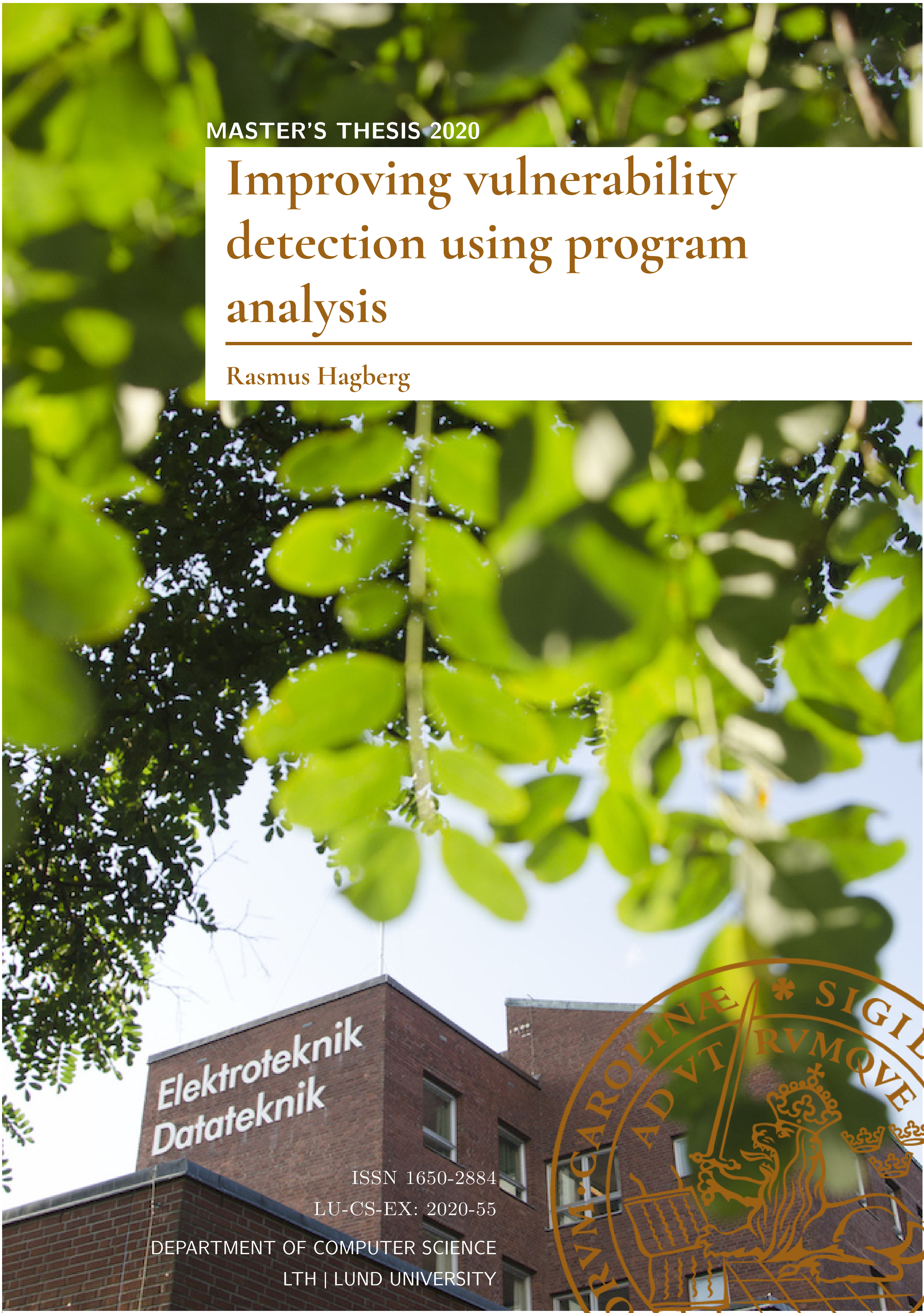
MASTER'S THESIS 2020

Improving vulnerability detection using program analysis

Rasmus Hagberg



ISSN 1650-2884
LU-CS-EX: 2020-55

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-55

Improving vulnerability detection using program analysis

Rasmus Hagberg

Improving vulnerability detection using program analysis

Rasmus Hagberg
dat14rha@student.lu.se

August 27, 2020

Master's thesis work carried out at Debricked AB.

Supervisors: Martin Hell, martin.hell@eit.lth.se
Christoph Reichenbach, christoph.reichenbach@cs.lth.se
Oscar Reimer, oscar.reimer@debricked.com

Examiner: Martin Höst, martin.host@cs.lth.se

Abstract

The use of open source libraries in all types of software has increased over the years, and this is likely to continue. The ability to use readily available libraries can be a great boon, saving lots of time and preventing needless recreation of existing functionality. This comes at a cost, however; these libraries can contain vulnerabilities that could allow ones code to be exploited despite there being no errors in the code that has been written. This problem is worsened with each additional dependency that is included, and with these dependencies often having dependencies of their own the situation quickly becomes hard to manage. With the industry trending towards relying more and more on freely available open source libraries these problems are likely to increase with time.

Conversely, simply including a library with a vulnerability does not necessarily make the code susceptible to exploitation. In order for this the actual vulnerable part of the library has to be used, and in such a way as to make it exploitable. The common way to determine if this is the case is expert judgement, but this is costly and time consuming.

This thesis proposes and evaluates a tool that automatically determines if a piece of software is susceptible to exploitation because of a vulnerable library by applying program analysis techniques to analyse in which way the library is used. We conclude that this analysis is able to reduce false positives by between 10% and 20%, with potential for further reductions as improvements are made to surrounding tools.

Keywords: Vulnerability, Dependency Management, Program Analysis, Call Graph, Static Analysis

Acknowledgements

I would like to thank my supervisors Martin Hell and Christoph Reichenbach for their advice in getting started as well as for their feedback on the thesis. I would also like to thank Debricked for allowing me to use their offices and vulnerability database while writing this thesis.

Contents

Chapter 1
Introduction

Vulnerable software dependencies are an ever increasing problem for the security of all kinds of applications. If not properly mitigated these vulnerabilities can have disastrous consequences for the users of these applications. Take for example the widely published Equifax breach which was caused by a failure to update an open source library [7]. This is a widely acknowledged problem, with `OWASP` including it in their "Ten Most Critical Web Application Security Risks" at number nine as "Using Components with Known Vulnerabilities" [16]. Awareness has grown following large and widely published and discussed vulnerabilities such as Heartbleed[1], but despite this, there are no industry standard practices for handling these issues.

Despite fixes to these libraries often being available, vulnerable versions often continue being used, either because of a lack of knowledge about the vulnerability and the potential impact of it, or because of a lack of will to mitigate it. Updating dependencies can take significant amounts of time and money, and brings with it risks to the stability of the product. The updated version might have bugs not present in the current version, or it might have been changed in such a way as to require rewriting parts of the rest of the program. Creators of software which is expected to be stable over an extended period of time might not want to introduce undue changes, software which runs on integrated devices might not have the connectivity required to easily be updated, etc. The reasons to hold off on updating are plentiful.

Furthermore, often significant effort is required to even know if a vulnerability in one of the dependencies does in fact affect the program, further increasing the unwillingness to take these costly steps when there might not even be an issue. Expert advice is often relied on for this, but this technique is fraught with problems. Expert advice is expensive and time consuming. Manually analysing how code is used is a significant task, and there is no guarantee that the expert opinion is actually correct, it is after all just an opinion. Since expert advice by nature can not automated, it also does not scale as well as a programmatic

[1] `http://heartbleed.com/`

solution does.

Currently there exists no simple way of handling these issues. Makers of libraries expect their users to always use the latest version in order to benefit from new features, bug fixes and security patches, but as discussed there are various reasons to not always do this. In these cases and others a call has to be made on whether it is worth the hassle and work to update the library, or if it is worth the risk to keep the vulnerable version. Evaluating the risk is an effort in and of itself, and often there is no good answer available even if the effort is put in, making this decision even harder.

In this thesis we present a vulnerability filtering program that aids in this decision making, helping to determine if a vulnerability does indeed affect a program, or if it can safely be ignored. This is done by using static program analysis to determine what part of a library is affected by a security vulnerability, and then determining if this part is used by the program, potentially making the program susceptible to exploitation.

1.1 Debricked

This thesis work was carried out at Debricked AB[2], and conceptually builds upon the work from a previous research project at LTH, SECONDS [8], further developed by Debricked into a SaaS tool. The vulnerability filtering program presented in this thesis will act as a supplement to a tool of this type, helping to reduce false positives.

The tool created by SECONDS analyses dependency files and compares them against a database to determine if there are any known public vulnerabilities against any of the dependencies. The vulnerability filter presented in this thesis uses this type of information as a baseline and goes one step further, analysing how the dependencies are used, in order to determine if a dependency is used in such a way as to expose a vulnerability. If it is not it does not matter that the dependency has said vulnerability against it, and a false positive is avoided.

While largely standalone the program does make use of some proprietary Debricked resources, such as a database that is queried to determine if a library is affected by a vulnerability, and if so what versions are. This could be done by instead parsing an open resource such as NVD [15], but the ability to utilise the Debricked database and its API has simplified work.

1.2 Outline

Section 2 starts off with some background about the problem, introducing some concepts later utilised. Section 3 introduces the idea behind the vulnerability filter and its limitations as well as explaining how it works in some detail. Section 4 presents and analyses the results produced by the vulnerability filter and Section 5 concludes with a short discussion about what can be learned from the process of writing this program and observing the results, as well as the viability of applying it in practice. Section 6 discusses the potential for improving and further developing the vulnerability filter.

