# Effectiveness of using call graphs to detect propagated vulnerabilities

Jakub Nguyen
Delft University of Technology
The Netherlands

Mehdi Keshani
Delft University of Technology
The Netherlands

Sebastian Proksch
Delft University of Technology
The Netherlands

## ABSTRACT

Nowadays software development greatly relies upon using third-party source code. A logical consequence is that vulnerabilities from such sources can be propagated to applications making use of those. Tools like Dependabot can alert developers about packages they use, which entail vulnerabilities. Such alerts oftentimes turn out to be false positives because the vulnerable functionality of the package is not used. Current research by the FASTEN Project revolves around analysing dependency networks using a finer granularity; moving from package-level to method-level analysis with the help of call graphs. Such analysis can theoretically be used to gain better insights into how vulnerable a dependency for an application is.

This report aims to display the practical effectiveness of using call graphs to detect propagated vulnerabilities. To evaluate the effectiveness, results generated through method-level analysis were studied with regards to whether a vulnerability in the corresponding project is reproducible. Furthermore, possible improvements to call graphs to detect vulnerabilities more accurately are described in this study. An experiment, based on call graph analysis, was conducted to detect propagated vulnerabilities in a set of public software repositories. The used data about the repositories and vulnerabilities was provided by the FASTEN Project. Each vulnerability detection was manually verified and studied on its impact based on public information about the corresponding vulnerability.

The results of this experiment show that none of the potential propagated vulnerabilities could be reproduced. This implies that a greater set of repositories needs to be analysed to draw meaningful conclusions for the effectiveness of call graphs to detect propagated vulnerabilities. The proposed improvements to call graphs display a fraction of the great potential of the precision that could be reached through such fine-grained analysis.

## 1 INTRODUCTION

Nowadays build automation tools like Maven or Gradle are oftentimes an integral part of software development. They allow the user to conveniently make use of external libraries hosted on centralised repositories. However, by reusing others code its flaws are also adopted which, in terms of security, can pose a great risk. Given the sheer amount of projects that make use of certain packages, a severe vulnerability in one of these could be fatal. A recent failure of such a dependency structure was the Equifax security breach which leaked over 100.000 credit card records due to an updated package [3].

The most used GitHub tool to keep dependencies up-to-date is Dependabot [1]. While this tool performs well for many of its purposes, detecting a vulnerable dependency has a major limitation. Dependabot analyses dependencies only on the package-level. This leads to triggering alerts that dependencies are vulnerable even if the user's application does not make use of the vulnerable functionality of the dependency. Such alerts are false positives. A project with dozens of dependencies has a great chance of receiving numerous alerts about vulnerabilities that are not affecting the project. Repeatedly notifying project maintainers about false vulnerabilities can reduce the incentive to handle such threats in general.

Recent studies have shown that the use of call graphs increases the accuracy of such dependency analyses [2][3]. These call graphs are a manifestation of analysis on method level. This means method calls are traced either statically or dynamically to generate a graph. A call graph is a graph in which vertices represent methods of the program and edges possible calls between them. In theory such analysis leads to significantly less false positives than package level analysis when detecting vulnerabilities.

An important milestone for this field of research would be to prove the value of fine-grained package analysis in practice. However, no research has yet been conducted showing how well such fine-grained information is related to the vulnerability. Therefore this research aims to display how effective call graphs are to detect propagated vulnerabilities. This is done by investigating how well method-level data reflect the reproducibility of vulnerabilities. The outcome of this study can show the great potential of such vulnerability detection, fortifying the relevance of further research into this topic.

The following sections take on related work, the methodology used, performed experiments, several theoretical improvements for vulnerability detection through call graphs, responsible research, a discussion, a conclusion and future work.

## 2 RELATED WORK

Studies that are closely related to this research are highlighted in this section.

