Keeping The Build Up-To-Par
An empirical study on in-range breaking updates in the npm ecosystem

Benjamin J. Rombaut[1]
[1] *School of Computing, Queen's University, Canada*

***Abstract*—Software dependency relationships allow a client package to reuse certain versions of a provider package. These provider packages often release versions containing bug fixes, new functionalities, and security enhancements, so it is important for the clients to keep their dependencies up to date. Client's can specify a range of versions from providers they would like to accept, usually including small upgrades and fixes, which will automatically be installed whenever the client's project is built. However, these in-range updates can sometimes break a client's build, which is problematic as users of the client package will also automatically receive the in-range update and not be able to build the client's project. To understand the characteristics of these in-range breaking updates and how they are resolved, we examine in-range breaking build issue reports opened by the Greenkeeper bot and actions that are taken by clients to get their build passing again. We find that the majority of issues are created for patch updates, and that packages with higher total and more frequent releases tend to cause more breakages. Often, the client simply updating their dependency specifications is enough for them to resolve their build, and we found no indication that releases that break a relatively high proportion of client's builds prompts a response in the provider packages.**

Source-code Link: https://github.com/brombaut/LOG6307_Project

***Keywords*-Dependency Management, Software Ecosystems, Breaking changes**

I. Introduction

Today's software systems are large and complex. They are rarely built from scratch, and it is commonplace for developers to leverage others' code that has been built in the past to facilitate code reuse. Prior research shows that code reuse is related to the improvement of developers productivity, software quality, and time-to-market of software products [1] [2]. However, this often comes at an increased cost of having to manage these dependencies [3].

Greenkeeper[1] is a popular automated dependency management tool that GitHub[2] and npm[3] users can integrate into their projects. Each time one of their dependencies releases a new version, Greenkeeper opens a new branch with that update. If the repositories continuous integration pipeline fails with the new dependency version and the dependency release is an in-range update, Greenkeeper will open up an issue report in the client's repository with information stating which dependency update caused the problem.

These in-range updates that break client's builds should not be overlooked. Semantic versioning[4] works so that clients can specify a range of versions they would like to be able to use from a provider. This is done so that they can automatically receive simple bug fixes and patch updates as the provider releases them. If Greenkeeper is opening an issue report for an in-range update that is breaking a client's build, it means that if a user of the client's package were to download and install the package after the dependency has released their in-range update, or even simply try to update the dependencies on the client's project, it would fail to build. Therefor, it is expected that there would be a certain level of urgency from practitioners when they receive these issue reports.

In our study, we examine these in-range breaking build issues to determine characteristics that cause these build failures, as well as how client's respond to these issues and how they resolve their build. Specifically, we investigate the following research questions:

- *RQ1: What are the characteristics of in-range updates that break a client's build?*
- *RQ2: How do clients respond to in-range breaking updates?*
- *RQ3: Do broken builds in clients prompt a response from the breakage-inducing provider?*

The remainder of this paper is structured as follows. We start by describing the background information in Section II. Section III discusses the data set we used for the study. Section IV presents the results of our research questions. Section V presents the threats to validity of our study. The related work is presented in Section VI. Finally, Section VII draws conclusions and discusses future work.

II. Background

In this section, we provide a more in-depth description of dependency-relationships, semantic versioning, and Greenkeeper.

A. Dependency Relationships

Software dependency relationships allow a client package to reuse a certain version of a provider package stored in online package distributions. There are over 1.6 million packages available on npm alone[5]. These provider packages are constantly evolving, with newly added features and patches that

[1]https://greenkeeper.io/
[2]https://github.com/
[3]https://www.npmjs.com/
[4]https://semver.org/
[5]https://libraries.io/npm

fix bugs in older versions. Additionally, the availability of such a huge amount of reusable packages facilitates software development and evolution. However, dependency relationships can also cause problems, such as software becoming out of date with respect to more recent package releases, or, if the packages are kept up-do-date, there is a risk that the new version will break existing functionality, so developers may resort to version pinning their dependencies or other less-than-ideal solutions [4] [5].

