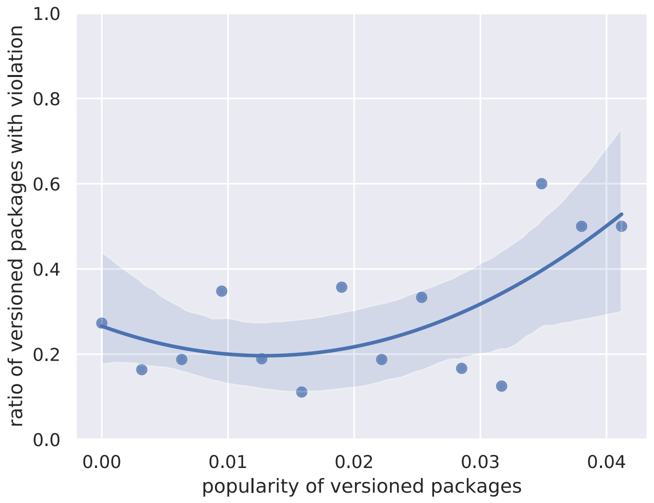


Fig. 14. Semantic versioning violations and popularity of the *versioned packages*.

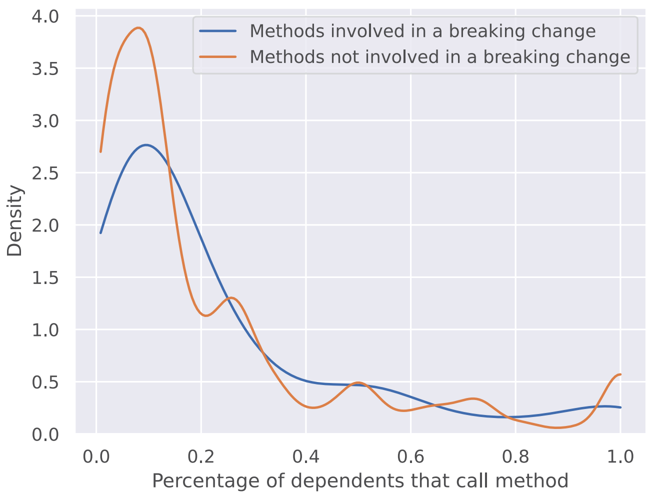


Fig. 15. Distribution of percentage of dependents that call methods. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

between the population curve of these scores for methods that are involved in breaking changes and those that are not.

To explore the relationship between popularity and breaking changes we first analyze the popularity of the libraries. As described in Section 3.8 we measure what portion of the dependents are attracted to each target *versioned package*. Then we connect this information to the breaking changes to understand whether or not the popularity of libraries has any effects on making breaking changes. As shown in Fig. 14 there exists a slight tendency for more popular *versioned packages* to introduce more breaking changes. This can be due to the higher number of change requests that popular libraries receive from their users. Note that the overall number of unique dependents is 7190. Therefore, when we divide the number of dependents of each *versioned package* by such a big number, the maximum popularity value lies around 0.04.

To look closer at the popularity of breaking changes we inspect the relation between method popularity and breaking changes. Fig. 15 shows the population of the popularity of methods that are not involved in breaking changes compared to the popularity of methods involved in breaking changes. The popularity is represented by the percentage of dependents that call the method. For example, when the x-axis is equal to one, the method is called by every dependent of the *versioned package*. The orange line in the figure shows the density of the population of the methods that are defined in our target *versioned packages* and are not identified as a breaking change. The blue curve, however, is the methods of the same *versioned packages* that are involved in the breaking changes. Methods with zero popularity are filtered from this figure since they make the rest of the figure invisible due to their high number. More specifically, 61k methods that were not used by any other library were filtered from the orange curve and 2.2k were eliminated from the blue curve.

In Fig. 15, we can compare the difference between two populations by investigating the difference in the density at a certain point of the x-axis. If the curve of breaking changes has a higher density than the curve without breaking changes on a certain point x, this means that on average, a method with a breaking change more often has popularity x than a method without a breaking change. However, in this figure, it can be noted that both lines follow a very similar distribution. Whether or not the method is involved in a breaking change does not seem to be related to its popularity. To prove this, we performed a *T-test* on both populations and observed that there is no significant difference between them. The results of this test show a $p - value = 0.48$ for a *two-sided* independent samples, showing that these populations do not have a significant difference.

One observation that we can also make from this figure is that the orange curve contains several spikes. Our manual inspection shows that the popularity values of methods can be concentrated on the same numbers. This means many methods are assigned with similar popularity values. Since *versioned packages* have a limited number of dependents and there are not many permutation options for method usage, these numbers can be similar. For example, assume a *versioned package v* has 10 dependents and 20 methods. If 4 of these methods are related to one functionality and are usually used together, when five of the dependents only use this functionality, the popularity value will be 0.5 for all four methods. The popularity of the remaining 16 methods can also be 0.5 if they are used by five of the dependents. Hence these spikes can be observed when creating the population curve for all data points. However, despite these spikes, the overall distributions are similar as validated by the *T-test*.

