Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing

Lyuye Zhang
School of Computer Science and Engineering, Nanyang Technological University
Singapore, Singapore
zh0004ye@e.ntu.edu.sg

Chengwei Liu
School of Computer Science and Engineering, Nanyang Technological University
Singapore, Singapore
chengwei001@e.ntu.edu.sg

Zhengzi Xu*
School of Computer Science and Engineering, Nanyang Technological University
Singapore, Singapore
zhengzi.xu@ntu.edu.sg

Sen Chen
College of Intelligence and Computing, Tianjin University
Tianjin, China
senchen@tju.edu.cn

Lingling Fan
College of Cyber Science, Nankai University
Tianjin, China
linglingfan@nankai.edu.cn

Bihuan Chen
School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University
Shanghai, China
bhchen@fudan.edu.cn

Yang Liu
School of Computer Science and Engineering, Nanyang Technological University
Singapore, Singapore
yangliu@ntu.edu.sg

ABSTRACT

To enhance the compatibility in the version control of Java Third-party Libraries (TPLs), Maven adopts Semantic Versioning (SemVer) to standardize the underlying meaning of versions, but users could still confront abnormal execution and crash after upgrades even if compilation and linkage succeed. It is caused by semantic breaking (SemB) issues, such that APIs directly used by users have identical signatures but inconsistent semantics across upgrades. To strengthen compliance with SemVer rules, developers and users should be alerted of such issues. Unfortunately, it is challenging to detect them statically, because semantic changes in the internal methods of APIs are difficult to capture. Dynamic testing can confirmingly uncover some, but it is limited by inadequate coverage.

To detect SemB issues over compatible upgrades (*Patch* and *Minor*) by SemVer rules, we conduct an empirical study on 180 SemB issues to understand the root causes, inspired by which, we propose Sembid (Semantic Breaking Issue Detector) to statically detect such issues of TPLs for developers and users. Since APIs are directly used by users, Sembid detects and reports SemB issues based on APIs. For a pair of APIs, Sembid walks through the call chains originating from the API to locate breaking changes by measuring semantic diff. Then, Sembid checks if the breaking changes can affect API's output along call chains. The evaluation showed Sembid achieved 90.26% recall and 81.29% precision and outperformed other API checkers on SemB API detection. We also revealed Sembid detected over 3 times more SemB APIs with better coverage than unit tests, the commonly used solution. Furthermore, we carried out an empirical study on 1,629,589 APIs from 546 version pairs of top Java libraries and found there were 2∼4 times more SemB APIs than those with signature-based issues. Due to various version release strategies, 33.83% of *Patch* version pairs and 64.42% of *Minor* version pairs had at least one API affected by any breaking.

ACM Reference Format:
Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2022. Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556956

1 INTRODUCTION

The frequent updates of Third-party Libraries (TPLs) prompt downstream users to upgrade their dependencies accordingly to embrace necessary new features and bug fixes [69, 70], while the potentially incompatible changes could lead to abnormal execution or even crashes of downstream projects according to [50]. To address this problem, modern package managers adopt Semantic Versioning (SemVer) [23] to control the compatible upgrading of TPLs. SemVer divides upgrades into three types, namely, *Major*, *Minor*, and *Patch* and requires changes in *Minor* and *Patch* upgrades to be backward

*Zhengzi Xu is the corresponding author.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556956

compatible. Unfortunately, it is unlikely to guarantee all versions strictly satisfy SemVer rules, especially in legacy platforms like Maven [19] for Java. For example, *Hadoop-hdfs* [9], a widely-used framework of Apache was susceptible to a breaking issue [10] during *Patch* upgrade (3.0.0-3.0.1). Not limited to popular Java projects like *Hadoop*, the breaking issues commonly exist in the Maven ecosystem. As revealed by Ochoa et al. [52], 20.1% of non-*Major* upgrades in the Maven repository [20] have introduced breaking changes that could induce massive unexpected downstream issues.

In recent years, works have been proposed to detect the breaking compatibility issues at the API level of TPLs. Since APIs serve as the entry points for downstream projects to access TPLs, an upgrade is usually considered compatible if all APIs are compatible. Many API compatibility checkers [1, 3, 6, 7, 11, 22] are proposed to uncover signature-based errors, such as *ClassNotFoundException* and *NoSuchMethodError*. We call these errors *Syntactic Breaking* (SynB) in this paper. However, these checkers fail to detect the breaking issues caused by the internal behavioral changes, which are often exposed at run-time, such as abnormal output, unexpected exceptions, and even crashes [37, 59]. That is because internal changes often dwell in the bodies of indirectly called methods that are not considered by these checkers. We call the internal behavioral changes without syntactic errors *Semantic Breaking* (SemB).

Unlike SynB, SemB is much more difficult to be detected. Existing works mostly rely on manually curated dynamic regression tests to detect them. For example, Mostafa et al. [50] found only 13 of 126 real-world SemB APIs could be detected by existing unit tests. Chen et al. [32] have proposed DeBBI which accelerates unit tests from downstream projects to facilitate the detection, but it still relies on testing which is handicapped by the coverage of limited test cases. Therefore, detecting SemB issues properly is challenging, without which, SemVer rules cannot be completely enforced. Then, unexpected breaking upgrades would sabotage the community.

To bridge this gap, we seek to detect SemB statically for better coverage of potential issues. We are facing the following **challenges**. **C1**: It is unclear how to statically infer dynamic behavioral changes by source code diff, which has hardly been studied. **C2**: Syntactic changes (including inter-method changes), such as the refactoring, which hardly changes the semantics, are difficult to be excluded. **C3**: Some behavioral changes, such as bug fixing, are considered compatible, and thus allowed by SemVer. They are supposed to be ruled out, but there is no specific criterion to identify them. **C4**: A considerable amount of internal changes implicitly optimize the logic or performance, which hardly changes the output of APIs, thus not observable from the user side. Without the impact analysis of these changes, naive detection may lead to false positives.

To this end, we propose, to the best of our knowledge, Sembid (Semantic Breaking Issue Detector), the first static tool to detect potential SemB over *Patch* and *Minor* upgrades against SemVer rules based on APIs. First, to address the challenge **C1**, we conducted a study on causes of SemB which indicated inconsistent behaviors were reflected by the diff of dependency relationships derived from static slicing [63]. Then, to address syntactic changes in **C2**, Sembid extracts abstract semantic information free from syntactic changes. It starts with Intermediate Representation (IR) and traverses the methods within the call chains of a given API pair to form clusters of changed methods to neutralize inter-method changes. Given a cluster, Sembid recursively backward slices the global output inter-procedurally to derive semantic relationships between global input and output, which eliminate local syntactic changes. Sembid heuristically constructs semantic graphs based on sliced statements and measures their semantic diff based on subgraph isomorphism to infer the potential SemB. For **C3**, during the semantic diff measurement, benign changes are identified and excluded by pre-summarised patterns to avoid false alarms. Last, for **C4**, along call chains from the cluster, Sembid verifies if the captured SemB can influence the API's output by checking its triggerability as well as whether it can propagate back to the API.

To evaluate the accuracy of SemB detection of Sembid, we first experimented for the benchmark with other API checkers. Due to the lack of a benchmark dataset, we manually formed one consisting of 77 version pairs with 671 APIs. Sembid outperforms other state-of-the-art Java API compatibility checkers [6–8, 11, 22] with 90.26% recall and 81.29% precision. Then, over all APIs of breaking *Patch* upgrades, we compared Sembid with unit tests in terms of the effectiveness of breaking API detection. It demonstrates that Sembid with better API coverage could detect over 4 times more SemB APIs than unit tests. Furthermore, we carried out a study on 21 top Java libraries with 546 version pairs from Maven Repository [20] to evaluate the compliance with SemVer at the levels of API, version pair, and library respectively. We found that, compared with 0.38% and 1.04% of APIs affected by SynB for *Patch* and *Minor* respectively, 1.10% and 4.06% of APIs additionally brought in SemB issues. For version pairs, due to various upgrading strategies adopted by libraries, 33.83% of *Patch* and 64.42% of *Minor* upgrades have breaking pairs. We conclude our main contributions as follows:

- We conducted a study to understand the root causes of SemB and summarized the patterns of benign semantic changes.
- We proposed the first static SemB issue detector (Sembid) to detect potential SemB issues against SemVer rules.
- We built a benchmark dataset of APIs for SemB detection to facilitate further research, which is accessible on our website [13].[1]
- We carried out a study on compliance with SemVer rules in the top 21 real-world Java libraries. We found version release strategies adopted by libraries varied greatly so that SemVer rules were not reliable for users. The prevalence of SemB proved the necessity of detecting SemB when upgrading TPLs.

