Using Others' Tests to Identify Breaking Updates

Suhaib Mujahid
Data-driven Analysis of Software (DAS) Lab
Dept. of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
s_mujahi@encs.concordia.ca

Rabe Abdalkareem
Software Analysis and Intelligence Lab (SAIL)
School of Computing
Queen's University
Kingston, Ontario, Canada
abdrabe@gmail.com

Emad Shihab
Data-driven Analysis of Software (DAS) Lab
Dept. of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
eshihab@encs.concordia.ca

Shane McIntosh
Software Repository Excavation and Build Eng. Labs
Dept. of Electrical and Computer Engineering
McGill University
Montreal, Quebec, Canada
shane.mcintosh@mcgill.ca

ABSTRACT

The reuse of third-party packages has become a common practice in contemporary software development. Software dependencies are constantly evolving with newly added features and patches that fix bugs in older versions. However, updating dependencies could introduce new bugs or break backward compatibility. In this work, we propose a technique to detect breakage-inducing versions of third-party dependencies. The key insight behind our approach is to leverage the automated test suites of other projects that depend upon the same dependency to test newly released versions. We conjecture that this crowd-based approach will help to detect breakage-inducing versions because it broadens the set of realistic usage scenarios to which a package version has been exposed. To evaluate our conjecture, we perform an empirical study of 391,553 `npm` packages. We use the dependency network from these packages to identify candidate tests of third-party packages. Moreover, to evaluate our proposed technique, we mine the history of this dependency network to identify ten breakage-inducing versions. We find that our proposed technique can detect six of the ten studied breakage-inducing versions. Our findings can help developers to make more informed decisions when they update their dependencies.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Risk management*; *Software version control*; **Software libraries and repositories**; *Software configuration management and version control systems.*

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387476

KEYWORDS

JavaScript, Node.js, Empirical Studies, Software Quality, Software Ecosystems, Software Testing

ACM Reference Format:
Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. 2020. Using Others' Tests to Identify Breaking Updates . In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387476

1 INTRODUCTION

Today's software systems are large and complex. Many of these software systems are not built from scratch, but rather leverage others' code that has been built in the past to accelerate their own development. One particular driver of this code reuse is the growing popularity of software ecosystems such as *Node.js* Package Manager (`npm`),[1] which provides a platform for developers to share their own and use others' code. Thus, developers commonly publish their reusable code as packages on `npm`, which can be used in current and future projects developed by members of the `npm` ecosystem [35].

Code reuse has many advantages, including allowing software systems to be developed faster, include richer features, and even achieve higher quality [1, 2]. However, this often comes at an increased cost of having to manage these dependencies [21]. Specifically, as the software evolves (and its dependencies do as well), updating these dependencies can become more risky [5, 9, 12].

The question of whether one should update to the newest released version is an important development decision. On the one hand, updating means that developers will get the newest features and important patches [8, 11]. On the other hand, the fear of an update breaking existing functionality often lingers on the minds of developers, making them resort to version pinning their dependencies, or other suboptimal solutions [10, 19, 38].

To ensure the stability and quality of newly released dependencies, developers often run their own tests. This has proven to be a good solution and some tools (e.g., Greenkeeper[2]) support the

[1] https://www.npmjs.com
[2] https://greenkeeper.io

automation of such approaches. However, in many cases, developers are still forced to "roll back" updates to packages because they introduce regression in their system functionality. Indeed, Mirhosseini and Parnin [21] found that there is a need for new techniques to increase the confidence in automated dependency updates.

To tackle the aforementioned issues, we set out to *leverage knowledge from the crowd to provide insights about the risk of a newly released version of a package*. Specifically, we propose a technique that runs the tests of *other* projects that depend on a specific version and use their test outcome(s) as crowd-sourced indicators of the risk of adopting a newly released package.

The technique runs tests from dependent projects before and after updating a target dependency from a prior version to a newer version. Unless an update is intentionally breaking backwards compatibility (e.g., a major release), the tests from the prior version should continue to pass in the newer version [25, 30].

To detect breakage-inducing versions, we execute the tests of dependent projects that depend on the prior version of the target dependency. For those tests that pass on the prior version, we re-execute them after updating the target dependency to the newer version. Tests that pass the execution on the prior version but not the execution on the newer version may indicate that the newer version has introduced a breakage.

To evaluate the proposed technique, we perform an empirical study of ten cases where an upgrade was rolled back because of a breakage-inducing version. Our study evaluates: 1) the coverage of the tests from other dependent projects and 2) the ability of the technique to indicate potential problems with a newer version of a target dependency.

We find that the tests from other dependent projects have varying test coverage, and in some cases, this coverage can be as high as 55%. Also, we find that of the 10 cases where a dependency was rolled back, tests from other dependent projects were able to indicate a failure 60% of the time.

Our work makes the following contributions:

- We propose an approach to detect breakage-inducing versions of third-party packages by leveraging tests from "the crowd".
- We perform an empirical study of ten cases of real word breakage-inducing versions to demonstrate the effectiveness of our approach.
- We make our dataset publicly available to facilitate further research [24].

Paper organization. The remainder of this paper is structured as follows. We start by describing the background information using a motivational example in Section 2. Section 3 provides an overview of the study design. Section 4 presents the results of our research questions. We discuss our results in Section 5. The related work is presented in Section 6. Section 7 presents the threats to validity of our study. Finally, Section 8 draws conclusions.

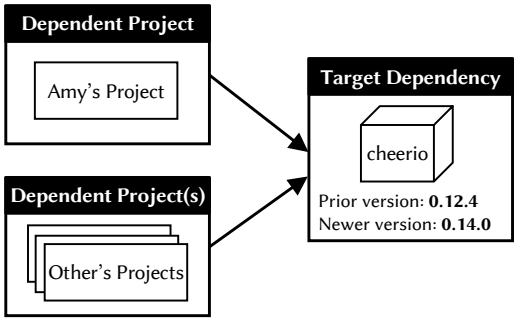

Figure 1: Motivating example overview and used terminology.

2 BACKGROUND AND MOTIVATING EXAMPLE

To help illustrate how our approach works and the challenges of updating the dependencies in the context of the `npm` ecosystem, we provide a simple motivating example.

Amy is as a web developer that is responsible for developing and maintaining web applications for three projects in her company. Her projects depend on open source projects from `npm` to leverage backend and frontend functionalities for her company's applications. Each of the applications uses on average 50 `npm` dependencies. As with many packages on `npm`, the dependencies she uses get updated frequently. Amy wants to be more proactive in managing her software dependencies, so she uses Greenkeeper, a tool that automatically checks for dependency updates. If a dependency has a newer version available, Greenkeeper updates the dependency to the newer version and runs the tests that Amy wrote for her application. If the newer version passes the tests, Greenkeeper creates a pull request to update the dependency.

