

# A Large-Scale Empirical Study on Semantic Versioning in Golang Ecosystem

Wenke Li<sup>†‡</sup>, Feng Wu<sup>§</sup>, Cai Fu<sup>\*†‡</sup>, Fan Zhou<sup>§</sup>

<sup>†</sup> School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China

<sup>‡</sup> Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security

<sup>§</sup> Platform and Content Group, Tencent Technology (Shenzhen) Co.Ltd, Shenzhen, China  
winkli1@hust.edu.cn, barryfwu@tencent.com, fucai@hust.edu.cn, fanzhou@tencent.com

**Abstract**—Third-party libraries (TPLs) have become an essential component of software, accelerating development and reducing maintenance costs. However, breaking changes often occur during the upgrades of TPLs and prevent client programs from moving forward. Semantic versioning (SemVer) has been applied to standardize the versions of releases according to compatibility, but not all releases follow SemVer compliance. Lots of work focuses on SemVer compliance in ecosystems such as Java and JavaScript beyond Golang (Go for short). Due to the lack of tools to detect breaking changes and dataset for Go, developers of TPLs do not know if breaking changes occur and affect client programs, and developers of client programs may hesitate to upgrade dependencies in terms of breaking changes.

To bridge this gap, we conduct the first large-scale empirical study in the Go ecosystem to study SemVer compliance in terms of breaking changes and their impact. In detail, we propose *GoSVI (Go Semantic Versioning Insight)* to detect breaking changes and analyze their impact by resolving identifiers in client programs and comparing their types with breaking changes. Moreover, we collect the first large-scale Go dataset with a dependency graph from GitHub, including 124K TPLs and 532K client programs. Based on the dataset, our results show that 86.3% of library upgrades follow SemVer compliance and 28.6% of no-major upgrades introduce breaking changes. Furthermore, the tendency to comply with SemVer has improved over time from 63.7% in 2018/09 to 92.2% in 2023/03. Finally, we find 33.3% of downstream client programs may be affected by breaking changes. These findings provide developers and users of TPLs with valuable insights to help make decisions related to SemVer.

**Index Terms**—Semantic Versioning, Breaking Change, Third-Party Library, Mining Software Repositories, Go Ecosystem

## I. INTRODUCTION

Third-party libraries (TPLs) nowadays play an important role in modern software systems, avoiding reinventing the wheel and accelerating the development [1]–[3]. Most TPLs provide application programming interfaces (APIs) with downstream client programs to share their new features, fix bugs and refactor code [4], [5].

However, frequent updates of TPLs may introduce breaking changes and result in compilation errors or runtime errors in client programs, which impose extra costs on client programs to adapt to breaking changes [6], [7]. In order to communicate with client programs regarding API compatibility, Semantic Version (SemVer) [8] is used by TPLs to signal the state of

changes to APIs. SemVer constrains the version number to follow the form of *major.minor.patch.pre-release.build*. The updates of *major* and *pre-release/build* may have incompatible changes (breaking changes), while the updates of *minor* and *patch* compatible changes (no breaking changes).

Unfortunately, not all releases of version numbers are compliant with SemVer. Ochoa et al. [5] found that in the Maven central, 83.4% of library upgrades do comply with SemVer, 20.1% of non-major upgrades introduce breaking changes, and only 7.9% of breaking changes affect client programs. Decan et al. [9] proposed an evaluation based on the “wisdom of crowds” principle to infer SemVer compliance, and their results showed that packages in Cargo, Npm, and Packagist are more compliant with SemVer than packages in Rubygems. Lots of work studies SemVer in Java, Javascript, PHP, Ruby, and Rust ecosystems, but there is a gap that we do not know the situation of SemVer in the Golang (Go for short) ecosystem.

The Go programming language is becoming increasingly popular among developers as the number of Go repositories starred on GitHub has increased from rank 8 in *Q1/2014* to rank 3 in *Q1/2023* [10], indicating that more Go TPLs are available. However, not all TPLs adhere to SemVer to meet backward compatibility, for example, the users of popular web framework *gin* found that there were breaking changes in a *minor upgrade* [11]. Despite these breaking changes were tagged in the release notes, they still led to confusion for developers of downstream client programs [12]. As a result, it is significant to investigate SemVer compliance in the Go ecosystem.

To bridge this gap, in this paper, we perform the first large-scale empirical study to explore SemVer compliance in the Go ecosystem, there are two main challenges.

- There is no reliable tool to identify breaking changes and analyze their impact on client programs for Go.
- There is no large-scale dataset for Go, which can identify TPLs and corresponding client programs.

To address the first challenge, we implement a new tool, *GoSVI (Go Semantic Versioning Insight)*, to detect module-based breaking changes and compare usages of client programs with breaking changes based on source code. In detail, first, we extract exported objects from all packages of TPLs, such as interface, function, and struct. Second, we

\* Corresponding author

report breaking changes according to 39 conditions based on the Go apidiff tool [13]. Finally, we resolve identifiers of client programs to extract their types and compare them with breaking changes to calculate the usage rate. To address the second challenge, first, we crawl all Go repositories on GitHub with stars greater than five and remove invalid, duplicate, and outdated repositories. Second, The dependency graph is extracted among the repositories by parsing the *go.mod* file, which is stored in the graph database neo4j [14]. Finally, we identify TPLs and client programs based on their relationships, where entry nodes are treated as client programs and exit nodes are treated as TPLs.

In our work, we set up three questions to explore SemVer compliance in the Go ecosystem.

- RQ1: How are semantic versioning compliance applied in the Go ecosystem in terms of breaking changes?
- RQ2: How much adherence to semantic versioning compliance has increased over time?
- RQ3: What about the impact of breaking changes on client programs?

In our study, we collect 124K TPLs and 532K client programs according to the dependency graph of our dataset. Our results find that (1) 86.3% of libraries upgrades follow SemVer compliance, 28.6% of non-major upgrades (minor and patch upgrades) introduce breaking changes, where *remove* in *function* type is the most common breaking change condition at 36.2%. (2) the tendency to comply with SemVer has improved over time from 63.7% in 2018/09 to 92.2% in 2023/03. (3) only 3.1% of breaking changes affect client programs, while 33.3% of client programs may be affected by breaking changes. Furthermore, we give some suggestions for developers of TPLs and client programs regarding finding breaking changes, maintaining the project layout, and when to upgrade dependencies. Our dataset and tool are accessible on the website <sup>1</sup>.

The main contributions of our study are as follows.