### 2.1 Software Ecosystem Call Graph for Dependency Management (2018)

This study [3] describes the use case of call graphs to span across entire ecosystems of software. In contrast to how dependency networks are analysed currently the aforementioned call graphs would allow for impact analysis on method-level. The paper outlines the construction of such call graphs but also a preliminary evaluation of a security issue. Thousands of packages were found to be package-level vulnerable but only a single package was analysed on method-level and verified to contain calls to vulnerable methods. Furthermore an algorithm on call graphs for impact analysis is outlined.

[1] https://dependabot.com/

With regards to the aforementioned work this research aims to expand upon the idea and concept of detecting the impact of vulnerabilities through dependencies in practice. This is done by analysing thousands of projects and verifying the results for correctness. An algorithm with great similarities to the proposed impact analysis one was implemented for this research. This report does not cover the feasibility of constructing call graphs on an ecosystem level but instead shows the potential of such call graph analysis.

### 2.2 Fine-Grained Network Analysis for Modern Software Ecosystems (2020)

The research conducted by Boldi and Gousios [2] revolves around displaying the risks and limitations of how dependency networks work currently. Furthermore they propose the usage of call graphs to lift such limitations; envisioning entire ecosystems to move from package-level to method-level. Potential use cases that are mentioned are about analysing ecosystem-wide impacts, more flexible software licensing and more precisely determining the impact of a vulnerability on an individual project.

Similarly to the study mentioned in the subsection before, the key difference to the research conducted for this report is showcasing the practical usage of call graph analysis. This report brings the mentioned opportunity of analysing whether an application calls into vulnerable code or not to the test and further describes more opportunities to increase precision of such analysis.

## 3 METHODOLOGY

In order to investigate how well fine-grained dependency analysis relates to a vulnerability, a set of projects that call vulnerable methods from dependencies was determined. The following sections cover the steps taken to find this set.

### 3.1 Approach overview

Figure 1 depicts the overall process of finding the method-level vulnerable projects. Starting from the data gathered by the FASTEN Project, thousands of projects were selected to be analysed on package-level vulnerabilities. Method-level vulnerability analysis was conducted on the package-level vulnerable projects. Finally the method-level vulnerable projects were manually verified and studied on their possible vulnerabilities.

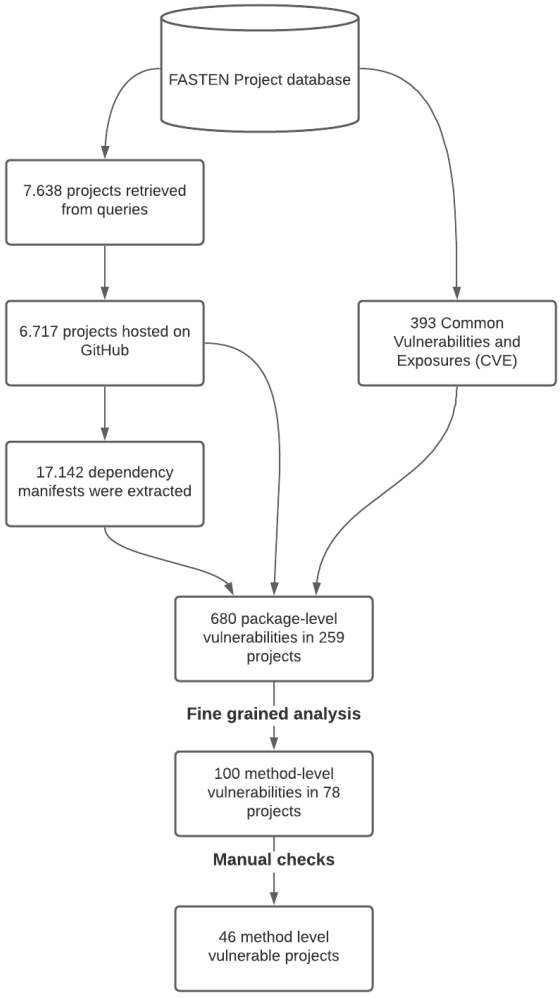
### 3.2 Project selection

A preliminary set of 7.638 public java projects was selected based on data which was gathered during the research conducted in the FASTEN Project. The FASTEN Project is storing data of thousands of deployments of public projects including their dependencies and vulnerabilities. By searching for projects that are hosted on GitHub, with dependencies that contain vulnerabilities in their latest recorded deployment the preliminary set of projects was determined.

