# Improvement Analysis of Function-Level over Package-Level Vulnerability Recommendations

Niels Mook
Delft University of Technology
The Netherlands

Mehdi Keshani
Delft University of Technology
The Netherlands

Sebastian Proksch
Delft University of Technology
The Netherlands

## ABSTRACT

Software reuse in the form of dependencies has become widespread in software development. However, dependencies have the potential to suffer from vulnerabilities, thereby potentially putting depending projects at risk. Dependency analysis software can be used to manage vulnerable dependencies, such as Dependabot. Yet, such programs are generally inaccurate as a result of false positives, due to the limitations of package-level analysis.

In the case of a false positive vulnerability recommendation, a software project imports a vulnerable dependency, but does not use any of its vulnerable functions. While most developers already do not pay enough attention to using vulnerable dependencies, false positives can only make this worse. Instead, function-level vulnerability analysis has the capability to eliminate package-level false positives.

In this paper, research is performed to gain quantitative insight in the improvement of function-level over package-level analysis in terms of recommendation correctness. A package-level analysis simulation in combination with a function-level analysis was performed, built with the FASTEN framework. The latter uses RTA call graph generation and method tracing to remove package-level false positives. In total, 4071 open-source repositories were analyzed with 393 open-source vulnerabilities, of which 259 projects had positive recommendations. Comparison shows that 85% of package-level recommendations are false positives, which are removed by performing function-level analysis instead. This indicates significant improvement by function-level analysis. Research on greater data sets would be needed for further insight in this improvement.

## 1 INTRODUCTION

*Vulnerabilities and False Positives.* Java programmers often use build automation tools, such as Apache Maven, Apache Ant, or Gradle. Among other things, these tools can make external projects usable within the project in the form of packages, while the programmer only has to declare used libraries in a specified file.[1] External libraries used in a project this way are called the *dependencies* of the project.

Dependencies allow programmers to reuse each other's code. However, the dependencies could have vulnerabilities, possibly leaving depending projects vulnerable as well. Exploitation of these unattended vulnerabilities by malicious persons or organizations can lead to dire consequences. One striking example of this was the Equifax data breach in 2017, where failure to update a vulnerable dependency resulted in sensitive personal information of 143 million Americans being stolen [4]. With the continuous rise in overall cyber-attacks [9], the risk of vulnerability exploitation is here to stay.

To fight such inheriting of vulnerabilities from dependencies, vulnerability analysis software has been developed to track down and report vulnerable dependencies in projects in the form of recommendations. One such analysis tool is Dependabot.[2] Dependabot analyzes dependencies on a *package-level*. This means that a dependency is flagged as a vulnerability when one of its methods is known to be vulnerable. Note that this is independent of whether the method is used or not.

However, when none of the vulnerable methods of a flagged vulnerability is directly or indirectly called by the project, such a vulnerability warning becomes a *false positive*. In that case, there is a vulnerable method inside one of the project's dependencies, but it is not used by the analyzed project and therefore not a direct vulnerability. To that end, Dependabot is capable of giving many such false positive warnings. How many is not certain, but it is very likely that false positive warnings waste the time of developers and annoy them, as they are counterproductive.

When warnings are regarded as mostly false positives, correct vulnerability warnings might be discarded, leaving the project and dependents vulnerable, acting as a crying wolf [2]. Furthermore, the importance of increased awareness on dependency vulnerability is emphasized by the fact that 69% of developers are not aware of vulnerable dependencies in their products, and 81.5% do not care about updating dependencies at all [8]. This is even further emphasized by Synopsys' OSSRA report, which stated that 84% of 1,546 scanned commercial codebases contained at least one open source vulnerability [10].

*Fine-Grained Analysis and Related Work.* Instead of package-level, methods are researched to perform vulnerability analysis on a more fine-grained *function-level* [2, 5]. This means that a dependency is only flagged vulnerable when a vulnerable method in that dependency is called by the dependent project. However, this form of analysis is more complex. In order to check whether a vulnerable external method is called, a *call graph* (CG) must be constructed for the project. Call graphs describe which methods call each other, and thus whether the vulnerable dependency method has been called.

Reasons to change to function-level analysis are mainly improvements on the before mentioned inefficiency of package-level analysis, and the resulting annoyance and (indirectly) reduced vulnerability awareness. Keeping in mind the increased complexity and difficulties of function-level analysis on ecosystem-scale, actionable dependency management resulting from function-level analysis is considered a big step [5]. All this is only amplified by the immense

[1] https://maven.apache.org/what-is-maven.html

[2] https://dependabot.com/

size of code repositories and their releases. In addition, impact analysis for JavaScript dependencies by Decan et al. [3] shows that for every vulnerable package version, there are more than tenfold depending packages at risk on package-level. This shows that implementation of a working function-level analysis software is meaningful and can realize significant impact.

