Analysing the effectiveness of fine-grained dependency analysis to convince developers of updating their dependencies

CHRISTOPHE COSSE[1] , **SEBASTIAN PROKSCH**[1] , **MEDHI KESHANI**[1]
[1]TU Delft

Abstract

Dependency maintenance is a critically important part of software development as vulnerabilities and exploits are constantly being discovered. Unfortunately it is extremely tedious for developers to manually keep track of these vulnerability discoveries and update their dependencies consequently. Dependency maintenance tools such as Dependabot and WhiteSource help to make this job easier for developers but still many developers never update their dependencies even with notifications from these tools. As such this research paper aims to find if giving more information to the developer as to how the vulnerability affects their code entices developers more to update their dependencies. This research found that developers seem to not care much for extra information about vulnerabilities and in whole maybe a different approach is required to educate developers on the critical importance of dependency maintenance.

1 Introduction

In modern software development, software engineers often reuse code from others through the use of dependencies. Such dependencies are handled through the use of tools like Maven or Gradle (for the Java programming language). However, by reusing code, developers often include many bugs and security issues in their programs which are often not yet discovered and come from the dependencies, [5] goes further into detail between the correlation of dependencies and vulnerabilities in software projects. These bugs and security issues can then be exploited by hackers to access confidential and sensitive information. For example in 2017 a dependency vulnerability was responsible for the leak of personally identifying information of over a 100 thousand Equifax clients [6].

This leak could have been avoided had Equifax simply updated their dependencies more regularly, as the exploit had been discovered and patched in a previous update. However, dependency management is extremely tedious to do manually. That is why nowadays several automated programs (bots) exist to automatically give recommended updates for dependencies to developers. Examples of these bots are WhiteSource [1] and Dependabot [2]. For the purposes of this research Dependabot will be used.

Unfortunately, these bots have one critical flaw; they analyze repositories at the package-level. This leads to Dependabot not being able to know if the vulnerable dependency is actually vulnerable in the context of the repository. To understand this further let us assume a software-project uses some dependency that contains a vulnerable method. This method is never directly, or indirectly, called in the software project hence the vulnerability caused by the method is never actually exploitable for this project, even thought this is the case, Dependabot will still flag this dependency as needing to be updated even though it does not actually cause a security concern. This leads to Dependabot recommending updates that are completely unnecessary and therefore the high percentage of false positives which leads the developers to not really paying attention to Dependabot recommendations.

To solve this issue, recent studies [6] [4] have shown that a different type of dependency analysis, this time at the method-level, is much more reliable in proposing correct updates to developers. However no study as so far concluded on whether these more accurate suggestions lead to a higher receptivity and attention from the repository maintainer. In this study we aim to understand what the impact of such analysis is on the actual actions of developers towards these dependency management recommendations compared to more traditional package-level analysis. Hence the research question of this project is: Do people react to fine-grained information more than package-level information?

The structure of this document is as follows: section 2 will delve deeper into the background of this field and show the methodology used to select sample repositories, analyze them using both package-level and method-level techniques as well as how we use this analysis to create pull requests that help us gather data and assess the reactions of developers towards different types of data. Section 3 goes into detail into what the main contributions of this paper are for the wider computer engineering community. Section 4 will dive

[1]https://www.whitesourcesoftware.com/product-overview/
[2]https://dependabot.com/

into the experimental setup and the methodology of analysis used on the data to obtain the results. Section 5 will deal with the responsibility of this research and its reproducibility. Section 6 compares the results of this paper to other known results from related work. Finally, section 7 will lay down the conclusion of this work and harp away at possible future work.

2 Background

2.1 The importance of dependency management

Dependency management is a critically important task that must be performed on all projects, whether they be large-scale, medium-scale or even toy projects. Code reuse has permitted object-oriented projects to be developed much quicker then starting from scratch as shown in [3], which empirically shows how code reuse increases productivity.