One day, Amy started to receive complaints from her application's users about unexpected behaviour. When she debugged the issue, she found that a recent change that she made by updating to a newer version (`0.14.0`) of the `cheerio` dependency introduced the issue. Even though all tests passed, the tests that Amy wrote were not able to detect the breakage behaviour in the updated dependency - the tests simply did not cover the case causing the unexpected behaviour. The immediate solution was to rollback the dependency update to the prior version (`0.12.4`). Even though this procedure fixes the issue, Amy starts to become concerned about breakage-inducing versions. Through a quick web search, Amy finds that she is not the only one that suffers from this breakage-inducing version problem.

Our proposed technique aims to help developers like Amy, be more confident when they update their dependencies. In this example, Amy's tests did not detect the breakage-inducing version, however, as we illustrate in Figure 1, Amy is not the only one that uses the target dependency. Other developers also use the same dependency in their projects. If Amy's tests failed to detect the breakage-inducing version, other's tests may have potentially caught that breakage-inducing version. When a newer version of a dependency is out, why not wait until other developers update to the newer version and based on their test results determine whether

or not we should update. If the newer version breaks others' code, there is a high chance that it may break Amy's code as well.

Our technique simulates something similar, but at a very high level. Rather than waiting for others to update, we update the target dependency from the prior version to the newer version for the dependent projects that use the target dependency and check whether it breaks their tests or not. If the update breaks the tests, we flag the newer version of the target dependency as a breakage-inducing version. Even though when we mark a version as a breakage-inducing version, it may not mean it will be a breakage-inducing version for every target dependency, however it means that newer version might be risky since it broke other's tests. Hence, Amy, for example, should be careful when she wants to update to this specific newer version.

3 STUDY DESIGN

In this section, we discuss the main data used in our study and its collection process.

3.1 Corpus of Candidate Packages

Since the main goal of our study is to detect breakage-inducing versions of packages in npm, we first collect a large dataset of packages that are published on npm. Even though, the main intent of packages published on npm is to be used as third-party libraries by other JavaScript projects, these packages also depend on other packages to perform their tasks. To perform our study, we retrieve the list of all packages published on npm thorough its registry [26]. We were able to collect a total of 664,204 of npm packages as March 29th, 2018.

We choose to study packages on npm that are written in JavaScript since 1) we manually examine the source code changes of some packages and to give us confidence, we choose a programming language that the authors have expertise in, 2) JavaScript is one of the most popular programming languages on GitHub and also npm the most growing packages management systems in recent years [12, 34]. In addition, npm has a well structured software ecosystem with a large amount of packages.

That said, it is essential to highlight that our approach is not language or platform dependent and can be applied on dependencies written in any languages and published on any dependency ecosystem. Figure 2 illustrates the steps used to build our data corpus. We describe each step in more detail next.

Apply Data Filtering (Step 1). After obtaining the list of 664,204 packages, we want to analyze the commit history and then use the packages' tests to detect the breakage-inducing versions. However, the suggested/common practice on npm is to exclude the tests and non-production code files from the published packages [27]. To recover the missed data, we rely on the `git` repositories of the packages to retrieve their test code and their development history. Thus, we filter out packages that do not have a git repository. We found 391,553 npm packages in our dataset that have a valid link to their GitHub repository.

To eliminate immature and dummy packages, we filtered out packages that have less than two commits that touch the `package.json` file, which is the dependency configuration file. It is worth mentioning that this filtering process is important to allow us to keep only packages that do update their dependencies. After applying this filter, we were left with 290,417 repositories to analyze and use as our set of dependent projects.

Once we obtain the lists of 290,417 GitHub repositories, we extracted their dependencies and tracked all changes that the developers performed on their dependency versions. Specifically, we tracked all commits that touch the package configuration file (`package.json`), which we explain next.

Extract Dependencies and Versions (Step 2). Since our approach relies on identifying dependent projects to test the candidate update of a dependency, we need to identify the dependencies of the dependent projects. However, dependencies and versions can change across the history of a project. Thus, we want to collect these changes for two reasons: 1) to extract dependency downgrade cases, which indicate problematic updates i.e., breakage-inducing versions, and 2) to build a precise dependency graph between the dependent projects and the dependencies based on different points in the history, which will be used later to select the dependent projects.

In order to extract the dependencies and their versions across the history of a project, we analyze all commits that touch the `package.json` file, which is a file that npm use it to recognize the package dependencies and its versions. We use the GitHub GraphQL API[3] to collect all commits that touch the `packages.json` file for each project in our dataset. As a result, we collected 4,200,936 commits that touch the package configuration file. On each commit, we retrieve two versions of the `packages.json` file, one showing the file before the commit ($File^P$) and the other showing the file after the commit ($File^C$). We parse the files and extract the dependency list from $File^C$. For each dependency in $File^C$, we extract its version from $File^C$ and $File^P$. At the end of this process, we were able to extract more than 53,019,774 dependency records across the history of the projects.

Identify the Explicit Versions (Step 3). Developers of JavaScript projects usually do not specify the explicit version number for each of their dependencies. Instead, it is popular to use version ranges for their dependencies. Hence, we cannot link these dependency ranges to a specific version. In such cases, it is not possible to pin point the exact dependency version that was used. For example, if the range is `1.2.x` and the latest version is `1.2.1`, npm will point the dependency to this version. Later, if a newer version, e.g., `1.2.2`, is released, npm will point the dependency to it, and so forth. npm ensures that the updates respect the version ranges specified by developers. For example, if the newer version is `1.3.0`, then npm will not update to it since it does not satisfy the specified version range `1.2.x`.

Thus, to identify the version of a dependency that was used to satisfy a version range, we map version ranges to the latest satisfied version that is released before the date of the commit that introduces $File^C$. In order to perform this step, we: 1) replicated the npm registry locally; and 2) built a registry proxy that takes the commit date as an argument and simulates the registry as if it was at that specific date. Then, we use the built proxy to intercept the result from the npm registry and remove versions that are

[3] https://developer.github.com/v4/

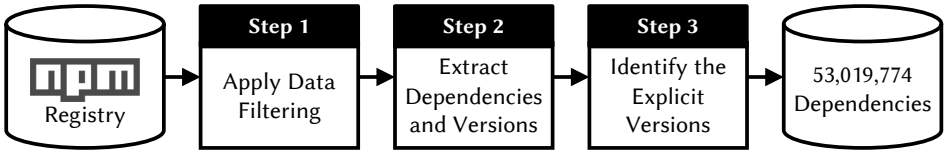

Figure 2: Data collection overview

Table 1: The Selected Ten Downgrading Cases.