The popularity of a method does not play a role in whether or not it is involved in breaking changes, no significant difference exists between the popularity of the methods that break the semantic versioning and the ones that do not.

7. Discussion

We found that a large number of *versioned packages* contain breaking changes. Users run into issues when they unsuspectingly attempt to upgrade their dependencies, as the API is no longer compatible with their artifact. This happens even though the versioning convention of the library promises compatibility. Furthermore, illegal API extensions do not pose a threat to the compatibility of users. Therefore, it might seem intuitive that maintainers can be less concerned with these illegal API extensions. However, the maintainers that extend their API illegally are not only employing bad practices but also increasing the chances of having more breaking changes and violations in the future. This is because having more methods in a library increases the chance of violations as we showed in Section 4. By doing this study, we found that maintainers on one hand need to pay more attention to semantic versioning and any type of library API expansion as a preventive measure to decrease the chance of semantic versioning violations. Researchers on the other hand can continue our work by investigating what type of tooling, design decisions, and development methodologies lead to more stable APIs.

We did not find a meaningful relationship between the popularity of a method and whether it is involved in a breaking change or not. However, there might be some factors at stake. For example, on one hand, more popular methods may get involved in more issues by the users. This can potentially help maintainers understand which methods are more popular. On the other hand, attracting more attention to the issues naturally increases the chance of changes in the methods and consequently more involvement in breaking changes. Similarly, in the case of unpopular methods, maintainers may realize that not as many people open issues related to a particular method. Hence they are not very popular. Maintainers may change such methods more easily since they assume they would not affect many people. We speculate that there are many similar factors affecting the behaviors of maintainers in case of breaking changes that are worth understanding for the software community. Hence, we encourage researchers to conduct more research to understand such aspects. Our results emphasize the need for such studies by showing that breaking changes are still a big problem for members of the ecosystems. However, in this study, we only focused on finding the breaking change usages in the ecosystem. It is also important to understand the collective cost that breaking changes impose on the ecosystem members in terms of time and money since the cost may differ from one dependent project to another. We encourage researchers to reuse some parts of our approach as a starting point to study these collective costs.

We speculate that library maintainers do not have sufficient data to accurately differentiate between popular and unpopular methods. They can use this information before introducing breaking changes and during their maintenance. An established popularity ranking of the usage of their own methods across the whole ecosystem would allow library maintainers to improve their workflow. Utilizing such a popularity ranking, one could prioritize a list of issues to work on based on their importance within the ecosystem; a change to a popular method will have more impact compared to an unpopular one. During this study, we realized that there are numerous opportunities for client-side and platform-based tools. IDEs can recommend maintainers to adjust their version appropriately whenever a change in method signatures is detected. Build tools can offer warnings when finding an inappropriate version upgrade. GitHub can warn users when they break a popular method in their *pull requests*. Maven Central can warn developers about incompatibilities, impose strict requirements for versioning of artifacts, or give a high score to projects without any semantic versioning violations. To create a safer and more trustworthy environment within the ecosystem researchers and engineers can focus on building such tools in the future as we indicated the need in this study.

Method popularity also introduces opportunities for the users of the libraries. Method popularity might be used as a recommender system that supports developers during the coding activity. Currently, library popularity in Maven helps users determine which library to use. However, users can also pay attention to the popularity of the methods that they need or compare the popularity of methods with similar functionalities. Another use case that is worth further research is integrating method popularity ranking within auto-complete engines to improve their recommendations.

In this study, we only focused on public methods as the main mean of library reuse. While it is possible for the developers to use package private and protected library methods, such methods may not as often be intended for API usage. Maintainers intentionally expose the public methods for the reuse of their library. Hence, we limit our scope to the intended and most common modifier for library reuse. We also limited the scope of violations to the defined methods since the CGs only add edges to such methods. We ran our popularity analysis experiment on all methods and we observed minimal difference in the overall results. We believe a more detailed investigation in this regard is out of this study's scope. Future research is needed to investigate this in detail. We realized that despite the existing studies there is room for further research. For example what parts of code are more likely to break or whether public methods break more often than protected methods. Answering these questions benefits the community to understand what parts of the code they should pay more attention to.

7.1. Threats to validity

In this study, we limit our dependent resolution to a recent six months while multiple decades of evolution are available on Maven Central. We believe our dataset is already extensive and that a larger study would substantially increase the cost of execution, while there is no reasonable intuition to expect that the insights would differ for a bigger period. A more extended period surely increases the number of *versioned packages* and their dependents. However, we do not expect the overall usage pattern of *versioned packages* to change.