[2] `https://debricked.com/`

Chapter 2
Background

In order to understand the rest of this thesis some background and discussion is needed on the situation as well as various concepts discussed, and techniques utilised. This section aims to provide this by introducing terminology, some important underlying technologies and concepts, as well as related work.

2.1 Terminology

During the course of this thesis some terms will be used that describe properties of the program and of different types of analysis. In this section we will describe a few of the key terms, and how they are used in the thesis.

True and False Positives and Negatives

A *positive* is something that is marked as a vulnerability by analysis. A *true positive* is something that is marked as a vulnerability, and is in fact a vulnerability. A *false positive* is something that is marked as a vulnerability, but is not in fact a vulnerability.

A *negative* is something that is *not* marked as a vulnerability by analysis. A *true negative* is something that is not marked as a vulnerability, and is not a vulnerability. A *false negative* is something that is *not* marked as a vulnerability, but is in fact a vulnerability.

Having significant amounts of *false positives* leads to low *precision*. Having significant amounts of *false negatives* leads to low *recall*.

Recall

Recall is the fraction of existing vulnerabilities actually captured. Having high recall means one can be reasonably sure that most vulnerabilities will be marked as such by the analysis.

If 100% recall can be guaranteed the analysis is called *sound*. Mathematically recall is defined as:

$$r = \frac{TP_c}{TP} \tag{2.1}$$

Where TP_c is the amount of *true positives* captured, and TP is the amount of *true positives* that exist.

Recall is important for the analysis to have value to the user. If one can show that the analysis has high recall the user can be confident that once the analysis shows no positives, it is likely that there are in fact no positives.

Soundness

Soundness is a property of a type of analysis. An analysis is called *sound* if it is guaranteed to have full *recall*. For a longer discussion on soundness and how it relates to and applies in our analysis see Section 2.3.

Precision

Precision is the share of the captured *positives* that are *true positives*. Having high precision means that one can be reasonably sure that a positive is in fact a true positive, and not a false alert. Mathematically precision is defined as:

$$p = \frac{TP_c}{P_c} \tag{2.2}$$

Where TP_c is the amount of *true positives* captured, and P_c is the amount of *positives* captured.

Precision is important to instil trust in the analysis. If the analysis has low precision the user can not know if a reported positive is in fact really an issue. This can lead to users dismissing the results of the analysis, since they do not know if the potential issues presented really are issues.

2.2 Call graphs

Call graphs are a central concept in the methodology of this analysis, and a key area to get working well. For this reason we spend some extra time explaining how they work, and how they are utilised by the vulnerability filter.

2.2.1 Introduction to call graphs

A call graph is a graph describing the potential execution paths of a program. Each method is represented by a node, and is linked by edges to the methods (nodes) that can be called from the method in question. Following the edges from a node allows one to determine which other methods can be called from a particular method, including transitively. See Figure 2.1 for an illustration of an example.

Call graphs can be generated in different ways, and these can be achieved through two different types of analysis: Static and Dynamic Analysis. The difference between these types

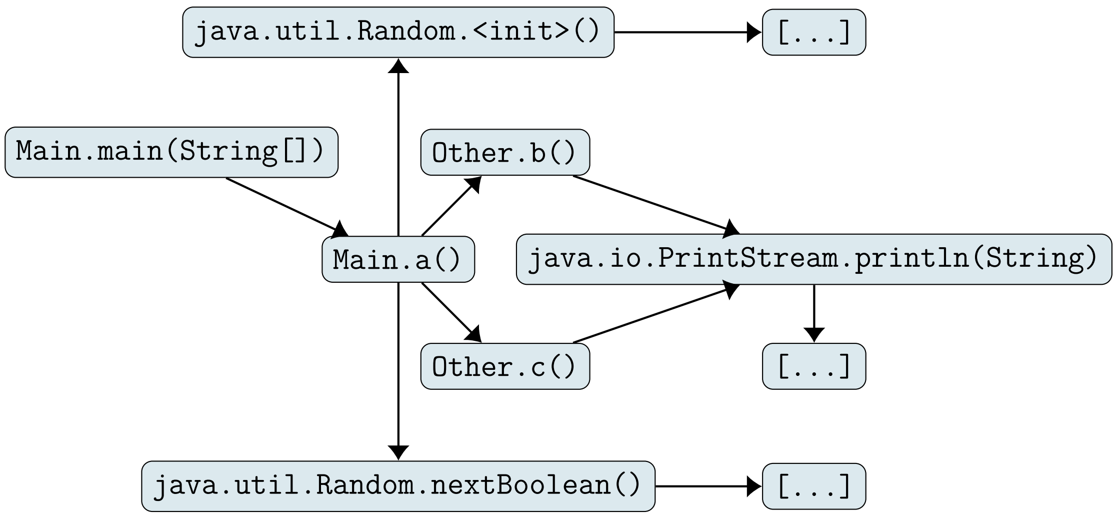

java.util.Random.<init>()
[...]
Main.main(String[])
Other.b()
Main.a()
java.io.PrintStream.println(String)
Other.c()
[...]
java.util.Random.nextBoolean()
[...]

Figure 2.1: Illustration of a call graph. The corresponding code is available in appendix A. Some methods are omitted for brevity.

of analysis, including up- and downsides and how these apply to the vulnerability filter are discussed in Section 3.1.3.

2.2.2 Difficulties

Generating call graphs is a significant part of this thesis, and a proper research problem in its own right (see for example [9]), so finding previous work in this area will be a good springboard to work off, and simplify future work. Since call graph generation software generally targets one language, the availability of tools in this area depends greatly on the language, and is a significant factor in deciding what language or languages to support. Novel software will not be written for this purpose, since that would be a large project on its own, and as such out of scope for this thesis.

Since call graph generation is very much nontrivial, the quality will vary between languages and implementations. While there being good call graph generation libraries available is a prerequisite when selecting what language or languages to support, no detailed comparison of different call graph generation libraries, such as done in for example [1], will be performed during the course of this thesis.

The call graph generation is necessarily language specific, so compartmentalising this properly and separating it from the rest of the program logic is important. This will allow us to add support for more languages by modifying only part of the code.

2.3 Soundness

When working in security it is always good to make conservative estimates. It is better to err on the cautious side and claim there is a vulnerability when there is not than to falsely claim a program to be safe when in fact it is vulnerable. In short; false positives are better than false negatives, and we prioritise recall over precision while not ignoring precision altogether. Since this analysis by nature will assume the program to be safe unless a vulnerability can be found, we need to be careful in minimising the amount of false negatives, maximising recall. With this in mind one can not let the precision get too low, or it might lead to users dismissing true positives as false, defeating the point of the analysis.

2.3.1 Soundness in call graph generation

The question of whether to prioritise soundness when designing call graph generation algorithms is an interesting one. As mentioned false negatives can have dangerous consequences, but overwhelming developers with false positives can have equally dangerous consequences if it causes them not to find and fix the true positives. Interviews with users of a similar tool [18] showed that too many false positives can have as large of a negative impact on tool usage as too many false negatives.

Some language features such as reflection and dynamic code generation are hard to follow and static analysis alone is incapable of producing sound call graphs in this scenario, without including so many false positives as to make the analysis useless [12]. Dynamic analysis on the other hand is inherently unsound, since it relies on every possible call actually being made during analysis, which in practice one can never be sure is the case.

This can be solved in a few ways. One sound way is to assume that any reflective call could call any method. This means that any node (method) in the call graph containing a reflective call will have an edge to every other node that it can reach. As one can imagine, this will generate so many false positives that it makes the analysis practically useless, but in the general case this is the only way to guarantee the absence of false negatives.

If this amount of false positives can not be accepted some assumptions have to be made about what methods the reflective call can reach. One way of doing this is by deploying dynamic analysis, and simply observing what methods it calls in practice, for example while running tests, as in [17]. This is then assumed to be the complete set of methods this reflective call will reach. This is not sound, and carries the risk of introducing false negatives.

Another way to reduce the amount of false positives is to make some assumptions about what methods the reflective calls can reach, without resorting to dynamic analysis. In their 2015 paper Livshits et al. introduce the concept of "soundiness" [14], where sound analysis is used for most parts of a language, but unsound assumptions are made where this is not feasible, for example in the case of reflection. This way one can get a result that, while not being sound in theory, in practice contains most of the actual methods called using only static analysis, without generating unwieldy amounts of false positives.

2.4 Related work

A similar approach to ours is presented by [17], through relying on dynamic analysis, gathering call traces during the execution of tests. This means they have to rely on extensive test coverage, including of paths not commonly traversed in normal usage, but that could be exploited by attackers. This is an adaption threshold we would like to avoid. It should be easy to add this analysis to ones CI/CD pipeline without changing the actual software, or requiring lots of work be put into writing additional tests, possibly only for this purpose.

The authors later expand upon this [18], turning it into a concrete product, but abstracting away from specific libraries and versions, modelling vulnerabilities as AST constructs independent of the library they are in. We want to stick to describing vulnerabilities in the form of subsections of versions of libraries. Furthermore, dynamic analysis is still a big part of their detection phase, whereas we use static analysis.

A different way of tackling the problems faced by both static and dynamic analysis is presented in TamiFlex [2], a tool for aiding existing static analysers for Java where these have a hard time handling certain language features such as reflection and loading or generating custom classes on the fly. It does this by recording one or several runs of the program, and warning the user if the static analysis performed later is unsound in regard to these recorded runs. Such a tool could have been used to improve the static analysis, but since it relies on recording runs of the user code it is not feasible to integrate it with the vulnerability filter as it works currently.