To realize function-level dependency management, the FASTEN Project[3] is TU Delft led project that has built a framework to perform Fine-Grained Call Graph (FGCG) vulnerability analysis on software ecosystems in Java, Python, and C. The framework is still in active development though and not yet easily usable or regarded as stable by its developers. Still, with the help of the FASTEN Project and access to its database and code, this research was able to perform function-level vulnerability analysis.

*Research Question.* Ref. [2, 5] provide insight on how function-level analysis improves package-level analysis on a theoretical level. However, no research has been yet done on this improvement in practice. No numbers are available on the amount of false positives generated by package-level vulnerability analysis on publicly hosted projects. The difference might be significant or negligible.

The aim of this work is to fill this gap and provide quantitative insight in the improvement in recommendation correctness that fine-grained function-level analysis has over coarse-grained package-level analysis by elimination of false positives. This will be investigated by taking the difference in the amount of package-level and function-level recommendations, resulting in the amount and ratio of false positive recommendations produced by package-level analysis. Recommendations will be generated for a set of repositories on function-level and package-level using the framework of the FASTEN Project.

This research found that 85.3% of the package-level recommendations are false positive. This shows the significant improvement of function-level analysis over package-level analysis in correctness. On the other hand, this result also shows the lack of relevance of package-level recommendations.

However, consideration should be given to the research scope and limitations. The research makes use of a relatively small subset of open-source Java repositories hosted on GitHub[4] and open-source vulnerabilities.[5] In addition, the RTA CG generation algorithm whose limitations [6] are to be considered. Still, this paper hopes that the results are relatively representative to larger sets of repositories and vulnerabilities.

*Structure.* Section 2 offers background information to help understand the concepts in this paper. Section 3 lays out the overall methodology used to gather the sample repositories, to perform package-level and function-level analysis, and to compare the vulnerability warnings. Section 4 gives exact details on the experimental setup used to generate the recommendations. Section 5 perform the comparison of recommendations and presents the results. Section 6 will describe considerations on ethics and reproducibility during the performed research and writing of this paper. Section 7 will discuss results in consideration with found shortcomings of the methodology and limitations of implementation. Finally, section 8 will provide the conclusion of this paper.

[3]https://www.fasten-project.eu/
[4]https://github.com/
[5]https://github.com/fasten-project/fasten/wiki/Vulnerability-Analyzer

## 2 BACKGROUND

This section provides background information on call graphs, and package-level and function-level vulnerabilities. Due to only Java projects being analyzed, function-level analysis will from now on also be called method-level analysis.

### 2.1 Call Graphs

A simplified view of a call graph of a project is a directed graph $G = (V, E)$. Here, nodes $V = V_{int} \cap V_{ext}$, containing all nodes representing methods internal and external to the analyzed project. Edges $E \subseteq V \times V$ represent the performed method calls between methods [2]. The notion of *stitching* external nodes to internal nodes, as described in detail in the aforementioned paper, is assumed to already have happened and instead $V_{ext}$ simply means all methods contained in dependencies of the analyzed project.

Call graphs can be generated either statically or dynamically. For this project, static generation was chosen. However, static call graphs are neither complete nor sound [2]. Static call graph generation can miss method calls by dynamic dispatch. When this happens to a vulnerable method, this would lead to a false negative vulnerability recommendation. Alternatively, the static call graph can identify execution paths that (are designed to) never occur during runtime, potentially causing a dynamic false positive [2].

Finally, there are multiple call graph generation algorithms available, most notable Class Hierarchy Analysis (CHA) and Rapid Type Analysis (RTA). RTA produces a more precise call graph that contains less false negatives, while CHA is faster to generate [1, 6]. So, if possible, RTA generation is preferred.

### 2.2 Package-Level Vulnerabilities

It is important to note that a limited form of package-level analysis is used in this research. As Boldi et al. [2] explain, vulnerability analysis is done on *resolved source call graphs*, that recursively and uniquely contain all dependencies, including those of all dependencies themselves. However, the scope of this research only includes projects with vulnerable dependencies for which developers are able to update the dependencies and resolve the associated vulnerability. This way, the results from performed vulnerability analysis could be used as recommendations to a developer. Therefore, only

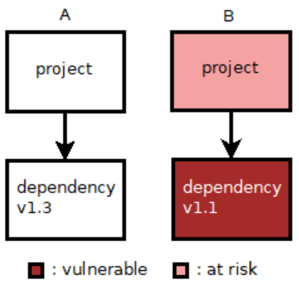

**Figure 1: Direct package-level vulnerability**

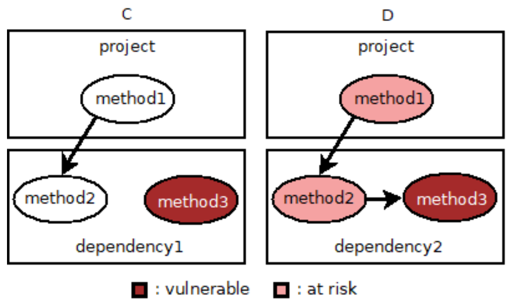

**Figure 2: Direct method-level vulnerability**

direct dependencies that are included in the project dependency file of a project are analyzed.