B. Semantic Versioning and Dependency Constraints

With many software packages being created and updated every day, it is important to standardize the way of versioning and keeping track of package releases and dependencies. Semantic Versioning, referred as *semver*, has become a popular policy for communicating the kinds of changes made to a software package. It allows dependent software packages to be informed about possible "breaking changes". A semver-compatible version is a version number composed of a major, minor and patch number. The version numbers allow to order package releases. For example, 1.2.3 occurs before 1.2.10 (higher patch version), which occurs before 1.3.0 (higher minor version), which occurs before 2.1.0 (higher major version). Backward incompatible updates should increment the major version, updates respecting the API but adding new functionalities should increment the minor version, while simple bug fixes should increment the patch version. Unfortunately, the semantic versioning policy is not always respected by package maintainers.

npm recommends software packages to follow a specific flavor of semantic versioning[6]. Package releases specify their version number in the metadata stored in a json file[7], and use dependency constraints to specify the version ranges of other packages they depend upon. These constraints are built from a set of operators that specify versions that satisfy the range. Table I summarizes the types of dependency constraints for npm, their interpretation and an example of each constraint type.

C. Greenkeeper and In-Range Issues

As previously mentioned, Greenkeeper is a tool that practitioners can integrate with their project and has been shown to be an effective tool for dependency management. Projects that use Greenkeeper upgraded 1.6x as often as projects that did not use any tools [3]. It sits between npm and GitHub, watching the modules their repository depends on. Each time one of the dependencies is updated, Greenkeeper opens a new branch with that update. The repository's CI tests run, and Greenkeeper watches the results to see whether they pass or not. Based on the test results and the client's dependency version definitions, Greenkeeper will open an issue report in the client's repository with information stating which dependency update caused the problem. Figure 1 shows an example of an issue report opened by Greenkeeper.

[6]https://docs.npmjs.com/misc/semver
[7]https://docs.npmjs.com/files/package.json

Figure 1: A Greenkeeper in-range issue report

An in-range update means that a client will accept the update without having to change their version specification. For example, If a client specifies an accepted version range of ^1.0.0 for a dependency, and that dependency releases version 1.0.1, that update is in-range. Users of the client's package would receive this update when they run *npm install*. If an update is out of range of a client's version specification, they will not receive the update. So if an an out-of-range update breaks a client, their user's will not be effected.

III. Data Set

In this section, we discuss how we collected the data set used in this study.

A. Greenkeeper Issue Reports

We use Google Big Query[8] to find 12,134 GitHub projects that have integrated with Greenkeeper. From this list of repositories, we use the GitHub API[9] to retrieve the repository information, including the projects development and run-time dependency version specifications, as well as all of the issue reports for the project authored by the Greenkeeper bot. This leaves us with 123,197 in-range breaking build issue reports. For each of these issue reports, we collect any comments that have been made on the issue report, as well as any events that occurred on the issue report. Issue events[10] can be any actions that concerns the event, such as closing an issue or referencing the event on a pull request. In total, we collect 365,625 comments and 209,750 issue events. If there were any commits that referenced the issue report, we collect information on those commits, for a total of 17,623 commits.

B. Provider Package Releases

After each of the projects development and run-time dependencies had been extracted, we collected each of the dependencies release history from *libraries.io*[11]. In total, we collect information on 556,742 releases across 7,361 unique dependencies.

[8]https://cloud.google.com/bigquery
[9]https://docs.github.com/en/free-pro-team@latest/rest
[10]https://docs.github.com/en/free-pro-team@latest/rest/reference/issues#events
[11]https://libraries.io/

Constraint	Interpretation	Example	Satisfied Versions
fixed	exact version required	=2.3.1	2.3.1
minimal	only use releases above the declared version	>=2.3.0	$\geq$2.3.0
maximal	only use releases below the declared version	<2.3.1	<2.3.1
latest	use latest available release	latest	$\geq$0.0.0
hyphen ranges	only use releases between two versions	1.2.3-2.3.4	$\geq$1.2.3 $\wedge$ $\leq$2.3.4
x ranges	only update where "x" or "*" is	1.2.x	$\geq$1.2.0 $\wedge$ <1.3.0
tilde ($\sim$)	only update patches	$\sim$2.3.0	$\geq$2.3.0 $\wedge$ <2.4.0
caret ($\hat{}$)	only update patches and minor releases	$\hat{}$2.3.0	$\geq$2.3.0 $\wedge$ <3.0.0

Table I: Types of dependency constraints for npm package dependencies

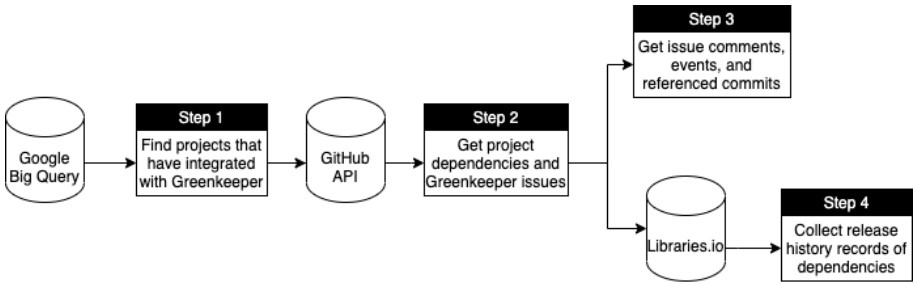

Figure 2: Data collection overview

An overview of the data collection process is shown in Fig. 2 and the data set is summarized in Table II.