2 BACKGROUND AND MOTIVATION

2.1 Semantic Versioning Rules

The Semantic Versioning [23] stipulates rules for three upgrades:

- **Patch version Z (x.y.Z)**: *"MUST be incremented if only backward-compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior."*
- **Minor version Y (x.Y.z)**: *"MUST be incremented if new, backward-compatible functionality is introduced to the public API. It MUST be incremented if any API is deprecated."*
- **Major version X (X.y.z)**: *"MUST be incremented if any backward-incompatible changes are introduced to the public API. It MAY also include minor and patch level changes."*

[1] Data set is accessible at https://sites.google.com/view/ase22semverdetection/homepage

Listing 1: A Motivating Example from httpcore:4.2-4.3

```
1
2  -if (this.contentDecoder != null &&
3  -   (this.session.getEventMask()&SelectionKey.OP_READ)>0) {
4  +if (this.contentDecoder != null) {
5  +   while ((this.session.getEventMask()&SelectionKey.OP_READ)>0) {
6          handler.inputReady(this, this.contentDecoder);
7          if (this.contentDecoder.isCompleted()){
8              resetInput();
9  +           break;
10 +       if (!this.inbuf.hasData())
11 +           break;
12 }}}
```

The breaking changes are not allowed in *Patch* and *Minor* upgrades. Hence, to enhance compliance with SemVer rules, SEMBID aims at detecting breaking changes based on API over *Patch* and *Minor* upgrades to alert developers and users. Unlike SynB, SemB can hardly be detected by existing tools so SEMBID focuses on detecting SemB to bridge the gap.

2.2 Motivating Example

The example in Listing 1 is used to demonstrate our motivation.

The code was collected from a Jira issue [5] of *http-core* [15]. The upgrade caused the decoder to be stuck in an infinite loop when it reaches the end of the buffer if the input buffer contains a character '\r' that is never consumed. First, the change was not documented as a breaking, which suggests the breaking was unexpected. Second, the unit tests failed to uncover the case. Last, the *Minor* upgrade from version 4.2 to 4.3 intuitively indicated backward compatibility. Therefore, the judgment of the author is not always reliable so a static tool with better coverage is needed to detect potential breaking issues for *Patch* and *Minor* upgrades.

3 EMPIRICAL STUDY

In this section, we study the causes of SemB and the code patterns of benign changes.

3.1 Study of Root Causes of Semantic Breaking

To detect SemB, we conducted a study regarding its root causes.

3.1.1 ***Causes of SemB.*** Mostafa et al. [50] studied the Behavior Backward Incompatibilities in Java software, which also refers to the breaking APIs beyond signatures like SemB. They categorized the immediate causes into three: usage change (32.77%), e.g. enable/disable poor input; better output (55.74%), e.g. output format change; and other reasons(11.49%), e.g. internal structure changes.

Despite the immediate causes, Mostafa et al. [50] only focused on the user side of APIs but failed to dive into the internal code to locate the root causes. Since no other works have studied the root causes of SemB, we conducted one by analyzing the internal code changes of APIs that caused SemB with the aid of existing static java analysis tools, BCEL [12] and Soot [60]. The study included 180 real-world SemB issues (126 from [50], 54 collected by ourselves in recent 3 years) with commits. We found SemB had various forms which could hardly be summarised as patterns at the source code level, but they usually occurred along with the following changes:

- (73.33%) The changed execution logic. In Listing 1, the conditions remained the same (L2 & L4), but the *if* statement was altered to a *while* loop, which led to an infinite loop.
- (91.67%) The changed calculation of the output. For example, in a self-increment function, the change of calculation from $a+=1$ to $a+=2$ obviously modifies the output, while the execution logic remains the same.

The two types of causes overlapped in over 60% of issues. The first change in the execution logic embodied the inconsistency in Control Flow Graph (CFG). The second change in the calculation process, embodied in the changed data flow, can reflect the inconsistency of output values. For 86% of cases, the SemB changes dwelt in internal methods called by APIs instead of entry methods. Therefore, internal code analysis is required to detect SemB.

3.1.2 ***What Changes Will NOT Cause SemB?.*** Since successful upgrades have no specific indicator (no-issue is not enough), we could only study successful regression tests to understand why some changes do not cause SemB. We firstly collected 20 most used Maven libraries according to Maven Repository [20] with 77 version pairs and ran regression tests by testing the new implementation with old unit tests. 20,373 tested APIs were derived. Then, we filtered out APIs with no changes in called methods to obtain 2,191 APIs. 500 successful cases with binary change were randomly selected. For each case, we manually analyzed the diff dwelling in the methods called by the APIs to check if there existed an input to trigger the failure of unit tests. If the input did not exist, the reasons were collected. The cases of each reason may overlap with one another. The APIs passed regression tests due to:

- (27.4%) **Inadequate input to trigger SemB.** We manually examined these cases and found there existed input that could trigger SemB. SEMBID is designed to overcome such limitations of tests.
- (19.6%) **Refactoring.** By [21], refactoring is the process of restructuring the existing factoring without changing its external behaviors, which is prevalent in Java by [61]. In general, the relevant API-level refactoring can be categorized as inter-method and intra-method. Inter-method refactoring, e.g. Extract Method, and Inline Method, is the most common type because it is frequently used to extend API flexibility based on Java polymorphism [25].
- (2.4%) **SemB not triggerable by old input.** The inconsistent behaviors cannot be triggered by the old variables, but only by the variables in the new version. This situation mostly occurs for new functionality, because the new input serves as the option to trigger additional behaviors.
- (14.6%) **Internal SemB has no impact on API output** The changed output of breaking changes has no impact on the output of the API to be observed by users. For example, if the logs are changed, while the return value remains the same, the output of the API is considered unchanged.
- (36.0%) **Benign changes** As allowed by SemVer, benign changes, such as bug fixes, and new functionality, do not substantially break the original semantics. They usually would not break downstream projects, and thus are considered false positives.

We found that except for the first reason (limitation of unit tests), the rest is the false alarm of SemB to be ruled out. For the refactoring, SEMBID extracts inter-procedural semantic graphs to eliminate syntactic refactoring. For the third and fourth reasons, SEMBID excludes them by checking the impact of the captured SemB

changes. As for the benign changes, we further conduct a study to identify them by patterns.

3.2 Study of Benign Changes

SEMBID aims at identifying the benign changes to avoid reporting them as breaking. Since the benign changes stipulated by SemVer are not well defined, we summarise the syntactic patterns of benign changes by manually studying commits with such intentions. Then, these patterns will be categorized by their semantic representations on PDG and CFG to be automatically identified. According to [45], the intentions of commits can be categorized into (1) Bug fixing; (2) Performance improvement; (3) Feature introduction, deletion, and modification. Since the performance improvement does not explicitly change the output, it would usually not be caught by SEMBID, thus unnecessary to identify it. As feature deletion and modification are not allowed by SemVer, they should be directly reported instead of being ruled out. Hence, we focused on bug fixes and new functionality. We collected 584 commits from the most starred 25 Java projects on Github to summarise the patterns by searching for the keywords in commits, which are *fix*, *correct*, *improve*, *address*, *tweak*, *clean up*, and *add/new feature/functionality*. Then, the diffs were manually categorized into several patterns.

3.2.1 *Bug Fixes (*393 *cases) Categories*.

- **Additional/ conditions and branches.** (41.85%) The change introduces additional conditions to narrow down or broaden the input range. The conditions mostly introduce new branches of statements for additional handling. This is usually to enforce the original rules by disallowing illegitimate input.
- **Changed/deleted conditions and branches.** (13.28%) The original conditions are changed or deleted to fix unreasonable behaviors. Usually, the change handles the illegitimate input to enforce the original rules, but it sometime introduces regression errors.
- **Similar substitution.** (11.60%) Variable types or methods are substituted with similar ones with semantically equivalent method names for better implementation. The original functionality is meant to be maintained.
- **Additional *try/catches*.** (10.08%) It introduces additional *try/-catch* pairs or adds *catches* to existing *try*. This kind of change handles more exceptions to avoid unexpected crashes.
- **Assignment revision.** (10.76%) The output assignments are changed partially or completely to correct wrong behaviors or improve sub-optimal behaviors. Such changes can also be found in breaking issues because they could break downstream projects unexpectedly if they are used incorrectly.
- **Auxiliary variables.** (6.68%) Auxiliary variables are introduced to control the process, which often participates in *if* conditions. For instance, a counter is used to avoid infinite loops.
- **Other.** (5.75%) It consists of *changing internal type, initializing variables, improving method/variable modifiers*, etc. They slightly change syntax without modifying the original semantics.