In one study [19] the consequences of using traditional call graph algorithms designed for application code to analyse libraries are analysed, concluding that this leads to both missing and spurious edges. They suggest a new type of call graph algorithm designed for analysing libraries. Since the part of our analysis producing call graphs is done in conjunction with an application however, this does not apply to us. The analysis done on libraries in isolation, without considering the application, is limited to producing and analysing diffs, and does not include call graph generation.

On the topic of refining what versions are affected, the possibility of tracing a vulnerability backwards in time through an affected piece of software in order to find out at what point it was first introduced in the cases where this information is not available has been examined [3]. While this could be used to figure out a lower bound for which versions are affected in the cases where this is not known, potentially excluding some false positives, this is considered out of scope for this thesis. In the cases where there is no lower bound we assume all versions lower than the first safe version to be vulnerable.

Chapter 3
Approach

The basic idea behind this thesis is to use program analysis techniques to determine if a vulnerable library is used in such a way as to make it exploitable. This is done by figuring out what parts of a library are affected by a vulnerability, and determining whether these parts are used by the program we analyse. If they are not the vulnerability can not be exploited, and using the vulnerable library in this particular way is perfectly safe.

Section 3.1 talks about the limits of the vulnerability filter, Section 3.2 explains how it works, and Section 3.3 discusses potential sources of errors.

3.1 Scope

The vulnerability filter developed during the production of this thesis will analyse Java code. Only static analysis will be utilised, and only dependencies added through the Maven dependency management program will be analysed. This section explains the reasoning behind these choices, and some discussion on related areas.

The vulnerability filter is written in such a way as to allow a broadening of this scope, adding support for additional languages, types of analysis and dependency management systems. This is discussed in Section 6.

3.1.1 Language support

A significant limitation is that of supported language(s). When choosing which language to support a number of factors must be considered.

The difficulty of this type of analysis varies greatly depending on the language supported, with features such as reflection and dynamically generated code causing significant increases in complexity, in some cases making it infeasible. Reflection is a significant hurdle for any static analysis, very much so including statically generating call graphs. Take for example JavaScript with the common and varied usage of the `eval()` function. In most cases there is no feasible way to know what code will actually be executed, and as such it cannot be soundly analysed. Since usage of `eval()` is common in JavaScript code, and used in many different ways [20], this makes it a tricky language to support.

The tools available for a language are also an important factor. Since call graph construction is a difficult problem (see discussion in Section 2.2) and writing novel call graph construction tools for a language is out of scope for this thesis, there need to be pre-existing tools available for this purpose for a language to be considered. Since the quality of the call graph has a significant effect on the outcome of the analysis, the quality of the available tools for call graph generation will affect the choice of language.

JavaScript, C and Java are three common and well studied, yet different, languages that could be interesting for this analysis. A short discussion on the pros and cons of each of these languages as pertaining to this analysis follows.

JavaScript

JavaScript is a very common language used for a variety of projects, but due to the prevalence of execution of dynamic code, as explained in the previous section, it is tricky to analyse. There are tools available however [1, 4, 6] so it is not entirely infeasible, given one is willing to accept the reduced recall and accuracy inherent in static analysis of JavaScript compared to Java or C.

C

C is a simple language that does not have dynamic dispatch or reflection, and where the only way to call code dynamically is by making shell calls from within the code, something that is not used in normal production. This makes C a language easier than most to support.

Java

Java is common in this area, and thus of interest. There is a good amount of tooling available for this language, further increasing its viability. While Java also features reflection, causing significant issues for static analysis [12], there are more ways to reason about what code can be called than there is in JavaScript. Still this is a serious issue, with one paper describing the common informed opinion in the static analysis community as being that "Java reflection is a dynamic feature which is nearly impossible to handle effectively in static analysis" [13, p. 3]. The difficulty of analysing this type of code, and the prevalence of it, is further affirmed by a study [12] which shows that in a representative corpus of 461 Java programs, 78% use dynamic language features that then-current state-of-the-art static analysers had a hard time handling.

Conclusion

In conclusion the vulnerability filter will initially support Java. Java is a good middle ground between JavaScript and C, having more features than C while being more analysable than JavaScript.

3.1.2 Dependency management

Retrieving the code to be analysed is a problem in and of itself, and one that unfortunately does not have a trivial solution. In order to encourage wide adoption of the vulnerability filter it should be simple to use, and as much of the process as possible should be automated. This most certainly applies to acquiring libraries, forcing the user to supply them alongside the code is overly cumbersome and should be avoided if possible. The user should not have to submit anything outside of what they normally submit to CI/CD systems.

Attaching to the dependency management system is a reasonable point of integration for acquiring libraries. This way we parse the same files that the version management system does, ensuring that the user only needs to supply files they have readily available and that normally are uploaded to version management and CI/CD systems, while also ensuring that the vulnerability filter fetches exactly the same libraries that the code normally uses. This is the same entry point that LTH SECONDS [8] used, and that Debricked currently uses.

The downside of this approach is that it makes it impossible to support libraries gotten elsewise, such as those manually downloaded and imported, or those imported using an unsupported dependency management system. It also means specialised code must be written for each dependency management system supported.

This thesis will support the Maven dependency management system since it is a popular dependency management system for the targeted language, using it for dependency resolution, including of transitive dependencies, as well as some version management.

3.1.3 Type of analysis

Another important decision is that of static versus dynamic analysis of the code. This decision will affect the outcome, but also the amount of work needed to use the tool, and to add support for additional languages.

Static analysis

Static analysis is the easiest to integrate into existing development practices, since it simply analyses the source code, which in the case of VCS or CI/CD integration is already available. Using static analysis for building call graphs has some drawbacks however; it is essentially incapable of detecting edges caused by dynamic code generation or reflection, though some tools have some success in predicting what code will be executed [1]. The severity of this problem varies greatly depending on the language, with languages that do not support such execution of code obviously not experiencing this problem at all.

Dynamic analysis

Dynamic analysis can go some way towards alleviating the problems caused by dynamic code generation and reflection, but it has plenty of drawbacks of its own. Since using dynamic analysis to create call graphs relies on all possible calls actually occurring the suite used during analysis needs to be very thorough. If using the execution of normal tests to generate the graphs, as in [18], one needs to be mindful of ensuring proper test coverage. Furthermore it is important to keep in mind that the type of behaviour being tested is not necessarily the same as is being used by attackers. This could lead to missing edges that could be leveraged by an attack. In additional to full code coverage one would need coverage of inputs, flows, and other factors not typically considered vital when writing test.

Mixed analysis

Some tools such as TamiFlex [2] for Java use a combination of static and dynamic analysis, augmenting the static analysis with information gleaned through dynamic analysis. Such a system could be utilised in order to improve upon the results gained from purely static analysis, but it relies on there being an already existing thorough suite of tests or similar to execute during the dynamic analysis, something that might not always be the case.

Conclusion

This thesis will be limited to purely static analysis. In addition to not having to supply files the user does not normally supply to CI/CD systems the user should not have to annotate or otherwise modify their code in ways they normally do not.

Further work could extend the vulnerability filter by adding support for dynamic analysis in addition to the static analysis, or by replacing or complementing the existing static analysis. The consequences of this are discussed in Section 6.4.

3.2 Methodology

The vulnerability filter handles two distinct tasks. The first is to analyse vulnerable and fixed versions of a library in order to construct a database of vulnerable method and class signatures for a specific library and version for later use. The second is to utilise this database in analysing user code, to find out if vulnerable methods or classes are being used by the user code. In practice these tasks are performed in sequence, the second initiating the first. The results from the library analysis, that is to say the signatures of the classes and methods analysed to be vulnerable, are saved for future use, so as to not have to unnecessarily repeat it. Figure 3.1 shows a flow diagram, showing a simplified view of how data moves through the vulnerability filter.

In order to encourage wide use, it should be easy to use the vulnerability filter. The user should not have to supply files or information they do not have readily available, and they should not have to add additional annotations or similar to the code.

3.2.1 Library analysis

The library analysis takes as input the source code for two versions of a library, one just before and one just after fixing a given vulnerability. The difference between the two versions is computed, and analysis is performed in order to figure out what methods and classes have changed, and can be assumed to be affected by the vulnerability.

Since this version change is the smallest easily attainable difference between a vulnerable and fixed library, this change is assumed to only fix the vulnerability and not for example fix unrelated bugs or add or remove functionality. The precision of this step could be improved