However with this code reuse, vulnerabilities are transmitted from project to project as shown in [5], which studied over 1200 projects and showed a correlation between the number of dependencies a project used and the number of security issues affecting the project.

2.2 The hunt for dependency vulnerabilities

Dependency vulnerabilities have been a wide known problem for many years and therefore there exists a large database of each know vulnerability and thousands new ones are discovered every year either by vulnerability hunters, or simply by accident. These vulnerabilities are published in CVEs (Common Vulnerabilities and Exposures explained well in this blog post [1]). These CVEs are critical in the pursuit of more secure software ecosystems since they are the data used by all tools which manage dependencies, including the research done by this paper explained in section 3

2.3 Automating dependency management

To automate the combat against dependency vulnerabilities tools such as Dependabot are used currently. These tools must be added individually by projects are scan the dependencies of these projects for vulnerable dependencies. If such a dependency is found, a pull request is generated to update the dependency. Figure 1 shows an example of such a pull request. As can be seen the pull request also contains the security vulnerability information to notify the developer of what the vulnerability is and what the previous and updated version of the dependency is.

While Dependabot gives the developer the vulnerability information and the fix to update the dependency and get rid of the vulnerability in the project, it is up to the developer to apply the fix by merging which is unfortunately not something that is always done.

2.4 Issues with these tools

The main issue with tools such as Dependabot is that they only scan repositories for a vulnerability on the *package-level*; that means that they only check if the project contains a dependency which has been detected as vulnerable. As such they do not check if the vulnerability actually affects the project.

That means many Dependabot pull requests are actually false since the vulnerability doesn't actually affect the project itself. Indeed vulnerabilities are often just a simple error in a single method of a dependency. If that method is never called, either directly or indirectly, then the vulnerability doesn't affect the project.

The gross method of vulnerability detection might have something to do with the general nonchalant attitude that developers show towards security vulnerabilities. As can be seen in [2], a study that performed a statistical analysis of the effectiveness of Dependabot security issues on large-scale projects, a large majority of issues (65%) result in a dependency update however this number is still very low considering the fact that most security vulnerabilities are of high risk.

2.5 The FASTEN Project

The FASTEN project [3] is large project in development which aims to make software ecosystems more robust by making dependency management more intelligent. They do this by relying on the creation of an ecosystem-wide **Fine-Grained Call Graph** directly at the *function-level*. These call graphs permit FASTEN to track function calls across an entire application and therefore determine if a vulnerable method is ever called.

The reader is indicated to [4] [6] to further understand call graphs and their application for dependency mangement.

3 Methodology

The following section describes in great detail the methodology used to answer the research question and provide a contribution to the field of dependency management. Starting with the selection of a usable project set we go over the package level and method level analysis produced and ending with the explanation of the information gathering. Each subsequent subsection represents one step in the methodology used.

3.1 Selection Criteria

Before analysing projects, it is necessary to first obtain a project set that fits the usage of the project since it is simply unfeasible to analyze the millions of java projects publicly available on Github. For this several factors must be taken into account. Firstly the project must be in the Java programming language. Next projects need to have been package level vulnerable at a minimum and they must use either Maven or Gradle as dependency management tools. The popularity and activity of projects is also taken into account. Luckily FASTEN contains a database, the FASTEN Metadata Database [4], of projects which fit this description.

3.2 Package level analysis

Performing package level analysis is a rather simplistic task. All that needs to be done is to check the dependency file (pom.xml for Maven [5] and build.gradle for Gradle [6]) of the