3.2.2 *New Functionality (*191 *cases) Categories*.

- **New classes/methods.** (61.29%) New classes/methods are introduced as the entries for new implementations which can be called by the existing APIs.
- **Additional branches.** (22.58%) New implementations/handlers are optionally accessible via new *if* or *switch* branches.
- **Additional parameters.** (9.68%) New parameters are added to existing internal methods to control or optimize existing functionalities by providing more options.
- **Additional fields.** (6.45%) New fields of returned objects are added with corresponding data and control dependencies. They are used to provide additional information loaded in the fields.

The new functionality cases were only considered *new* if the original functionalities were intact. Thus, the New Functionality cases were mostly additive changes. These patterns will be summarized to heuristics based on PDG and CFG. The commit IDs are provided on our website [13].

4 METHODOLOGY

According to Gunter et al. [38], since the change of semantics can be interpreted to infer the change of output which usually leads to the breaking issues, SEMBID is designed to infer the SemB by measuring semantic diff.

SEMBID aims at detecting SemB issues over *Patch* and *Minor* upgrades based on APIs. As shown in Figure 1, SEMBID consists of five major steps, (1) **Group clusters from call graphs**: Given the API pairs, SEMBID constructs the call graphs of them, from which it groups the consecutive changed methods as clusters. (2) **Derive Dependencies Summaries**: SEMBID backward slices the output of clusters to obtain inter-procedural dependency summaries of data/-control/exception. (3) **Match patterns for benign changes**: we heuristically summarized the patterns from Section 3.2 for SEMBID to identify the statements of benign changes. (4) **Measure semantic diff**: Inter-procedural semantic graph pairs are constructed from dependencies summaries. Semantic diff is measured by topological similarity based on subgraph isomorphism by Weisfeiler-Lehman (WL) algorithm. If changes in semantics are too large, the cluster is considered to be affected by SemB. (5) **Check the impact of SemB**: SEMBID checks if the SemB of each cluster is triggerable as well as if the breaking change can propagate back to API's output.

4.1 Grouping Clusters from Call Graphs

Given the byte code, SEMBID first identifies the APIs that are worth checking by narrowing down the API candidates to those with identical signatures only. Then, it constructs the call graphs for the APIs to derive the subsequent method calls for further analysis.

4.1.1 *Generating Call Graphs for Candidate APIs*.

SEMBID starts with identifying the set of API pairs that have no SynB. Specifically, $A_{cand} = \{\langle api_{old}, api_{new}\rangle \mid Sig_{api_{old}} = Sig_{api_{new}}\}$ is the API pair candidates. Note that the return type of Sig_{api} has to remain consistent, except it is a widening cast. For example, if the return type of the old API is changed from *long* to *int* in the new API, the compilation would succeed after the upgrade. To obtain A_{cand}, Soot [60] was leveraged to collect API sets A_{old} and A_{new} from class files of the old and the new libraries respectively.

Based on A_{cand}, SEMBID generates call graphs of each API with Soot by the Spark algorithm [24]. For method bodies, Soot transforms them into Jimple [4], a typed IR. The size of the call graph can grow exponentially based on the depth, which results in the inefficiency of the analysis of methods at deep levels. According to

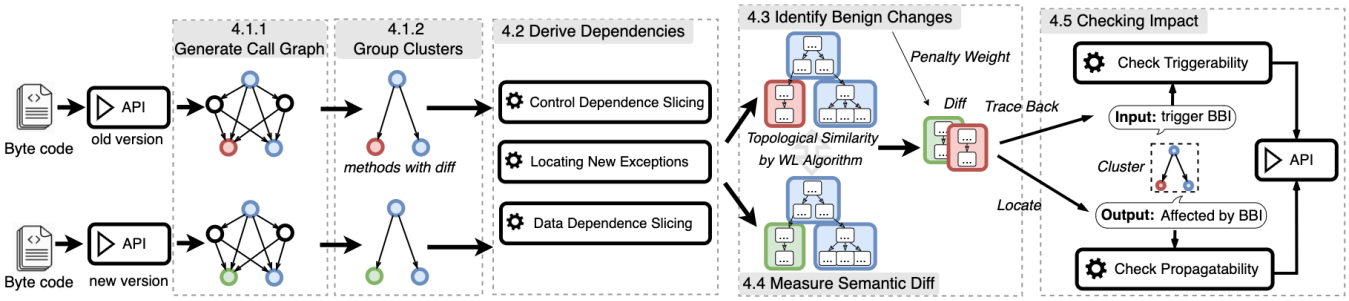

Fig. 1. Overview of SEMBID

[57], the semantics of a method decays along the calling chain to be negligible at the depth of around 10 stacks. Thus, conservatively, we set the depth of the call graph $x \in [1, 15]$ to boost the performance.

4.1.2 ***Grouping Clusters.*** As discussed in Section 3.1.2, clusters are constructed by grouping methods that have signature or body changes to mitigate the inter-method refactoring. Given a pair of call graphs CG_{old} and CG_{new}, in terms of the method's signature and body code, methods from both call graphs can be classified as changed methods $M_{changed}$ and unchanged methods $M_{unchanged}$. According to [61], inter-method refactoring involves methods that are directly connected in a call graph. For instance, "Extract method" creates a new method to replace the removed block. Thus, SEMBID groups as many directly connected (consecutive) methods as possible of $M_{changed}$ as clusters. The clusters are analyzed and sliced altogether as a whole to neutralize the inter-method refactoring.

4.2 Deriving Dependencies Summaries

For a cluster pair $\langle c, c' \rangle$, SEMBID relies on three kinds of inter-procedural semantic summaries to model the relationship between the input and output, which are Data Dependency Summaries (DDS), Control Dependency Summaries (CDS), and Exception Summaries (ES). Each dependency in the summary is embodied as a Static Single Assignment (SSA) statement from the IR. These summaries are used to capture the factors that potentially change the output regarding the data calculation, control logic, and exception handling. Listing 1 will be used as a running example.

The DDS and CDS of the output of clusters are extracted based on backward slicing of the output in the Program Dependence Graph (PDG) recursively. We first define cluster $c = \{m_{root}, m_j \mid j = 0, ..., n\}$ where m is the method, and n is the number of methods, excluding the root. The ES is a set summarised from the unhandled exception exits of all methods within a cluster. Next, we define the output of a cluster as the non-local variables that are written after the execution of the cluster. The output can be in three forms: (1) Variables returned by m_{root}; (2) Class fields written in case m_{root} returns void; (3) Exceptions thrown, as exception variable is a special non-local variable to be handled out of c. In general, the output is considered as the impact that the c imposes on the global environment. The semantic dependency summaries are robust against syntactic intra-method refactoring, such as moving, renaming, pushing down/pulling up, and splitting/merging local variables, because SEMBID directly models the relationship among non-local variables without the interference of local variables.

4.2.1 ***Data Dependencies Summary.*** DDS is used to model the relationship between the input and output as an aspect of data calculation. Based on PDG, output statements are backward sliced to derive data dependence statements. If any non-local variable, such as parameter, is met, SEMBID associates all statements met before with the non-local variable to check the triggerability later. Since the operands in SSA are prone to renaming, the operands are normalized as *var*, *parameter*, and *field* based on their roles.

To support inter-procedural analysis, when SEMBID meets a called method m_0 within the cluster during the slicing, it dives into m_0. In m_0, the operations saving is executed with the same pattern as the root method m_{root}. The only exception is that parameters of m_0 are mapped to the local variables in m_{root} instead of non-local variables. If SEMBID meets a method not included in the cluster, which are either unchanged methods or methods from other libraries, such as Java built-in classes, SEMBID does not dive into it, as analysis of them does not make difference on SemB detection.