### 3.3 Package level vulnerabilities

To decide which projects from the preliminary set are package level vulnerable in their current development state the currently used dependency manifest files (e.g. *pom.xml* or *build.gradle*) were retrieved. To achieve this, a customised script from the Dependabot project was used. By searching for certain keywords in the manifests, depending on the build automation tool, the dependencies and dependency versions for each project were determined. In total 17.142 Maven dependency manifests were extracted. A set of 393 vulnerabilities in 211 unique dependencies, from the FASTEN Project [2], containing data about associated methods, was used to compare to the dependencies of the preliminary projects. 259 package-level vulnerable repositories were found.

**Figure 1: The procedure to determine method-level vulnerable projects**




### 3.4 Method level vulnerabilities

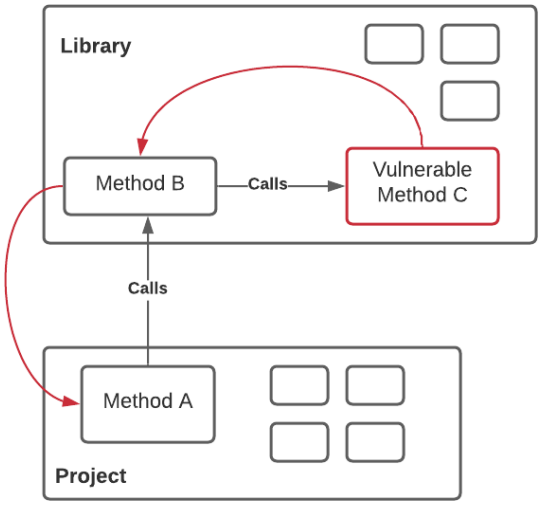
Once a package-level vulnerability has been verified in a project, fine-grained analysis was conducted. By making use of the FASTEN Project's call graph generation feature (Rapid Type Analysis algorithm [4]), graphs for the project itself and the vulnerable packages

[2] https://www.fasten-project.eu/view/Main/

were generated. All call graphs of a project were then merged into one to allow for the tracing of methods. This merging is implemented in the FASTEN project and attempts to match all calls to external sources to their corresponding nodes.

Figure 2 depicts an example of how a vulnerable method is traced. The vulnerability propagates through methods B and C making method A indirectly vulnerable.

**Figure 2: Minimal example of vulnerability tracing. Through tracing the methods Method A is identified as vulnerable.**




For each package-level vulnerability detected in a project, the associated vulnerability methods were used to perform an algorithm similar to breadth first search. Starting from the vertices representing all vulnerable methods (in a dependency), the methods calling the vulnerable ones were considered vulnerable as well. This repeats for each vulnerable method until either a method that is part of the project is encountered or there are no methods making use of the current one. If a method that is part of the project is encountered during this analysis it means that in the current project a method call to a vulnerable method from a dependency exists, in other words a method-level vulnerability has been detected. 78 projects were analysed to be method-level vulnerable.

### 3.5 Relation to the actual vulnerability

Once the set of method-level vulnerable projects is determined, the relation to the actual vulnerability can be investigated. To filter out the actual vulnerabilities the supposedly vulnerable dependency versions were manually verified to match for each project; 46 method-level projects remained. The corresponding deployment of the vulnerable dependency was retrieved from Maven Central [3] and checked to contain the analysed method calls.

Furthermore, the description and other publicly available information of the corresponding vulnerability were considered to determine how vulnerable the propagated vulnerabilities are. The exact procedure for each considered vulnerability can be found in the experimental setup and results section.

## 4 EXPERIMENTAL SETUP AND RESULTS

The following sections cover the experimental setup and gathered results.

### 4.1 Experimental setup

7638 projects were retrieved from the database of the FASTEN Project. From these projects 6717 remained which were hosted on GitHub [4] and had functioning links to the corresponding repository associated. For 5201 projects the dependency manifests (build.gradle, pom.xml) could be retrieved. Data about 393 different vulnerabilities gathered by the FASTEN Project was used to filter out 259 package-level vulnerable projects. For each of these projects fine-grained analysis was performed resulting in 78 method-level vulnerable projects.