- We conducted the first large-scale empirical study to analyze SemVer compliance in the Go ecosystem and the impact of breaking changes on client programs.
- we implemented a new tool, *GoSVI*, to detect breaking changes and analyze their impact on client programs.
- We proposed a large-scale dataset with a dependency graph in the Go ecosystem, in which TPLs and corresponding client programs are identified for further research.
- We offer some suggestions for Go developers of TPLs and client programs from our study of compatibility analysis.

## II. BACKGROUND

### A. Semantic Version

According to Semantic Version [8], a version number (x.y.z-pre-release+build) consists of three required parts (Major,

Minor, Patch version) and two optional parts (Pre-release and Build Metadata label). Each version change should respect the following rules:

- Major version (x) increases when you make incompatible API changes, for example, the version number changes from 1.0.0 to 2.0.0.
- Minor version (y) increases when you add functionality in a backwards compatible manner, for example, the version number changes from 1.0.0 to 1.1.0.
- Patch version (z) increases when you make backwards compatible bug fixes, for example, the version number changes from 1.0.0 to 1.0.1.
- Pre-release label (pre-release) indicates that the version is a pre-release and may not guarantee compatibility requirements compared to the associated normal version, for example, version 1.0.1-alpha may not be compatible with version 1.0.1.
- Build metadata label (build) indicates that the version only has different build metadata compared to the associated normal version, for example, version 1.0.1+001 may change build metadata compared to version 1.0.1.

Moreover, If the major version is 0, we treat it as the initial development stage which can introduce breaking changes.

### B. Go Package Management and Module System

Package managers are used as tools to create the environment to download, update, and remove project dependencies, allowing developers to build projects more conveniently, such as Maven, Gradle in Java, Npm in JavaScript, and Pip in Python, the Go team also introduced Go modules to manage dependencies [15].

In the module system, a module consists of multiple packages and is used as a unit for managing dependencies [16]. Each Go module has a *go.mod* file and a *go.sum* file.

- The *go.mod* file describes the module's properties, including module path, its dependencies on other modules, and the Go version.
- The *go.sum* file lists down the checksum of direct and indirect dependency required along with the version, which is used to confirm that none of them has been modified.

In the package management system, the main process consists of a decentralized package distribution system, a package search engine, a version naming convention, and Go command tools [17].

- The decentralised package distribution system allows developers to publish modules in their repositories and download other modules.
- Package search engine [18] provides developers with the ability to locate and discover useful modules and packages.
- Version naming convention helps developers to select the right version in terms of compatibility, it serves the same purpose as SemVer.

<sup>1</sup><https://github.com/liwenke1/GoSVI>

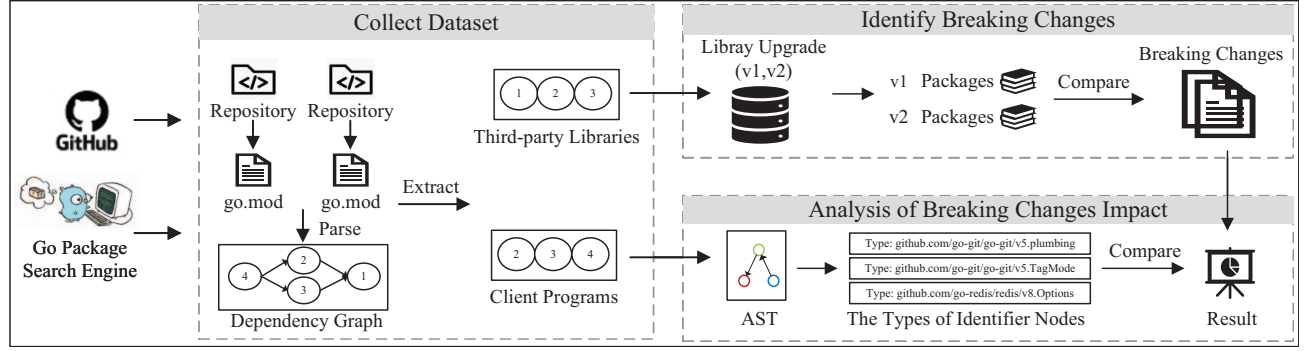


Fig. 1. Overview of the *GoSVI* tool

- Go command tools support an interface to manage dependencies, making it easy for developers to add, update, and remove dependencies.

In this paper, we use modules as the unit for detecting breaking changes because modules, which as the unit of the package distribution system, can be imported by client programs. Moreover, we extract the dependency graph by parsing the *go.mod* file of each Go module.

### III. STUDY DESIGN

In this section, we describe how we collect the dataset and the implementation of *GoSVI* to detect breaking changes between two valid versions and analyze the impact on client programs, as shown in Figure 1.

#### A. Data Collection

The purpose of data collection is to get a large number of go modules and the dependency graph among them, which can be used to identify TPLs and client programs. In the package search engine [18], we can locate useful modules with their dependency graph, but there are two limitations: (1) it is difficult to get all modules in the engine; (2) there are many duplicate and deprecated modules in their dependency graph. Therefore, to ensure the confidence and quality of the dataset, we collect the Go repositories from GitHub and extract the dependency graph among them.

**Crawl Dataset.** We use official Search API [19] published by GitHub to crawl Go repositories. Regarding the limitations on the length of query response, we design two query strategies based on *Star* to collect all the repositories. For repositories with more than 100 stars, keyword *Star* is used to order the query response during pagination. Otherwise, repositories whose *Star* are the same and smaller than 100, will trigger the query response limitations (1,000), resulting in missing some search results, we add another keyword *CreateTime* to order the query response during pagination to solve the problem. Finally, we only reserve repositories with stars greater than five.

**Clean Dataset.** The repositories, which have many defects, will affect the detection results, so we summarized three situations in which the repositories should be removed from our dataset.

- **Invalid repositories.** Repositories that are empty or have build errors or their version number is smaller than two, can not support detecting breaking changes. We check the extension of every file in each repository, if all files do not equal `*.go`, it is considered as an empty repository; we use command `"go build"` in the default branch of the repository to check whether it has build error; we use `"git tag -l"` to calculate the number of versions and remove the repositories whose versions number is smaller than two.
- **Duplicate or similar repositories.** Duplicate repositories are treated as copies of other repositories, as a result, we remove those fork or mirror repositories by searching through GitHub API. Similar repositories, defined as having cloned, migrated, or deprecated repositories, will have the same module path and version number list, so we keep the latest repositories based on the update time.
- **Repositories with Go version before 1.11.** We remove these repositories because Go 1.11 adds preliminary support for a new concept called "modules", an alternative to *GOPATH* with integrated support for versioning and package distribution, and support locating and loading packages of Go source code, what's more, the dependence is defined and managed with *go.mod* file. We remove the repositories that do not contain *go.mod* and have errors during loading and resolving packages.