4.2.2 ***Control Dependency Summary.*** CDS describes the controlling semantics of the cluster, which determines the execution branching. Towards a pair $\langle c, c' \rangle$, CDS is derived by backward slicing the output of m_{root} with Control Dependency edges recursively. Boolean conditions, e.g. *if*, directly and indirectly, lead to the output will be collected. The inter-procedural analysis is conducted in the same manner as DDS. To normalize the conditions against syntactic changes, e.g. refactoring, SEMBID calculates the Disjunctive Boolean Summary (DBS) which is a joint logic symbol of the original logic and the reversed one. For instance, $a > 0$'s DBS is $\langle > 0 \mid <= 0 \rangle$. Moreover, the DDS of variables used in the conditions is extracted to capture the possible changes in variable values. These Data dependency statements are associated with the conditions.

Recall the motivating example, the DDS and CDS are extracted based on the output (L7 *handler*) slicing in Figure 2. These summaries on the left are organized based on the control flows from CFG. Since only L11-L12 introduces new conditions, the CDS changes, but DDS does not change. The switch from *if* (L2) to *while* (L5) changes control flow without changing CDS or DDS. Hence, It is necessary to include CFG in the analysis to capture all possible semantic changes.

4.2.3 ***Exception Summary.*** According to the study from [50], unexpected exception throwing is a common SemB issue, which

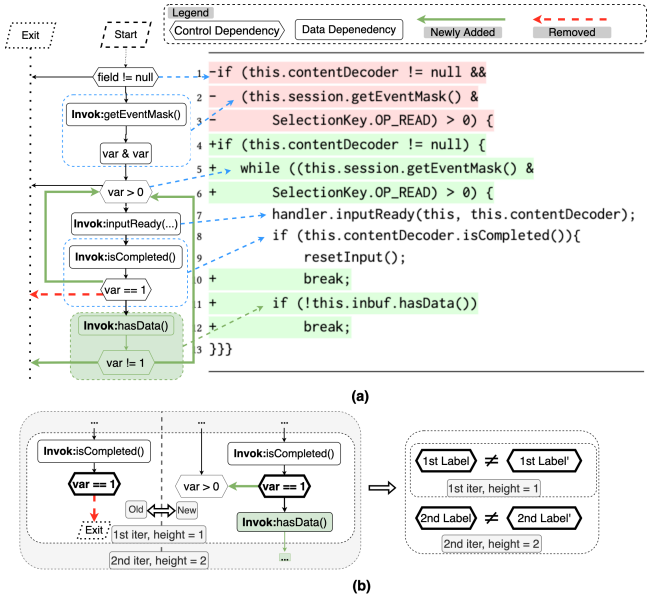

Fig. 2. (a) Data & Control Dependencies Summaries in the semantic graph of Motivating Example (b) Example of subgraph comparison of Node *var==1* at height 1 and 2

is caused by un-handled new exceptions. Thus, it is necessary to check if the exceptions are consistent during the upgrade for the cluster. Because exceptions have no data dependencies, but only control dependencies, the exceptions extracted from the cluster are associated with the corresponding conditions from the CDS.

4.3 Matching Patterns for Benign Changes

In order to match benign changes in PDG and CFG. We first summarize semantic patterns from the syntactic source code patterns from Section 3.2. Given a cluster c, statements of benign changes are identified and collected to a set S^c_{ben}. According to Campos et al. [31] and Pan et al. [53], there were 9 categories of bug fixes patterns in Java. We further generalize them into 5, namely, from major to minor, controlling conditions (CC), field, method call, variable assignment, and try/catch. Also, we found the patterns of new functionality rely on the first 3 objects. Since these categories that can be identified from CFG/PDG based on IR could cover most cases of benign changes, heuristics to identify the overall benign changes were designed based on these categories.

- **CC Adjustment:** The benign changes adjust the conditions for legitimate use. If the subsequent implementations along the branch are not changed, the change is considered benign. **Steps**: (1) Get changed conditions from two lists of old and new conditions of CDS from Section 4.2.2. (2) Locate subsequent blocks of changed conditions by CFG and compare the DDS statements in those blocks to check if changed conditions lead to the same implementation. (4) If so, add the changed conditions as well as the backward sliced data dependency statements of them to S_{ben}, because they all contribute to the adjustment.
- **CC and *Try/Catch* Extra Handling:** Because the old implementation is not sufficient to cover all legitimate input, developers may add extra handling to the original to expand the input domain. The extra handling can either fix a corner case or introduce new functionality to support a new option. But in either way, they are reflected as extra branches in exceptional CFG. **Steps**: (1) Locate only the new conditions in CDS and new *Catch* in ES. (2) Derive statements in two subsequent blocks of boolean conditions and *Catch* by CFG. (3) Since the extra handling would not tamper with the original implementation, one of the blocks should remain identical and the other one is new. Compare the blocks with the old implementation to get the new ones. (4) Add these new conditions/exceptions, statements from new blocks, and their data dependency statements to S_{ben}.
- **Field: Augmented Output:** In Java, the instance as output is augmented to accommodate more fields. Another case is non-static methods returning void set more fields as the output. **steps**: (1) Identify new fields by comparing the output's old and new field lists by name and type. Note that SEMBID only captures the newly added fields, which means the old fields should remain the same. If the number of output fields does not increase in the new version, it is not considered augmented, but possible breaking changes, as the old implementation could rely on the types of original fields. (2) Backward walk through the def-use chains of the fields recursively to add relevant statements to S_{ben}. Backward slicing does not work in this case, because it returns dependencies of the output instance, instead of specific fields. Hence, fine-grained slicing at the field level is applied.
- **Method Call: Similar Substitution:** Classes or methods are substituted with similar analogies to enforce the original rules. In this case, the semantics of the substituted component is reflected by its role in the context. For example, if one statement of invocation is replaced, but the contextual statements remain the same, the role of the substitution is not changed. SEMBID captures this semantics by using a neighbor-preserving algorithm in Section 4.4, which considers two nodes are identical if all 1_{st} neighboring nodes remain the same.
- **Variable Assignment Revision:** Some variables are assigned different values or new assignments are included. If the variable is local and not a data dependency of the output, it is likely to be an auxiliary variable that controls correct behaviors without directly tampering with the value of output. It is identified by (1) Obtain data dependency statements of conditions from CDS. (2) Compare them based on each condition to get new statements, and check if they belong to new variables. (3) If so, add them to S_{ben}. If the output or data dependency of output is assigned different values, we do not explicitly classify them to S_{ben}, because it is likely to introduce breaking issues by yielding abnormal output.

Theoretically, benign changes should be identified and ruled out to avoid false positives. However, the identification cannot be perfectly accurate and they sometimes still cause SemB, such as the regression issue, because either developers fail to anticipate the breaking or the downstream projects use the API illegitimately. Thus, SEMBID fuzzily measures the semantic diff to determine the SemB with the de-emphasized benign changes instead of completely

ignoring them. However, if the accumulated semantics is changed greatly, it is still considered SemB.

4.4 Measuring Semantic Diff

To measure the semantic diff for a cluster pair, SEMBID constructs an inter-procedural semantic graph as Figure 2 by connecting Exception/Data/Control Dependencies Summaries with execution paths from CFG. The semantic graph preserves the execution logic among relevant dependency statements to model the behaviors of non-local variables of clusters. SEMBID infers the extent of semantic change by calculating the topological similarity of semantic graphs by subgraph matching algorithm based on Weisfeiler-Lehman (WL) graph kernel [58] with weighted statements of S_{ben}. If the final value is above a threshold, the cluster is affected by SemB.

Here are the reasons for the adoption of the WL graph kernel. WL graph kernel can convert the original graph to a sequence of substructures defined as kernels that sort and compress topological and labeling information of adjacent nodes. The kernel pairs preserving the neighboring semantics can be used to calculate the semantic similarity based on the number of matched kernels. Besides, the runtime scales linearly in the number of edges better than other kernels, such as Random-Walk or Shortest-Path [58]. WL graph kernels are designed for directed discrete large graphs which suit the scenario of semantic graphs.