Entity	Records
Projects	12,134
In-range breaking build issue reports	123,197
Issue comments	365,625
Issue events	209,750
Referenced commits	17,623
Release Versions	556,742

Table II: Summary of data

IV. Results

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

A. RQ1: What are the characteristics of in-range updates that break a client's build?

1) Motivation: In this research question, we look to see whether there are any common characteristics of packages that release in-range breaking updates, and what types of updates end up breaking a client's build. In-range breaking updates are problematic for clients and finding common features of these updates is the first step in attempting to eliminate them.

2) Approach: We first wanted to see how the in-range breaking updates were distributed across different update types. Recall that semantic versioning splits updates into three primary categories: *patch*, *minor* and *major* updates. On the breaking build issue reports, we extract the version the dependency was being updated from and the version the dependency was being updated to in order to determine whether the release was a major, minor, or patch update. We also want to compare how the number of in-range breaking updates by type compare to how often these update types are being released, as well as how often these update types are actually accepted by client version specifications.

To determine what proportion each update type has of all releases, we examine all of the versions each of the dependency to determine how many patch, minor, and major updates the package has released. We determined what types of updates clients accept from providers by extracting each client's development and run-time dependency version specifications, and determining the type of updates they accept based on the rules outlined in Table I.

Another factor we investigate was whether the release frequency of provider packages tends to have any affect on how often they release in-range updates that break their clients. We tally the number of in-range updates each provider has that broke their client's, and compare the distributions across the total number of releases that package has, as well as the frequency at which that package releases new versions.

3) Results: We were able to extract the version ranges of the breaking dependency being updates for 63.77% of the issue reports. This is because Greenkeeper will bundle upgrades for the packages that have to be upgraded together, and if one of the updates in the bundle breaks the client's build, the issue report is opened for all of the updates in the bundle. We are not able to determine the specific breaking dependency in the bundle, and we therefor omit these issue reports from our analysis.

We found that 65.15%, 34.69%, and 0.16% of in-range breaking updates are patch, minor, and major updates, respectively. These results would suggest that package developers are not following the semantic versioning scheme, which states that only major updates should be used to signify a breaking change in the package. However, patch and minor updates occur much more frequently than major updates. We found that 64.37%, 28.25%, and 7.37% of package releases were

patch, minor and major updates, respectively. This explains to some degree why the number of in-range breaking updates that are patch and minor updates is larger than major updates, since patch and minor updates occur more frequently then major updates.

Additionally, we found that major updates are far less likely to be accepted as in-range updates by clients. Recall that, in order for an update to be considered in-range, the client must specify the range of updates they are willing to accept. We found that 12.53% of dependencies are pinned by clients, which means the provider will never be able to release an in-range update. Of the remaining dependencies, 3.74%, 82.31%, and 1.08% are accepting patch, minor, and major updates, respectively, which would further reduce the number of major updates that are considered to be in-range for clients. These results are summarized in Figure 3

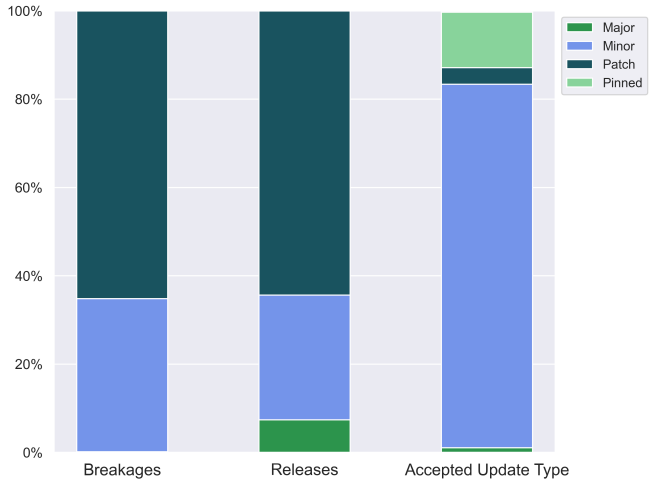

Figure 3: Distribution of update types for in-range breakages, package releases, and accepted update types