**Extract Dependency Graph.** The purpose of extracting the dependency graph is to identify TPLs and their corresponding client programs in the dataset. To this end, we parse the *go.mod* file of each repository, analyze the *module* field and the *require* field, and obtain the module path and dependency information on other modules, including the module paths, version numbers, and relationship. We filter out all modules that are not directly imported since no package from these modules is called directly. Note that multiple modules may exist in a repository, so each module will be parsed separately and stored separately. Finally, the dependency graph is stored in the graph database *neo4j* [14], where different versions of each module are treated as nodes and the dependency relationships are treated as edges. In the dependency graph, nodes will be identified as TPLs only if they are dependent

```

1 Module: github.com/pinpoint-apm/pinpoint-go-agent
2 Library Upgrade: v1.1.3 -> v1.2.0, Minor Upgrade
3 Package: github.com/pinpoint-apm/pinpoint-go-agent/
  protobuf
4 Change Node: NewAgentClient
5 Change Category: Function
6 Change Condition: Param Change
7 Change Message: func(google.golang.org/grpc.
  ClientConnInterface) AgentClient -> func(*google
  .golang.org/grpc.ClientConn) AgentClient

```

Fig. 2. The Output of Identifying Breaking Changes

on other modules, and their downstream nodes are treated as client programs.

### B. Breaking Change Detection

In our work, we utilize and enhance the Go apidiff tool [13], proposed by Go maintainers, to detect breaking changes between versions of TPLs. In this section, we introduce the methodology of detecting breaking changes and the catalogue of breaking changes.

The Go apidiff tool is designed to display the differences in exported objects between any two packages, including compatibility and incompatibility changes (breaking changes). We use it to detect breaking changes, however, there are two limitations: (1) the brief report messages make it difficult to classify the results for further research; (2) it cannot detect breaking changes between modules. To address the first limitation, we optimized the tool to report the location and type of breaking changes, as shown in Figure 2, including module, package, type, condition, and detailed message of change nodes. To address the second limitation, we parse all packages of the modules, detect breaking changes between packages with the same path, and finally merge all results. Moreover, the detection process is built completely automatically on repositories hosted on git workflows.

Table I presents the catalogue of breaking changes. There are 13 categories of breaking changes, consisting of *Package* change, 11 data types changes in Go syntax, and *Data Type Change*, which is defined as the change between the above 11 data types. It should be noted that for *basic* data type, we set *basic (Const)* because of its extra breaking change condition of *value change*. A data type may be declared in multiple scenarios, for example, *Map* can be declared by using the keywords *var* or *type*. Furthermore, we summarize 39 conditions of breaking changes based on the change, removal, and addition of its components. Note that *Add Unexported Method* is adding a method to an interface without an existing unexported method, which will make a type that implements the old interface fail to implement the new one.

### C. Analysis of Breaking Changes Impact

To measure the impact of breaking changes on the client programs, we parse the ASTs of packages in the client programs based on static analysis and resolve identifiers to obtain their types. By comparing the breaking change nodes with the identifier nodes of client programs, we are able to get the results of the impact analysis.

TABLE I  
CATALOGUE OF BREAKING CHANGES IN THE GO ECOSYSTEM

Category	Condition
Package	Remove
Basic (Const)	Type Change, Value Change, Remove
Basic	Type Change, Remove
Array	Element Change, Length Change, Remove
Slice	Element Change, Remove
Map	Key Change, Value Change, Remove
Struct	Field Number Change, Field Anonymous Change, Field Type Change, Field Name Change, Field Tag Change, Comparability Change, Remove
Interface	Method Number Change, Method ID Change, Add Unexported Method, Add Interface Method, Remove
Pointer	Base Change, Remove
Channel	Element Change, Direction Change
Function	Param Change, Return Change, Variadic Change, Remove
Named	Element Change, Remove
TypeParam	Type Change, Remove
Data Type Change	Data Type Change

### Algorithm 1: Algorithm of Impact Analysis

**Input:**  $L$ : third-party libraries.  $C(i)$ : clients for third-party library  $i$ .  $B(i)$ : nodes of breaking changes in third-party library  $i$ .

**Output:**  $R$ : affected nodes in the clients

```

1  $R \leftarrow \phi$ ;
2 foreach third-party library  $l$  in  $L$  do
3    $t(b) \leftarrow$  types of  $B(l)$ ;
4    $p(b) \leftarrow$  package paths of  $B(l)$ ;
5   foreach client  $c$  in  $C(l)$  do
6      $p(c) \leftarrow$  import package paths of  $c$ ;
7     if  $p(b) \cap p(c) = \phi$  then
8       continue;
9     end
10    foreach Identifier Node  $n$  in AST of  $c$  do
11       $t(n) \leftarrow \text{Resolve}(n)$ ;
12      if  $t(b) \cap t(n) \neq \phi$  then
13         $R \leftarrow \langle c, n \rangle$ ;
14      end
15    end
16  end
17 end
18 return  $R$ 

```

Algorithm 1 gives the process of analyzing the impact on client programs. First, for each TPL, we extract the type list  $t(n)$  and package path list  $p(n)$  of the breaking change nodes  $n$  (line 2-4). Second, we extract the package path list  $p(c)$  of client  $c$ , and by comparing  $p(b)$  and  $p(c)$ , we filter out client programs that do not import packages of breaking change nodes (lines 5-9). Otherwise, we resolve each identifier node  $n$  to obtain its type  $t(n)$  by parsing the ASTs of the client program, when the type of the identifier node  $t(n)$  is in the type list  $t(n)$  of breaking change nodes, it means that the client program will be affected by this breaking change (line 10-15).



TABLE II  
SCOPE OF DATASET

Dimension	Validity	Category	Number
Repository	Total		102,420
	Invalid		73,325
	Valid		29,095
Version	Total		559,693
	Invalid		112,513
	Valid	Total	447,180
		Major Upgrade	6,691
		Minor Upgrade	45,634
		Patch Upgrade	118,912
		Development	200,859
		Pre-release/Build	75,084
	Valid	TPL Repository	5,604
		Version	124,532
		Client Repository	23,929
		Version	532,832

#### IV. STUDY RESULTS

In this section, we present the scope of our dataset and answer each question through experiments and analysis. All experiments were performed under Go version 1.19.6.