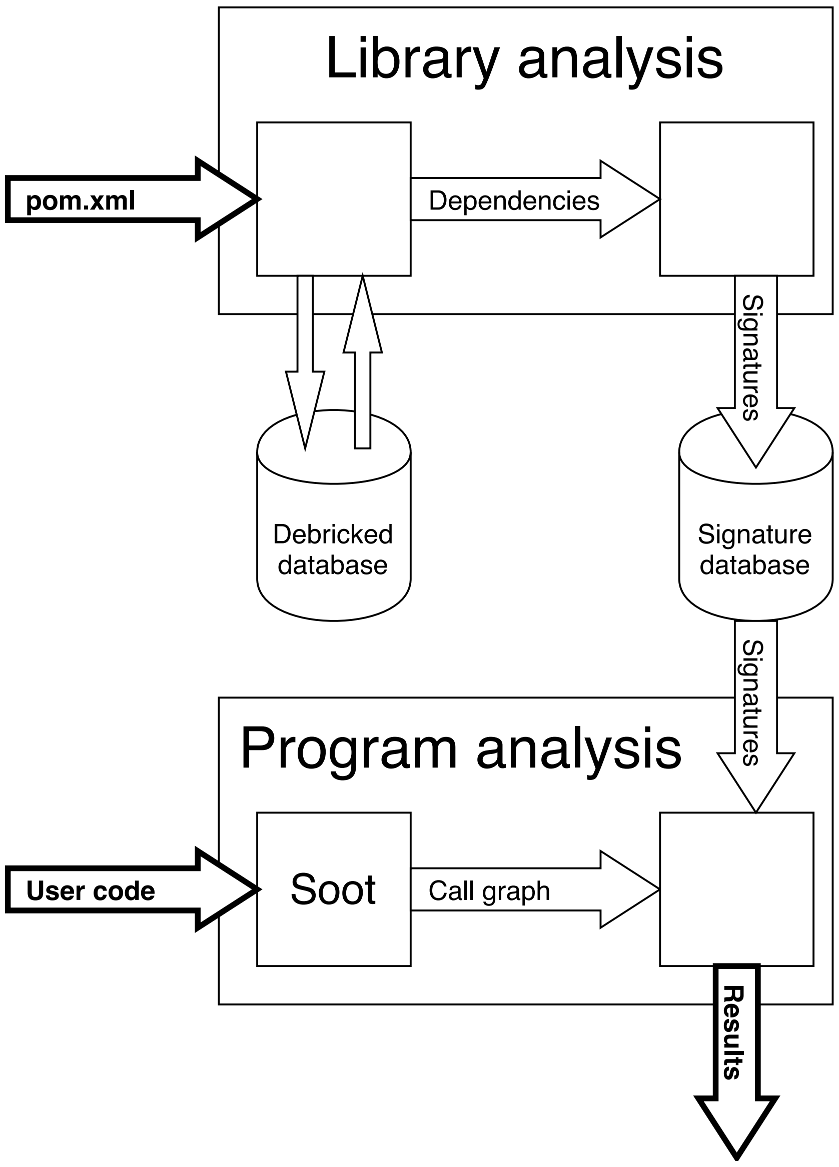


Figure 3.1: Simplified view of how data moves through the vulnerability filter. Inputs and outputs are bolded.

by increasing the granularity, for example by analysing changes on a commit level. This is discussed further in Section 3.3.2, but considered out of scope for this thesis.

In practice this is done in two steps. First we use the `java-diff-utils` [11] library to produce the diff between the versions, and the list of line numbers that are changed, added or removed, as well as what if any new files are added to the library. Then we use `ExtendJ` [5] to translate this list of files and lines into a list of class and method signatures. This list is stored in a database for later use.

If a library can not be analysed for some reason, perhaps a version is not available in the version management system, or a fix for a vulnerability is not yet released, the library is marked as not analysable and saved in the database. This database is used when analysing user code.

3.2.2 User code analysis

The user code analysis takes as input compiled user and library code, and the list of vulnerable signatures generated during the library analysis phase. We use Soot [21] to compute a call graph for the user code as a whole, including the libraries. For a full list of options used when doing this, see Appendix B. We then check this graph for the presence of any of the vulnerable signatures, and if found these, the library they belong to, and the sequence of calls leading to their (potential) execution are noted down. These signatures and their corresponding libraries as well as any libraries used that in the previous phase were marked as not analysable are then presented to the user as (potentially) vulnerable.

3.2.3 The vulnerability filter as a whole

In addition to these two interesting steps we need to do some amount of minutiae to make it all work together. First of all we need to identify the vulnerable libraries. We do this by using Maven to list all dependencies, and passing these to the Debricked vulnerability database to see if they have any vulnerabilities listed against them, and if so if the version being used is affected by these. This is not without complication, see further discussion in Section 3.3.1.

When we know a library is vulnerable we need to know between which versions the vulnerability is fixed. Sometimes this is not possible to know, for example if no fix is yet released, in which case our analysis can not be performed, and the library is marked as vulnerable without analysis, in which case any use of the library is considered exploitable. Sometimes only one of the “last vulnerable” or “first fixed” versions are available, in which case we parse the Maven version information file[1] to figure out the closest version after or before the specified version, respectively. Sometimes both versions are supplied in which case no version resolution need be performed.

We then need to obtain the source code for these versions of the library, which is done using Maven. This as well is not without complication, as sometimes versions or names are different in the vulnerability database and in Maven. We try to work around this where it is possible, but where it is not, no analysis can be done and so the library is marked as vulnerable

[1]For an artifact `artifact` by the group `group.id` published at `http://repository` this file is available at `http://repository/group/id/artifact/maven-metadata.xml`

without analysis. For further discussion on this issue see Section 3.3.1. Once all source code is retrieved analysis can be performed as described in the two previous sections.

When using the vulnerability filter only the Maven `pom.xml` file and the user code needs to be supplied, everything else is done automatically.

3.2.4 Where the vulnerability filter fits in

The vulnerability filter is written to be able to be attached to an already existing CI/CD pipeline, since this is the part of the code life cycle most suited to this type of analysis [23]. Running it earlier makes it difficult to properly analyse all libraries, and running it later makes it harder to push changes through since one might not want to introduce instability by changing a dependency. Doing the analysis at this point ensures developers are informed about potential issues early in the life cycle, while still being able to count on the code being compilable in full, something that is a requirement for many static analysis tools.

3.3 Potential errors

During the process errors can be introduced at several stages. It is important to be aware of these so that their impact can be minimised, and so that one does not make any claims that can not be substantiated. Many of these discussion points also offer possibilities for improvement.

3.3.1 Name translation

Unfortunately the naming schemes for a product or version is not necessarily the same in the vulnerability database as in the dependency management system. This can lead to false negatives in the case of improperly translating the dependency management names or versions to the database names or versions, and false positives when improperly translating database names or versions to dependency management names or versions.

The spring framework is an example of the former. In Maven these are distributed as separate packages with the `org.springframework` group ID and with artifact IDs like `spring-web`, `spring-orm` etc. In the vulnerability database however no such distinction is made and CVEs are filed against “`spring_framework`”. This makes it tricky to find these CVEs given the information available in Maven. Neither a search for the artifact nor group ID would find these CVEs, and so they are missed. This makes it seem like there are no CVEs affecting the libraries, when in fact there are. Since this problem does not relate to analysis of how dependencies are used, but rather to translating between different naming schemes, solutions for this problem are considered out of scope for this thesis.

The other type is exemplified by the Google `guava` library. In Maven there was only one type of release available until version 22, after which there are two releases available for each version; `version-jre` and `version-android`. In the vulnerability database however, no such distinction is made, bugs are filed against a version without qualifier. This can be handled fairly easily even in the general case; strip the qualifier from the version number when consulting the vulnerability database, and reattach it when consulting the dependency management system. This assumes, however, that all qualifiers are specified in a predictable

```
         }

         @Override
+        public String toString() {
+            return name;
+        }
+
+        @Override
         public boolean equals(Object o) {
             if (o == null) {
                 return false;
```

Figure 3.2: Traditional diff algorithms will cause the `equals()` method to incorrectly be marked as changed. This output was produced by GNU diffutils[3] version 3.7.

way[2], and does not handle the case where a qualifier is changed, added or removed between versions, such as before and after version 23.0 of Google `guava`. For example if one uses version 18 affected by a bug fixed in version 24 the vulnerability filter will not know to attach a qualifier, or which qualifier to attach.

Unfortunately there is no easy way to solve this issue, and even worse there is no easy way to detect if we are subjected to it. There is no way to know if we mistranslated a name and therefore got no matches, or if we correctly translated the name and there truly have never been any CVEs issued against this particular dependency. Unless we want to issue a warning for every dependency that has never had any CVEs against it we must assume the latter is the case, and so we do. When integrating the vulnerability filter into a larger analysis system that has better name and version translation this is an area where it will benefit greatly from work not directly related to it.

3.3.2 Diffing versions

False positives can be introduced when computing the diff between the last vulnerable and first fixed version of a library, either due to more being marked as changed than actually is changed, or more being changed than is required to fix the vulnerability.

The first case can happen when adding a new method, as shown in Figure 3.2. Due to how the diff algorithm works the second `@Override` is marked as added rather than the first, and `ExtendJ` will (correctly) associate this with the `equals()` method defined below it. This will cause the `equals()` method to be marked as having been changed when in fact it has not been, potentially introducing false positives. Since `java-diff-utils` supports custom diff algorithms one could be written that handles this case better, though this is considered out of scope for this thesis.

If the two versions compared differ in more than the vulnerability being fixed, false pos-

[2]In the vulnerability filter a version is defined as a series of dot-separated numbers optionally followed by a qualifier made up of printable characters.

[3]`https://www.gnu.org/software/diffutils/`

itives can also be introduced this way. For example a version change that includes a patch for a vulnerability could also include bug fixes that are not security related, or introduce new functionality. Since we assume that all changes are related to fixing the vulnerability, the code changed for other reasons will also be marked as vulnerable, introducing false positives.

The amount of false positives introduced this way could be reduced, improving accuracy, by increasing the granularity of change that we examine. For example instead of comparing two released versions we could compare two versions of the same pre-release build separated by a commit, if we know that this commit fixes the security vulnerability in question. This requires finding out what commit does this though, which is not easily solved in the general case. What versions are susceptible to a vulnerability (and by method of elimination which versions are not) is available from the vulnerability database, but what commit includes the fix is not. Because of this we limit ourselves to analysing released versions, improving the vulnerability filter by analysing finer changes is left as future work.

3.3.3 Getting signatures

When generating the list of signatures affected by a change, as described in Section 3.2.1, errors can be introduced. A changed or deleted line could be marked as affecting a different signature than it actually does, or it could be falsely marked as not affecting any signature, or vice versa.