### 4.2 Evaluation setup

Due to the nature of the Rapid Type Analysis algorithm and merging of call graphs, used for call graph generation, some unreachable method calls were followed to determine these vulnerabilities. Therefore several method traces for each vulnerability were manually verified. Furthermore vulnerable dependency versions were verified to be matching (due to partially incorrect vulnerability data). This procedure resulted in 46 remaining projects that were further investigated on the corresponding vulnerabilities.

### 4.3 HTTPCLIENT-1803

Twenty-five projects are potentially affected by the HTTPCLIENT-1803 vulnerability in Apache's HttpComponents package [5]. It revolves around passing a malformed URL argument to the constructor of the URIBuilder which resulted in not being able to change the host of this client with the *setHost(host)* method. Out of the 25 projects only nine projects made use of the URIBuilder. Out of the 9 projects 6 made use of the vulnerable constructor. None of the projects exposed the URIBuilder to the user which leads to the result that the vulnerability cannot be reproduced with any of these projects.

### 4.4 CVE-2019-14379

Sixteen projects are potentially affected by the CVE-2019-14379 vulnerability in the FasterXML jackson-databind package [6] [7]. It allows for remote code execution through deserialisation from external sources given that settings like *default typing* that allow for the instantiation of objects from unsafe sources, are enabled. To filter out actually vulnerable projects it was searched for usage of the following settings: The method *enableDefaultTyping()* or the annotation *@JsonTypeInfo* using *id.CLASS* or *id.MINIMAL_CLASS*. None of the 16 projects made use of either setting.

[3] https://mvnrepository.com/repos/central

[4] https://github.com/

[5] https://issues.apache.org/jira/browse/HTTPCLIENT-1803

[6] https://access.redhat.com/security/cve/CVE-2019-14379

[7] https://bugzilla.redhat.com/show_bug.cgi?id=1752962

### 4.5 Other vulnerabilities

The remaining five projects covered vulnerabilities by three unique CVEs. Two projects were checked on the reproducibility of CVE-2019-20445 [8] and CVE-2019-20444 [9] related to the *Netty* package. No usages of *HttpObjectDecoder.java* were found and one project did not even make use of any functionality from the *Netty* dependency. Both projects did not use the package's HTTP feature rendering them not vulnerable to the aforementioned CVEs.

The last three projects revolved around CVE-2018-17196 [10] which allows to bypass Kafka's Access Control Lists (ACL)[11] checks. Out of various security advisories and sources [12] [13] [14] it was not possible to infer what exact structure a manually crafted request needs to have to reproduce the vulnerability. Only one of the projects made use of a Kafka Producer. That producer's configuration was provided only serialisation formats and adhered to a default producer otherwise. There were no projects to be found vulnerable by CVE-2018-17196.

## 5 IMPROVEMENTS TO STATIC CALL GRAPH GENERATION

This section covers proposals of theoretical improvements to static call graph generation with the aim to increase the accuracy of propagated vulnerability detection.

### 5.1 Simple parameter constraints

The vast majority of vulnerabilities considered in this research, are only triggered by very specific inputs to functions in certain states. In general, parameters can fundamentally lead to a very different execution of a method. The OPAL and FASTEN Projects initial objectives were not revolving around vulnerability detection such that the current implementations do not take method parameters into consideration.

It is a given that statically analysing code to identify constraints on how arguments are passed to a method is an unsolvable problem. Nevertheless some basic cases can be ruled out and simple statements could be evaluated. Naturally also the vulnerability information must entail data about argument constraints to make use of this improvement.

There already exists research and software suited for this problem. For example the Julia Static Analyzer [8] is capable of converting Java byte code into set-constraints. By making use of the constraints of variables in the scope of a method it is possible to specify outgoing method calls more accurately. Consider figure 3 for an example code snippet for the HTTPCLIENT-1803 vulnerability.