Inspired by the graph matching algorithm of PDG in CCGraph [71], SEMBID relies on the subgraph isomorphism based on h iterations of WL graph kernel calculation to determine the semantic diff between semantic graphs $\langle G, G' \rangle$. CCGraph targets detecting code clone only based on PDG, while the semantics from PDG is not sufficient for SemB detection. For example, the crucial control flow change in Listing 1 is not reflected in PDG. Thus, SEMBID measures the semantic diff between sliced statements connected by control flows for better semantic representation. Also, SEMBID further de-emphasizes the benign changes to align with SemVer rules. The procedures are described below in Algorithm 1:

- Labels of nodes in graphs are hashed as the initial values.
- In i_{th} iteration of WL algorithm [58], labels of each node as well as its neighbors in i hops are compressed into a new label by local sensitive hashing according to WL algorithm.
- In i_{th} iteration, if labels of two nodes are identical, the subgraphs of the node pair are considered as isomorphic at height i.
- After all iterations, the number of non-identical node pairs multiplies with a deteriorating weight w and benign penalty p per pair as the final graph kernel value K. K is normalized to compare against the threshold T. If $K > T$, the cluster pair has SemB. K is calculated as

$$K = \frac{\sum_{i=1}^{h} \langle l(n) | l(n) \neq l(n') \rangle * (h-i+1)/h * sizeof(S_{ben})/sizeof(G')}{min(sizeof(G), sizeof(G'))}$$

Same as [71], deteriorating weight w is calculated as $(h-i+1)/h$ for i_{th} iteration, because the closer the neighbors of node n are, the more they affect the node n. The benign change penalty p is calculated as $sizeof(S_{ben})/sizeof(G')$ to dynamically adapt to the size of G'. Since lower T lowers the precision of SEMBID, while higher T lowers the recall, the T is empirically set as 0.1 to balance the precision and recall. In Figure 2(b), the node *var==1*'s neighbors are converted to subgraphs and then compressed by WL algorithm to form a label at height 1. Height 2 is formed in the same way. Evidently, none of the label pairs of node *var==1*' is identical.

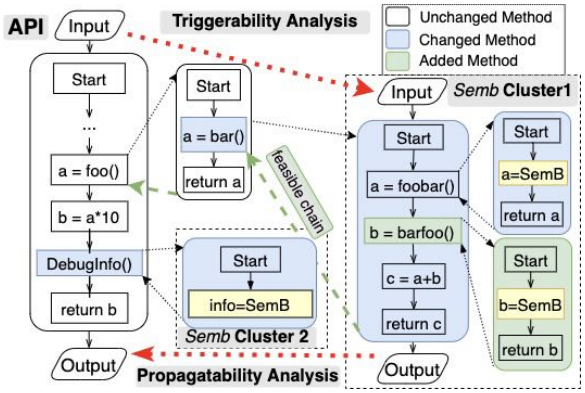

Fig. 3. Triggerabitliy and Propagatability Analysis

Algorithm 1: Algorithm of Measuring Semantic Diff

Input: $\langle c, c' \rangle$: clusters pair with nodes n, label $l(n)$. h: iteration number of WL algorithm. T: threshold, w_i: deteriorating weight at i_{th}, p: benign change penalty.
Output: R: Result of existence of SemB in $\langle c, c' \rangle$

```
foreach i_th iteration in h do
    foreach node n in cluster c do
        sg(n) ← WLGenSubGraph(n, neighbor(n, i_th))
        L(n) ← ∑ (l(n)|n ∈ sg(n))
        L(n) ← sort(L(n))
        l(n) ← l(n)   L(n)
        l(n) ← WLcompress(l(n))
    w_i ← (h − i   1)/h
    k_i ← sizeof(l(n) ≠ l(n')) * w_i * p
k ← ∑ k_i/min(sizeOf(c, c'))
if k > T then
    R ← 1
return R
```

4.5 Checking Impact of Semantic Breaking

This procedure only proceeds when a SemB cluster is caught in the previous step. Only if both triggerability and propagatability are feasible, the SemB cluster is considered a threat to the API.

4.5.1 ***Verifying Triggerability.*** Triggerability defines if SemB can be triggered by old input. In Section 4.2, during the backward slicing, input is associated with relevant statements. If the statements are caught as the SemB changes, the associated input would be checked, including parameters and fields. If the problematic input is introduced in the new version, the SemB cannot be triggered, because the old implementation cannot access the new input.

4.5.2 ***Verifying Propagatability.*** Propagatability determines if the SemB output can propagate from the cluster $c = \{m_{root}, m_j \mid j = 0, ..., n\}$ along call paths to API to affect users. Since SemB is introduced in the new version, SEMBID only verifies the propagatability in the call graph of the new version. SEMBID uses JGraphT [49] serving as the graph infrastructure which first derives all call paths by Dijkstra Algorithm [14] from the API to the m_{root} as $P = \{path_i \mid i = 0, ..., n\}$. Second, For each call path, SEMBID calculates PDG for every method along the call path to verify if the

inter-procedural dependency chain is feasible along every PDG from m_{root} to API's entry method. Finally, propagatability Pg is obtained by $if\, f \exists path \in P, depChain(path) == True, Pg = True$.

For example, in Figure 3, *cluster 1*'s output is propagatable to the API's output. The output of m_{root} of *cluster 1*, $return\, c$ has a feasible dependency chain $a = bar() \rightarrow a = foo() \rightarrow return\, b$ as indicated by the green dotted line. Thus, the output of the API indirectly depends on the SemB change in *cluster 1*. In contrast, an example of an un-propagatable SemB change is *cluster 2* which yields debugging information not used as the API's output.

5 EVALUATION

SemBid was implemented in 9.2K LOC based on Jimple IR of Soot 4.2.1 in Java. We aim at answering the following research questions:
RQ1: What is the accuracy of SemBid in terms of SemB detection?
RQ2: How is the effectiveness of SemBid against unit tests?
RQ3: How do top Java libraries comply with SemVer rules?

5.1 Evaluation Setup

5.1.1 ***For RQ1.*** To evaluate the accuracy of SemBid, we first construct a high-quality ground truth dataset for the benchmark with other API checking tools and the baseline.

Benchmark Dataset Collection. Since SemBid is designed to detect SemB across *Patch* and *Minor* upgrades, we collected broken API pairs of *Patch* and *Minor* upgrades from the 20 most used Maven libraries. The steps are (1) We located Github repositories of those libraries. (2) We conducted regression tests by running unit tests from the old versions against the implementations in the new versions to detect SemB. (3) After ruling out the SynB, the failures that caused *AssertionError*, unexpected exceptions, and crashes are considered as SemB, because *AssertionError* means the program is executed abnormally. Since Mostafa et al. [50] conducted the same regression tests on some of the libraries before 2018, we extended the dataset by extracting APIs from their testing logs in the same steps. Eventually, we derived 308 API pairs with SemB from 77 version pairs. For the dataset of compatible API pairs aligning with SemVer rules, we used the dataset from Section 3.1.2. 363 API pairs with binary changes without SemB serve as the negative data set of SemB detection.

Metrics. The outcomes of SemBid are categorized into (1) True Positive (TP): APIs reported have SemB. (2) False Positive (FP): APIs reported do not have SemB. (3) True Negative (TN): The API not reported by SemBid has no SemB. (4) False Negative (FN): The API not reported by SemBid actually has SemB. Precision, recall, and F-measure are used as evaluation metrics.

5.1.2 ***For RQ2:*** We conducted another experiment to verify the effectiveness of detecting SemB APIs over version pairs between SemBid and the commonly used SemB detection solution, unit tests. As the number of APIs of popular libraries is considerable (406, 826 APIs of 77 pairs), the efforts of manual ground truth checking would be overwhelming. Hence, we selected the top 4 most used libraries with 1 SemB *Patch* version pair each, as they are more likely to have the best testing coverage. In total, 3, 846 APIs were collected.

5.1.3 ***For RQ3:*** All semantically successive version pairs published in the last 20 years of 21 most used Java TPLs from the Maven repository [20] were collected for the large-scale analysis. In total, 546 version pairs and 1, 629, 589 APIs were tested. To analyze the compliance of the SemVer rules, we classified them into (1) *Patch*: 334 pairs; (2) *Minor*: 163 pairs; (3) *Major*: 49 pairs. The versions collected were stable unless none was available.

Table 1: Benchmark Accuracy of SemB Detection based on APIs

Tools	TP	FN	Recall	FP	Precision	F-measure
SemBid	**278**	**30**	**90.26%**	**64**	**81.29%**	**85.54%**
baseline	302	6	98.05%	363	45.41%	62.07%
revapi	30	278	9.74%	21	58.82%	16.71%
japicc	21	287	6.82%	14	60.00%	12.25%
japi-cker	14	294	4.55%	10	58.33%	8.44%
clirr	1	307	0.32%	0	100.00%	0.64%
sigTest	0	308	0.00%	0	N.A.	N.A.