##### A. Dataset Scope

Table II shows the results of crawling and cleaning the dataset in different dimensions. There are 102,420 repositories with stars greater than five, we filter out invalid repositories according to the rule proposed in Section III-A and obtain 29,095 valid repositories. For the remaining repositories with 559,693 versions, we filter out all versions that do not meet the SemVer constraint because we cannot analyze their compatibility. Furthermore, we classify the valid versions into five categories based on the type of library upgrade, including *major upgrade* (6,691), *minor upgrade* (45,634), *patch upgrade* (118,912), *development* (200,859), and *pre-release/build* (75,084). *development*, with a major version number of 0, is considered to be the initial development stage and can introduce breaking changes like *major upgrade* and *pre-release/build*. *minor upgrade* and *patch upgrade* cannot introduce breaking changes to ensure compatibility with client programs. We find that *development* is the most frequent of the five library upgrades, meaning that there are more Go repositories in the unstable development stage. Finally, we identified the TPLs and corresponding client programs based on the dependency graph, there are 5,604 TPLs with 124,532 versions and 23,929 client programs with 532,832 versions respectively. Note that the versions of the TPLs must meet the SemVer constraint, while the versions of the client programs do not.

We also present the distribution of valid library upgrade numbers ranging from 2018/09 to 2022/04, as shown in Figure 3. We chose the start date of 2018/09 because the Go package management and module system was launched in Go version 1.11 in 2018/09 [15]. In 2023/04, the number of library upgrade number drops sharply relative to the previous date since we collect repositories with stars greater than five before 2023/04 and recent repositories have less time to get five stars.

TABLE III  
TOTAL AND BREAKING UPGRADES IN THE DATASET

Level	Total		Breaking	
	Count	%	Count	%
Repository	5,604	100	1,674	29.9
Version	Major	1,926	1.5	1,147
	Minor	16,305	13.1	6,173
	Patch	43,222	34.7	10,836
	Development	63,079	50.7	17,904
	Non-Major	59,527	47.8	17,009
Total	124,532	100	36,060	29.0

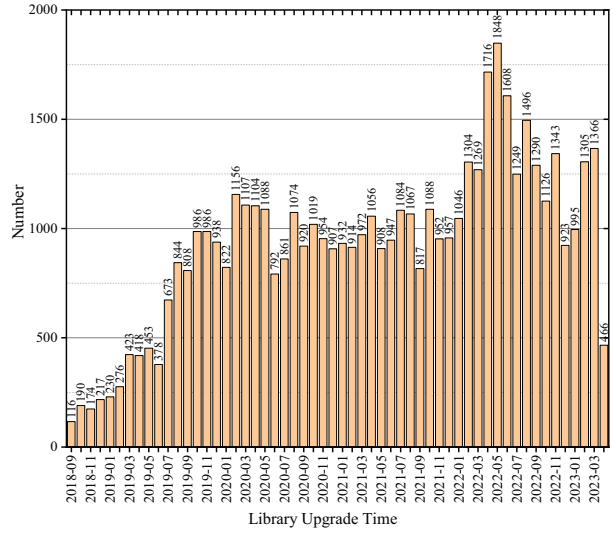


Fig. 3. Statistics on the number of version upgrades over time

Overall, the number of library upgrade number is increasing from 116 in 2018/09 to 1,366 in 2023/03. Therefore, with the rapid increase in library upgrades, it is meaningful to detect whether library upgrades meet SemVer compliance and do not introduce breaking changes.

*Summary:* We identified 124K TPLs and 532K client programs from the collected dataset. Moreover, the number of valid library upgrades is rapidly increasing over time and it is significant to explore compliance with SemVer.

B. RQ1: How are semantic versioning compliance applied in the Go ecosystem in terms of breaking changes

1) *Method:* With RQ1, we analyze the extent of SemVer principles applied in the Go ecosystem from three perspectives: breaking repositories, breaking library upgrades, and the types of breaking changes. First, we identify how many repositories have introduced breaking changes. Second, we analyze how many breaking changes are usually introduced in each library upgrade to know the distribution of breaking changes. Finally, we study the frequency of each type of breaking change to find out the most common breaking changes in our dataset.

To do so, we identify breaking changes between two valid versions of TPLs and count the types of breaking changes. The types of library upgrades are distinguished into *major*

TABLE IV  
THE DISTRIBUTION OF BREAKING CHANGES AND THEIR IMPACT

Index	Category	Condition	Breaking Change		Usage in Client Program			Breaking Library Upgrade	
			Number (B)	%	Number (U)	%	% (U/B)	Number	%
1	Package	Remove	4,132	1.1	165	1.5	4.0	2,091	4.1
2	Basic (Const)	Type Change	1,332	0.4	22	0.2	1.7	53	0.1
3		Value Change	26,955	7.4	594	5.3	2.2	18,788	37.2
4		Remove	22,738	6.3	529	4.7	2.3	265	0.5
5	Basic	Type Change	1,608	0.4	1	0.0	0.0	15	0.0
6		Remove	13,223	3.6	3	0.0	0.0	20	0.0
7	Array	Element Change	12	0.0	0	0	0	0	0
8		Length Change	60	0.0	0	0	0	0	0
9		Remove	73	0.0	0	0	0	0	0
10	Slice	Element Change	928	0.3	3	0.0	0.3	10	0.0
11		Remove	3,709	1.0	0	0	0	0	0
12	Map	Key Change	63	0.0	0	0	0	0	0
13		Value Change	247	0.1	0	0	0	0	0
14		Remove	1,350	0.4	6	0.1	0.4	8	0.0
15	Struct	Field Number Change	195	0.1	3	0.0	1.5	3	0.0
16		Field Anonymous Change	0	0	0	0	0	0	0
17		Field Type Change	140	0.0	1	0.0	0.7	1	0.0
18		Field Name Change	6	0.0	0	0	0	0	0
19		Field Tag Change	48	0.0	0	0	0	0	0
20		Comparability Change	4,567	1.3	367	3.2	8.0	8,325	16.5
21		Remove	970	0.3	0	0	0	0	0
22	Interface	MethodNumber Change	4	0.0	0	0	0	0	0
23		Method ID Change	2	0.0	0	0	0	0	0
24		Add Unexported Method	101	0.0	0	0	0	0	0
25		Add Interface Method	30,563	8.4	2,660	23.5	8.7	22,899	45.4
26		Remove	102	0.0	0	0	0	0	0
27	Pointer	Base Change	969	0.3	2	0.0	0.2	170	0.3
28		Remove	4,968	1.4	1	0.0	0	6	0.0
29	Channel	Element Change	15	0.0	0	0	0.0	0	0
30		Direction Change	0	0	0	0	0	0	0
31		Remove	71	0.0	0	0	0	0	0
32	Function	Param Change	37,864	10.4	786	7.0	2.1	8,141	16.1
33		Return Change	17,351	4.8	418	3.7	2.4	1,862	3.7
34		Variadic Change	3,917	1.1	193	1.7	4.9	5,054	10.0
35		Remove	131,565	36.2	829	7.3	0.6	2,329	4.6
36	Named	Element Change	10,039	2.8	3,783	33.5	37.7	1,254	2.5
37		Remove	35,701	9.8	853	7.5	2.4	1,469	2.9
38	TypeParam	Type Change	21	0.0	0	0	0	0	0
39		Remove	0	0	0	0	0	0	0
40	Category Change	Data Type Change	7,819	2.2	85	0.8	1.1	1,012	2.0
41	Total		363,428	100	11,304	100	3.1	50,485	100