Optimally, during call graph generation a constraint for the *baseURL* constant would be identified, namely that the *baseURL* is equal to the string "somebaseURL.com". When analysing the *generateURIBuilder* method this information can be applied to improve the precision of the definition of the method call by not only storing that there exists a call to the constructor of *URIBuilder* but also that the argument passed begins with "somebaseURL.com/".

**Figure 3: An example code snippet to demonstrate possible variable constraints.**

```
private static final baseURL = "somebaseURL.com";

public URIBuilder generateURIBuilder(String path) {
    return new URIBuilder(baseURL + "/" + path);
}
```

With regards to the HTTPCLIENT-1803 vulnerability, by making use of current implementations the code snippet would be flagged as method-level vulnerable. With the improvement of tracking simple variable constraints it can be seen that a syntactically correct URL is passed to the constructor, resulting in no vulnerability detection which is correct.

In terms of practicality it would be possible to make use of existing call graph generation algorithms and augment the resulting graph's edges with parameter constraints. This could be implemented by iterating through all vertices and re-evaluating the corresponding method's source code. For each method call detected the corresponding parameter constraints would be added to the edge in the call graph.

### 5.2 Certainty of method calls

The rapid type analysis algorithm and merging of graphs used to generate the call graphs over-approximate some types of method calls. For example, a call to several methods is detected if an *Object* type is calling a not unique method name (within the scope of the call graph) like *toString*. The majority of such calls turn out to not exist in practice rendering the potential vulnerability false. To improve vulnerability detection it would be beneficial to mark such edges as a possible over-approximation when generating a call graph.

### 5.3 Order of execution

The order of method calls yields crucial knowledge for vulnerability detection. The nature of vulnerabilities can be quite intricate and complex [7] and many require more than a single method call to be produced. Due to this nature it would be a logical choice to track the order in which methods are called during call graph generation.

The methods are examined in order already such that keeping track of a single index would suffice. Similarly to the parameter constraint improvement the order of execution could also be used to augment existing call graphs which would require re-analysing the corresponding source code. In resulting call graphs the directed edges would contain id's representing the ordering of calls where the edge comes from.

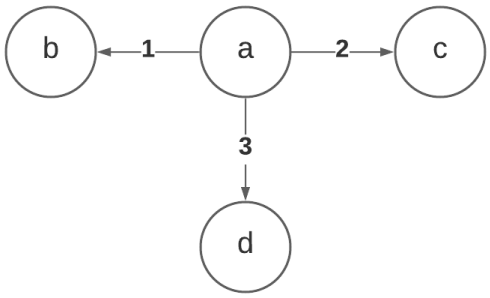
Figures 4 and 5 depict an example of adding order of execution to a call graph for Java source code.

[8] https://access.redhat.com/security/cve/cve-2019-20445
[9] https://access.redhat.com/security/cve/cve-2019-20444
[10] https://access.redhat.com/security/cve/cve-2018-17196
[11] https://docs.confluent.io/platform/current/kafka/authorization.html
[12] https://www.mail-archive.com/dev@kafka.apache.org/msg99277.html
[13] https://security-tracker.debian.org/tracker/CVE-2018-17196
[14] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17196

**Figure 4: An example method displaying the order of method calls.**

```
public void a() {
    b();
    d(c());
}
```

**Figure 5: The partial call graph of example snippet from figure 2.**




### 5.4 Precise vulnerability information

Building on top of the aforementioned improvement, the order of method calls, it would be possible to store more accurate method-level information for a vulnerability. Optionally the order in which methods can be called to reproduce the vulnerability should be specified.

## 6 RESPONSIBLE RESEARCH

This research revolves around finding propagated vulnerabilities through dependencies in public GitHub repositories. Data about such vulnerabilities can be sensitive information and is not publicly available therefore great attention to integrity is paid within the scope of this research. Furthermore results are only presented in a manner where it is not possible to trace back to a specific project.