[3] https://www.fasten-project.eu/

[4] https://github.com/fasten-project/fasten/wiki/Metadata-Database-Schema

[5] https://maven.apache.org/

[6] https://gradle.org/

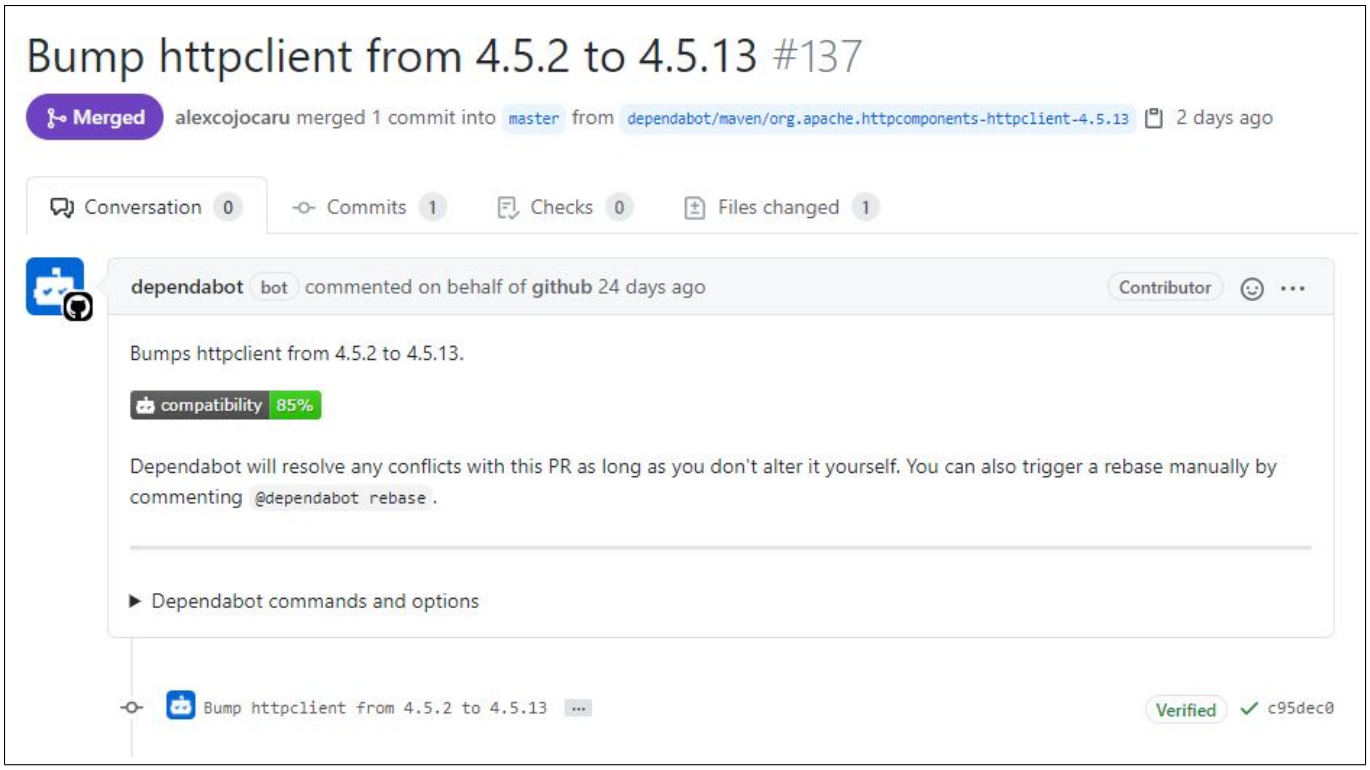

Figure 1: Example Dependabot pull request

project for vulnerable dependencies. Figure 2 shows an example of such information in a pom.xml file. A script was written for this task. The dependency file is parsed and all dependencies and their versions are extracted.

These dependencies and their versions are then cross-referenced with a list of known vulnerable dependencies obtained from CVEs. If such a dependency and its correct vulnerable version is present in the dependency file, the project is marked as package-level vulnerable.

```
60          <dependency>
61            <groupId>com.fasterxml.jackson.core</groupId>
62            <artifactId>jackson-core</artifactId>
63            <version>2.9.10</version>
64          </dependency>
```

Figure 2: Dependency information in a pom.xml file

3.3 Method level analysis

Once a list of all package-level vulnerable projects is obtained from the previous step, method-level analysis can be done. This is done using the FASTEN project which has a java library containing many functions and features for the creation of call-graphs. These call-graphs track which methods are called by which other methods and as such permits the user to trace method-calls. So one method A calls method B which in turns calls method C which finally in-turn calls method D. Using call-graphs it is possible to trace a vulnerable method, lets say D is vulnerable, all the way to method A, the method of the developer. These call graphs hence permit developers to know if vulnerabilities in dependencies are actually affecting their repositories.