*upgrade*, *minor upgrade*, *patch upgrade*, *development*, and *non-major upgrade*, ignoring *pre-release/build* because it is usually complementary to the normal library upgrade. *non-major upgrade* consists of *minor upgrade* and *patch upgrade* and cannot introduce breaking changes based on SemVer compliance.

According to the layout of Go projects [20], not all packages are imported by client programs. Therefore, We filter out the following folders that are not related to breaking changes.

- **/cmd:** Main applications for this project.
- **/internal:** Private application and library code that do not want others importing in their applications or libraries.
- **/vendor:** Application dependencies.
- **/config:** Configuration file templates.
- **/init:** System initialization and management files.
- **/scripts:** Scripts to perform build, install, etc. operations.
- **/build:** Packaging and continuous integration.
- **/deployment:** Deployment configuration files.

- **/test:** Additional external test applications and data.

2) *Results:* From the perspective of breaking repositories, as shown in Table III, 1,674 repositories (29.9%) have at least one breaking change, meaning that almost one-third of TPLs violate the SemVer compliance and there is a high risk for client programs to pick dependencies that do not adhere to compatibility constraints.

From the perspective of breaking library upgrades, we found that 50.7% of library upgrades are in the initial development stage, followed by *patch upgrades* (34.7%), *minor upgrades* (13.1%), and *major upgrades* (1.5%). This distribution indicates that half of the library upgrades are not ready for the public API. For breaking library upgrades, as expected, most *major upgrades* (59.6%) introduce breaking changes, followed by *minor upgrades* (37.9%), *development* (28.4%), *patch upgrades* (25.1%). There are 28.6% of *non-major upgrades* that do not meet the SemVer compliance, as many as the breaking repositories, meaning that one-third of library

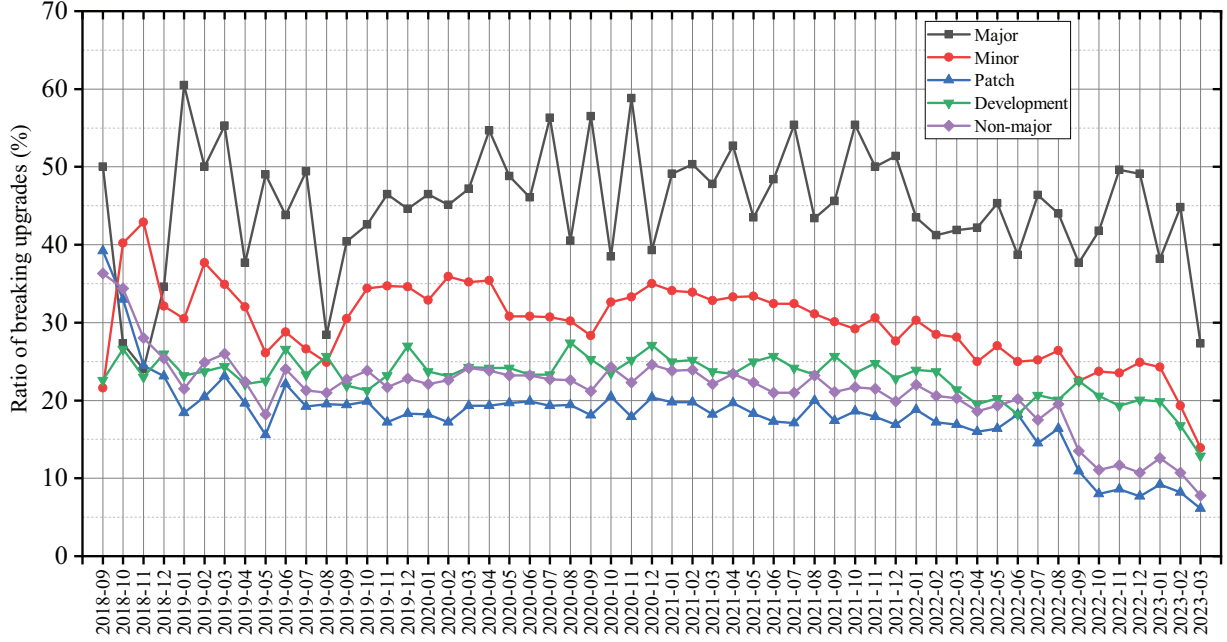


Fig. 4. SemVer compliance in the Go ecosystem over time

upgrades introduce breaking changes. It is also important to note that the rate of breaking library upgrades in *development* differs significantly from *major upgrade* but only slightly from *non-major upgrade*, pointing to greater attention to semantic version compliance during the initial development stage.

From the perspective of the types of breaking changes, as shown in Table IV, there are 40 breaking change conditions, divided into 14 categories, including *package*, *basic (const)*, *basic*, *array*, *slice*, *map*, *struct*, *interface*, *pointer*, *channel*, *function*, *named*, *typeParam*, and *category change*. We identified 363,428 breaking changes, and the five most common breaking change types are *remove in function* type (36.2%), *param change in function* type (10.4%), *remove in named* type (9.8%), *add interface method in interface* type (8.4%), and *value change in basic (const)* type (7.4%). Moreover, for breaking change actions such as *remove*, *change*, and *add*, we found that *remove* is the most common action (60.2%).