5.2 RQ1: SemB Detection Accuracy

To evaluate the accuracy of SemBid against existing tools, we have selected 5 Java API compatibility checking tools (i.e., revapi [22], japicc [11], japi-checker [7], clirr [8], sigTest, [6]), which are commonly used in benchmarks [42] and industrial software, such as Apache Httpclient [16]. Besides existing tools, we also implemented a baseline tool that relies on the same call graph construction procedures as SemBid, but SemB is considered as positive if any binary diff exists in any method called by APIs.

Table 1 presents the accuracy evaluation of SemBid and other tools on the benchmark data set from Section 5.1. SemBid achieves 90.26% recall, 81.29% precision, and 85.54% F-measure. It is concluded that SemBid outperformed other tools because these tools could only detect SynB instead of the SemB. There were two reasons why some tools could still have TP. First, tools, such as revapi, not only evaluated the compatibility based on signatures but took other features, such as method accessibility and abstraction into account. Second, some of the tools only returned the incompatible class names without method names. If any API signature from the data set had the same class name as the returned class names, we considered the SemB API was detected.

False Negative Cases Discussion. We manually examined the 30 false-negative cases and summarised 4 reasons why SemBid made false decisions. A detailed name list is provided in [13].

- (46.67%, 14 cases) **Secondary Output**: It is the output embodied as debugging messages, written files, and other auxiliary information conveyed by the APIs. Unlike the primary output, the secondary outputs are not global variables passed to the users' programs, but the unit tests sometimes still include them by *assertion*. Since, unlike the secondary output, users directly use the primary one which could break downstream projects, SemBid focuses on the primary output to cover the mainstream scenarios.
- (23.33%, 7 cases) **Falsely Identified Benign Changes**: The patterns of these changes fall into the summarised benign changes. However, no evidence suggested they are benign from commit messages or documentation. Although de-emphasizing benign

changes is not perfect, it does improve the precision at the relatively low cost of false negatives.

- (20.00%, 6 cases) **Signature Reflection**: The breaking was caused by different signature reflections [18]. SEMBID cannot capture the behaviors that can only be triggered dynamically, and thus can complement testing for a more comprehensive detection.
- (10.00%, 3 cases) **Subtle Changes**: The changes were too subtle to be detected by the semantic diff measuring, such as the change of index number of an array, which may not always cause changed output. Although the threshold may overlook some subtle breaking changes, it reduces many more false positives.

False Positive Cases Discussion. 64 false-positive cases were listed with 3 reasons why the results were incorrect.

- (53.13%, 34 cases) **Large Accumulated Semantic Diff**: Although SEMBID already assigned low weights to the benign changes, multiple benign changes can still have a considerable stacked impact on measuring semantic diff. Because benign changes cannot be accurately identified and filtered out, we have to trade off between the under-fitting and over-fitting by weighting.
- (37.50%, 24 cases) **Equivalent Re-implementation**: The code with the same functioning was re-implemented in another way in the new version. One example is the Java feature evolution. Java 8 [17] introduced a feature, *stream*, for parallel aggregate operations. Although it can be used with *forEach* to work the similar way as primitive *for loop*, they are written in totally different byte code. Another example is the change from *array.getElementAt(n)* to *array.subString(n, n+1).get(0)*. Both of them return the same element in the array, but they are implemented in different ways, which resembles the *Type 4 Code Clone* problem that hardly has efficient and effective solutions so far. Hence, SEMBID fails to identify the semantic equivalence between them at the current abstraction level.
- (9.38%, 6 cases) **Unhandled Exception**: They were detected as BBI APIs because the newly thrown exceptions are not correctly handled in the new version. But they are actually handled by the super-type exception catchers. SEMBID checks the exception handling by comparing the exception signatures of the thrown and the catcher. If they are not the same or the catcher is not a general exception, such as "Exception", the thrown exception is considered not caught. In fact, if the thrown is an inherited sub-type of the catcher with different signatures, the thrown can still be caught. SEMBID made such wrong decisions due to the lack of knowledge of the exception inheritance hierarchy.

The **baseline** tool achieved high recall but low precision, which failed to detect 6 cases of signature reflection due to its static basis. All APIs from the compatible test set were false positives.

Conclusion of RQ1: SEMBID outperformed other API compatibility checking tools and achieved 90.26% recall, 81.29% precision, and 85.54% F-measure in terms of SemB API detection. SEMBID achieved much better precision than the baseline tool (45.41%), which indicates that SEMBID is able to effectively filter out the false-positive changes.

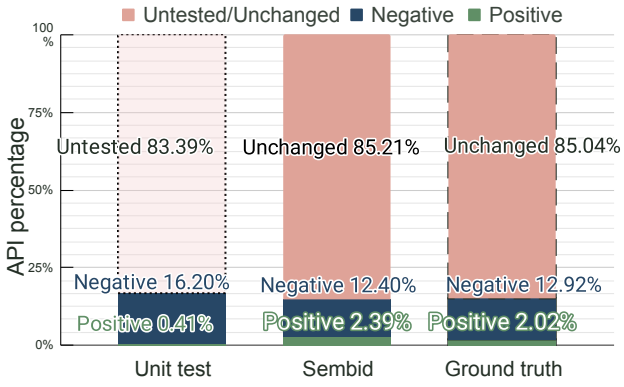

Fig. 4. Comparison between unit tests and SEMBID over all APIs during upgrades as well as the ground truth

5.3 RQ2: Effectiveness of Detecting SemB against Unit Tests

As unit tests are widely used to detect SemB based on APIs, we compared SEMBID against unit tests regarding the number of detected SemB APIs. A similar tool, DeBBI, was proposed by Chen et al. [32] to detect SemB based on augmented unit tests, but it requires manual analysis, and no public data or source code is available. Hence, DeBBI is not involved in the evaluation. Based on the selected libraries from Section 5.1, we first obtained APIs that have binary change as set $A_{changed}$. Then, we derived all tested methods from the testing source code. Because developers would not explicitly mention what methods or classes are tested, we made an overestimation by assuming all public and instantiable classes used in the tests along with their public methods are tested. Based on this assumption, we processed the testing classes in the following manner: (1) Irrelevant testing methods were filtered out according to the rules of testing frameworks. For example, if a framework, Junit, was used, methods annotated with $@before$, $@after$, $@ignore$ would be ignored. 2) From the relevant testing methods, we constructed call graphs for each of them, then directly called methods that meet the aforementioned conditions were collected. (3) During running the tests, the dynamic call graphs were calculated to derive the dynamically called APIs. APIs collected are formed as a set A_{tested} to denote the changed APIs covered by tests.

In Figure 4, the results are illustrated. In total, 3,846 APIs were evaluated in 4 version pairs. The ground truth of them was manually confirmed. It is evident that from the first bar unit tests covered averagely 16.61% of APIs, while the second bar indicates that SEMBID can cover 100% APIs and focus on detecting potential SemB in 15.00% (577) APIs with binary changes. The unchanged APIs are naturally compatible. Apart from the coverage of APIs, even for APIs covered by unit tests, unit tests only uncovered 0.48% (16) SemB APIs of all APIs. According to the ground truth denoted by the third bar, unit tests failed to detect 79.46% (62/78) of SemB APIs, but SEMBID successfully covered 92.30% (72/78) of SemB APIs. Although SEMBID made some false alarms (21.73% = 20/92), generally SEMBID detected more SemB APIs than the unit tests with 4.5 times more TP. However, SEMBID as a static tool has its limit. As unit tests

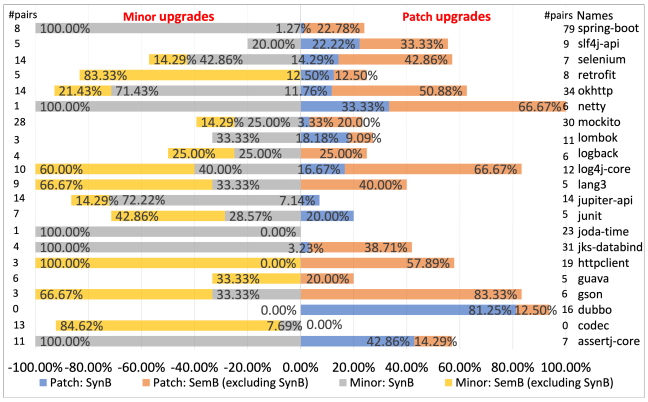

Fig. 5. Proportions of version pairs affected by SemB and SynB of *Patch* and *Minor* upgrades

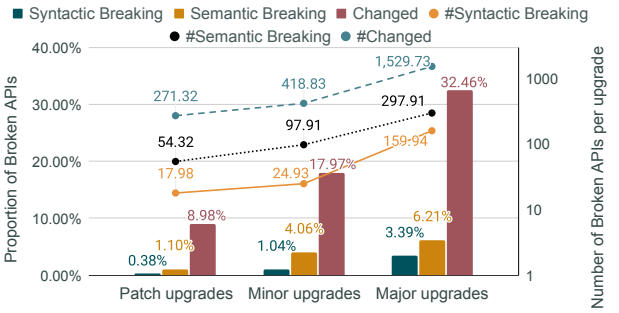

Fig. 6. Average number and percentage of APIs over three upgrades

can dynamically detect SemB in certain APIs with reflection, but SEMBID is not able to cover them statically.