This could lead to false positives in that unchanged lines are marked as changed, or false negatives in that changed lines are marked as unchanged. In order to avoid this we need to ensure that all lines changed, and only those lines, are marked as such and that they are translated to their proper method or class signatures. In most cases this is fairly straightforward, with the notable exception of the problem described in Figure 3.2.

3.3.4 Call Graph

As discussed in Section 2.3 any call graph of reasonable precision for a language such as Java will not be sound in the general case, in other words errors could be introduced when creating the call graph. This is in practice unavoidable, since requiring the graph to be sound would make it imprecise to the point of uselessness.

The amount of errors introduced at this point is largely out of our control, since it depends on the library used for call graph generation, and as such will not be considered. For practical purposes the call graph is assumed to be both sound and precise, even though this is not actually the case. The implications of this are not analysed, this is left as future work. See Section 6.5.

Though necessary, this assumption unfortunately makes it hard to in depth analyse the effectiveness of the vulnerability filter, since interesting errors would be introduced at this stage. This is a practical necessity however, since otherwise a very large amount of work would have to be put into analysing and comparing call graph construction algorithms, an area large enough for a thesis of its own (see for example [4, 9]).

In order to improve upon errors in this area one could perform a study on the performance of call graph generators, such as done in [1], and pick a better one. One could also create several call graphs and check to see if a vulnerable signature is contained in any of them, or only include those included in both. This would increase recall at the cost of precision and performance or precision at the cost of recall and performance respectively, so the benefits would have to be weighed against the drawbacks. This is discussed in Section 6.4.

Chapter 4
Results

In order to evaluate the vulnerability filter it was run against a number of Maven projects downloaded from GitHub. The projects were selected by searching for projects with a `pom.xml` file in the root directory of the repository, and sorting by the time GitHub indexed them. This produces a result random enough for the purposes of this thesis. Once downloaded the programs were compiled using the Maven `compile` command.

In the following, a "project" is a repository cloned from GitHub, and a "dependency" is one version of an artifact in Maven used by a project, directly or transitively. The call graph generation was done using Soot. For a full list of options passed to Soot see Appendix B.

4.1 What share of projects are we able to analyse?

Of the 200 downloaded projects 50 failed to compile, and so were excluded. An additional 9 had no code to analyse, leaving 141 programs.

Since Soot can only analyse Java version 8 or lower code, some projects had to be excluded due to Soot not being able to analyse them. 14 projects were excluded for this reason. Some programs also had to be excluded due to having erroneous or no longer working Maven configurations. These programs compiled, but it was not possible to list their dependencies and/or the repositories these were downloaded from. Since this information is necessary for our analysis these also had to be excluded. 11 programs were excluded for this reason. One program was manually excluded since the repository supposedly containing many of the dependencies of said program no longer did so. This left us with a total of 115 projects that compiled, had code to analyse, and that Soot was capable of analysing. Figure 4.1 illustrates this breakdown.

If we ignore the programs that do not compile or do not contain code, since we can not reasonably be expected to be able to analyse these, we get a more realistic view of what share

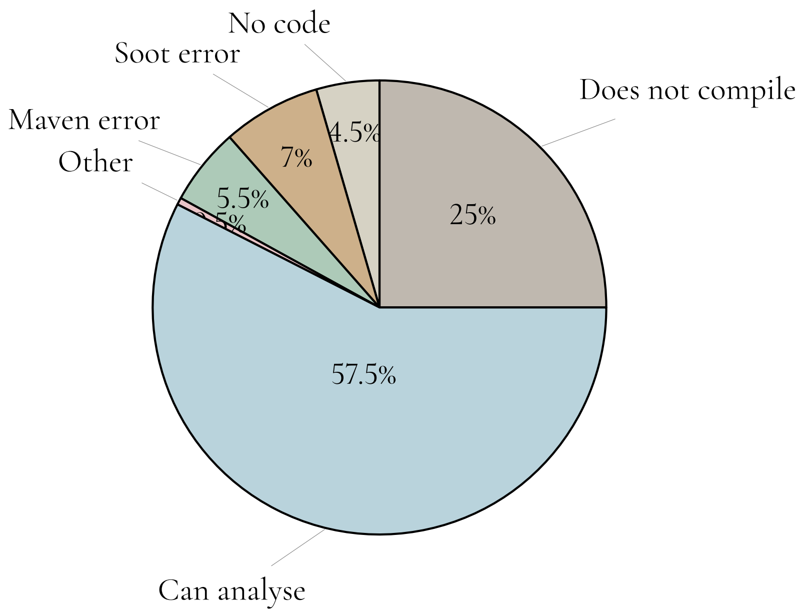


Figure 4.1: Overall share of projects we are able to analyse

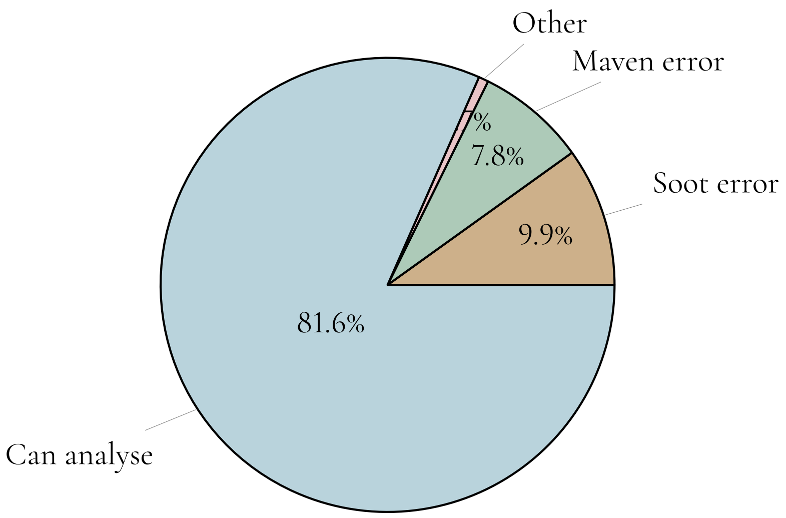


Figure 4.2: Share of projects containing compiling code we are able to analyse

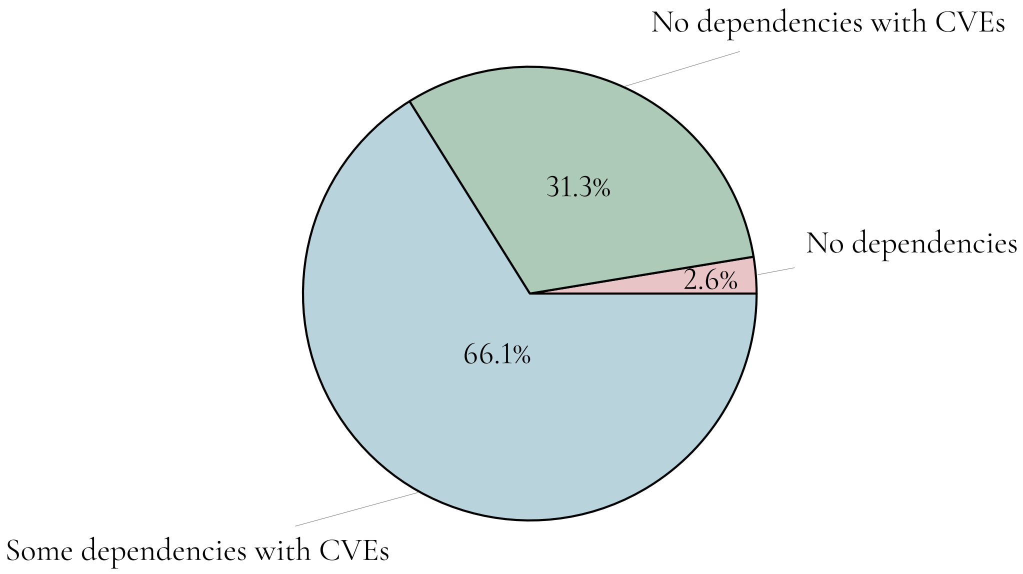


Figure 4.3: Share of projects using dependencies with CVEs listed against them

of programs we are able to analyse, approximately 82%. This is illustrated in Figure 4.2. The fact that Soot is incapable of analysing a significant part of our projects is unfortunate, but not something we can do anything about.

4.2 What share of projects and dependencies are vulnerable?

Of the 115 projects, 112 had dependencies. Of these, 76 were analysed to have dependencies that have a CVE listed against the version in use, meaning about 66% of the projects made use of dependencies with known vulnerabilities. Figure 4.3 illustrates this breakdown.

In these 112 projects there were 7568 usages of 3583 dependencies. Of these, 1015 were of a dependency with a CVE listed against it, meaning that about 13% of the time a dependency was included that dependency had a CVE listed against it.

Note that these numbers are not necessarily representative of most projects. Since the projects were picked at random many of them were quite old and presumably abandoned, thus increasing the likelihood of vulnerable dependencies.

4.3 What share of dependencies are we able to analyse?

Of these 1015 usages of dependencies with CVEs listed against them, 378 were unable to be analysed, 524 were analysed to use the dependency in such a way as to expose the vulnerable parts of the dependency, and 113 were analysed not to. See Figure 4.4.