*Summary:* 86.3% of library upgrades adhere to SemVer compliance, but 28.6% *non-major upgrades* introduce breaking changes. It is important to note that even in the initial development stage, developers of TPLs pay more attention to avoid introducing breaking changes relative to *Major Upgrades*. Moreover, *remove* is the most common breaking change action at 60.2% while *remove in function* is the most common breaking change condition at 36.2%.

C. RQ2: How much adherence to semantic versioning compliance has increased over time?

1) *Method:* To answer RQ2, we study the tendency of compliance with SemVer over time in terms of library up-

grades. In detail, library upgrades are still divided into five categories, including *major upgrade*, *minor upgrade*, *patch upgrade*, *development*, and *non-major upgrade*. Moreover, We count the rate of breaking library upgrades per month to reflect the tendency of SemVer compliance over time. We do not study the tendency of compliance with SemVer over time in terms of repositories because too few repositories may lead to high data volatility while the number of library upgrades is nearly 22 times the number of repositories.

2) *Results:* Figure 4 shows the tendency of the rate of different breaking library upgrades in the latest five years, from 2018/09 to 2023/03. We focus on the tendency of the rate of breaking *non-major upgrade* because this upgrade cannot introduce breaking changes. Also, We divide the tendency into the following three parts.

- **From 2018/09 to 2019/01.** Due to the recent release of the Go module system and the low number of library upgrades, the rates of breaking library upgrades change a lot during this period. However, the rate of breaking *non-major upgrades* is decreasing from 36.3% to 21.5%.
- **From 2019/01 to 2022/08.** The rates of breaking library upgrades tend to be stable during this period, where the rate of breaking *non-major upgrades* ranges from 17.5% to 26.0%. Regarding *major upgrade*, the rate of it changes a lot because of its small number.
- **From 2022/08 to 2023/03.** The rates of all breaking library upgrades are decreasing, in which the rate of breaking *non-major upgrade* is decreasing sharply from 19.5% to 7.8%.

*Summary:* The rate of breaking *major upgrade* fluctuates over time because of its small number, while the rates of other breaking library upgrades are decreasing. Furthermore, more library upgrades meet SemVer compliance from 63.7% in 2018/09 to 92.2% in 2023/03.

*D. RQ3: What about the impact of breaking changes on client programs?*

1) *Method:* In RQ3, we examine to what extent breaking changes affect client programs in terms of the types of breaking changes and how many client programs are affected by breaking changes. In detail, for a breaking library upgrade from version *v1* to *v2*, we extract the client programs that depend on version *v1*. If the developers of the client programs update the dependent version to *v2*, breaking changes will affect the client programs. Moreover, we ignore *major upgrades* and *developments* because these library upgrades can introduce breaking changes while *non-major upgrades* need to ensure that there are no incompatible changes. We analyze the impact of breaking changes on client programs by calculating the usage related to breaking changes.

2) *Results:* According to Table III, there are 17,009 *non-major upgrades* and we collected 151,589 downstream client programs based on the dependency graph. We present the data of breaking changes used in client programs and library upgrades affected by different types of breaking changes, as shown in Table IV.

There are 11,034 breaking changes used in client programs, in which the five most common breaking change types are *element change in named type* (33.5%), *add interface method in interface type* (23.5%), *remove in named type* (7.5%), *remove in function type* (7.3%), and *param change in function type* (7.0%). Compared to the five most common breaking changes, *element change in named type* is added while *value change in basic (const) type* is missing. *Named types* are data types customized by the developers of TPLs to achieve different functionality, therefore, breaking changes on them are more likely to be introduced by client programs. We ignore the missing of *value change in basic (const) type* because it still occurs frequently (5.3%).

we also found that 96.9% of breaking changes do not affect client programs. Among the high frequency of breaking changes, only a very low rate affects client programs, for example, only 0.6% of *remove in function type* have an impact on client programs. For *element change in named type*, although its breaking changes occur at a rate of 2.8%, one-third of them will have a significant impact.

We also explain the impact of breaking changes from the perspective of the client programs. There are 50,485 client programs affected by breaking changes, which means that one-third of the client programs should make more efforts due to violations of SemVer compliance. Moreover, the two most common breaking changes affecting client programs are *add interface method in interface type* (45.4%) and *value change in basic (const) type* (37.2%). *add interface method in interface type* requires the developers of client programs to

write additional code to adapt to the new behaviour. And *value change in basic (const) type* indicates that the developers of client programs need to keep up with the times using newer parameters and configurations for program operation.

*Summary:* We observe that only 3.1% of breaking changes are used by client programs, but they affect 33.3% of downstream client programs due to numerous dependencies. This tells us that just a small number of breaking changes can have a serious impact on client programs in the Go ecosystem.

## V. DISCUSSION

### A. Implications for TPLs Developers

Changes often occur in software development to add new features, fix bugs, and refactor code [21]. When breaking changes exist, developers cannot perform *minor upgrades* and *patch upgrades* according to SemVer [8]. This is simple to do for TPLs maintained by a single or a small number of developers, while in a collaborative TPL, there is no guarantee that everyone will strictly adhere to SemVer compliance. Therefore, it is important for developers to use *GoSVI* to detect breaking changes and help determine the version number.

Not all exported objects should be referenced by client programs. Developers can limit the scope of packages by defining *internal* packages, which cannot be used by client programs. Also, if the interfaces do not want to be used by client programs, an unexported method should be added to the interface so that client programs cannot implement them.

Moreover, it makes sense to use *GoSVI* to detect which exported objects are more popular with client programs, enabling developers to refactor their code more sensibly while affecting client programs as little as possible.

### B. Implications for Client Programs Developers

As shown in RQ1 and RQ3, although 86.3% of library upgrades adhere to SemVer compliance, 33.3% of client programs are affected by breaking changes. Therefore, it is essential for developers of client programs to decide which version to upgrade. Compared to reading the documentation and source code, *GoSVI* can automatically identify breaking changes of library upgrades and help meet requirements without causing too many interruptions.

Furthermore, breaking changes exist in most ecosystems [9], such as Cargo, Npm, Packagist, and Rubygems, and Go is no exception. If developers do not want to be affected by breaking changes, the most conservative approach is to not update dependencies. Delaying updates to dependencies and checking feedback to decide whether to update is also a recommended approach.

## VI. THREATS TO VALIDITY

### A. Internal Validity

We collect the dataset from 2018/09 to 2023/04, while SemVer was proposed in 2013/06, meaning that the analysis results are missing between 2013/06 to 2018/09. We ignore



those data to ensure that the dataset is confident because it is not possible to extract dependency graphs and identify TPLs based on the module system.