Therefore a script is then built with input the list of package-level vulnerable projects from the previous step and CVEs which contain information such as vulnerable dependency name, version and the methods that are responsible for the vulnerability. This script then has as output a list of projects that are actually vulnerable as well as the affected methods in the repository and the methods that are at the source of the vulnerability for each of the repository methods.

3.4 Pull request generation

With all this information, pull-requests are manually created with method-level information see Figure 3 for an example of this. These pull requests also contain a small very simple yes/no quiz (in the form of assertions with tick-boxes) for the repository manager to answer. These assertions ask for simple information such as if the method-level information was found to be relevant to the project or if the manager was convinced to update the dependency because of the extra method-level information.

As can be seen in Figure 3, the title of the pull request starts with the vulnerability severity in all caps, the dependency name, old version and version to which it is updated. This information is obtained directly from the CVEs. The pull request then follows a similar styling as dependabot pull

requests with information about the vulnerability and a link to the concerned CVE as well as a link to the patch notes of the dependency release which fix this vulnerability. Then the pull request shows, the affected methods in the project (which are hyperlinks to the place in the code), the method from the dependency called in the project and finally the method from the dependency that is at the origin of the dependency.

It is important, for the method-level information to be taken seriously, that all three methods are shown such that the developer is able to trace the vulnerable method directly to his code and as such be certain that this vulnerability does indeed affect his project.

The pull request then has a small description and explanation of the information presented above as well as how this information was generated and the scope and aim of the research project tied to this pull request. This was added to entice the developers to hopefully do more research about this topic and convince them to answer out the quick survey.

4 Results

In this section, we present our findings.

4.1 Responses to the method level pull requests

The following Table 1 shows the responses to the method level pull requests sent out to vulnerable projects. The projects are divided into active and inactive groups based on if the project had a contribution in the last 4 months.

Activity	Active	Inactive
# Pull Requests	25	12
# Merges	3	0
# Interactions	7	0
# Ticked Box	1	0
Average Time to Respond	10 days	NA
# Responses within 1 day	4	0
# Responses after 3+ weeks	3	0

Table 1: Obtained responses from PRs

As can be seen from Table 1 a total 37 pull requests containing method level information were sent out, 25 of these being on active repositories which had recent contributions and therefore are expected to respond to the dependency information. A further 12 pull requests were also sent out to inactive projects to check whether developers continue to maintain their projects after having published them.

From the data it is quite obvious that the responses are rather lacking with only seven of the active projects (28%) having any interaction at all with the pull requests while only three (12%) actually merged the pull request and updated their dependency.

Furthermore while the average response time was ten days this result is extremely skewed since four of developers replied almost instantly within the first one of the PR being sent out while the other three responses each took over three weeks to reply.

Unfortunately while some developers did interact with the pull requests only one actually managed to tick any of the boxes in the pull requests that asked for feedback on the usage of method-level information in these types of security pull requests. This is why comments were published on each PR with a survey link, with the same questions as the tick boxes in the pull requests, to hopefully obtain more information which resulted in two more responses. These responses are presented in Section 4.2.

Finally it can be noted that none of the inactive projects responded to the pull requests or interacted with them in any way showing that these projects will most likely be vulnerable forever.