Situation B of Figure 1 illustrates a package-level vulnerability as used in this research. If a direct vulnerable dependency version is present, then the project is concluded as at risk, and bots would report the vulnerable dependencies to the developers of the project as recommendations to update. In this case the vulnerability applies to the project on a package-level.

### 2.3 Method-Level Vulnerabilities

A project is vulnerable on method-level when its call graph contains a path from a vulnerable external method to an internal method. Again should be noted, that this is a limited form, because only the call graph of the project with direct dependencies is considered. Normally, the path could go through multiple dependencies.

Similar to how Boldi et al. [2] illustrate it, Figure 2 shows a simplified method-level analysis of package-level vulnerable situation B from Figure 1. Note here that in both situations, *dependency1* and *dependency2* are vulnerable on a package-level like in Figure 1 In situation C, there is no path from the vulnerable *method3* to internal *method1*. Therefore, the project is not at risk and the package-level recommendation would be a false positive. As is the case in situation D, when method tracing results in a path being present from vulnerable *method3* to internal *method1*, the project is at risk at method-level.

## 3 METHODOLOGY

This section will expand on the steps taken to generate and analyze results of the two vulnerability analysis methods. Each following subsection represents one step of the process. In overview, first projects are selected and vulnerability information is acquired. With this information, package-level analysis is performed first, generating the package-level vulnerability recommendations for a subset of selected repositories. Then, method-level analysis is performed on this repository subset, resulting in method-level vulnerability recommendations. Finally, both sets of recommendations are compared for false positives and analyzed in further detail.

### 3.1 Project Selection and Vulnerabilities Acquisition

Firstly, before the recommendations of the two analysis methods can be generated, a set of projects is needed to generate them for. GitHub hosts over 128 million public repositories [7]. To narrow the project selection down, the FASTEN Project database was queried for GitHub hosted projects that have been known to have method-level vulnerable dependencies in past releases on the Maven Central Repository.[6]

Although the FASTEN Project already has a method-level vulnerability analysis procedure available for the research to use, another procedure was written that analyzes the most recent project repository code. This is done, because FASTEN's analyzer relies on the latest *release* of a project, which may be months ago. Therefore, a vulnerability in a repository could have already been resolved in the current state of the default branch of the project repository. The written procedure instead downloads the most recent version of the project repository's default branch.

Secondly, vulnerabilities (also called advisories) need to be acquired to perform analysis against. In order to perform this package-level and method-level comparison, at the very least vulnerabilities need to have some form of identification to distinguish them, a list of vulnerable dependency versions, and a list of vulnerable methods affected for each version. This vulnerability information was queried from available data on the FASTEN Project server.

### 3.2 Package-Level Vulnerability Analysis

To generate package-level vulnerability recommendations, the dependency file of each project is analyzed on the presence of any vulnerable package versions acquired during the previous step. Although this analysis done for thousands of projects, it only takes a few seconds to execute. If a vulnerable package version is present, it should be logged together with its associated vulnerability. From what was observed, one vulnerability is associated with one package version per project. Hence, this is also the definition of a package-level recommendation during this research.

This research did not use external package-level analysis software, but simulates it instead. Normally, package-level vulnerability analysis is performed by vulnerability analysis software bots such as Dependabot. However, for this research it was possible to simulate package-level analysis by such tools in Java code, with help of the vulnerability information provided by the FASTEN Project. Equal recommendations were produced with a simulation written[7] using the Ruby source code of Dependabot, which is publicly hosted on Github.[8][9] Therefore, the Java-based simulation recommendations are assumed as representative of other package-level analysis software.

### 3.3 Method-Level Vulnerability Analysis

Method-level vulnerability recommendations are generated in several steps. First, one merged call graph is generated of both the projects and the vulnerable dependency in question. This generation makes use of the RTA algorithm. Afterwards, the call graph is checked for the presence of vulnerable methods part of the dependency. Finally, when such a method is present, method tracing checks for the presence of calling methods internal to the analyzed

[6]https://search.maven.org/
[7]https://github.com/jakub014/CG-dependency-analyzer/blob/master/scripts/depbot-script/security-script.rb
[8]https://github.com/dependabot/dependabot-core
[9]https://github.com/dependabot/dependabot-script

project. These internal methods are at risk of the associated vulnerability. During this research, one method-level vulnerability recommendation is defined as the set of all internal methods at risk to one vulnerability.

Fine-grained method-level analysis is, however, costly to perform on a large amount of projects. Luckily, as vulnerable methods are always part of a vulnerable dependency, all method-level vulnerable projects must also be vulnerable at package-level. Therefore, the resulting vulnerable projects from package-level analysis will serve as input to method-level analysis.