In terms of how the release practices of packages affect how often they release in-range breaking updates, we found that the number of broken builds increases by 0.07 for every additional release a package makes. Intuitively, this result is expected. The more releases a package has, the more opportunities there are that a release will break a client's build. We also look at the distribution of build breakages caused by a provider against the frequency at which they release new versions. We found that the number of in-range breakages decrease by $3.7e-10$ as the release frequency increases, which is not strong evidence that a higher release frequency is less likely to cause in-range breaking updates, or vice versa. These results are visualized in Figure 4.

While the majority of clients will accept minor updates, most breakages are caused by patch updates. Packages with higher total and more frequent releases tend to cause more breakages.

B. RQ2: How do clients respond to in-range breaking updates?

1) Motivation: As previously mentioned, in-range breaking updates should be treated with a sense of urgency by client developers, as any attempt by users to install their package after the in-range dependency update will fail. In this research question, we look at how client developers respond to these in-range updates, and what actions they have to take in order to resolve their build.

2) Approach: To answer this research question, we examine a number of attributes related to the in-range breaking build issue reports opened in the client projects. We look at the content of the issue reports themselves, the comments on the issue reports, as well as the types of changes that are made by developers in order to fix their builds, Additionally, we look at the effectiveness of Greenkeeper's automatic attempts at pinning the breaking dependency as a first attempt to resolve the client's build.

3) Results: We found that 79.82% of in-range breaking update issue reports were eventually closed, with a median time to close the issue of 4 days and 11 hours. We examine the comments on in-range breaking build issue reports in order to gain insight into how clients are responding to these issues. We found that 85.93% of issues have at least 1 comment. This is expected, as Greenkeeper will attempt to pin the breaking dependency immediately after opening the majority of issue reports, and will automatically comment the build result of pinning the dependency on the issue report. 47.95% of issues have at least 2 comments. We hypothesised that approximately half of all comments on the issue reports were from users. However, further analysis revealed that only 2.93% of comments were from non-bot users on GitHub, and that 96.89% of all comments are from the Greenkeeper bot itself, with the remaining comments being from other bots.

Specifically examining the user comments, we found that the median response time was 2 days and 12 hours. We classified these comments using regular expressions to match specific phrases and patterns. We found that users are specifically referencing a fix for the issue 34.5% of the time, while the simply mention the issue has been fixed 17.4% of the time. 18.3% of comments indicate the build failure is a false alarm, referencing a flaky test, CI hiccup, or that they simply re-ran the build pipeline and it passed without incident. The proportion of user comment types, as well as the average response times for each type are summarized in Figure 5

We saw that practitioners were referencing fixes on these issue reports, so we investigated what these fixes entail. We found that 69.7% of referenced commits include changes the *package.json* file, 28.5% include changes to the *package-lock.json* file, and 25.6% include changes to the *yarn.lock* file. These are all files that are used to specify a projects dependencies. The two lock files are automatically generated and so have a relatively high commit churn, with the *package-lock.json* file having a median commit churn of 101 additions and 130 deletions, and the *yarn.lock* having a median com-