4.2 Responses to the survey

From the survey sent out to the developers and the responses to the checklist style tick boxes in the pull requests, a total of three responses were recorded which are presented in the following table:

Question	Yes	No
I was aware of the vulnerability affecting my project before being informed by the Pull Request.	0	3
I was convinced by the provided method call data that the vulnerability indeed affects my project.	2	1
I plan on merging the PR in the near future.	3	0
The provided method call information has made my process of dealing with the vulnerable dependency easier	1	2
I have given priority to the task of fixing the vulnerability over other project tasks that are yet to be completed.	2	1
I would like to receive this kind of method information in future vulnerable dependency Pull Request descriptions.	2	1

Table 2: Obtained responses from Survey

It is interesting to note that from the three respondents one of them (who did not give priority to this issue in his project) has yet to merge the pull request and update their dependency. These responses will be further discussed in Section 6.

4.3 Developer behaviour from the literature

While developers of active projects had a low interaction rate with the pull requests (28%), it is important to put these results into perspective and understand how developers react to normal package-level pull requests (dependabot). From the literature study [2] shows that 65% of normal security pull requests are merged. However it is important to realize that these are pull requests on projects which are subscribed to dependabot. This means these projects are generally aware of the importance of proper dependency management.

MethodLevelAnalyzer commented 6 days ago

Vulnerability Information

Bumps apache-httpclient from 4.5.1 and 4.5.2 to 4.5.3.

Listed dependency **org.apache.httpcomponents:httpclient** contains vulnerable methods which are called from this project. This vulnerability appears to affect *httpclient* package versions lower than 4.5.3 (excluding). The vulnerability has been fixed in version 4.5.3, as can be seen from the package release notes.

Property	Value
Linked CVE	HTTPCLIENT-1803
Number of affected methods	2
Severity	**MAJOR**
Current version	4.5.1 and 4.5.2
Updated version	4.5.3
Backwards Compatibility	True

Vulnerable method calls

Methods in this repository	Used package methods	Origin vulnerable method
com.simplifyOM.HttpUtility/ApiUtil.get(String url, Map<String, String> queryParam, Map<String, String> headers)	org.apache.http.impl.client/CloseableHttpClient.execute(HttpUriRequest request)	org.apache.http.client.util/URIBuilder.normalizePath(String path)
com.simplifyOM.HttpUtility/ApiUtil.get(String url, Map<String, String> queryParam, Map<String, String> headers)	org.apache.http.client.utils/URIBuilder.build()	org.apache.http.client.util/URIBuilder.normalizePath(String path)

What do the columns represent?

The 1st column in the table indicates the method in this repository that was found to be affected by vulnerable methods from the *httpclient* package.

The 2nd column indicates the *httpclient* method that was directly called from this repository.

The 3rd column indicates the origin vulnerable method in the *httpclient* package. According to our dataset, this is one of the methods that produces the **HTTPCLIENT-1803** vulnerability. This method was found to be internally chain-called in the *httpclient* package by the method listed in column 2.

How were the results generated?

This vulnerability was analyzed specifically for usage in this project using the FASTEN Project. Statical method-level analysis was used to check for usage of vulnerable methods in the project.

Method calls between your project and *httpclient* have been mapped into a directed graph. From this graph, it could be then be seen whether any vulnerable *httpclient* methods are being called from within your project.

Research Scope

We are a team of 3 BSc Computer Science students at the TU Delft. Our goal is to conduct research on how developers react to method-level vulnerability information that affects their projects. We would highly appreciate if you could help us with our research and please tick statements which apply to you below.

First impression checklist

- [ ] I have read this pull request description.
- [ ] I was aware of this dependency vulnerability affecting my project before being informed by this Pull Request.
- [ ] I was convinced by the provided method information that this vulnerability indeed affects my project.
- [ ] After seeing the provided method-level information, I plan on fixing the vulnerability.

After fixing vulnerability checklist

- [ ] I found that the provided method information has made my process of dealing with the vulnerable dependency easier.
- [ ] I have given priority to the task of fixing the vulnerability over other project tasks that are yet to be completed.
- [ ] I would like to receive this kind of method information in future vulnerable dependency Pull Request descriptions.