Conclusion of RQ2: Over 4 version pairs, 3,846 APIs in total, SEMBID detected 4.5 times more SemB APIs than unit tests (72 v.s. 16) and achieved better coverage of APIs (100% v.s. 16.61%). Unit tests are able to detect 6 more SemB caused by dynamic operations than SEMBID. It indicates that SEMBID can serve as the complement of unit tests to detect more SemB for *Patch* and *Minor* upgrades.

5.4 RQ3: Study of Compliance with SemVer

To verify the compliance of SemVer rules in popular Java TPLs, we evaluated the TPLs at library, version pair, and API levels respectively. During the evaluation, both SynB and SemB are considered as evidence of breaking (either SemB or SynB is a breaking). For each version pair, the APIs affected by SynB and SemB as well as the APIs with binary changes were collected. The detection of SynB depends on the aforementioned API checking tools. The SynB APIs are the union of them. The SemB was detected by SEMBID. The changed APIs were collected with Soot and BCEL.

Beginning with library and version pair levels, Figure 5 illustrated the proportion of SemB and SynB version pairs of 21 libraries. If one breaking API was detected in a version pair, this version pair was considered to be broken. Note that the SemB version pairs were counted when the version pairs were free from SynB. In other words, the sum of SemB proportion and SynB proportion is the proportion of all broken version pairs. The left side is the *Minor* upgrades, and the right side is the *Patch* upgrades. The numbers of version pairs are annotated at the ends of bars. We found that

- **Patch upgrades on average**: 14/21(66.67%) libraries were subject to SynB for at least one version pair (1.27% – 81.25%). 19/21 (90.48%) libraries have at least one breaking version pair. It is seen that SemVer rules are hardly applied to popular libraries. However, the situation is getting better at the version pair level. On average, SynB affects 10.45% of version pairs, but SemB affects additional 23.35%, which makes 33.83% of version pairs affected by either breaking. The SemB version pairs are over 2 times of SynB. It suggests that SemB detection is necessary in *Patch* upgrades.
- **Minor upgrades on average**: All 20 libraries with minor upgrades are affected by any breaking. It is observed that developers are more likely to include breaking changes in *Minor* upgrades than *Patch*. At the version pair level, SynB affects 37.42% of version pairs, and SemB additionally affects 26.99%, which makes 64.42% of either breaking. Due to legacy reasons, libraries adopt various version release strategies, and many libraries' administrators allow breaking changes in *Minor* upgrades.
- **Particular libraries**: The performance varies tremendously among libraries, because they adopt different, even opposite version release strategies. Some libraries have a high ratio of breaking version pairs, such as *dubbo*, *netty*, *log4j*, *gson*. Because they have very few or 0 *Minor*/*Major* upgrades, they frequently make *Patch* upgrades. They usually make *Major* upgrades cautiously when the entire structure is rewritten, such that *log4j* upgrades from 1.x to 2.x, *httpclient* 4.x-5.x. Then almost all normal upgrades with breaking changes were accommodated in *Patch* and *Minor* upgrades. If *Minor* upgrades are rare, *Patch* upgrades would be mostly broken. For this kind of library, SemVer rules are not properly applied so users have to identify the strategies case by case. There are also some libraries adopting the opposed strategy. For instance, *guava* made *Major* upgrades frequently (13 times in 7 years) with a few *Patch* upgrades. Although *guava* still has a few unexpected SemB, SemVer rules are basically well applied, thus users can upgrade it by SemVer rules with ease.

Considering version release strategies vary greatly among libraries, SemVer rules are not generally followed by popular libraries. Apart from SynB, SemB is also prevalent over *Patch* and *Minor* upgrades. For developers, *Patch* and *Minor* upgrades should check SemB too before publishing to avoid breaking downstream projects. For users, to avoid identifying version release strategies of dependencies case by case, SEMBID can be used on dependencies to pre-check the breaking APIs before upgrading along with SynB checking tools.

The SemB and SynB proportions at the API level are illustrated in Figure 6, *Changed* denotes the number of APIs with binary changes. It is observed that, for SemB, SEMBID significantly filtered out non-breaking APIs from *Changed* APIs. The *SynB* APIs were way fewer

than *SemB*, which means *SynB* checking is far from enough. Moreover, there were still 1.10% APIs affected by the SemB in *patch* upgrades and 4.06% in *minor* upgrades.

Since these libraries were dependencies of over 110k artifacts (libraries) in the Maven ecosystem, the unexposed SemB APIs could unexpectedly detriment the functionalities of those artifacts.

Conclusion of RQ3: From the experiment with 1,629,589 APIs in 546 version pairs, 1.10% of APIs from *Patch* and 4.06% of APIs from *Minor* upgrades were affected by SemB. They are 2-4 times more than SynB APIs (0.38% and 1.04%). In terms of the 497 *Patch* and *Minor* version pairs, *Patch* upgrades have 33.83% breaking pairs, and *Minor* upgrades have 64.42% breaking pairs because version release strategies adopted by libraries vary greatly.

6 THREATS TO VALIDITY

The primary threat is that benign behavior filtering cannot ideally reflect the real intention of the developers, because the rules to filter out harmless behaviors were made based on the empirical summary. The commit intention classification technology can be used to facilitate the accuracy of benign change identification, but they would introduce heavy procedures, such as Machine Learning models, which handicap the scalable and efficient deployment.

Another threat is the scope of *API*. We took public methods of instantiable classes as APIs which are a superset of client-used APIs. However, since Java has no built-in indicator to mark exposed APIs, it is hard to accurately locate the actually used APIs. Taking advantage of usage data of TPL methods is plausible, but it is still not accurate.

7 RELATED WORK

7.1 Study of Semantic Versioning Compliance

Many research works [26, 33, 44, 47, 52, 54, 55] have studied the compliance of SemVer since its release. Raemaekers et al. [54, 55] found around 1/3 of all releases introduce at least one breaking change in seven years release history of Maven Central Repository. Decan et al. [33] studied 4 ecosystems (Cargo, NPM, Packagist, and Rubygems) to understand to what extent developers rely on SemVer to determine dependency constraints and found situations varying greatly among them. Ochoa et al. [52] revealed that 83.4% of upgrades of Maven comply with SemVer rules, and most breaking changes do not affect clients with only 7.9% of clients affected. The works drew conclusions based on signature-based incompatibility instead of SemB, which is not complete. Thus, Sembid is required to provide a more comprehensive analysis of the disobeying of SemVer by including SemB into the picture.

7.2 API Compatibility Checking

Only a limited number of research works [32, 50] regarding SemB of Java program were published in recent years. Mostafa et al. [50] conducted an empirical study on behavioral incompatibility phenomena in popular Java libraries and analyzed published issues from Jira. DeBBI [32] used cross-project testing to amplify the testing coverage to detect Behavioral Incompatibility. Their implementations relied on unit tests, thus subject to coverage. But Sembid relies on static analysis so that Sembid can conduct a more comprehensive analysis. Towards analyzing or detecting the signature-based API compatibility issues of Maven or Android programs, massive empirical studies [2, 34, 41–43, 62, 67, 68] have been conducted. RAPID [68] detects the status of incompatible APIs in the Android ecosystem. Huang et al. [41] studied callback compatibility issues of Android and developed a tool based on CFG to detect such issues. Jezek et al. [42] evaluated 9 commonly used syntactical incompatible API detection tools. Apidiff [30] determined the incompatibility at the name level of methods used in the target library. CiD [46] tried to alert the users by modeling the life cycle of APIs used in specific versions, while ACRYL [56] was a complementary method for CiD based on an alternative data-driven approach . The studies and detection tools based on the signature of APIs did not entail the semantics, thus, they cannot be used to detect SemB APIs like Sembid.