We filter packages designed not to be imported based on the layout of Go project [20], but this rule is not followed by all developers. Therefore, this threat may lead to a low rate of compliance with the Semver.

Although the Go apidiff tool is officially implemented and widely used, we were unable to verify its accuracy in detecting breaking changes because of the absence of any standard benchmark. In order to understand as much as possible about the effectiveness of *GoSVI* in detecting breaking changes, we verified the precision on 100 randomly selected cases of breaking changes. The result <sup>2</sup> shows that *GoSVI* can identify the majority of breaking changes with 91% precision.

The process of resolving identifier nodes to extract the types may have errors. However, to relieve this threat, we implement this functionality based on a well-known TPL: *packages* [22].

### B. External Validity

We collect the Go repositories hosted on GitHub, which attracts over 100 million software developers. There is an error in using the analysis results of repositories on GitHub as the analysis results of the whole ecosystem.

## VII. RELATED WORK

### A. Study of Semantic Versioning Compliance

SemVer is commonly accepted by package management systems to inform developers whether releases of software packages introduce possible breaking changes. Hence, many empirical works [5], [23], [24] have studied compliance of SemVer in different package management systems, such as Java with Maven, Node.js with Npm, Rust with Cargo, PHP with Packagist and Ruby with Rubygems. Cogo et al. [23] scoped the downgrade from client view with more than 600K reusable packages of JavaScript from Npm and the result showed the clients decide to downgrade mostly because the provider version introduces a failure and the downgrade is correlative with the upgrade caused by the provider. Ochoa et al. [5] scoped SemVer on Java with Maven of almost 120K library upgrades and found that 20.1% of non-major upgrades contained Breaking Changes and only 7.9% of client programs are affected by breaking changes. Raemaekers et al. [24], [25] scoped SemVer on Java with Maven of more than 100K libraries and found that nearly one-third of all releases introduce at least one breaking change, the tendency of complying with SemVer in non-major releases do not become better over time and developers of client programs do not follow deprecation guidelines suggested by SemVer as expected. In this paper, we focus on the scope of the Go package management system and found that 86.3% of library upgrades comply with SemVer while 33.3% of client programs may be affected by breaking changes.

<sup>2</sup>The data is accessible at <https://drive.google.com/drive/folders/1Cf9KITHz5p04xZJCKQo5BZEP6h4Bov8>

### B. Detection of Breaking changes

There are many tools that are designed to detect API breaking changes [4], [26]–[29]. Brito et al. [4] proposed APIDiff to detect syntactic changes on Java with type, method and field and recognized breaking changes and non-breaking changes based on different conditions. Mezzetti et al. [26] proposed techniques named type regression testing to automatically detect breaking updates on the public interface on Node.js based on dynamic analysis. Mujahid et al. [27] proposed a technique to detect breakage-inducing versions of third-party dependencies. Moreover, Jezek et al. [30] have developed a compact benchmark data set of less than 200KB to evaluate the accuracy of the tools, such as japicmp [28], JAPICC [31], and SigTest [32]. Zhang et al. [29] detected semantic breaking change based on static analysis and found that there were 24 times more semantic breaking changes than those with signature-based issues. In this paper, we focus on the signature-based breaking changes and propose *GoSVI* tool to detect breaking changes between versions in the Go module system.

### C. Analysis of breaking change impact

There are many works have been proposed to analyze the impact of breaking change on client programs [5], [21], [24], [33], [34]. Xavier et al. [21] and Bavota et al. [34] analyzed whether the types of breaking changes are imported by client programs (using keyword *import*). This approach overestimates the impact of breaking changes because there are cases where they are imported but not used. Robbes et al. [33] manually calculated the ripple effects of breaking changes across an entire ecosystem. This approach is not appropriate for large-scale studies. Raemaekers et al. [24] proposed a new technique to inject each breaking change individually and analyze its impact by counting the number of compilation errors. It is difficult to ensure that the injection process does not introduce other breaking changes to mislead the experimental results. Ochoa et al. [5] used Rascal  $M^3$  model [35], which stores the relationship between Java elements, to link breaking changes to the usages in client programs. This approach is limited by the  $M^3$  model and cannot analyze breaking changes related to overridden methods. Prior works have focused on analyzing the impact of breaking changes in the Java language except for the Go language. In this paper, we resolved all identifiers in the client programs to extract their types and calculated the affected elements of client programs by comparing the types with breaking changes. We implement this functionality based on the Go underlying toolkit *packages* [22] to make the results more accurate.

### D. API Evolution

Many research works [33], [36]–[41] have studied the API evolution from different perspectives. Lamothe et al. [36] found that understanding, mitigating, and leveraging the impact of APIs and API evolution on software development are necessary to design and use APIs. Koçi et al. [37] summarized that API evolution can be divided into changed API elements,

changed performing actions from the producer of view, and changed API elements from the consumer of view. The reason to perform changes includes improving understandability and readability, improving error tolerance from various consumers, improving security by fixing bugs or meeting consumers' common requirements. Robbes et al. [33] point out that the median time of developers reacting to API changes is two weeks, and the time of adaption takes nearly a month or higher. Hora et al. [40] showed that client reaction time to API changes is longer than deprecation. Lima et al. [41] analyzed that popular APIs provide a large proportion of API elements and contain more code comments, which indicates the stability of popular APIs is better. In this paper, we aim to investigate API compatibility in terms of API breaking changes according to SemVer [8].

## VIII. CONCLUSION

In this paper, we conduct the first large-scale empirical study in the Go ecosystem to investigate the compliance of the SemVer in terms of breaking changes. We aim to analyze the frequency of breaking changes, the tendency of breaking changes over time, and the impact of breaking changes on client programs. To do so, we implement and use *GoSVI* to detect breaking changes between two versions in the Go modules and analyze the impact of breaking changes on clients by resolving identifier nodes and comparing the types with breaking changes. Also, we collect a large-scale dataset with a dependency graph, including 124K TPLs and 532K client programs. We found the following results.

- The number of library upgrades is rapidly increasing over time, and it is significant to explore whether each library upgrade adheres to Semver compliance.
- Library upgrades often introduce breaking changes and 28.6% of *non-major upgrades* violate SemVer compliance. Moreover, *remove in function* type is the most common breaking change condition at 36.2%.
- More library upgrades do not introduce breaking changes over time. The tendency to comply with SemVer has significantly increased from 63.7% in 2018/09 to 92.2% in 2023/03.
- There are only 3.1% of breaking changes used in client programs, but one-third of client programs may be affected, meaning that few breaking changes will have a serious impact on client programs.