*Call Graph Generation.* All call graphs are generated statically, due to dynamic analysis being out of the scope of this project, and due to limited time and resources. Program analysis in general can be done statically or dynamically, and the same holds for call graph generation. However, the FASTEN Project fine-grained analysis framework that is used in this research is based on static analysis. Therefore, dynamic analysis is out of the scope of this research. Furthermore, dynamic call graph generation is more complex and time consuming, while static call graph generation is already complex and can cause heap space errors on desktop computers. Due to these reasons, the amount of projects and limited research time, static call graph generation is chosen as generation method. Still, the shortcomings of static call graphs should be taken into consideration when interpreting the results.

## 3.4 Recommendation Analysis

Recommendations on method-level can be seen as a subset of recommendations on package-level, which makes false positive calculation simple. Taking the difference between method-level and package-level recommendations results in *false positive* package-level recommendations. This way, method-level analysis functions as a validation of package-level analysis, as one can conclude from previous explanation of both analysis methods. The resulting ratio of false positive recommendations to the total (package-level) recommendations is what is of interest to this research.

In order to create the conditions of a fair comparison between both recommendations, only package-level recommendations will be compared if their repositories also completed method-level analysis. In the event that repositories did not complete method-level analysis, but do suffer from package-level recommendations, these recommendations would automatically become false positives. This, because there are no method-level recommendations to compare against. As a result, the false positive ratio would become higher than might be the case in reality.

To help interpreting results and giving insight into the reliability of results, additional statistics have also been gathered on performed project analysis, such as the amount of at-risk internal methods (also called *impact points* versus originating vulnerable dependency methods, and spread of false positives over projects.

## 4 EXPERIMENTAL SETUP

This section clarifies in detail the experimental setup used for gathering repository and vulnerability data, and for generating recommendations.

### 4.1 Data Collection

*Repository Selection.* The project repository URLs are queried from metadata database of the FASTEN Project. The repository URL resides in the *repository* field in the *packages* table.[10] In this query, it is important that projects have the *vulnerabilities* JSON field defined in the *metadata* field in the *package_versions* table, such that the query includes:

```
metadata -> 'vulnerabilities' IS NOT NULL
```

The resulting 7,638 repository URLs are the individually matched with regular expressions to extract the GitHub username and the repository name. Together, they form a combination *"username/repository"* that can be used to download dependency files and generate a standardized GitHub URL to download the complete repository. This generated repository URL was then queried over the GitHub API[11]. If it returned repository information, the URL was assumed as valid. This resulted in 6,717 of such combinations.

*Vulnerability Acquisition.* The FASTEN Project stores found vulnerability data from multiple sources in the form of JSON files called Vulnerability Object Definitions (VODs),[12] which also contain the generalized information record of the vulnerability called CVE. This includes the CVE identifier, a brief description of the vulnerability, and any pertinent references.[13] This information is useful to get more information on the vulnerability.

To be able to perform vulnerability analysis of repositories, the data contained in following fields of the VOD are required:

- *vulnerable_purls:* Package coordinates of vulnerable packages, in the form of standardized *package URLs.*[14] Just like the URL found in the Metadata Database, package URLs contain the *"username/repository"* combination, in addition to the version.
- *vulnerable_fasten_uris:* Vulnerable callables (also called methods), listed using FASTEN URI format.[12] This is a unique identifier generated by the FASTEN Project fine-grained analysis framework for vulnerable methods of a specific package version.

A script was constructed in the Python programming language to gather VODs having a non-empty *vulnerable_fasten_uris* field, which also guarantees that the *vulnerable_purls* is non-empty. This resulted in 393 VODs found spread over 211 dependencies.

*Dependency File Extraction.* To perform package-level vulnerability analysis on a project, its dependency file needs to be analyzed. During the first few weeks of research, the complete repository was downloaded to perform this analysis, while only the dependency file was needed. As stated in section 2.3, all method-level vulnerable projects must first be package-level vulnerable. As a consequence, the repository only needs to be completely downloaded for method-level vulnerability analysis once vulnerable dependencies have been found inside its pre-extracted dependency file. This saves time when analyzing thousands of projects.

[10]https://github.com/fasten-project/fasten/wiki/Metadata-Database-Schema
[11]https://docs.github.com/en/rest
[12]https://github.com/fasten-project/fasten/wiki/Vulnerability-Analyzer
[13]https://cve.mitre.org/cve/identifiers/index.html#defined
[14]https://github.com/package-url/purl-spec

A dependency file can reside in the root directory or a child directory of the repository. For the collected projects, a considerable portion of dependency files resided in a child directory. Only requesting dependency files in the root folder over the GitHub API would, thus, skip over child directory project dependency files.