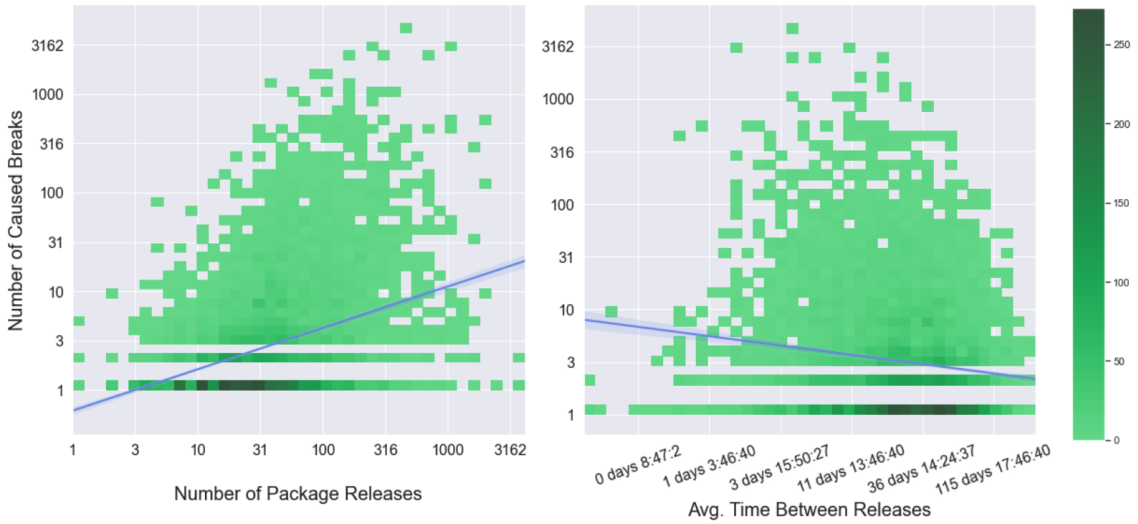

Figure 4: Distribution of number of cause breakages and total package releases (left) and release frequency (right)

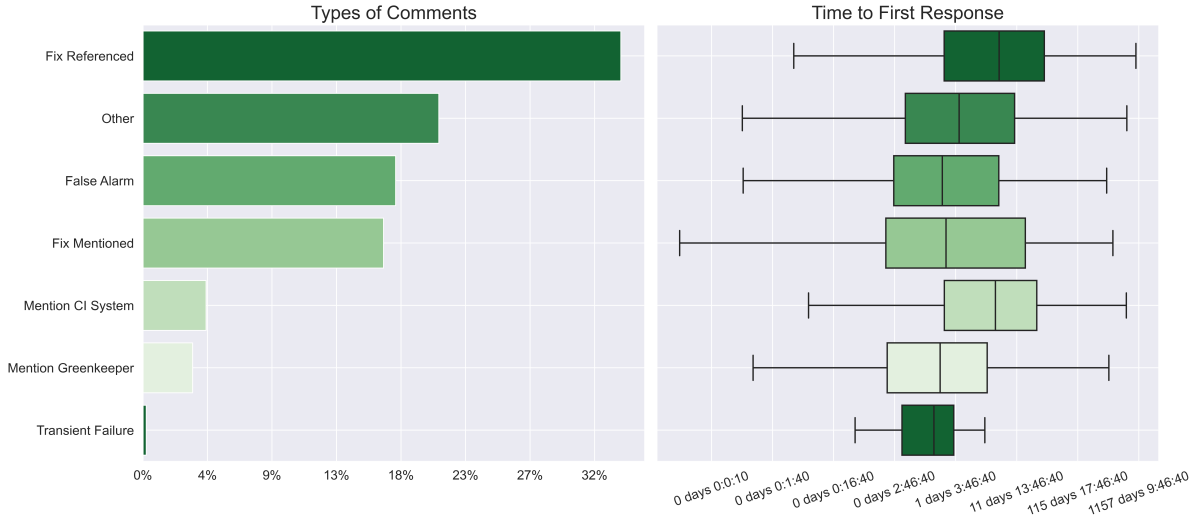

Figure 5: Most common types of comments left by users on in-range breaking build issue reports (left) and how long users take to make each response (right)

mit churn of 44 additions and 59.5 deletions. However, the *package.json* file, which is manually maintained, has a median commit churn of 1 addition and 1 deletion per commit when fixing an in-range breaking build issue. This suggests that, in order to fix their build, users are simply updating their accepted dependency version range, such as pinning the dependency that is failing the build. Figure 6 summarized the most common files changed in referenced commits on in-range breaking build issues, as well as their commit churn. Additionally, these results shows that it is uncommon for clients to modify their source code that uses these dependencies in order to fix their builds.

When the Greenkeeper bot opens a breaking build issue report, it will automatically attempt to pin the dependency to the previous version and re-run the build to determine if the issue can be resolved. Pinning a dependency is a legitimate option when developers don't have the time or resources to fix a problem introduced by a dependency update. Greenkeeper will bundle upgrades for the packages that have to be upgraded together, and it won't attempt to pin all of the packages in the bundle, which is why 28.1% of issues don't have a pin attempt. However, of the issue reports that did have a pin attempt, only 33.1% resulted in the client's build passing again. This indicates that the majority of these in-range breaking build update issue reports are not actually caused by the dependency being updated. In order to better understand the reasons as to why so many pin attempts fail, we manually analyze a sample of the issue reports that have a failed pin attempt. We found that clients improperly configuring their projects build pipeline was the primary reason for the dependency updates failing, including missing configuration files, invalid credentials, or inconsistent environment versions. We also saw