7.3 Software Evolution Studies

Many works [27–29, 35, 36, 39, 40, 48, 51, 64–66] were dedicated to studying the evolution of software across platforms. Dig et al. [35] discovered that 80% of breaking changes belonged to refactoring over the evolution. Mcdonne et al. [48] discovered that the adoption of API is much slower than the API evolution. Researchers of [28, 36] summarized the best practice for developing web application APIs. Bavota et al. [27] revealed the impact of dependencies upgrade based on 14 years of published maven projects. Wu et al. [64–66] found that APIs in frameworks are more susceptible to the missing method or class. These studies established the foundations of API compatibility. With their contributions to the overall understanding of API compatibility, we can locate and resolve the pain points of API compatibility.

8 CONCLUSION

We proposed Sembid to statically detect SemB based on APIs during *Patch* and *Minor* upgrades to enhance the compliance of SemVer rules. Experimental results demonstrated that Sembid achieved 90.26% recall and 81.29% precision. Another experiment proves that Sembid with larger coverage detected 4.5 times more APIs than the commonly used solution, unit tests. Furthermore, a study was conducted on the top 21 Java libraries for over 1.6 million APIs, and 546 version pairs to evaluate the compliance with SemVer rules at the library, version pair, and API levels, which revealed that 33.83% *Patch* upgrades and 64.42% *Minor* upgrades had at least one API affected by any breaking. And on average, there were 2-4 times more APIs affected by SemB issues than SynB issues.

ACKNOWLEDGMENTS

This research is partially supported by the National Research Foundation, Singapore under its the AI Singapore Programme (AISG2-RP-2020-019), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRF-NRFI06-2020-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, the Ministry

of Education, Singapore under its Academic Research Fund Tier 2 (MOE-T2EP20120-0004) and Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] 2004. Japitools. https://savannah.nongnu.org/projects/japitools/.
[2] 2007. Evolving Java-based APIs. https://wiki.eclipse.org/Evolving_Java-based_APIs.
[3] 2008. Jour. http://jour.sourceforge.net/signature.html.
[4] 2012. Jimple. https://en.wikipedia.org/wiki/Soot_(software)#Jimple.
[5] 2013. Http-core motivating example. https://issues.apache.org/jira/browse/HTTPCORE-367.
[6] 2014. sigtest. https://docs.oracle.com/javacomponents/sigtest-3-1/user-guide/toc.htm.
[7] 2015. jchecker. https://github.com/trohovsky/japi-checker.
[8] 2016. clirr. https://www.mojohaus.org/clirr-maven-plugin/index.html.
[9] 2019. Apache Hadoop. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.
[10] 2019. Hadoop HDFS API breaking issue. https://issues.apache.org/jira/browse/HDFS-14595.
[11] 2019. japi-compliance-checker. https://lvc.github.io/japi-compliance-checker/.
[12] 2021. BCEL. https://commons.apache.org/proper/commons-bcel.
[13] 2021. Data Set. https://sites.google.com/view/ase22semverdetection/homepage.
[14] 2021. Dijkstra Algorithm. https://en.wikipedia.org/wiki/Dijkstra_algorithm.
[15] 2021. Http-core. https://hc.apache.org/httpcomponents-core-4.4.x/index.html.
[16] 2021. Httpclient. https://hc.apache.org/httpcomponents-client-5.1.x/.
[17] 2021. Java 8. https://www.oracle.com/java/technologies/java8.html.
[18] 2021. Java Reflection. https://www.oracle.com/technical-resources/articles/java/javareflection.html.
[19] 2021. Maven. https://maven.apache.org/.
[20] 2021. Maven Repository. https://mvnrepository.com/.
[21] 2021. Refactoring. https://en.wikipedia.org/wiki/Code_refactoring.
[22] 2021. revapi. https://revapi.org/revapi-site/main/index.html.
[23] 2021. Semantic Versioning. https://semver.org.
[24] 2021. Soot Spark Call Graph. https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/options/soot_options.htm#phase_5_2.
[25] 2022. Java Polymorphism. https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html.
[26] Rabe Abdalkareem, Md Atique Reza Chowdhury, and Emad Shihab. 2022. A Machine Learning Approach to Determine the Semantic Versioning Type of npm Packages Releases. *arXiv preprint arXiv:2204.05929* (2022).
[27] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *2013 IEEE international conference on software maintenance*. IEEE, 280–289.
[28] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20, 5 (2015), 1275–1317.
[29] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 109–120.
[30] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 507–511.
[31] Eduardo Cunha Campos and Marcelo de Almeida Maia. 2017. Common bug-fix patterns: A large-scale observational study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 404–413.
[32] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 112–124.
[33] Alexandre Decan and Tom Mens. 2019. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering* 47, 6 (2019), 1226–1240.
[34] Jens Dietrich, Kamil Jezek, and Premek Brada. 2016. What Java developers know about compatibility, and why this matters. *Empirical Software Engineering* 21, 3 (2016), 1371–1396.
[35] Danny Dig and Ralph Johnson. 2006. How do APIs evolve? A story of refactoring. *Journal of software maintenance and evolution: Research and Practice* 18, 2 (2006), 83–107.
[36] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. 2014. Web API growing pains: Stories from client developers and their code. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 84–93.
[37] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 408–419.
[38] Carl A Gunter. 1992. *Semantics of programming languages: structures and techniques*. MIT press.
[39] André Hora, Anne Etien, Nicolas Anquetil, Stéphane Ducasse, and Marco Tulio Valente. 2014. Apievolutionminer: Keeping API evolution under control. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 420–424.
[40] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. 2015. How do developers react to API evolution? the pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 251–260.
[41] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for Android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 532–542.
[42] Kamil Jezek and Jens Dietrich. 2017. API Evolution and Compatibility: A Data Corpus and Tool Evaluation. *J. Object Technol.* 16, 4 (2017), 2–1.
[43] Kamil Jezek, Jens Dietrich, and Premek Brada. 2015. How Java APIs break–an empirical study. *Information and Software Technology* 65 (2015), 129–146.
[44] Patrick Lam, Jens Dietrich, and David J Pearce. 2020. Putting the semantics into semantic versioning. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 157–179.
[45] Stanislav Levin and Amiram Yehudai. 2017. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. 97–106.
[46] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of API-related compatibility issues in Android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–163.
[47] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*.
[48] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of API stability and adoption in the Android ecosystem. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 70–79.
[49] Dimitrios Michail, Joris Kinable, Barak Naveh, and John V Sichi. 2020. Jgrapht—a Java library for graph data structures and algorithms. *ACM Transactions on Mathematical Software (TOMS)* 46, 2 (2020), 1–29.
[50] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 215–225.
[51] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. 2002. Evolution patterns of open-source software systems and communities. In *Proceedings of the international workshop on Principles of software evolution*. 76–85.
[52] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. 2021. Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central. *arXiv preprint arXiv:2110.07889* (2021).
[53] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14, 3 (2009), 286–315.
[54] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2014. Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 215–224.
[55] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158.
[56] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. 2019. Data-driven solutions to detect API compatibility issues in Android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 288–298.
[57] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do stack traces help developers fix bugs?. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 118–121.
[58] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. 2011. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research* 12, 9 (2011).

[59] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2020. Why my app crashes understanding and benchmarking framework-specific exceptions of android apps. *IEEE Transactions on Software Engineering* (2020).

[60] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[61] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C Gall, and Alberto Bacchelli. 2019. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming* 180 (2019), 1–15.

[62] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in Java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–45.

[63] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[64] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. Aura: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 325–334.

[65] Wei Wu, Foutse Khomh, Bram Adams, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2016. An exploratory study of api changes and usages based on apache and eclipse ecosystems. *Empirical Software Engineering* 21, 6 (2016), 2366–2412.

[66] Wei Wu, Adrien Serveaux, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2015. The impact of imperfect change rules on framework api evolution identification: an empirical study. *Empirical Software Engineering* 20, 4 (2015), 1126–1158.

[67] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 138–147.

[68] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, et al. 2020. How Android developers handle evolution-induced API compatibility issues: a large-scale study. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 886–898.

[69] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in Android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1695–1707.

[70] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. 2021. Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review. *IEEE Transactions on Software Engineering* (2021).

[71] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 931–942.