According to our results, we give some suggestions to help developers of TPLs and client programs to make decisions about SemVer. Also, we recommend using *GoSVI* to assist with automated breaking changes analysis and downstream client programs data insight.

As future work, we plan to detect semantic breaking changes in the Go ecosystem to understand how breaking changes potentially disrupt client programs. Also, we will draw comparisons with other package ecosystems such as Npm and Maven to provide more valuable insights into the relative significance of breaking changes across different platforms.

## ACKNOWLEDGMENT

We appreciate the insightful insights provided by anonymous reviewers to improve the quality of the paper. This work is supported by the National Science Foundation of China under grant No. 62072200 and No. 6217071437.

## REFERENCES

- [1] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "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)*, 2021, pp. 1695–1707.
- [2] A. Zerouali, T. Mens, J. Gonzalez-Barahona, A. Decan, E. Constantinou, and G. Robles, "A formal framework for measuring technical lag in component repositories—and its application to npm," *Journal of Software: Evolution and Process (J Softw)*, vol. 31, no. 8, p. e2157, 2019.
- [3] X. Zhan, T. Liu, L. Fan, L. Li, S. Chen, X. Luo, and Y. Liu, "Research on third-party libraries in android apps: A taxonomy and systematic literature review," *IEEE Transactions on Software Engineering (TSE)*, 2021.
- [4] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Apidiff: Detecting api breaking changes," in *Proceedings of 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 507–511.
- [5] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, "Breaking bad? semantic versioning and impact of breaking changes in maven central," *Empirical Software Engineering (ESE)*, vol. 27, no. 3, 2022.
- [6] S. Mostafa, R. Rodriguez, and X. Wang, "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 (ISSTA)*, 2017, pp. 215–225.
- [7] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 408–419.
- [8] "Semantic versioning," 2021. [Online]. Available: <https://semver.org/>
- [9] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 6, pp. 1226–1240, 2019.
- [10] F. Beuke, "Github language stats," 2023. [Online]. Available: <https://madnight.github.io/github/#/stars/2023/1>
- [11] N. Meyer, "Why was backward compatibility broken without a major version upgrade," 2019. [Online]. Available: <https://github.com/gin-gonic/gin/issues/2174>
- [12] G. K. Raula, M. G. Daniel, O. Ali, and I. I. Takashi, "Do developers update their library dependencies?" *Empirical Software Engineering (ESE)*, vol. 23, pp. 384–417, 2017.
- [13] "Apidiff packages for checking go package api compatibility," 2022. [Online]. Available: <https://pkg.go.dev/golang.org/x/exp/apidiff>
- [14] "Neo4j graph data platform - graph database management tool," 2023. [Online]. Available: <https://neo4j.com/>
- [15] "Go 1.11 release notes," 2018. [Online]. Available: <https://go.dev/doc/go1.11>
- [16] "The go.mod file reference," 2018. [Online]. Available: <https://go.dev/doc/modules/gomod-ref>
- [17] "How to managing dependencies," 2018. [Online]. Available: <https://go.dev/doc/modules/managing-dependencies>
- [18] "Go package search engine," 2023. [Online]. Available: <https://pkg.go.dev/>
- [19] "Github docs - search repositories," 2022. [Online]. Available: <https://docs.github.com/en/rest/search#about-the-search-api>
- [20] K. Quest, "Standard go project layout," 2023. [Online]. Available: <https://github.com/golang-standards/project-layout>
- [21] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in *Proceedings of 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 138–147.
- [22] "packages package," 2023. [Online]. Available: <https://pkg.go.dev/golang.org/x/tools/go/packages>
- [23] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "An empirical study of dependency downgrades in the npm ecosystem," *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 11, pp. 2457–2470, 2019.

- [24] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the maven repository," *Empirical Software Engineering (ESE)*, vol. 129, pp. 140–158, 2017.
- [25] S. Raemaekers, A. Van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository," in *Proceedings of 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (ICSME)*, 2014, pp. 215–224.
- [26] G. Mezzetti, A. Möller, and M. T. Torp, "Type regression testing to detect breaking changes in node.js libraries," in *Proceedings of the 32nd european conference on object-oriented programming (ECOOP)*, 2018, pp. 7:1–7:24.
- [27] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, "Using others' tests to identify breaking updates," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, 2020, pp. 466–476.
- [28] M. Mois, "japicmp," 2023. [Online]. Available: <https://github.com/siom79/japicmp>
- [29] Z. Lyuye, L. Chengwei, X. Zhengzi, C. Sen, F. Lingling, C. Bihuan, and L. Yang, "Has my release disobeyed semantic versioning? static detection based on semantic differencing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022, pp. 1–12.
- [30] K. Jezek and J. Dietrich, "Api evolution and compatibility: A data corpus and tool evaluation," *The Journal of Object Technology (J. Object Technol.)*, vol. 16, no. 4, pp. 2–1, 2017.
- [31] A. Ponomarenko, "Java api compliance checker," 2023. [Online]. Available: <https://github.com/lvc/japi-compliance-checker>
- [32] K. Looney, "Sigtest," 2023. [Online]. Available: <https://wiki.openjdk.org/display/CodeTools/sigtest>
- [33] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation? the case of a smalltalk ecosystem," in *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11.
- [34] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in *Proceedings of 2013 IEEE International Conference on Software Maintenance (ICSM)*, 2013, pp. 280–289.
- [35] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju, "M3: A general model for code analytics in rascal," in *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, 2015, pp. 25–28.
- [36] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A systematic review of api evolution literature," *ACM Computing Surveys (CS)*, vol. 54, no. 8, pp. 1–36, 2021.
- [37] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló, "Classification of changes in api evolution," in *Proceedings of 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, 2019, pp. 243–249.
- [38] T. Espinha, A. Zaidman, and H.-G. Gross, "Web api growing pains: Stories from client developers and their code," in *Proceedings of 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 84–93.
- [39] S. Sohan, C. Anslow, and F. Maurer, "A case study of web api evolution," in *Proceedings of 2015 IEEE World Congress on Services*, 2015, pp. 245–252.
- [40] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "Api evolution miner: Keeping api evolution under control," in *Proceedings of 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 420–424.
- [41] C. Lima and A. Hora, "What are the characteristics of popular apis? a large-scale study on java, android, and 165 libraries," *Software Quality Journal (Softw. Qual. J.)*, vol. 28, no. 2, pp. 425–458, 2020.