Package	Downgraded From		Downgraded To	
ESLint	2.4.0	(^2.2.0)	2.2.0	(~2.2.0)
Express	3.4.0		3.3.4	
Express	4.2.0	(^4.0.0)	3.4.8	(~3.4.x)
Intl.js	1.2.5	(^1.2.4)	1.2.4	
jQuery	2.2.0	(^2.1.1)	2.1.4	(~2.1.4)
Marked	0.3.5	(^0.3.2)	0.3.3	
Marked	0.3.9	(^0.3.6)	0.3.7	
Nodemon	1.12.1	(^1.11.0)	1.11.0	
Passport	0.3.0	(^0.3.0)	0.2.0	
Request	2.83.0	(^2.53.0)	2.81.0	

newer (i.e., come after the date of the analyzed snapshot). The proxy helps us simulate the status of npm result at the snapshot time (commit date). Using this approach, we were able to determine the exact version of each dependency, which we later use to determine dependency down grades and provide us with a precise list of dependent projects.

3.2 Selection of Case Studies

In order to examine the practicality of our proposed approach, we want to extract a baseline of breakage-inducing versions. Since the normal behaviour is upgrading the dependencies, a dependency downgrade can be a perfect indicator of unusual behaviour i.e., upgrades that break the tests of the dependent projects. Thus, in this study, we resort to use the downgraded cases to select our studied breakage-inducing versions that have cases of breakage-inducing versions

For every commit in our dataset, we compare the explicit versions for the ranges extracted form $File^P$ and $File^C$. If the explicit version of the dependency on $File^P$ is greater than the explicit version on $File^C$, we consider this as a downgrade case. We were able to identify 9,046 possible breakage-inducing versions from 3,255 npm dependency packages by analyzing the commits form their dependent projects and detect dependency downgrades.

To isolate the downgrade behaviour from other changes, we only kept commits that do not perform any other changes besides the dependency downgrade change. In other words, we select downgrade cases where the commit only changes one line, which is the line that changes the dependency version. By adding this constraint, we were left with 1,880 possible breakage-inducing versions from 909 npm dependency packages.

In addition, to make sure the process correctly identifies cases of downgrade versions, the first two authors also performed a sanity check of randomly selected 100 downgrading commits by checking the commit messages and examining the `packages.json`. In all cases, the commit messages confirmed our results that the commits were only downgrading the dependencies.

Since the number of identified packages is a large number and it does not make sense to examine all of these cases, we decide to focus our analysis on cases that we can manage to analyze manually and perform an in-depth analysis. To evaluate our proposed approach using different real-word npm dependencies and breakage-inducing versions, we randomly selected ten downgrade cases to be used as the baseline in the evaluation of our proposed technique. In the selected cases, we consider the prior versions that the developers downgraded from as the breakage-inducing versions and the newer versions that they downgrade to as the stable versions.

Table 1 presents the ten randomly selected cases. The second column shows the breakage-inducing versions that the developers downgrade from as it was specified in $File^P$ (version ranges is shown in brackets). In the third column, the table shows the versions that the developer downgrade to, as specified in $File^C$ (version ranges is shown in brackets). Table 1 shows that the selected cases belong to eight npm dependency packages that are well-known and commonly used within the JavaScript developer community.

3.3 Detection of Breakage-Inducing Versions

To detect a breakage-inducing version, we rely on running the tests of projects that depend on the prior version before and after updating the target dependency to the newer version. We argue that the tests of the dependent projects can reveal the breakage-inducing versions. To do that, we propose an approach that is composed of three main steps that include, 1) identify the projects that use the prior version of the target dependency, 2) prioritize the dependent projects to run sufficient tests, and 3) automatically run the tests of dependent projects. Figure 3 presents our proposed approach and next, we explain these steps in more details.

Identify Dependent Projects. To identify dependent projects that have candidate tests for our selected ten breakage-inducing versions, we use the records of explicit package dependencies that we explained earlier in Section 3.1. To checkout the code on a specific point in history, i.e., where it depended on a prior version, we retrieve all commits that point to that prior version. In most cases, a project's repository history can have several commits that point to the same version. In such case, we choose the first commit that introduces the prior version. Then, we checkout the work directory based on that commit. We were able to find 5,853 dependent projects that use the prior version of the selected cases. Finally, we want to

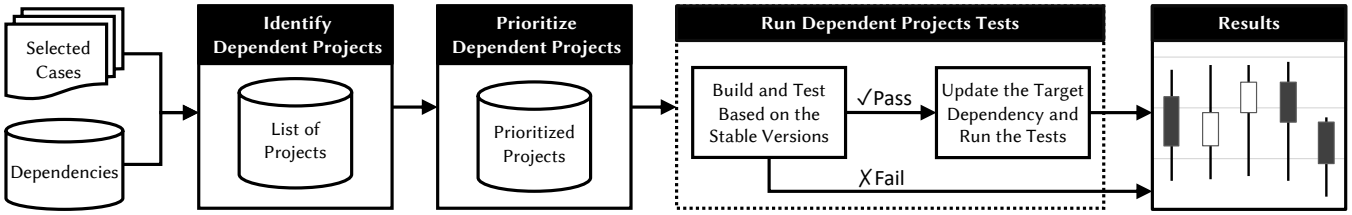

Figure 3: The approach overview.

exclude projects that do not have tests. To do so, we examine the the `package.json` file of each project and check whether it specifies a test script. This process left us with with 3,473 dependent projects that expose test scripts. Later, we use these scripts to run the the tests.

Prioritize Dependent Projects. The number of dependent projects can scale to thousands of projects. Building all of them may add no value. Therefore, in practice, a budget of number of builds needs to be specified, which will impact how many dependent projects projects one can consider. To include the most valuable dependent projects, for each breakage-inducing version in our case studies, we order its dependent projects in a queue based on their test coverage percentage. To retrieve the test coverage of the dependent projects, we rely on the API of the `npm` search engine (*npms*).[4] If more than one package has the same test coverage percentage, we prioritize the package that has a higher ranking score in *npms*. The *npms* scores are based on quality, maintenance and popularity - more details about how these scores are calculated can be found on *npms* [28].

Run Dependent Project's Tests. To detect breakage-inducing versions, we need to build the dependent projects, which include installing their dependencies and running their tests. To perform this process, we build the dependent projects in an isolated environment using Docker containers. Our implementation keeps a record of the output for every stage, the time that each stage spent and the detailed test coverage reports. We achieve this by performing the following.