TU Delft
Delft University of Technology

Figure 3: Example method level pull request

When it comes to general projects [7] showed that the vast majority of active projects (81.5%) do not update their dependencies and 69% of developers were not aware that their projects were vulnerable. These results fall somewhat inline with the data obtained from the method-level pull requests and surveys where 87% of the projects did not update their dependency and none of the respondents were aware of their project being vulnerable.

5 Responsible Research and Reproducibility

5.1 Responsible research and ethical aspect

Given the nature of this research, finding and alerting project managers of vulnerabilities in their projects, it is critical to take into account responsibility.

Although no personal or identifiable information is presented in this document, it would be rather easy for anyone with knowledge of Github to find all the PRs that were made by the research group and with that all the vulnerability information on the different projects. Although it is true that anyone can, just as simply as was done for the purpose of this document, analyze thousands of repositories for vulnerabilities, the fact of the matter is with this research a couple dozen projects are directly publicly announced as vulnerable with all the details of these vulnerabilities. This is quite sensitive information so special care needs to take place to make sure this information is as protected as can be. That is why all the pull requests (be they merged or not) have been deleted before the publication of this document and no information that permits readers from identifying the projects used for this research is present in this document either.

5.2 Reproducibility of the research

Another important discussion that needs to be mentioned is the reproducibility of the research and the results. While the exact same results are impossible to reproduce (given that the projects have all either updated their dependencies or are inactive), the publishers of this document do believe that the general consensus and contribution done by this research can be reproduced simply by following the exact same methodology described in section 3 with a new base set of projects and up to date CVEs.

6 Discussion

6.1 The developer responses

The biggest takeaway from the experiment that was conducted was that there were very little responses from the developers which were contacted. Only one of the developers checked any of the check boxes in the PRs (even though the first one was essentially "I have read the pull request") which shows how careless developers are to external information on their projects. Only seven project maintainers interacted with the PR generated mostly for them to ask someone else to have a look at it and nothing came of those. In the end from 37 pull requests only three were merged.

From this, two main takeaways can be considered: firstly developers are, for the large majority, not receptive to external contributions. Secondly when they are receptive, they do not take the time to fully read the contribution and understand it. This leads them to miss out on critical information and probably if the driving force behind vulnerability fixes that aren't addressed as soon as they are discovered and published through CVEs.

6.2 Analyzing the responses received

Now that the lack of responses has been discussed it is possible to analyze the few responses that were obtained, even if they are not sufficient enough to come to any hard conclusion. The analysis of these responses might indicate that further research in this field is necessary to understand the added value of method-level information.

We will list all seven interactions with the pull and discuss briefly what these interactions entail and how the developer responded.

1. **elasticsearch-maven-plugin** This developer took 21 days to respond but decided to ignore it fully because they already had a dependabot pull request open about the same topic and preferred to merge this one instead.
2. **terrier-core** This developer took 20 days to respond and answered the survey in which he mentioned that he plans on merging the pull request but has not done so as of yet. He also mentioned that he didn't want to give preference to this issue over others in his project.
3. **hazelcast-tomcat-sessionmanager** This developer acknowledged the pull request nearly instantaneously, however only to ask an admin to have a look at it. Since then nothing has come of it and the pull request has remained ignored for over 3 weeks.
4. **openstack4j** This developer reacted within a day to the pull request however the dependency update failed to build and the developer decided to ignore the dependency update instead .
5. **jcabi-http** This developer merged after a total of 22 days however he did not respond to the survey and later ignored a comment prompting him to answer the survey.
6. **stream-java** This developer merged the pull request only a few hours after receiving it however he did not respond to the survey neither on the pull request or on the comment sent to prompt him.
7. **trex-java-sdk** This developer merged the pull request a day after receiving it and responded to the survey after a comment prompted him to do so.

When taking a look at the three responses from the survey it is clear that developers are generally unaware of vulnerabilities in their projects. Furthermore only two out of the respondents said they were convinced their projects were vulnerable from the provided method-level information. This might be because of a lack of knowledge on the topic or because of the way the information was presented. Only one of the respondents also said that the extra method call information made their process of dealing with the vulnerability easier and finally one of the developers did not wish to receive more of these pull requests with extra method-level information while the other two did.