In terms of reproducibility of the results, this research makes use of non-public data sets about public projects and vulnerabilities from the FASTEN project [15] and detects the vulnerability by generating call graphs with open source tools. Given access to the aforementioned data set the applied methods and results can be reproduced when following the instructions in the methodology.

This research overall aims to display the relevance of fine-grained dependency analysis to detect propagated vulnerabilities. Raising awareness of this topic may lead some developers to pay more attention to vulnerability alerts and deal with them. Furthermore this research can support the decision to develop a tool that is feasible to be used in industry and capable of fine-grained analysis. Eventually this can lead to reducing many false positive vulnerability alerts stemming from the current state of the art, package-level analysis. Contrary to the aforementioned points, with an increasing accuracy of vulnerability alerts developers might be more inclined to consciously decide to not update a dependency. Not updating software in general is considered a risk and waste of opportunities as bugs might get fixed or additional features can be added [9].

[15] https://www.fasten-project.eu/view/Main/

## 7 DISCUSSION

### 7.1 Suitability of call graphs to detect vulnerabilities

In general, identifying software vulnerabilities is a complex enough task to require some degree of human labour [10]. But considering that the task is to detect propagated vulnerabilities originating from existing vulnerabilities in dependencies the actual detection process is less complex.

Boldi and Gousios [2] argue that through call graphs, analysing security alerts on function level, it is possible to obtain more precise information about the impact of a vulnerability on an application. But even with this more accurate information it is expected that a developer needs to manually estimate how vulnerable the affected methods, which were generated by the call graph analysis, are. Depending on the kind of analysis performed, even these results can yield false positives.

As shown in this research, detecting a call to a vulnerable method from a dependency alone oftentimes does not mean that the vulnerability can actually be reproduced. In the projects analysed during this research the vast majority of vulnerabilities required some kind of setup in order to make use of the vulnerability.

A human procedure of determining whether a project is affected by a vulnerability would be triggered by a security alert. Relevant keywords and understanding need to be extracted out of the vulnerability description. By searching for those keywords in the source code, code snippets that can be related to the vulnerability will be found. Such analysis can yield similar results as a simple call graph analysis because in essence statically created call graphs depict the method names similarly to source code. To retrieve the understanding and relevant keywords out of a not standardised vulnerability description is a task rather suited for a human but the main advantage of call graph analysis is the impact the graph can display.

Given that this field of research is in its early stages and that this report only shows a glimpse of the possibilities call graphs open, it becomes clear that it can be worth to use call graphs to detect vulnerabilities. The highlighted improvements in this research also hint at the great potential of call graph analysis.

### 7.2 Better information about vulnerabilities

The preciseness of the results from vulnerability detection through call graphs is highly correlated to the correctness and value of the vulnerabilities that are searched for. For instance in this research oftentimes many methods that are considered internal calls were associated to a vulnerability whereas only a handful of methods are the actual triggers of the vulnerability such that many 'vulnerable method calls' were found to not be related to the actual vulnerability, based on information available from several security advisories.

The methods associated to a vulnerability were obtained by the FASTEN Project in the following manner: In the changes of the patch which fixed the corresponding vulnerability were analysed

and every method encountered was associated to the vulnerability. Through this procedure the actually related methods are captured but there is a chance of including others that are not related. Naturally, given the current standard of information provided about vulnerabilities, e.g. description, affected products, mitigation, this is a very good first step but obviously not perfect.

As the considered vulnerabilities of dependencies are already discovered, a challenge for detecting propagated vulnerabilities more accurately becomes obtaining precise and meaningful information about the existing vulnerabilities. While Hejderup, Deursen and Gousios envision the concept of ecosystem-level call graphs [3], allowing for a vast majority method-level analyses, the information provided about software vulnerabilities should also include more detailed and standardised data about associated methods. In the optimal scenario developers that discover a vulnerability should additionally to the current information provide the methods with argument constraints needed to reproduce the vulnerability and the order in which they can be executed. With such information propagated vulnerability detection through call graphs could become very precise.