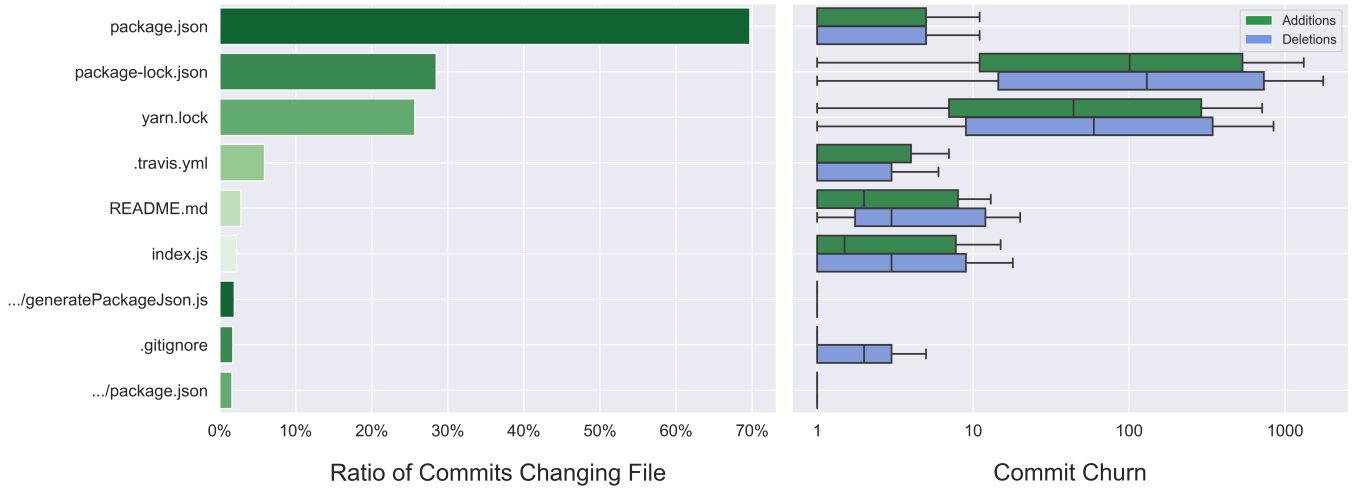

Figure 6: Most common files changed in commits referenced by in-range breaking build issue reports (left) and the commit churn of each file (right)

that the builds can fail due to flaky tests, request timeouts, and even attempting to run the build before the branch had been created in the project repository.

Additionally, we found that Greenkeeper does not take into account the type of error that occurs in the pipeline when testing new dependency updates, and will consider any error that occurs in the pipeline as a build failure. This means that, for example, if a developer that does not conform to the project's linter configuration, then all subsequent builds will fail, and Greenkeeper will open in-range breaking update issue reports for any dependencies that release a new in-range version. If the dependency continues to release new versions, the issue report will become flooded with comments from Greenkeeper confirming that the build is still failing, introducing a lot of noise into projects and distracting from valid issue reports, which practitioners say is one of the main reasons for not using automated dependency management tools [3].

> The majority of client responses mention a fix or confirm the issue is invalid. Most fixes involve small changes to dependency specification files. While pinning the dependency is a legitimate option, it is not as successful as expected due to builds failing for reasons not related to the dependency being updated.

C. RQ3: Do broken builds in clients prompt a response from the breakage-inducing provider?

1) Motivation: In the previous research question, we looked at how clients respond to in-range breaking updates. However, while it is the client's build that is failing, the root cause of the issue could originate from the provider. In this research question, we look to see whether in-range breaking updates prompt a response from the provider packages.

2) Approach: Initially, we examined all of the comments made by users on the in-range breaking build issue reports looking for potential references to issue report made in the dependency package that were related to the client's breaking build issue report. We were not successful in finding any, so we took a different approach. We hypothesis that, if a release from a package provider causes an in-range breaking builds in a high proportion of client, the next release the provider package makes would happen quicker than usual. To test this, we compare the average release frequency of packages against the time it takes a package to release the next update immediately following an update that causes an in-range breaking build in at least 20% of clients that depend on it.

First, we determine the number of clients that are using each of the provider packages we have release information on in our data set. We then determine how many issue reports were opened for each release made by each provider package. Using this information, we determine the ratio of in-range breaking update issue reports that each release made by every provider package caused. This allows us to categorize each release either as a breaking release or a non-breaking release. We then compare the release frequency of the non-breaking releases to the time to the release immediately following each breaking release.