First, for each prior version of our selected cases, we build and run tests of its dependent projects. Our proposed approach relies on builds and tests of dependent projects that pass the prior version. The build requires installing the dependencies. However, the fact that developers can specify version ranges can be an additional point of failure. For example, a dependency could have a newer version that break backward compatibility. If a newer version satisfies the specified version range, our build will install the newer version which is incompatible. To mitigate the problem, we used the registry proxy that we implement (Section 3.1) to emulate the registry as it was on the commit date of $File^C$. In this case, our replayed build(s) will install the version or versions that were available before the commit date. The dependent projects whose tests are already failing on the stable version are not useful since they do not provide useful information (and would not provide useful information in a real-life scenario). Therefore, we exclude dependent projects whose builds fail on the prior version. In our case study, we use a budget of 80 successful builds to be the limit. This means that if a target dependency passed tests of 80 dependent projects, we stop running more test and flag its newer version as non breakage-inducing version. At this budget, we built 1,447 dependent projects from the 3,473 projects in our dataset. Out of all builds we were able to successfully build 904 cases.

Second, for dependent projects that passed the previous building and testing stage, we update their prior version of the target dependency to the newer version. We run the same tests from the dependent projects based on the newer version and save the result. Then, we examine the saved results and if a test failed after updating the target dependency from the prior version to the newer version, we flag that version as a breakage-inducing version. This is meant to be reported to developers in an effort to help them adopt a more data-driven decision about updating to a newer version of their dependencies.

4 CASE STUDY RESULTS

In this section, we present the results of our empirical study with respect to our two research questions. For each research question, we present our motivation, approach, results and implications.

RQ1. Do the tests of dependent projects complement each other in terms of coverage?

Motivation. Previous work showed that JavaScript tests tend to have low coverage [14]. Therefore, we would like to know if better test coverage can be achieved by considering the tests of dependent projects. Achieving improved coverage using such tests would suggest that our technique has the potential to detect additional breakage-inducing versions. In this research question, we examine whether the tests of the dependent projects contribute to improving the test coverage of a package that they depend on.

Approach. To answer this question, we use an approach that depends on measuring how the tests of dependent projects contribute to cover a target dependency. We use the `Istanbul`[5] tool to measure test coverage. We configure the tool to track the test execution for the target dependency. After running all the tests of the dependent projects, we collect the detailed test coverage reports. Then we aggregate the test report based on the paths of the covered files, including the percentages of statements, branches, functions, and lines. In cases where the same file is covered by test code of more

[4] https://npms.io

[5] https://istanbul.js.org

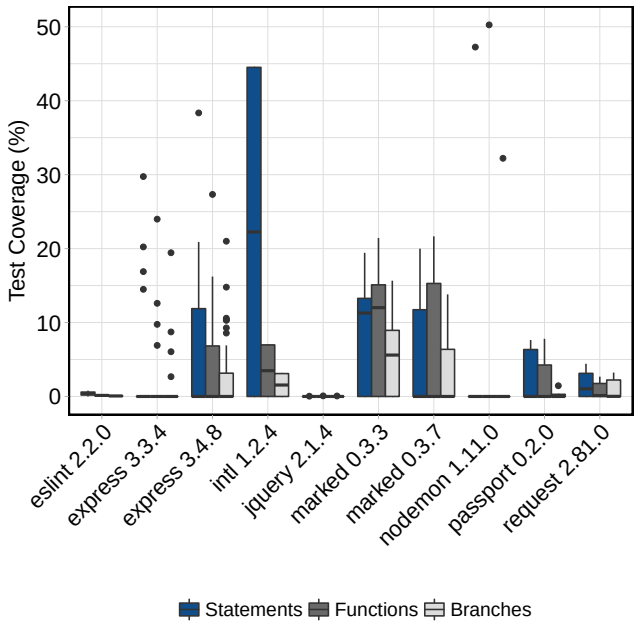

Figure 4: The distribution of test coverage for the studied cases based on running the tests of their dependent projects.

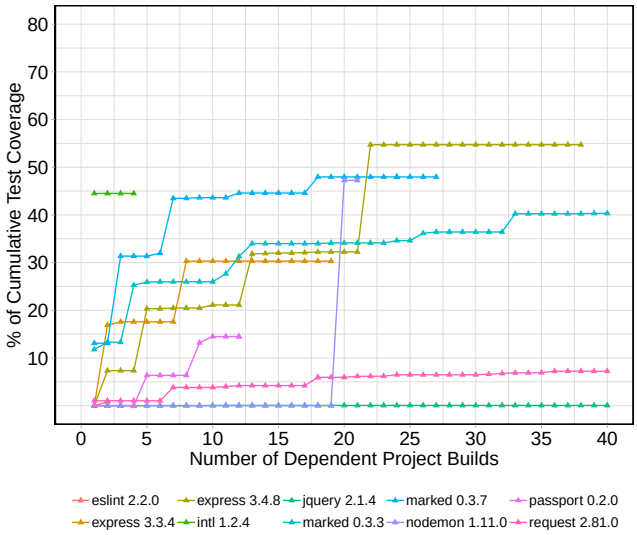

Figure 5: The cumulative statement test coverage for the selected ten cases based on running the testes of their dependent projects.

than one dependent project, we aggregate based on the coverage map of the file, i.e., we check the coverage map for each statement, branch, and function, if an element is covered in one report but not the other, we consider it as a covered element, so we count it only once. Finally, we measure the increase in test coverage based on the order of dependent projects that we produce using the prioritization described in Section 3.1.

Results. Figure 4 shows the distribution of test coverage for each selected case based on statements, functions and branch test coverage. The tests of dependent projects individually covered, on median, up to 22% statement test coverage of the target dependency's code. However, in one particular case (`nodemon 1.11.0`), we found one dependent project that covers approximately 50% of its code. Figure 4 also shows that there is a case (`jquery 2.1.4`) where crowd-based testing does not improve test coverage.

We also examine the effect of the number of dependent projects on the amount of test coverage that they provide. Figure 5 shows the cumulative statement test coverage achieved by running the dependent projects' tests. Overall, we observe that adding more dependent projects increases the statement test coverage of the target dependency. Figure 5 shows that eight dependencies have an increase in the test coverage when we increase the number of dependent projects. However, the trend of statement test coverage of most of the dependencies remains stable after running the tests of approximately 20 dependent projects.

Moreover, we examine the degree to which test coverage is improved by adding crowd-based test results. Figure 6 shows the cumulative statement test coverage when the tests of the target dependency itself and the dependent projects' tests are combined.