### 7.3 Dependency updating

Dependency updating is a logical way to avoid becoming vulnerable to security issues from third-party libraries. Unfortunately it turns out that the mindset of regularly updating dependencies among developers is rather uncommon. A study showed that 69% of interviewed developers were uninformed about vulnerable dependencies that affected them [5]. Furthermore 81.5% of the systems that were studied make use of outdated dependencies [5]. Another study shows that even if developers make use of tools that support updating dependencies, e.g. Dependabot, still 34.58% of security pull requests are not merged [1]. A different study displays that two thirds of studied pull requests were not merged and that the fear of breaking changes is a great concern to developers [6].

With regards to all of the aforementioned research it becomes clear that software development is still far away from a good system and mindset to keep dependencies up-to-date. An interesting question would be, given that the envisioned ecosystem-level call graphs [3] would be in place, how developers would react to more precise updating information. It offers many opportunities like filtering out the most important changes of a dependency based on the current usage in the application.

## 8 CONCLUSION AND FUTURE WORK

This report aimed to display the effectiveness of using call graphs to detect propagated vulnerabilities in practice. The results of this study could fortify other research in this field.

An experiment, making use of call graphs, was conducted in which dozens of projects were found to be method-level vulnerable. For each of these projects, a manual verification was performed on how threatening the corresponding vulnerabilities actually are. Originating from working on this study, several theoretical improvements to call graphs were developed. These improvements mainly focus on the potential precision call graphs could reach when detecting propagated vulnerabilities.

The results of this study have shown that none of the potentially propagated vulnerabilities could be reproduced. This implies that a greater set of repositories needs to be analysed to draw meaningful conclusions for the effectiveness and precision of call graphs to detect propagated vulnerabilities. The proposed improvements for vulnerability detection hint towards the immense potential call graphs yield.

There are several directions for future work. An experiment on a greater scale could be conducted, possibly with hand picked method to vulnerability associations. If such experiment would yield any results in which the vulnerability is reproducible it would showcase a great success for call graph analysis. Another option would be to consider implementing a proposed improvement to call graph analysis and show its potential in practice.

Eventually, once research in this topic has advanced further a tool, that possibly builds on top of Dependabot, could be developed. It would conduct method-level analysis for every security alert Dependabot generates and provide its own, more precise, recommendation. Obviously the aforementioned tool should only be developed if the results of future research suggest that call graphs are indeed a good way to detect vulnerabilities more precisely.

Together with the three other students that conducted the same experiment, namely, Tudor Popovici, Niels Mook and Christophe Cosse, it is planned to co-write a paper summarising the accumulated work. This collaborative study will cover various aspects surrounding the use of call graphs to detect propagated vulnerabilities.

## REFERENCES


[1] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. [n.d.]. On the Use of Dependabot Security Pull Requests. ([n. d.]).

[2] P. Boldi and G. Gousios. 2020. Fine-Grained Network Analysis for Modern Software Ecosystems. *ACM Transactions on Internet Technology* 21, 1 (Dec. 2020), 1:1–1:14. https://doi.org/10.1145/3418209

[3] J. Hejderup, A. v Deursen, and G. Gousios. 2018. Software Ecosystem Call Graph for Dependency Management. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. 101–104.

[4] B. Holland. [n.d.]. Call Graph Construction Algorithms Explained. https://ben-holland.com/call-graph-construction-algorithms-explained/

[5] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2017. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (May 2017), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[6] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 84–94. https://doi.org/10.1109/ASE.2017.8115621

[7] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2011), 772–787. https://doi.org/10.1109/TSE.2010.81

[8] F. Spoto. 2016. The Julia Static Analyzer for Java. In *Static Analysis (Lecture Notes in Computer Science)*, Xavier Rival (Ed.). Springer, Berlin, Heidelberg, 39–57. https://doi.org/10.1007/978-3-662-53413-7_3

[9] K. Vaniea and Y. Rashidi. 2016. Tales of Software Updates: The process of updating software. 3215–3226. https://doi.org/10.1145/2858036.2858303

[10] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek. 2018. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. In *2018 IEEE Symposium on Security and Privacy (SP)*. 374–391. https://doi.org/10.1109/SP.2018.00003