Therefore, the open source Dependabot demonstration Ruby script,[15] which makes use Dependabot source code, is altered to return all dependency files present in the repository with the associated relative file path. This way, each child directory project can be analyzed separately, making sure that no projects are lost. The script results in 17,142 Maven POM files and 2,855 Gradle dependency files (which were converted to a simple *username/repository$version* format for ease of dependency extraction).

## 4.2 Vulnerability Analysis

A program is constructed in the Java programming language to perform package-level vulnerability analysis, and on method-level as well with the use of the fine-grained analysis framework of the FASTEN Project.[16]

Each project has its dependency file analyzed on package-level vulnerabilities, by parsing out used dependency package versions and intersecting them with known vulnerable package versions present in package URLs in the previously acquired VODs. This is inefficiently done on a line-by-line basis; all vulnerable package versions were checked for each line of the dependency file. For each project, found vulnerable packages are then logged . Due to this inefficiency, the associated CVE code is linked to the vulnerability only after method-level analysis. Finally, this results in the partial package-level vulnerability recommendations for that project, completed after fine-grained analysis.

Once vulnerable dependencies have been confirmed in a project's dependency file, fine-grained analysis is performed on it.

First, the project repository is downloaded from GitHub. Then it is built from the root folder, normally resulting in all child projects being built as well. The building process stores resulting JAR files in the target directory in corresponding child directories. Then, the target directory is analyzed for the existence of the corresponding project child directory.

Once the project JAR is built and found, a call graph is generated. This is done with help of the FASTEN OPAL plugin[17] which in turn uses the OPAL static analysis platform.[18] More specifically, CG generation is done using the *ExtendedRevisionJavaCallGraph* class, making use of the RTA algorithm. This algorithm is supplied as *algorithm* parameter to the *CallGraphConstructor* class.

Then, for each of the vulnerable dependencies found, the vulnerable dependency JAR are retrieved from the Maven repository, and each has its call graph generated as well, again using RTA. The resulting two of call graphs of both the project and vulnerable dependency are then merged by the *LocalMerger* class into one larger call graph, containing the FASTEN URIs of both the project and the vulnerable dependency. Note, therefore, that this process and following steps are done separately for each vulnerable dependency.

[15]https://github.com/dependabot/dependabot-script
[16]Source code of the vulnerability analysis program written for this research can be found at: https://github.com/jakub014/CG-dependency-analyzer
[17]https://github.com/fasten-project/javacg-opal
[18]https://www.opal-project.de/

**Algorithm 1:** Method-Level Analysis Procedure

```
Data: vulnDeps: vulnerable project dependencies,
CG: complete project call graph
Result: Project methods affected by vulnerable methods of
        provided vulnerable dependencies
1  $M \leftarrow \emptyset$
2  foreach $vulnDep \in vulnDeps$ do
3      $VODs \leftarrow$ getVODs($vulnDep$)
4      foreach $VOD \in VODs$ do
5          $URIs \leftarrow$ getURIs($VOD$)
6          foreach $URI \in URIs$ do
7              if $URI \in CG$ then
8                  $M \leftarrow M \cap$ getAffectedMethods($URI, CG$)
9              end
10         end
11     end
12 end
13 return $M$
```

This is done, for efficiency and to prevent exceptions being raised during call graph generation as much as possible.

To finally find out whether the project is vulnerable on a method-level (and to complete the partial package-level recommendations), Algorithm 1 is performed:

(1) For each dependency, get all associated VODs (line 1-2). In the currently used process, this is always one vulnerable dependency.
(2) For each VOD, get all associated vulnerable FASTEN URIs stored inside the *vulnerable_fasten_uris* field (line 3-4).

**Algorithm 2:** Method Tracing

```
Data: URI: vulnerable fasten URI,
CG: complete project call graph
Result: Project methods affected by the provided
        vulnerable method
1  $A \leftarrow \emptyset$
2  $visited, queue \leftarrow URI$
3  while $queue \neq \emptyset$ do
4      $currentMethod \leftarrow$ poll($queue$)
5      $callers \leftarrow$ getCallers($currentMethod$)
6      foreach $caller \in callers$ do
7          if $caller \notin visited$ then
8              if $caller \in CG_{project}$ then
9                  $A \leftarrow A \cap caller$
10             else
11                 $visited \leftarrow visited \cap caller$
12                 $queue \leftarrow queue \cap caller$
13             end
14         end
15     end
16 end
17 return $A$
```

(3) For each vulnerable FASTEN URI, perform an algorithm to find other methods calling this vulnerable method (line 5-7).

The method tracing function that is called *getAffectedMethods* on line 8 is performs an algorithm similar to breadth first search, and it is also akin to the impact analysis algorithm shown at page 3 of Ref. [5]. This results in Algorithm 2:

(4) Continue tracing through the vulnerable methods calls in the graph until either the current method is part of the analyzed project (instead of an external method part of the vulnerable dependency) or no more calls are made to this method.
(5) Return the resulting set of vulnerable internal project methods and the originating dependency method.