Overall, we see that in the majority of the cases (9 out of 10) there is an improvement in the statement test coverage. In fact, in some cases, the cumulative coverage reaches as high as approximately 80%. However, in one specific case (`jquery 2.1.4`), we see that there is no improvement, we investigate the case and we found that to run its tests, we need to setup a local server that supports PHP [16].

It is important to note that the coverage in this *RQ* is measured in terms of the covered statements. In addition, we also compared the percentage of covered statements, branches and functions and we did not observe a noticeable difference between them.

Implications. The results show that the tests of dependent projects individually and cumulatively can cover the target dependencies up to 47% and 55%, respectively. These results highlight the importance of using the dependent projects to improve the capacity for detection of breakage-inducing versions. Such an approach could also improve test generation tools to produce more effective tests in the context of the ecosystem dependency network.

> ***The dependent projects' tests can individually cover 23% on median and up to 47% of the code for the target dependency. However, leveraging the tests of the dependent projects can cover up to 55% of target dependency code.***

RQ2. How effectively can the proposed technique detect real-world breakage-inducing versions?

Motivation. In the previous research question, we found that dependent projects can provide tests that cover up to 55% of target

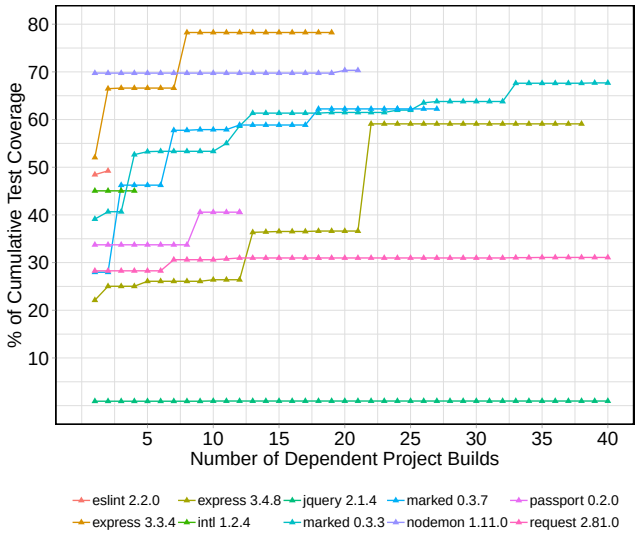

Figure 6: The cumulative statement test coverage for the selected ten cases based on running the testes of their dependent projects companied with their own tests.

dependency code. In this research question, we set out to see if using tests provided by dependent projects can catch real-world breakage-inducing versions.

Approach. To examine the effectiveness of the proposed approach in detecting breakage-inducing versions, we perform an experiment using the ten studied examples of breakage-inducing versions that are shown in Table 1. For each case, we build and run the tests of dependent projects using the prior version and once again based on the newer version using our approach described in Section 3.3. Cases where tests pass on the prior version(s), and have at least one failure on the newer version are flagged as breakage-inducing versions.

Results. The build results for all cases are shown in Table 2. Of the 904 successful builds, 314 builds passed the tests on the prior version. For each case, the second column shows the number of builds from dependent projects that proceed to the building stage. The third column shows the percentage of them that have a successful build, which range between 37.9% and 93.6%. In the fourth and fifth columns, we present the count and the percentage of dependent projects that passed the tests before updating the target dependency. Finally, the sixth column shows the percentage of dependent projects that failed the tests after updating the target dependency, which we use in the seventh column to flag if the newer version is a breakage-inducing version or not.

Table 2 shows that our proposed technique detects six of the ten studied breakage-inducing versions. For the four cases that our techniques failed to detect, we performed a manual investigation to gain insight into the reason why our proposed technique was not able to detect these cases.

`eslint 2.2.0`: ESLint is a tool that is used to identify and report problematic patterns or code that does not adhere to style guidelines in JavaScript code [37]. The tool is mainly used as a development dependency, which is not used in the production code. In our approach, we select the dependent projects based on their production dependencies. As a result, we were left with a small number of dependent projects.

In addition, the dependency package has the lowest percentage (9.5%) of passed tests from its dependent projects in the first testing stage. Out of the 19 dependent projects that had their tests fail, 16 of them produce the following error (`Cannot find module 'internal/fs'`). This error has a known workaround in the *Node.js* community. Applying this workaround may have helped, if it was known and applied in advance. Thus, we were left with only two dependent projects to test the update based on. Unfortunately, these two dependent projects only improve coverage by 0.8%, and thus, are unlikely to detect the breakage-inducing version.

`intl 1.2.4`: Intl.js is a package with five years of development history. The package is mostly used in client-side web applications to support legacy web browsers. Since web applications are not reusable dependencies by themselves, developers usually do not publish them on `npm`. Since we only used the dependent projects that are published on `npm` without considering dependent projects outside `npm`, we could only find 19 dependent projects for the target dependency version. Out of the 19 dependent projects, only four of them had their tests pass on the prior version before updating to the newer version. Including dependent projects in addition to the ones from `npm` (e.g., GitHub or Bitbucket) can help to increase the population for this case. We plan to investigate this in future work.

`jquery 2.1.4`: jQuery is a JavaScript library designed to simplify the client-side scripting of HTML. The package is mainly used to manipulate the Document Object Model (DOM) in web browser environments. Previous work showed that the DOM makes it hard for developers to test effectively [3, 15, 22]. Our result confirms the finding of a previous study by Fard and Mesbah [14], which shows that DOM-related tests lack proper coverage. In the case of `jquery 2.1.4`, the library test suite itself does not achieve any test coverage and also the dependent projects do not improve test coverage. This is due to a missing configuration setup (see RQ1). Hence, our approach cannot detect any breakage-inducing versions that is not covered by the test suites of the dependent projects.

`nodemon 1.11.0`: This case has 21 dependent projects that passed the tests of the prior version. However, all of them also passed the tests after updating the newer version. We investigated the case to figure out why our approach did not flag the case as a breakage-inducing version. By checking the commit messages for changes that the developers downgrade from the newer version (*nodemon 1.12.1*) to the prior version (*nodemon 1.11.0*), we find that developers downgrade to the prior version to maintain backward compatibility with older JavaScript standards (*ECMAScript 5* [32]). The following quote is an example of a commit message.

"Restrict version to pre-1.12 as it includes a dep requiring const"[6]

In other words, the newer version of the target dependency depends on a language feature that is not available in older JavaScript standards (*ECMAScript 5*). In our experimental setting, we only

[6]https://github.com/CoderDojo/cp-users-service/commit/59543709173c3af56baa216318cc4c954639d73b

Table 2: Builds and Tests Summary.