Generating a perfect static CG is an undecidable problem. Hence existing solutions use over-approximation and sacrifice precision for scalability. Our illegal API extension extraction also uses approximation. An AST-level solution is needed for better precision which was not practical for our scale. Although these design decisions may impact our precision, we opted to prioritize scalability, which enabled us to offer a more comprehensive perspective of the ecosystem. Nevertheless, our findings entirely align with previous research which shows the effects are minor.

Maven Central does not only contain Java artifacts; it also contains artifacts that are implemented in other JVM-based languages. In some cases, we had difficulty generating CGs for such *versioned packages*, and thus the artifacts were not included in the analysis. However, we found that no more than 1% of analyzed artifacts were Scala or Kotlin packages. Thus, we believe it is safe to only focus on Java projects in this study.

During this study, we developed several programs and used open-source libraries which may contain implementation errors. One can never ensure all implementations are bug-free. However, we tried to mitigate this by performing manual tests and code reviews as well as releasing our code and executable to the public.[2]

8. Summary

In this study, we showed that a large number of Maven libraries do not completely adhere to semantic versioning. We know this, as 63% of analyzed artifacts break compatibility within their major versions. Illegal API extensions also occur in 54% of artifacts. Therefore, these violations form a big problem for the trustworthiness of semantic versioning on Maven. Moreover,

2 https://github.com/ashkboos/semver-vs-popularity.

deletion or alteration of an unpopular method does not have the same impact as changing a popular method. We investigated whether a relationship between method popularity and involvement in breaking changes exists. We discovered that breaking changes occur in popular methods as frequently as in unpopular ones. By analyzing all interactions between software artifacts and their dependents we found that the majority (87%) of publicly defined methods are not used by others and 35% of the remaining are responsible for half of the dependent calls. Similarly, the number of dependents per library also follows a power law distribution.

CRediT authorship contribution statement

Mehdi Keshani: Conceptualization, Methodology, Software, Writing. **Simcha Vos:** Software, Visualization, Investigation. **Sebastian Proksch:** Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation program under grant number 825328 (FASTEN project). We wish to thank Thijs Nulle for his work on the research project related to this paper. We would also like to thank Dr. Maliheh Izadi for her invaluable feedback on this research.

References

Alrubaye, Hussein, Mkaouer, Mohamed Wiem, 2018. Automating the detection of third-party Java library migration at the function level. In: CASCON.

artifact, 2022. ShrinkWrap resolvers. https://github.com/shrinkwrap/resolver. Accessed: 2022-01-15.

Bavota, Gabriele, Canfora, Gerardo, Di Penta, Massimiliano, Oliveto, Rocco, Panichella, Sebastiano, 2015. How the apache community upgrades dependencies: an evolutionary study. Empir. Softw. Eng..

Benelallam, Amine, Harrand, Nicolas, Soto-Valero, César, Baudry, Benoit, Barais, Olivier, 2019. The Maven dependency graph: a temporal graph-based representation of Maven central. In: IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).

Dean, Jeffrey, Grove, David, Chambers, Craig, 1995. Optimization of object-oriented programs using static class hierarchy analysis. In: European Conference on Object-Oriented Programming.

Decan, Alexandre, Mens, Tom, Grosjean, Philippe, 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir. Softw. Eng..

Dunford, Rosie, Su, Quanrong, Tamang, Ekraj, 2014. The pareto principle. The Plymouth Student Scientist.

Harrand, Nicolas, Benelallam, Amine, Soto-Valero, César, Bettega, François, Barais, Olivier, Baudry, Benoit, 2022. API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages. J. Syst. Softw..

Hejderup, Joseph, van Deursen, Arie, Gousios, Georgios, 2018. Software ecosystem call graph for dependency management. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results.

Helm, Dominik, Kübler, Florian, Reif, Michael, Eichberg, Michael, Mezini, Mira, 2020. Modular collaborative program analysis in OPAL. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

Hora, André, Etien, Anne, Anquetil, Nicolas, Ducasse, Stéphane, Valente, Marco Tulio, 2014. Apievolutionminer: Keeping api evolution under control. In: Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE).

Hora, André, Robbes, Romain, Valente, Marco Tulio, Anquetil, Nicolas, Etien, Anne, Ducasse, Stéphane, 2018. How do developers react to API evolution? A large-scale empirical study. Softw. Qual. J..

Kim, Miryung, Cai, Dongxiang, Kim, Sunghun, 2011. An empirical investigation into the role of API-level refactorings during software evolution. In: Proceedings of the 33rd International Conference on Software Engineering.