When the returned set of vulnerable internal methods is non-empty, the project is vulnerable on a method-level. In addition, all CVE codes are added to package-level recommendations for which the project has completed method-level analysis, completing both sets of recommendations.

## 5 RESULTS

Each run of the combined package-level and method-level algorithm ran for around 4 hours on 4 desktop computers. The build process of Gradle projects, although successfully automated, took too much time to perform for this number of projects. Therefore, Gradle projects were discarded.

After getting as many projects to successfully complete package-level analysis, out of 4071 distinct Maven repositories that completed, 580 contained at least one POM file with a package-level vulnerability. Out of these 580 repositories, 259 inner Maven projects with a package-level vulnerable POM file were successfully analyzed on method-level, including call graph generation and method tracing. A majority of the projects failed method-level analysis mostly due to unsuccessful local building.

Table 1 shows the results found for the 259 analyzed Maven projects. Package-level analysis resulted in 517 vulnerability recommendations in 259 projects, while method-level analysis resulted in 100 recommendations in 78 projects. Taking the difference between the two, this results in 417 false positive package-level vulnerability recommendations spread over 181 projects. The percentage after each recommendation amount indicates the size of the vulnerability type compared to that of package-level vulnerability recommendations.

**Table 1: Package-level and method-level analysis results**

| Vulnerability type | Projects | Recommendations |
|---|---|---|
| Package-level | 259 | 680 (100%) |
| Method-level | 78 | 100 (14.7%) |
| False positive | 239 | 580 (85.3%) |
| False positive only | 181 | 417 (61.3%) |

Finally, over all analyzed projects this results in a false positive ratio of 85.3%. This ratio is well above the expected ratio of 67%. This high false positive ratio of 85.3% shows for the relatively small sample of projects analyzed in this research, that fine-grained vulnerability analysis on a method-level has significant improvement over package-level analysis.

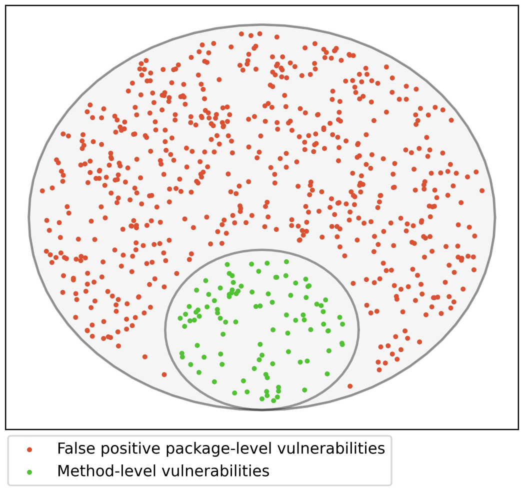

**Figure 3: Dot density Venn diagram of generated recommendations**

The dot density Venn diagram in Figure 3 illustrates this ratio by showing all individually found vulnerability recommendations into an intuitive picture, emphasizing the vast amount of false positive recommendations generated by package-level vulnerability analysis. The graph was generated with the script available at Ref. [11]. The area of the circles has the same ratio as the false positives.

Figure 4 shows the distribution of false positive package-level vulnerability recommendations over the 239 projects. A mean of 2.41 was observed and a standard deviation of 1.80 false positives. Therefore, although a vast majority of package-level recommendations are false positives, projects that contain false positive vulnerable dependencies mostly contain only 1 to 3 of them. This means that the 580 false positive vulnerable dependencies were evenly spread among the 239 containing projects, which makes these results more reliable.