Cases	Number of Builds	Succeessful Builds (%)	Passed First Tests (#)	Passed First Tests (%)	(%) Failed After the Dependency Update	Caught as Risky Version
`eslint 2.2.0`	34	61.8	2	9.5	0	No
`express 3.3.4`	63	54.0	19	55.9	15.8	**Yes**
`express 3.4.8`	196	45.9	38	42.7	5.3	**Yes**
`intl 1.2.4`	19	63.2	4	33.3	0	No
`jquery 0.3.3`	364	37.9	42	30.4	0	No
`marked 0.3.3`	214	67.8	64	44.1	4.7	**Yes**
`marked 0.3.7`	73	83.6	28	45.9	7.1	**Yes**
`nodemon 1.11.0`	98	77.6	21	27.6	0	No
`passport 0.2.0`	74	52.7	13	33.3	7.6	**Yes**
`request 2.81.0`	312	92.6	83	28.2	2.4	**Yes**
All	**1,447**	**62.4**	**314**	**34.7**	**4.1**	**60%**

use the latest version of *Node.js*. Our experiment runs on the *ECMAScript 6*. Hence, downgrades that are performed due to incompatibility with *ECMAScript 6* cannot detected. Note that this is a limitation of our experimental setup and not our approach. In theory, if one were to extend the experimental configuration to include *ECMAScript 5* environments, our approach would detect such cases.

Implications. With respect to the mentioned limitations, the results show that our technique is capable of detecting breakage-inducing versions in six of the ten real-world examples. The developers of both of the dependent projects and the dependencies themselves can benefit from our technique. Developers of dependent projects can use the approach to examine their dependency versions before applying the updates. Similarly, dependency developers can use the approach to check if version updates are likely to introduce regression into their codebases.

> ***The proposed approach was able to detect six of ten real-world breakage-inducing versions. However, our technique needs to have enough dependent projects.***

5 DISCUSSION

In this section, we discuss various aspects of our technique and how they might impact the technique's outcomes.

5.1 Technique Scalability

The first research question suggests that using more dependent projects to test a target dependency can extend the test coverage of the target dependency, which increases the chance to detect breakage-inducing versions. However, running these test cases, especially when there is a large number of dependent projects, can introduce a large overhead.

To investigate the scalability of our proposed technique, we perform an analysis to understand the time required to run the tests. To do so, we calculate the time required for tests of each dependent project and compare it to the percentage of dependent projects that we built. Figure 7 shows the distribution of time consumed to pass or fail the builds in our case study. We observe that the majority of passed builds consume more time than failed ones. Also, Figure 7 shows that the consumed time when the build pass is 66 seconds on average (median = 9).

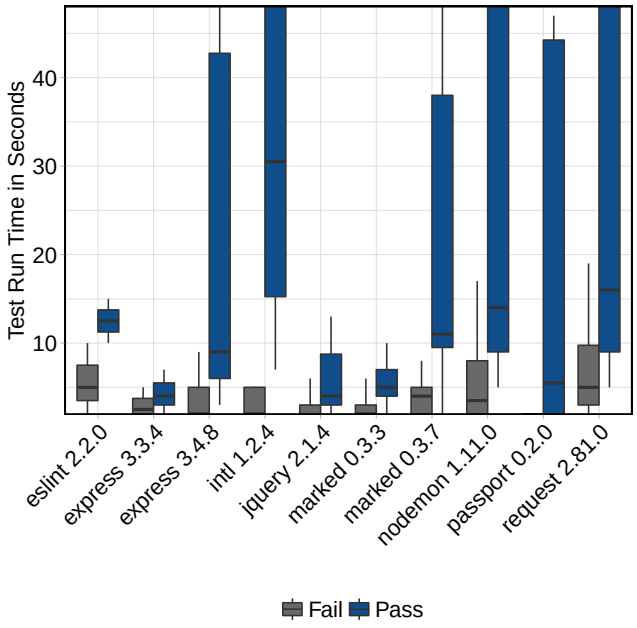

Figure 7: The distribution of the time that tests consume to pass or fail the builds.

Moreover, Figure 8 shows the accumulation of builds over time. Based on this, we observe that 90% of the builds consume less than 50 seconds. However, some builds consume over 890 seconds. Thus, setting a time limit for running builds can reduce the overall consumed time. For example, in our cases study, considering a time limit of 50 seconds of the total time, we can build 90% the candidate cases.

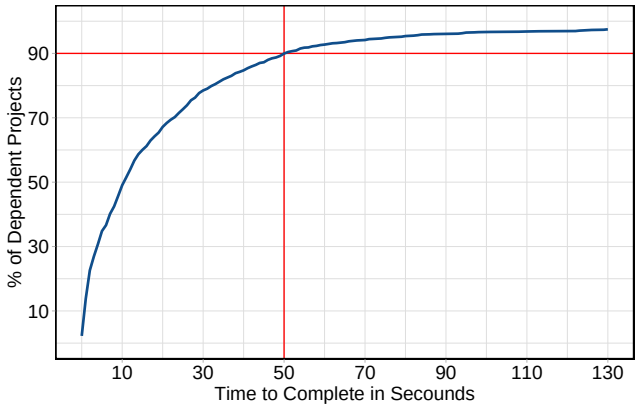

Figure 8: Time in seconds to the cumulative percentage of dependent projects that completed their tests.

5.2 Failed Builds

The results in Section 4 show that we were not able to build 37.6% of the candidate builds. Thus, we want to investigate the reasons behind the failed builds. To do so, the first two authors reviewed the logs of the failed builds and setup four classification categories. Then, they manually classified all logs and extracted the main error message. Based on the manual classification of each build log, we wrote specific regular expression to ignore the variable parts of the error messages. Then, we executed the regular expression to catch similar builds failures and classify them.

The result of the classification process is shown in Figure 9. In total we classify 543 failed builds. The most common reason (40.8%) is the failing in satisfy the dependencies. The next more frequent reason (22.2%) is missing a JavaScript environment requirement. For example, some projects depend on `Yarn`[7], which is a dependency manager that uses the `npm` registry to retrieve the dependencies. For such cases our setup fails to build and run the tests successfully.

In future work we are planning to mitigate some of these issues by considering the build configuration of the continous integration systems, if possible. For example, some of the projects use `Travis CI`[8], such projects include a configuration that specify pre steps and environment requirements for the build. For such cases we could satisfy the missed build requirements and increase the successful build percentage.

6 RELATED WORK

In this section, we present the work most related to our study. We divide the prior work into two main areas; work related to the study of API breakage changes and API testing.