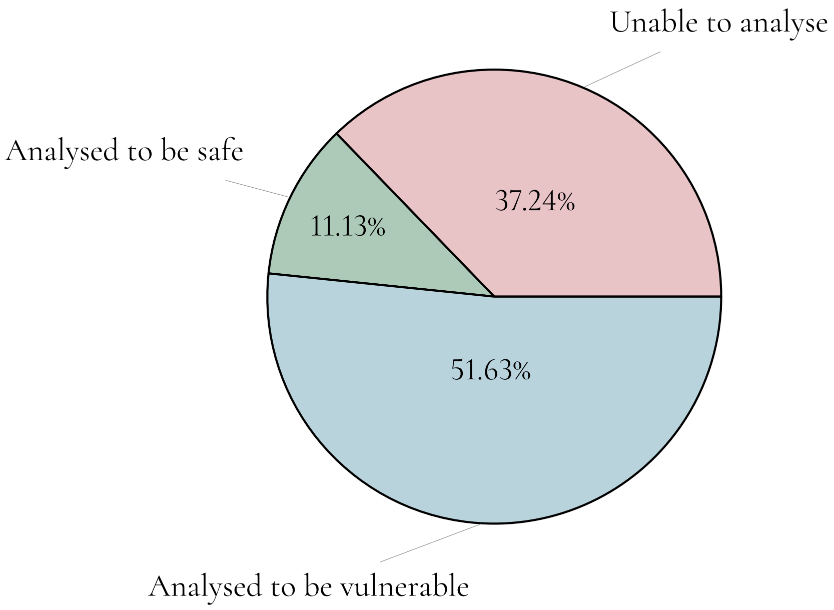


Figure 4.4: Analysis results for dependencies

In other words we find that in approximately 11% of cases where a warning would have been issued there was no need to do so, the analysed program did not use the vulnerable parts of the dependency. If we look only at the cases where we are able to do our analysis this number rises to 18%. Since improvements in the share of dependencies that we are able to analyse are expected to be made as the vulnerability filter is integrated into better tools for doing name translation the actual number is likely to move closer to 18%.

4.4 Why are we unable to analyse some dependencies?

The effectiveness of the vulnerability filter can be improved by increasing the number of dependencies we are able to analyse, so understanding our limitations and how, or indeed if it is possible, to overcome them is important.

Firstly, looking at what dependencies we are unable to analyse, presented in Figure 4.5, we quickly come to the conclusion that a small number of dependencies with many CVEs listed against them are responsible for a large share of the dependencies we can not analyse. For example, postgresql alone is responsible for a over third of the vulnerabilities we are unable to analyse, 130 of the 378.

Looking at the reason we are not able to analyse these dependencies, presented in Figure 4.6, we also see that most of these fit into a few categories. 48 are for reasons related to issues with qualifiers in version names, as described in Section 3.3.1. 85 are due to dependencies changing group name, so we are unable to find the correct version, another issue related to name translation. 81 are due to the version specified in the CPE or CVE not being available in Maven at all. 73 are due to the CVE or CPE not specifying, and we not being able to deduce, an end version, and 91 are for other reasons.

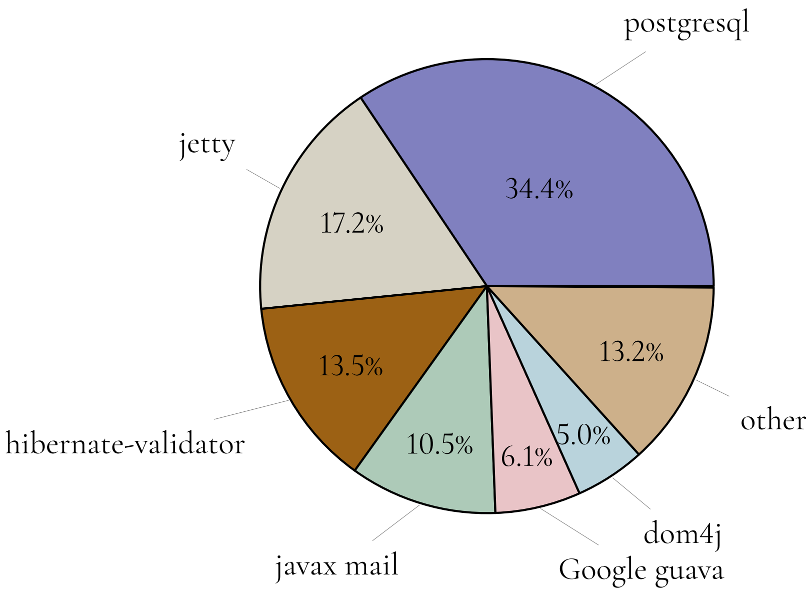


Figure 4.5: Which dependencies we are unable to analyse

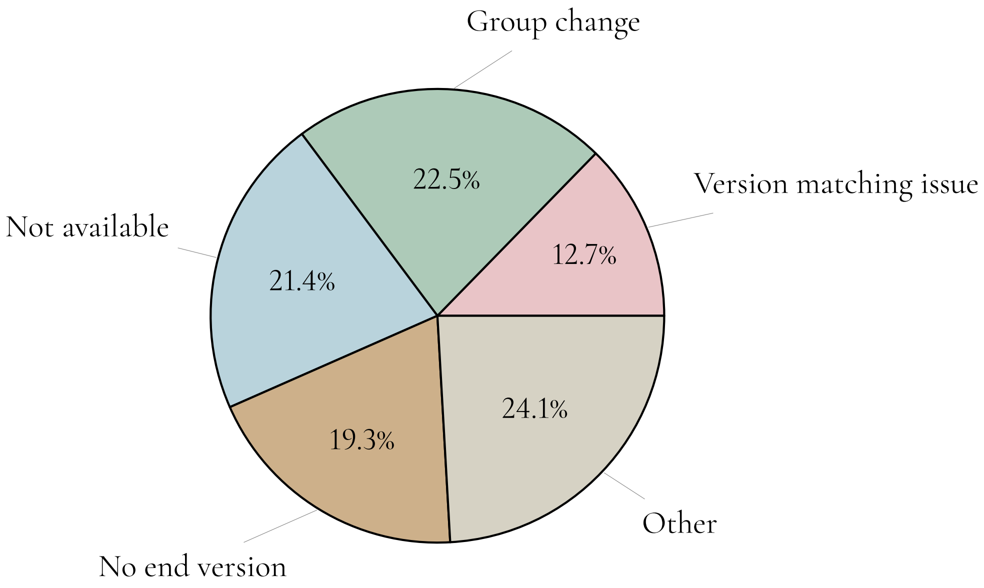


Figure 4.6: Why we are unable to analyse dependencies

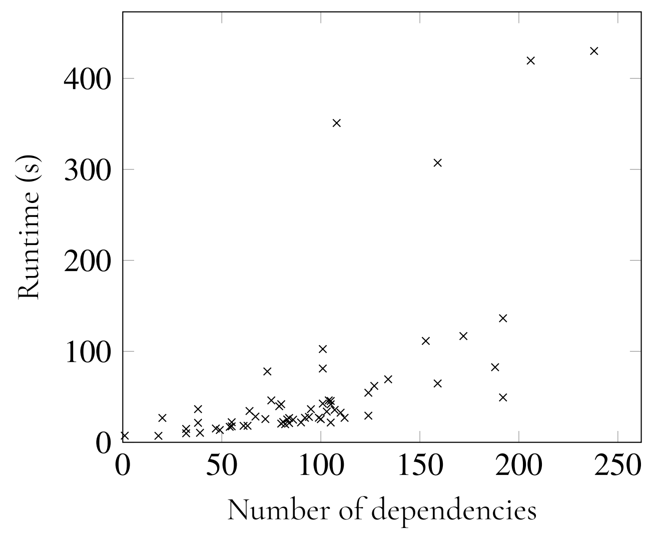


Figure 4.7: Running time[2] compared to the number of dependencies. The correlation coefficient[3] is 0.65

Issues with matching names and versions are likely to be reduced as the vulnerability filter gets integrated into an existing system for doing these translation. The current system is rudimentary, and so improvements should be expected in this area. In a related sense, sometimes the reason for a version not being available in Maven is due to us looking for the wrong version due to lax CPE matching, so this number as well is likely to decrease as said improvements are made.

The issue of group name changes could also potentially be handled by a similar name translation utility, but changes would have to be made to the vulnerability filter in order to support a group name changing as a version does, without treating it as a separate library.

Some of the reasons in the "other" category are also likely to be improved. All 40 of the dependencies that we can not analyse corresponding to javax mail (10% of the total number of dependencies we are unable to analyse) are false positives related to other programs with similar names. As name translation improves these will be filtered out.

4.5 How well does the analysis scale?

In order for this type of analysis to be viable for larger projects it needs to scale reasonably well with project size. When looking at scaling two factors were analysed; number of dependencies and size of the call graph. Scatter plots of the running time in relation to these are presented in Figures 4.7 and 4.9 respectively.

The y-axis in these graphs is the runtime in seconds when running the analysis on a Dell XPS 9350[1]. The actual runtime is not the interesting part however, but rather the scaling.

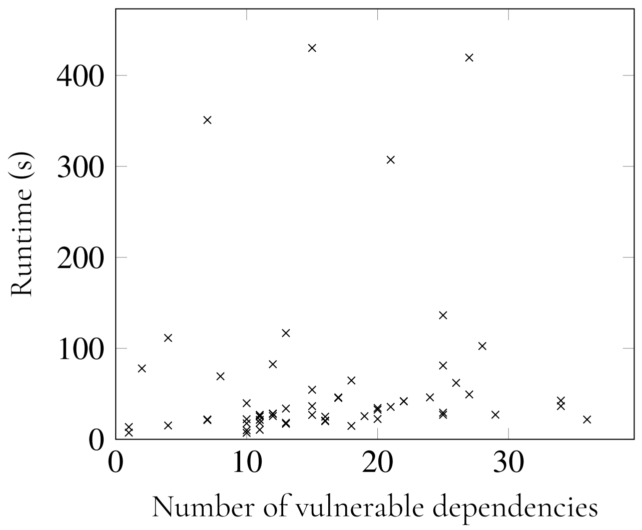


Figure 4.8: Running time[2] compared to the number of *vulnerable* dependencies. The correlation coefficient[3] is 0.11

4.5.1 Scaling with number of dependencies

Starting by looking at the number of dependencies, presented in Figure 4.7, we can see that most projects use between 50 and 110 dependencies, including transitive dependencies. We can see outliers, both in regard to number of dependencies and running time. It is hard to see a clear relation between number of dependencies and the running time, there are several instances where fewer dependencies lead to increased running time and vice versa. This is probably due to the dependencies varying greatly in size and complexity, so while simply looking at the number of dependencies can give us a hint towards the running time, it is not a good predictor of it. The correlation coefficient for this relation is approximately 0.65, confirming the picture of a weak relation.