Koçi, Rediana, Franch, Xavier, Jovanovic, Petar, Abelló, Alberto, 2019. Classification of changes in API evolution. In: IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC).

Kula, Raula, German, Daniel, Ishio, Takashi, Inoue, Katsuro, 2015. Trusting a library: A study of the latency to adopt the latest Maven release. In: IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER).

Kula, Raula, German, Daniel, Ishio, Takashi, Ouni, Ali, Inoue, Katsuro, 2017a. An exploratory study on library aging by monitoring client usage in a software ecosystem. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER).

Kula, Raula, German, Daniel, Ouni, Ali, Ishio, Takashi, Inoue, Katsuro, 2018. Do developers update their library dependencies? Empir. Softw. Eng..

Kula, Raula Gaikovina, Roover, Coen De, Germán, Daniel M., Ishio, Takashi, Inoue, Katsuro, 2017b. Modeling library popularity within a software ecosystem. Tech. Rep..

Lamothe, Maxime, Guéhéneuc, Yann-Gaël, Shang, Weiyi, 2021. A systematic review of API evolution literature. ACM Comput. Surv..

Lima, Caroline, Hora, Andre, 2020. What are the characteristics of popular APIs? A large-scale study on Java, Android, and 165 libraries. Softw. Qual. J..

Macho, Christian, Beyer, Stefanie, McIntosh, Shane, Pinzger, Martin, 2021. The nature of build changes: An empirical study of Maven-based build systems. Empir. Softw. Eng..

mavenIndex, 2022. Maven index. https://repo.maven.apache.org/maven2/.index/. Accessed: 2022-10-28.

Mileva, Yana Momchilova, Dallmeier, Valentin, Burger, Martin, Zeller, Andreas, 2009. Mining trends of library usage. In: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops.

Nguyen, Hoan Anh, Nguyen, Tung Thanh, Wilson, Jr., Gary, Nguyen, Anh Tuan, Kim, Miryung, Nguyen, Tien N, 2010. A graph-based approach to API usage adaptation. ACM SIGPLAN Not..

Qiu, Dong, Li, Bixin, Leung, Hareton, 2016. Understanding the API usage in Java. Inf. Softw. Technol..

Raemaekers, Steven, van Deursen, Arie, Visser, Joost, 2012. Measuring software library stability through historical version analysis. In: 28th IEEE International Conference on Software Maintenance (ICSM).

Raemaekers, Steven, van Deursen, Arie, Visser, Joost, 2013. The Maven repository dataset of metrics, changes, and dependencies. In: 10th Working Conference on Mining Software Repositories (MSR).

Raemaekers, S., van Deursen, A., Visser, J., 2017. Semantic versioning and impact of breaking changes in the Maven repository. J. Syst. Softw..

Soto-Valero, César, Benelallam, Amine, Harrand, Nicolas, Barais, Olivier, Baudry, Benoit, 2019. The emergence of software diversity in Maven central. In: IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).

Soto-Valero, César, Durieux, Thomas, Baudry, Benoit, 2021. A longitudinal analysis of bloated Java dependencies. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

Soto-Valero, César, Harrand, Nicolas, Monperrus, Martin, Baudry, Benoit, 2020. A comprehensive study of bloated dependencies in the Maven ecosystem. Empir. Softw. Eng..

VersionRanges, 2022. Maven version ranges. https://maven.apache.org/enforcer/enforcer-rules/versionRanges.html. Accessed: 2022-10-21.

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

Xavier, Laerte, Brito, Aline, Hora, Andre, Valente, Marco Tulio, 2017. Historical and impact analysis of API breaking changes: A large-scale study. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER).

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

Mehdi Keshani is currently a Ph.D. candidate in the Software Engineering Research Group, at Delft University of Technology, Netherlands. His research interests include program analysis and software ecosystems. His current project revolves around dependency management systems and how to make them more intelligent. He is also a key member of a European project called FASTEN. FASTEN aims at bringing method-level analyses such as call-graph-based impact analyses to the hands of ecosystem developers.

Simcha Vos is a graduate student at the Technical University of Delft, Netherlands. He is studying Software Engineering. In 2022, he successfully graduated from his bachelor's studies form Technical University of Delft. The focus of his undergraduate research was on semantic versioning on Maven Central.

Sebastian Proksch is a researcher interested in demystifying the software development process, both from the perspective of individuals and of teams. He studies the impact of novel technologies and envisions new tools to support software engineers in their day to day tasks. His most recent works in the CI/CD area have created tools that facilitate the adoption of CI/CD and help developers to spot anti-patterns in their build pipelines. Sebastian Proksch is Assistant Professor in the Software Engineering Research Group at Delft University of Technology (Netherlands) and the scientific coordinator of the FASTEN project.