6.1 Studying API Breakage Changes

Several studies investigated API evolution and stability and proposed techniques to detect breakage changes [13, 17, 23, 36]. Recently, Xavier et al. [36] performed a large-scale analysis on 317 real-world Java libraries with 9K releases, and 260K client projects.

[7]https://yarnpkg.com
[8]https://travis-ci.org

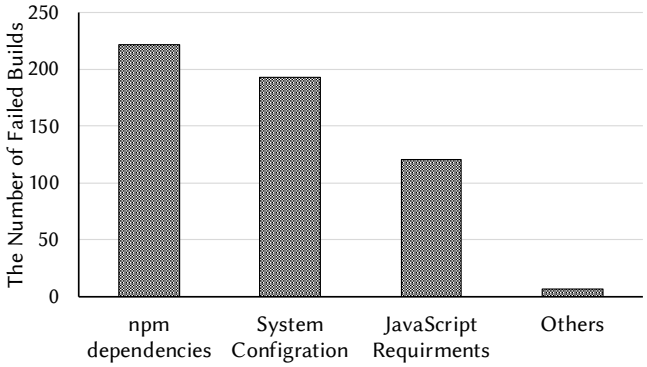

Figure 9: The classification of the failed builds

Their results show that 14.78% of the API changes are incompatible with previous versions and 2.54% of their clients are impacted. They also found that libraries with higher frequency of breaking changes are larger, more popular, and more active. Bogart et al. [4] empirically studied three software ecosystems, including `npm`, and found that fixing bugs, efficiency improvements, and addressing technical debt are the main reasons for inducing breakage changes API. Also, Businge et al. [6, 7] studied Eclipse interface usage by Eclipse third-party plug-ins and evaluate the effect of API changes and non-API changes. Dig and Johnson [13] proposed a catalog of API breaking changes and non-breaking changes. As a result, they found that 80% of the changes that break dependent projects are related to refactoring tasks.

Raemaekers et al. [29] investigate the use of semantic versioning in Java libraries. They found that breakage changes are prevalent in Java libraries. Zhong and Mei [39] conducted an empirical study on API usages focusing on how different types of APIs are used. Their empirical results showed that single API class usages are mostly strict orders, while multiple API class usages are more complicated since they include both strict orders and partial orders. Also, in recent work by Kula et al. [19], they studied more than 4,600 open source projects and found that 81.5% of studied projects are keeping their outdated dependencies libraries.

6.2 API Testing

Mostafa et al. [23] performed an investigation to gain insight on the behavioral backward incompatibilities of Java libraries. To do that, they proposed a method that use regression testing of 68 version pairs of 15 Java libraries, and examine more than 120 real world bugs. Their results showed that behavioral backward incompatibilities are not well understood by libraries developers and rarely documented. Kim et al. [18] propose a tool called `Remi` that predicts high-risk APIs in terms of producing potential bugs in the dependent projects. The main goal of `Remi` is to assists developers to write more test cases for the high risk APIs. Rodríguez-Baquero and Linares-Vásquez [31] present the use of 43 mutation test operations to test Node.js and JavaScript projects and leverage the `npm` platform to run test suites. They found that the proposed operations were able provide a mutation test coverage of 70.59% on average. Taneja et al. [33] proposed an automated test generation for database applications using mock objects, demonstrating that

with this technique they could achieve better test coverage. Abdalkareem et al. [1] studied the use of trivial packages on npm and found that even though developer believe that trivial packages on npm are well-test, their qualitative analysis showed that only 45% of the trivial packages have test case written for them.

The work by Mezzetti et al. [20] is closest to ours. In their work, the authors proposed a technique to detect packages that break the types of their public interface in the npm ecosystem. The study leverage the test suites of dependent projects and uses a dynamic analysis to learn models of the package interface types. Our work complements the prior work since we propose a technique that leverage tests from dependent projects to detect semantic and behavioural breakage-inducing versions of target dependency.

7 THREATS TO VALIDITY

In this section, we disuses threats to validity that might influence our study.

7.1 Threats to Internal Validity

Internal validity concerns factors that could have influenced our analysis and findings. First, to evaluate our technique, we select a sample of downgraded cases to be examined. However, downgrading the dependency can be triggered for many different reasons. Therefore the results can be affected by introducing invalid evaluation cases. To mitigate this threat, we have a restricted approach to select these cases 1) we selected cases where the commits perform only one specific change, which downgrades the dependency version and 2) we make sure that the commit message of the selected cases mentions that a dependency downgrade as the main reason for the change. Second, we randomly selected only ten cases of breakage-inducing changes. Even though this number seems to be modest, our analysis shows that using these cases we were able to systematically evaluate the practicality of our technique. Also, in the future we plan to perform a large-scale study considering the lessons learned from our current experiences.

We only use download measurement to prioritize the selected dependent projects. Other measurements could have been used, such as number of stars for the project. That said, we believe the selection of our measurement is right since it gives us a clear indication of the quality of the dependent projects (low quality projects will probably not be downloaded much). Finally, to measure the test coverage that our technique achieves through running the test from dependent projects, we use the `Istanbul` tool. Thus, our analysis heavily relies on the accuracy of the `Istanbul` tool. That said, its popularity and common usage gives confidence in our results.

In our experimental evaluation we examine ten cases and our technique was able to catch 60% of the breakage-inducing versions. This result is highly dependent on the building of dependent projects.

7.2 Threats to External Validity

Threats to external validity concern the generalization of our technique and findings. In our study, we only examine packages and dependent projects mainly written in JavaScript. Thus our findings may not generalize to other programming languages. We also examine packages published on the npm package manager and hosted on Github. However, using other dependency ecosystems may provide different result.

To prioritize the selected dependent projects that we use their tests, we rely on the measurement (i.e., number of downloads) provided by *npms*. Thus, our prioritization is heavily impacted by *npms*. That said, since *npms* is the main search engine for npm and its data has been used in prior work (e.g., [1]), these reasons gives us confidence in the data provided by *npms*.

8 CONCLUSION AND FUTURE WORK

Updating dependencies is an essential part of any software project. However, in open source ecosystems, where anybody can contribute by publishing reusable packages, the risk associated with updating dependencies could be problematic [9]. Previous work has shown that managing dependencies is one of the most cited drawbacks of using npm packages [1]. In this work, we propose a technique to detect breakage-inducing versions of third-party dependencies. The technique leverages tests from dependent projects to warn software teams about breakage-inducing versions. We evaluate our technique through an empirical study of 391,553 npm packages. We use the dependency network from these packages to identify candidate tests. We find that our proposed technique can detect six of the ten studied breakage-inducing versions. However, we can perfom better if we incloude more dependent projects.