The time consuming parts of the library analysis are only done to *vulnerable* dependencies, so looking at number of these instead of the number of overall dependencies might give a clearer picture. Figure 4.8 shows a plot of this. Unfortunately isolating vulnerable dependencies doesn't improve the results at all, in fact it worsens them. There are still several outliers, and the relation is weak if existing at all. Here the correlation coefficient drops to approximately 0.11.

4.5.2 Scaling with call graph size

Looking instead at the number of edges in the call graph, Figure 4.9, we see a much more obvious relation, with the coefficient correlation reaching all the way to 0.96, a very strong correlation. We also see that for the vast majority of projects the call graph has fewer than 500,000 edges, with a few outliers having significantly larger call graphs, and also significantly increased running time. While there are cases where increased call graph size does not lead to increased running time these are fewer than in the case of looking at the number of

[1]`https://www.dell.com/gh/p/xps-13-9350-laptop/pd`

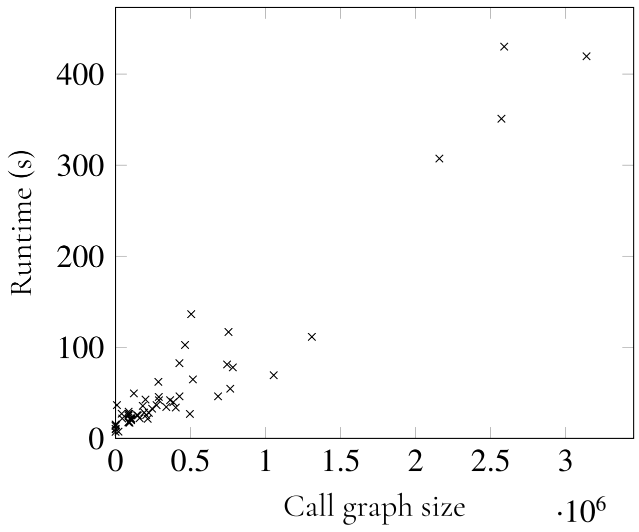


Figure 4.9: Running time[2] compared to the number of edges in the generated call graph. The correlation coefficient[3] is 0.96

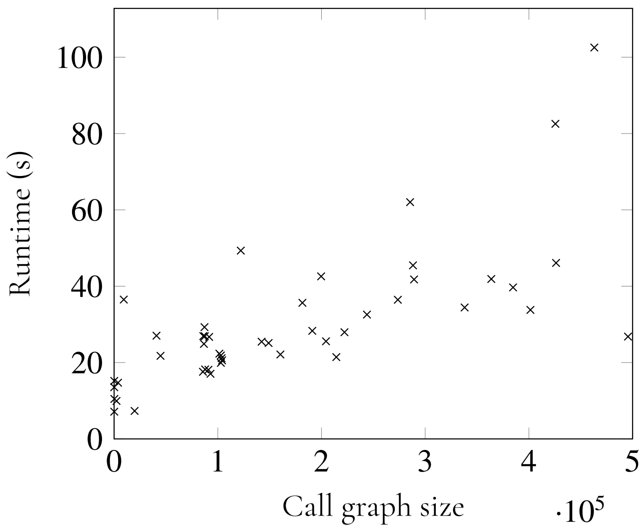


Figure 4.10: Zoomed in view of Figure 4.9, showing only those with less than 500 000 edges. The correlation coefficient[3] is 0.71

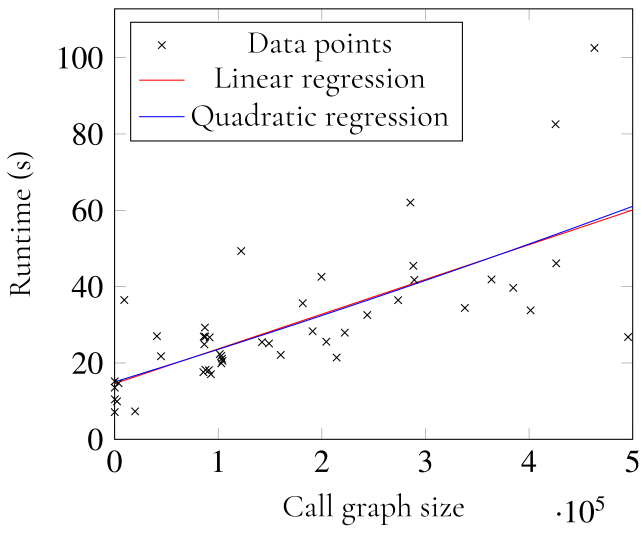


Figure 4.11: Figure 4.10 with regressions superimposed[3]

dependencies, and less extreme. There is a much closer relation between the size of the call graph and the running time than between the number of dependencies and the running time.

If we zoom in to those projects with a call graph size of less than 500,000 edges, Figure 4.10, we see largely the same pattern repeating, but with the correlation coefficient dropping down to approximately 0.76. There is a relation, but it is not very strong and there are outliers here as well.

Attempting to use the size of the call graph as a predictor has downsides though. The analysis consists of two phases, and the library analysis phase has nothing to do with the actual call graph, only with the number and size of vulnerable dependencies. While the program analysis phase is usually the more time consuming one, ignoring factors that affect the library analysis running time will give a skewed picture, as indeed we see in the figures. Furthermore the size of the call graph is not known until it has been generated, at which point the vast majority of the work has been done. Call graph size can then not be used to the predict the time it will take to do the analysis, since the analysis is practically already finished when this is known.

4.5.3 Conclusion

In conclusion it is hard to know the time the analysis will take before starting out. Looking at the number of dependencies can give us a little hint, but there are plenty of outliers in either direction. This analysis is computationally intensive, and analysing large and complex projects takes several minutes. While it is hard to find a single metric to look at to predict the running time, the call graph size seems to give a reasonable estimation.

Attempting to interpolate a function from the data, shown in Figure 4.11, hints at the analysis scaling linearly with the size of the call graph, with a not-insignificant baseline. This baseline is probably due to the library analysis, which is not dependent on the size of the call graph. The fact that the scaling is linear is encouraging, since this means we will be able to analyse even very large projects without consuming unreasonable amounts of resources. Note that we only consider runtime and not other metrics such as for example memory usage.

[2]Running time in seconds for the analysis as performed on a Dell XPS 13 9350 (`https://www.dell.com/gh/p/xps-13-9350-laptop/pd`)

[3]Regressions and correlation coefficients computed by numpy (`https://numpy.org/`) version 1.16.2.

Chapter 5
Conclusion

The initial results from this analysis are promising, showing a reduction in false positives between 10% and 20%, with further improvements expected. The analysis seems to scale linearly, meaning it can also be applied to very large projects. The analysis is quite complicated however, meaning it takes significant development effort to deploy. It is also computationally intensive, especially compared to the previous methodology of analysing dependency versions without looking at the code of the dependency or the program using it; analysing large projects can take several minutes.

In applying this analysis in practice there proved to be many difficulties in areas mostly not related to the core functionality that affect the results negatively. These issues are mostly not related to the actual meat of the analysis however, so they do not say much about the actual theory behind the analysis, just about how it can be tricky to make it work in practice. Using the vulnerability filter in a context where some of these problems are already solved or being worked on will alleviate these issues.

5.1 Areas for improvement

When looking for ways to improve the results we can look at the core analysis, i.e. the call graph generation and analysis, or the surrounding tools, i.e. name and version translation and parsing. Improvements in each area can be done independently.

5.1.1 Core functionality

As we saw in Section 4.1, Soot is unable to analyse about 10% of the projects that compile and have working Maven configs. This is of course an issue, but solutions are not straightforward. As of writing Soot only supports Java version 8 and lower, meaning that we lose out on a selection of projects due to this. Later versions of Soot will support later Java versions, but if one is unwilling to wait for these new versions the only other options are to patch Soot oneself or to use another tool for call graph generation. Since Soot is a large project patching it separately is a significant undertaking.

As for replacing Soot with another tool several are available, for example WALA [10] and Doop [22]. Replacing Soot with either of these tools is possible with some work, the vulnerability filter is designed to allow this type of slotting in and out of components, but there is no guarantee either of them perform better than Soot.

5.1.2 Surrounding tools

The area proving the most difficult to handle, and responsible for the largest share of dependencies we are unable to analyse, is the translation of names between as they are in the dependency management system of choice and in the vulnerability database (see Section 3.3.1 for a discussion on this). Integrating the service into already existing tools for doing this translation would bring significant improvements to the output of the tool, without changing the core functionality.

5.2 Final thoughts

In conclusion the analysis shows great promise. Deploying this analysis seems to lead to a reduction of false positives between 10% and 20%, depending largely on the sophistication of the name and version translation utilities.

Since the vulnerability filter in many ways will benefit from work not directly related to the core functionality there is reason to be hopeful for further increases in reduction of false positives. Increasing the share of libraries that can be analysed, which is almost exclusively down to improvements in areas not relating to core functionality, will greatly improve results.