The above results therefore do seem to suggest that this extra method-level information does provide a benefit to developers but there simply isn't enough data to formulate a strong opinion on the subject matter.

It is worth mentioning also that we obtained two responses from the survey posted in the comments sent to developers, yet only one developer actually confirmed that he completed the survey.

6.3 Reasons for the lack of responses

What should be discussed further is the reasons for the lack of responses. Why don't developers check in more detail information given to them about their projects? Below is a list of possible reasons and each will be discussed in some detail.

- ***After publishing their work, developers do not regularly check it for vulnerabilities***. Possibly the biggest reason. Developers are not thinking about the security of their work after it has been published. Many of the repositories did not see any activity since the release of the project. Even if the release was rather recent.
- ***Developers do not fully read vulnerability warnings in detail***. While this is impossible to fully confirm, it can be assumed that one of the big reasons that developers did not reply to the check boxes is simply that they did not read long enough to reach them. It is quite likely most to the developers simply read the PR title and decided "oh I will take care of this later" or "oh this is only dependency vulnerability I don't mind". Developers might think they have more important work and cannot allocate the simple 2 minutes it would take to fully read the pull request and respond to it.
- ***Developers do not take dependency vulnerabilities seriously***. This reason flows from the previous and that is the thought that developers seem to not much care for dependency vulnerabilities. They assume that the vulnerability is small. Or that their project is not important enough to be targeted.
- ***Dependency management bores developers***. No developer seems to care for dependency management. And that might be more because it bores them. Indeed it seems likely no developer enjoys spending their time sorting through dependency vulnerabilities to understand them and see if they need to perform an update.

7 Conclusions and Future Work

7.1 Conclusion

The main research question of this project was "Do people react to fine-grained information more than package-level information?" to be able to understand whether a lack of information on how vulnerabilities affect projects was the reason behind the low amount of dependency updates.

To answer this question a script was written to be able to automatize as much as possible the lengthy process of going method-level analysis on projects and then pull requests where sent out containing this extra method-level information. The objective was to obtain information about the interactions of developers with the pull request so that they could be compared to already available data from literature.

Unfortunately the number of responses were very low and very few conclusions can be taken from them. From what was gathered it is clear that, while the developers that did merge, seemed to be convinced by the extra information they only are a small proportion of the total developers contacted. Therefore it seems that the problem with low dependency updates is not with convincing developers that their projects are vulnerable but that these vulnerabilities are actually dangerous.

7.2 Future Works

While the research failed to get enough responses it is important to note that three of the seven total responses were obtained three weeks after the initial pull requests were sent. Therefore with more time and prompts to developers it might be possible to obtain more responses. Furthermore just expanding the project set will inherently increase the amount of responses. However this would require a great deal of work and investing into the creation of a dependabot-like script to automatically generate the pull requests.

A future paper which will be co-written with all the other researchers in this field, Jakub Nguyen, Tudor Popovici and Niels Mook shall include these new findings should they exist as well as incorporating all the research done by the fellow researchers.

References

[1] Kevin Casey October 29, Kevin Casey, October 29, Kevin Casey writes about technology, and business for a variety of publications. He won an Azbee Award. How to explain cve, common vulnerabilities and exposures, in plain english, Oct 2019.

[2] Mahmoud Alfadel, Diego Costa, Emad Shihab, and Mouafak Mkhallalati. On the use of dependabot security pull requests. 02 2021.

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10):104–116, October 1996.

[4] Paolo Boldi and Georgios Gousios. Fine-grained network analysis for modern software ecosystems. 21(1):14.

[5] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software*, 172:110653, 2021.

[6] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. Software ecosystem call graph for dependency management. 40:4.

[7] Raula Kula, Daniel German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23:1–34, 02 2018.