REFERENCES

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on Npm. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE '17)*. Association for Computing Machinery, New York, NY, USA, 385–395. https://doi.org/10.1145/3106237.3106267

[2] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the impact of using trivial packages: An empirical case study on npm and pypi. *Empirical Software Engineering* 25, 2 (2020), 1168–1204.

[3] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A Framework for Automated Testing of Javascript Web Applications. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 571–580. https://doi.org/10.1145/1985793.1985871

[4] 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 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE '16)*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/2950290.2950325

[5] Christopher Bogart, Christian KÃďstner, and James Herbsleb. 2015. When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW '15)*. IEEE, New York, NY, USA, 86–89. https://doi.org/10.1109/ASEW.2015.21

[6] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. 2012. Survival of Eclipse third-party plug-ins. In *Proceedings of the 28th IEEE International Conference on Software Maintenance* (Trento, Italy) *(ICSM '12)*. IEEE, New York, NY, USA, 368–377. https://doi.org/10.1109/ICSM.2012.6405295

[7] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. 2015. Eclipse API Usage: The Good and the Bad. *Software Quality Journal* 23, 1 (March 2015), 107âĂŞ141. https://doi.org/10.1007/s11219-013-9221-3

[8] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER '17)*. IEEE, New York, NY, USA, 516–519. https://doi.org/10.1109/SANER.2015.7081868

[9] Alexandre Decan, Tom Mens, and MaÃńlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER '17)*. IEEE, New York, NY, USA, 2–12. https://doi.org/10.1109/SANER.2017.7884604

[10] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Evolution of Technical Lag in the npm Package Dependency Network. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME '18)*. IEEE, New York, NY, USA, 404–414. https://doi.org/10.1109/ICSME.2018.00050

[11] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 181–191. https://doi.org/10.1145/3196398.3196401

[12] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2018. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (10 Feb 2018), 384âĂŞ417. https://doi.org/10.1007/s10664-017-9589-y

[13] Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve&Quest; A Story of Refactoring. *Journal of Software Maintenance* 18, 2 (March 2006), 83–107. https://doi.org/10.1002/smr.328

[14] Amin Milani Fard and Ali Mesbah. 2017. JavaScript: The (Un)Covered Parts. In *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST '17)*. IEEE, New York, NY, USA, 230–240. https://doi.org/10.1109/ICST.2017.28

[15] Amin Milani Fard, Ali Mesbah, and Eric Wohlstadter. 2015. Generating Fixtures for JavaScript Unit Testing (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 190–200. https://doi.org/10.1109/ASE.2015.26

[16] The jQuery Foundation. 2019. jQuery JavaScript Library. https://github.com/jquery/jquery. (Accessed on 01/20/2019).

[17] Puneet Kapur, Brad Cossette, and Robert J. Walker. 2010. Refactoring References for Library Migration. *ACM SIGPLAN Notices* 45, 10 (Oct. 2010), 726–738. https://doi.org/10.1145/1932682.1869518

[18] Mijung Kim, Jaechang Nam, Jaehyuk Yeon, Soonhwang Choi, and Sunghun Kim. 2015. REMI: Defect Prediction for Efficient API Testing. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 990–993. https://doi.org/10.1145/2786805.2804429

[19] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2017. Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration. , 34 pages. https://doi.org/10.1007/s10664-017-9521-5 arXiv:1709.04621

[20] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *Proceedings of the 32nd European Conference on Object-Oriented Programming* (Amsterdam, The Netherlands) *(ECOOP '18)*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:24. https://doi.org/10.4230/LIPIcs.ECOOP.2018.7

[21] Samim Mirhosseini and Chris Parnin. 2017. Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-date Dependencies?. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, Piscataway, NJ, USA, 84–94. https://doi.org/10.1109/ASE.2017.8115621

[22] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. JSEFT: Automated Javascript Unit Test Generation. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST '15)*. IEEE, New York, NY, USA, 1–10. https://doi.org/10.1109/ICST.2015.7102595

[23] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. A Study on Behavioral Backward Incompatibility Bugs in Java Software Libraries. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE, New York, NY, USA, 127–129. https://doi.org/10.1109/ICSE-C.2017.101

[24] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. 2019. Dataset: Using Others' Tests to Avoid Breaking Updates. https://doi.org/10.5281/zenodo.2549129

[25] npm Documentation. 2018. How to use semantic versioning. https://docs.npmjs.com/getting-started/semantic-versioning

[26] npm Documentation. 2019. npm-registry | npm Documentation. https://docs.npmjs.com/misc/registry. (Accessed on January 21, 2019).

[27] npm Documentations. 2018. How to publish and update a package. https://docs.npmjs.com/getting-started/publishing-npm-packages

[28] npms. 2016. About npms. https://npms.io/about. (Accessed on 19 April 2018).

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

[30] 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. https://doi.org/10.1016/j.jss.2016.04.008

[31] Diego Rodríguez-Baquero and Mario Linares-Vásquez. 2018. Mutode: Generic JavaScript and Node.Js Mutation Testing Tool. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 372–375. https://doi.org/10.1145/3213846.3229504

[32] Istvan Sebestyen. 2009. Ecma International finalises major revision of ECMAScript. http://www.ecma-international.org/news/PressReleases/PR_Ecma_finalises_major_revision_of_ECMAScript.htm. (Accessed on 01/22/2019).

[33] Kunal Taneja, Yi Zhang, and Tao Xie. 2010. MODA: Automated Test Generation for Database Applications via Mock Objects. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 289–292. https://doi.org/10.1145/1858996.1859053

[34] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 805–816. https://doi.org/10.1145/2786805.2786850

[35] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A Look at the Dynamics of the JavaScript Package Ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. Association for Computing Machinery, New York, NY, USA, 351–361. https://doi.org/10.1145/2901739.2901743

[36] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering* (Klagenfurt, Austria) *(SANER '17)*. IEEE, New York, NY, USA, 138–147. https://doi.org/10.1109/SANER.2017.7884616

[37] Nicholas C. Zakas. 2018. About - ESLint - Pluggable JavaScript linter. https://eslint.org/docs/about/. (Accessed on 19 April 2018).

[38] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. 2018. An Empirical Analysis of Technical Lag in npm Package Dependencies. In *New Opportunities for Software Reuse*, Rafael Capilla, Barbara Gallina, and Carlos Cetina (Eds.). Springer International Publishing, Cham, 95–110. https://doi.org/10.1007/978-3-319-90421-4_6

[39] Hao Zhong and Hong Mei. 2018. An Empirical Study on API Usages. *IEEE Transactions on Software Engineering* 45, 4 (December 2018), 319–334. https://doi.org/10.1109/TSE.2017.2782280