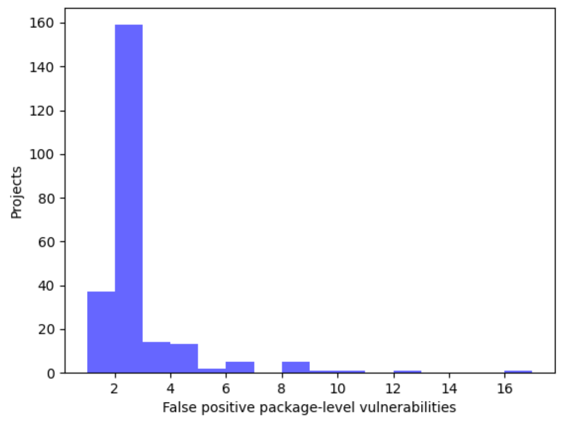

**Figure 4: Projects per number of false positive recommendations**

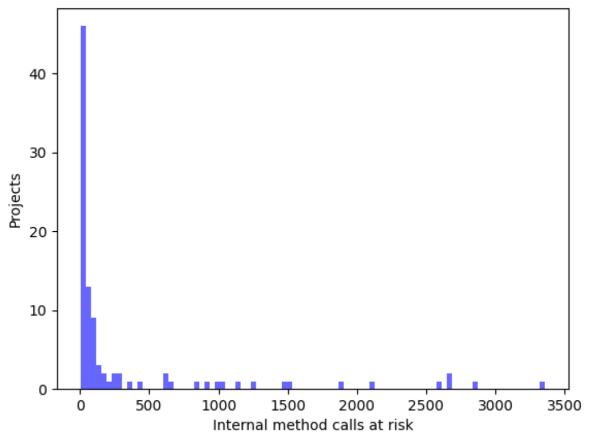

**Figure 5: Projects per number of internal methods at risk**

Figure 5 shows the distribution of impact points of vulnerable methods inside method-level vulnerable projects, being the number of internal method calls at risk by having a direct path to a vulnerable method. Note here that two outliers with above 10,000 impact points are left out for clarity. Excluding these, a mean of 346.49 impact points was observed and a median of 44. In fact, calculation results in 67% of projects suffering from method-level vulnerability recommendations having at most 100 internal method calls at risk.

On average there are 74.67 vulnerable URIs in a CVE, however, as Figure 6 illustrates, projects are affected by only 4.67 distinct vulnerable external methods on average, excluding one outlier of 418. Combined with the observation that 74 out of 78 projects vulnerable on a method-level (94.9%) only had one vulnerability recommendation, the previously determined number of internal methods at risk is high per external vulnerable method. Assuming a lenient prediction of 100 internal methods at risk per external vulnerable method used, the following is observed. Although only a small number of external vulnerable methods are used in a project on average, over 21 fold as many internal methods are at risk. Interpreted from the perspective of applying fine-grained analysis; for every vulnerable method caught by fine-grained analysis, over 21 compromised internal methods are likely to be caught as well according to the found data.

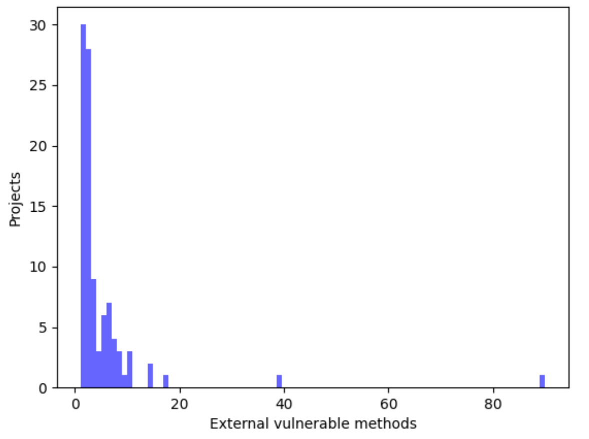

**Figure 6: Projects per number of distinct external vulnerable method used**

## 6 RESPONSIBLE RESEARCH

With regards to the ethical aspects of this research, attention was given to disclose neither repository names nor their vulnerabilities. Although the projects as well as all vulnerabilities are open source, it would still put projects in unnecessary danger of the exploitation of these vulnerabilities. In addition, care has been taken not to leave data out unless a valid reason is given, such as obvious outliers.

As is important in all research, reproducibility has also been considered throughout the duration of this work. One concern of reproducibility is the starting set of vulnerable repositories, which is obtained by querying the PostgreSQL metadata database hosted by the FASTEN Project. However, access to this database is private and was granted to this research. Anyone with access to it is able to rerun all queries performed by this research. In addition, the database is accessible through the REST API endpoints[19] such that one would be able to reproduce the queries in API calls. Finally and as mentioned in section 4.2, all scripts and programs written to generate given results are published at GitHub, and some of the written programs are also provided in pseudocode in this paper.

## 7 DISCUSSION

*Limitations.* The limited size of the repository and vulnerability set must be taken into account when interpreting the results. 680 vulnerability recommendations found over 259 projects cannot possibly represent all 128 million GitHub repositories, and this is not the goal of this paper. Instead, with the means that were possible and the lack of previous research in these quantities of false positives, these results are to be taken as a first insight in the level of improvement of fine-grained vulnerability analysis. Further insight can be obtained by performing similar research on a larger data set.

However, data set limitations are not the only imperfection of this research. Upon manual inspection of vulnerability information (VODs) provided by the FASTEN Project database, many VODs appeared to include unaffected versions of the vulnerable package. Manually inspection resulted in 47 out of 100 generated method-level recommendations containing an actually vulnerable version of the dependency.

In addition, manual inspection of method-tracings found non-existent edges from internal methods to the vulnerable dependency. Upon closer inspection, the majority of these errors resulted from merging both the project and dependency call graph, specifically with the *LocalMerger* class in the FASTEN core repository.[20] However, the exact cause of this inside the method was not investigated. Some of the non-existent edges were also generated by the RTA

[19]https://github.com/fasten-project/fasten/wiki/API-endpoints-for-Maven-projects
[20]https://github.com/fasten-project/fasten/tree/develop/core

CG generation algorithm. This probably happened due its limitations compared to dynamic generation algorithms and potentially inaccuracies.