3) Results: We first used all release records in our analysis including releases that did not cause any in-range update breaking issues in any clients. We found that the median release time after non-breaking releases was 1 day and 22 hours (IQR = 3 hours - 10 days 12 hours), while the median release time immediately following a breaking release was 8 days and 6 hours (IQR = 1 day and 3 hours - 39 days and 6 hours). From these results, it appears that the release that immediately follow an update that breaks a proportion of clients tends to take longer to be made available. However, we choose to perform further analysis on only breaking releases.

We examine only the set of releases that caused at least one in-range breaking build issue report to be opened in a

client. While all these releases caused at least one breaking build, we still classify a breaking release as a release that caused at least 20% of the package's client's builds to break. The median release time immediately following a breaking release was still 8 days and 6 hours (IQR = 1 day and 3 hours - 39 days and 6 hours), as that data set was unaffected, but the median release time after non-breaking releases was 8 days and 6 hours (IQR = 1 day 3 hours - 39 days 6 hours). These results can be seen in Figure 7.

> We did not find any indication that client's are filing issue reports with provider packages that release in-range breaking updates. There is no statistical difference that indicates that breaking releases are immediately followed up by a quicker release than normal.

V. Threats to Validity

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

A. Threats to Internal Validity

Internal validity concerns factors that could have influenced our analysis and findings. First, we did not perform any filtering of either the client projects or the provider packages we examined, such as removing inactive or "toy" projects. This may have affected our results pertaining how clients respond to the in-range breaking update issue reports, as well as the results involved with the release frequency of providers.

We only collect a single snapshot of each client's dependencies, and therefor we do not have the dependency history of each client project in our data set. In other words, we do not know at what date a client project integrates with each of it's dependencies, or whether they remove any dependencies over their lifetime. We therefor assume that a client uses each of the dependencies for it's lifetime for our analysis in Section IV-C. This may have biased our results when classifying each provider release as either a breaking or non=breaking release based on the ratio of client each released caused an issue report to be created.

B. Threats to External Validity

Threats to external validity concern the generalization of our technique and findings. First, we only analyse projects that have integrated with the Greenkeeper bot. While Greenkeeper is a popular tool used by many projects [3], they might not be representative of the general population, and while we analyzed a large sample of open-source repositories, these results may not extend to proprietary systems, which may operate under different constraints. Further, Greenkeeper was acquired by Snyk[12] in June 2020, and is no longer monitoring projects for in-range breaking updates. Additionally, one of the prerequisites for clients integrating with Greenkeepepr is that the project must have a *package.json* file specifying it's dependencies. This means that Greenkeeper is only able to monitor dependencies in the *npm* ecosystem for in-range breaking updates. Therefor, our analysis only examines npm packages, which can include many new and evolving packages. The dynamic nature of JavaScript can result in harder to detect breaking changes, which may be less of concern in other languages and ecosystems.

Finally, while performing our manual analysis to determine why so many attempts at pinning the dependency being updated did not resolve the client's build, we did not analyze enough samples to be statistically significant. We only analyzed a total of 50 sample, and while they all failed for similar reasons, this is not a statistically significant sample size, and might not lead to generalizable results.

[12]https://snyk.io/

VI. Related Work

In this section, we present the work most related to our study.

Mirhosseini et al. [3] conducted a study on why developers neglect to update software dependencies, and how effective Greenkeeper and other automated tools are at helping developers keep their dependencies up to date. They found these tools to be useful, with projects that use pull request notifications upgraded on average 1.6x as often as projects that did not use any tools. They also found that, although pull request notifications are useful, developers are often overwhelmed by notifications: only a third of pull requests were actually merged. Through a survey of developers, they found that one of the primary reasons why practitioners don't update their dependencies is due to the fear of breaking changes. This demonstrates a need for better tools that give developers higher confidence that updating their dependencies will not break their code.

Xavier et al. [6] performed a large-scale study on the historical and impact analysis of API breaking changes in the Java ecosystem. They assess the frequency of breaking changes, the behavior of these changes over time, the impact on clients, and the characteristics of libraries with high frequency of breaking changes. Alongside their results, they provide a set of lessons to better support library and client developers in their maintenance tasks. One of their more interesting results is that, despite their findings that 28% of all API changes break backwards compatibility, only 2.54% of clients are potentially impacted.