These non-existent edges resulted in only 30 out of 47 method-level recommendations actually calling the reported vulnerable method. Therefore, results have to be interpreted with the inaccurate RTA algorithm and *merger* method in consideration. Due to the amount of method-level recommendations becoming smaller with these inaccuracies, the false positive ratio climbs. As a result, one can expect only more impact of method-level analysis over package-level analysis. On the contrary, the elimination of false negatives would increase method-level recommendations. However, this was not possible to investigate, as this would require dynamic CG generation.

*Improvements.* Something to improve in the code implementation of the package-level analysis, is the inefficient line-by-line checking of dependency files against vulnerable package versions. The vulnerability data is already in an efficient JSON format to make this improvement. For this research, this improvement would not change the recommendations in any way, and execution of package-level analysis for all 4071 repositories was already finished in seconds. Still, it would result in the CVE codes already being logged for during package-level instead of after method-level analysis.

Another improvement would be the inclusion of Gradle projects. A specific Gradle version was downloaded for nearly every Gradle project. On the other hand, many Gradle project builds seemed to hang up, without error logs. This caused Gradle project analysis to take too long, and this issue could not be resolved in time. Only 16.7% of extracted dependency files were of Gradle type, so discarding them could be afforded.

## 8 CONCLUSIONS AND FUTURE WORK

In this research, the recommendations of two methods of dependency vulnerability analysis have been compared. The first is a written Java simulation of coarse-grained, package-level vulnerability analysis, observed as representative as the analysis performed by bots such as Dependabot. The second is a written Java execution of fine-grained, method-level vulnerability analysis, including call graph generation and method tracing. This was possible with the help of the FASTEN Project's framework. Package-level vulnerability analysis is known to be bugged with false positive vulnerability recommendations, but it was not clear by how much. Consequently, it was not known by how much method-level analysis would improve correctness of recommendations. By comparing package-level recommendations to method-level recommendations, the number of eliminated false positives is found. This way, this paper contributed in providing quantitative insight into the improvement in recommendation correctness of function-level analysis compared to package-level analysis.

The main contributions of this paper are as follows. The resulting recommendations for both methods, that have been generated for a limited set of open-source projects and vulnerabilities, show that a false positive ratio of 85.3% is present in package-level recommendations compared to method-level recommendations over the same data set. In addition, every vulnerable external method puts around 21 internal methods at risk, which shows the significance of using only a single vulnerable method to a project. The combination of both these results shows that function-level vulnerability analysis has more than significant improvement on correctness of vulnerability recommendations compared to package-level analysis.

Although not all false positives could be removed from method-level analysis, and false negatives are an inherent risk to a static call graph generation algorithm, the results still show significant improvement of method-level analysis. And as a result, a first insight is given in the potential impact of function-level analysis. These findings hopefully motivate the continued development of method-level vulnerability analysis software, specifically the elimination of method-level false positives. For future research, further insight can be obtained by performing similar research on a larger data set. Finally, this research shows that a promising analysis tool is in sight, able to raise software security to a higher level.

## REFERENCES

[1] David F. Bacon and Peter F. Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '96)*. Association for Computing Machinery, New York, NY, USA, 324–341. https://doi.org/10.1145/236337.236371

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

[3] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, Gothenburg Sweden, 181–191. https://doi.org/10.1145/3196398.3196401

[4] Dan Goodin. 2017. Failure to patch two-month-old bug led to massive Equifax breach. https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug/

[5] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. 2018. Software Ecosystem Call Graph for Dependency Management. *International Conference on Software Engineering: New Ideas and Emerging Results* 40 (2018), 4. https://doi.org/10.1145/3183399.3183417

[6] Ben Holland. 2016. Call Graph Construction Algorithms Explained. https://ben-holland.com/call-graph-construction-algorithms-explained/

[7] Neeraj Kashyap. 2020. GitHub's Path to 128M Public Repositories. https://towardsdatascience.com/githubs-path-to-128m-public-repositories-f6f656ab56b1

[8] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do Developers Update Their Library Dependencies? An Empirical Study on the Impact of Security Advisories on Library Migration. *Empirical Software Engineering* 23, 1 (Feb. 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[9] Connor Perrett. 2021. Major cyberattacks have rocked the US, and there are 'a lot of different ways that ransomware actors can disrupt everyone's lives,' experts say. *Business Insider* (June 2021). https://www.businessinsider.com/cyberattacks-are-on-the-rise-in-the-us-experts-say-2021-6

[10] Paul Sawers. 2021. Synopsys: 84% of codebases contain an open source vulnerability. *VentureBeat* (April 2021). https://venturebeat.com/2021/04/13/synopsys-84-of-codebases-contain-an-open-source-vulnerability/

[11] Dinesh Vatvani. 2019. Making aesthetically pleasing dot density Venn diagrams. https://dvatvani.github.io/dot-density-venn-diagrams.html