Brito et al.[7] conducted a follow up survey to their 2018 paper that introduced *APIDIFF*, a tool to identify API breaking and non-breaking changes between two versions of a Java library [8]. After identifying possible breaking changes, they asked the developers to explain the reasons behind their decision to change the APIs, and find that breaking changes are mostly motivated by the need to implement new features, by the desire to make the APIs simpler and with fewer elements, and to improve maintainability. To complement this first study, they conduct an analysis of 110 Stack Overflow posts related to breaking changes. they find that breaking changes have an important impact on clients, since 45% of the questions are from clients asking how to overcome specific breaking changes.

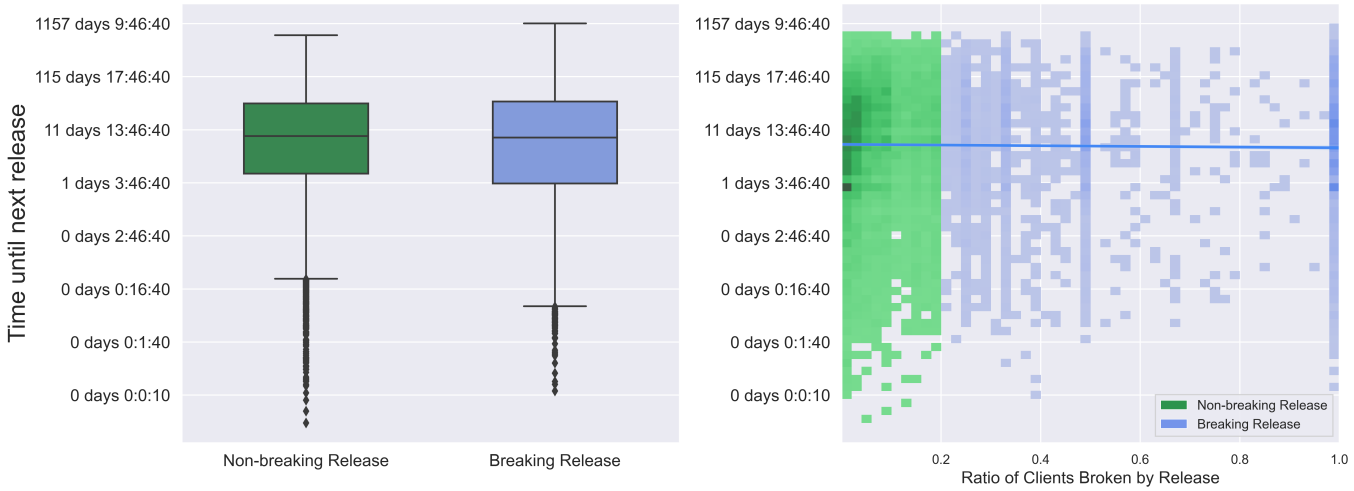

Figure 7: Time until next release for non-breaking and breaking releases

VII. Conclusion

It has become commonplace for developers to reuse code from multiple provider packages, and clients will often specify a range of versions they would like to use from provider packages so that they can automatically receive simple bug fixes and patch updates as the provider releases them. However, these in-range updates can sometimes break a client's build, and since the dependency update is accepted in the client's version specifications, the client package will also not build for any of it's users. In this study, we examined breaking build issue reports created by the Greenkeeper bot for in-range updates to determine characteristics that cause these build failures, as well as how client's respond to these issues and how they resolve their build. We found that the majority of issues are created for patch updates, and that packages with higher total and more frequent releases tend to cause more breakages. Often, the client simply updating their dependency specifications is enough for them to resolve their build, and we found no indication that releases that break a relatively high proportion of client's builds prompts a response in the provider packages.

References

[1] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 385–395. [Online]. Available: https://doi.org/10.1145/3106237.3106267

[2] R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab, "On the impact of using trivial packages: an empirical case study on npm and pypi," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1168–1204, Mar 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09792-9

[3] S. Mirhosseini and C. Parnin, "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*, ser. ASE 2017. IEEE Press, 2017, p. 84–94.

[4] 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*, pp. 1–1, 2019.

[5] R. G. Kula, D. M. Germán, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration," *CoRR*, vol. abs/1709.04621, 2017. [Online]. Available: http://arxiv.org/abs/1709.04621

[6] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 138–147.

[7] A. Brito, M. T. Valente, L. Xavier, and A. Hora, "You broke my code: understanding the motivations for breaking changes in apis," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1458–1492, Mar 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09756-z

[8] A. Brito, L. Xavier, A. Hora, and M. Valente, "Apidiff: Detecting api breaking changes," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2018, pp. 507–511. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER.2018.8330249