Chapter 6
Future Work

While the vulnerability filter works fairly well for the specific case it is written for, there are many ways to extend and improve upon it. The scope can be widened, and there are cases where both recall and accuracy can be improved. Additional research could also be done to better understand the results and how to interpret them.

6.1 Language support

Additional language support is one obvious way to expand the scope of the vulnerability filter. Where possible the code has been compartmentalised to allow for this.

Introducing support for different languages means replacing parts of the library analysis phase, and most of the user code analysis phase. Functionality needs to be implemented that translates a list of line numbers and files to signatures, and a good library for call graph generation needs to be found and integrated with the rest of the program.

6.2 Dependency management systems

The other obvious expansion of scope is to support additional dependency management systems, such as Gradle for Java.

When introducing support for other dependency management systems no part of the analysis needs to be changed, since they work on code only. Code needs to be written that takes a dependency management file for the dependency management system to be supported, and that translates names and versions to and from the way they are presented in this system to the way they are presented in the database. Code that downloads compiled and source versions of the libraries also needs to be written.

6.3 Granularity in version management

As discussed in Section 3.3.2, accuracy could be improved by increasing the granularity of the version difference in libraries being analysed. Instead of comparing released versions finer changes could be compared, such as those introduced by a commit. This requires finding out that a commit fixes a certain vulnerability though, a problem not easily solved in the general case. It could be done by linking certain dependencies to a git repository, and parsing commit messages to determine which commit fixes a vulnerability. Clearly this is not a general solution though.

6.4 Improving the analysis phase

It is possible to replace the call graph generation library with another, or add a supplementary one. In doing this dynamic analysis could also be introduced. This would entail moderate tweaks to the program analysis part of the vulnerability filter, but no part of the library analysis would need to be changed since call graphs are generated only for the analysed program as a whole, not the libraries in isolation.

Adding a supplementary call graph generation library would increase recall, but at the cost of accuracy and performance. Alternatively if one considers the intersection of reachable methods rather than the union accuracy would be increased, at the cost of recall and performance. The value of these factors would thus need to be weighed against one another to determine if this is desirable.

6.4.1 Plugging in dynamic analysis

As mentioned in Section 3.1.3 support for dynamic analysis could be introduced to complement or, if one wishes, replace the existing static analysis.

If one decides to complement the existing static analysis one could use either the intersection or the union of the two call graphs. Using the intersection would increase accuracy at the cost of recall, whereas using the union would increase recall without a significant impact on precision. Generating a dynamic call graph in addition to the static one would have performance implications however, and it is unclear how much of a benefit using the union would bring.

Additionally, unlike when using static analysis, dynamic analysis needs a suite to be run on. Creating this suite takes effort, increasing the threshold for using the vulnerability filter. If instead, as in [18], the tests are used to create the dynamic graph these tests need to be very thorough, or one risks introducing false negatives (if replacing the existing static analysis or using the intersection method). Furthermore, most tests are not written to use the code in the same way an attacker would, which could lead to missed paths, also introducing false negatives. More on this in section 3.1.3. Additionally, having good and thorough tests would then be a requirement in order to get the most out of the vulnerability filter, increasing the threshold required to start using it.

A system where dynamic analysis is used on top of the static analysis where possible, and where the union of this analysis and the regular static analysis is considered would benefit from most of the upsides without suffering from most of the downsides. If there is no good suite for dynamic analysis to be performed on baseline static analysis could still be performed, but if one is available we could use it to our benefit. Really the only downside of this approach is the performance implications, and the extra work required to support two types of analysis.

Another way of doing dynamic analysis is to attach to the program as it is running, as for example TamiFlex [2] does, but this incurs performance overheads, delays notification to runtime, and still has all the drawbacks of traditional dynamic analysis. The vulnerability filter is not written for this type of behaviour, so this would be a fundamental change to the program analysis phase, requiring substantial work.

6.5 Analysing call graph generation

Due to scope constraints the validity of the generated call graph is not analysed or considered, instead it is assumed to be perfectly sound and accurate. This is of course not true in reality. Research into how untrue this assumption is would allow us to better understand the actual reduction in false positives and introduction of false negatives.

References

[1] Gábor Antal, Péter Hegedűs, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. Static JavaScript Call Graphs: a Comparative Study. In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, IEEE, 2018.

[2] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 241–250, may 2011.

[3] S. Dashevskyi, A.D. Brucker, and F. Massacci. A screening test for disclosed vulnerabilities in foss components. *IEEE Transactions on Software Engineering*, mar 2018.

[4] Jorryt-Jan Dijkstra. Evaluation of static javascript call graph algorithms. Master's thesis, Centrum Wiskunde & Informatica, January 2014.

[5] ExtendJ Committers. ExtendJ. `https://web.archive.org/web/20190225210518/http://extendj.org/`. Accessed 2019-10-18.

[6] Asger Feldthaus, Max Schafer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *Proceedings - International Conference on Software Engineering*, pages 752–761, 05 2013.

[7] The Apache Software Foundation. Media alert: The apache software foundation confirms equifax data breach due to failure to install patches provided for apache® struts™ exploit. `https://web.archive.org/web/20190926085610/https://blogs.apache.org/foundation/entry/media-alert-the-apache-software`, 2017. Accessed 2019-09-26.

[8] Martin Hell, Jonathan Sönnerup, Linus Karlsson, and Paul Stankovski Wagner. SECONDS Secure Connected Devices. `https://web.archive.org/web/20191106141012/https://www.eit.lth.se/index.php?puid=220`. Accessed 2019-11-05.

[9] Elnaz Honar, Mortazavi Jahromi, and Seyed AmirHossein. A framework for call graph construction. Master's thesis, Linnaeus University, 2010.

[10] IBM. T.J. Watson Libraries for Analysis (WALA). `https://web.archive.org/web/20190902085321/http://wala.sourceforge.net/wiki/index.php/Main_Page`. Accessed 2019-09-02.

[11] java-diff-utils. java-diff-utils. `https://java-diff-utils.github.io/java-diff-utils/`. Accessed 2019-10-18.

[12] Davy Landman, Alexander Serebrenik, and Jurgen Vinju. Challenges for static analysis of java reflection - literature review and empirical study. In *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, pages 507–518, May 2017.

[13] Yue Li, Tian Tan, and Jingling Xue. Understanding and analyzing java reflection. *ACM Trans. Softw. Eng. Methodol.*, 28(2):7:1–7:50, February 2019.

[14] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[15] National Institute of Standards and Technology. National vulnerability database. `https://web.archive.org/web/20190902105701/https://nvd.nist.gov/`. Accessed 2019-09-02.

[16] OWASP. Owasp top 10 - 2017. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project`, 2017. Accessed 2019-09-26.

[17] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. Impact assessment for vulnerabilities in open-source software libraries. *CoRR*, abs/1504.04971, 2015.

[18] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2018.

[19] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 474–486, New York, NY, USA, 2016. ACM.

[20] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The Eval that Men Do: A Large-scale Study of the Use of Eval in JavaScript Applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 07 2011. Springer-Verlag.

[21] Sable. Soot - A framework for analyzing and transforming Java and Android applications. `https://web.archive.org/web/20190902085153/https://sable.github.io/soot/`. Accessed 2019-09-02.

[22] Yannis Smaragdakis. Doop - Framework for Java Pointer and Taint Analysis. `https://bitbucket.org/yanniss/doop/src/master/`. Accessed 2019-09-02.

[23] SourceClear. The busy managers' guide to open source security. https://web.archive.org/web/20190904103601/https://www.sourceclear.com/resources/TheBusyManagersGuideToOpenSourceSecurity.pdf, 2017. Accessed 2019-09-04.

Appendices

Appendix A
Call graph example code

```
import java.util.Random;

public class Main {
    public static void main(String[] args) {
        Main.a();
    }

    public static void a() {
        if (new Random().nextBoolean()) {
            Other.b();
        } else {
            Other.c();
        }
    }
}
```

Figure A.1: Main.java

```
public class Other {
    public static void b() {
        System.out.println("b");
    }

    public static void c() {
        System.out.println("c");
    }
}
```

Figure A.2: Other.java

Appendix B
Soot options

For all analysis call graph generation was performed using Soot version 3.3.0 with the following options:

Generic options

`-w` Whole Program Mode. Takes the whole program into consideration. Required for generating call graphs.

`-no-bodies-for-excluded` Does not generate bodies for excluded classes (namely default Java classes). Cuts down significantly on analysis time, but prevents the analysis from considering the JDK, meaning we miss any calls from the JDK back to the client code.

Call graph options

`all-reachable:true` Considers each method in the analysed program to be an entry point, rather than just the main method.

`jdkver:8` Targets Java version 8. This is the latest version currently supported.

Soot by default uses Class Hierarchy Analysis (CHA) when constructing the call graph. For a fairly accessible explanation of CHA, and a comparison to other common algorithms see [9, p. 3].

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-08-27

EXAMENSARBETE Improving vulnerability detection using program analysis

STUDENT Rasmus Hagberg
HANDLEDARE Martin Hell (LTH), Christoph Reichenbach (LTH), Oscar Reimer (Debricked)
EXAMINATOR Martin Höst (LTH)

Improving vulnerability detection using program analysis

POPULÄRVETENSKAPLIG SAMMANFATTNING **Rasmus Hagberg**

As the use of open source libraries increases, so does the amount of exploits and vulnerabilities relating to these, potentially making your code vulnerable. However, simply including a vulnerable library doesn't mean you are susceptible to exploitation. This thesis explores using program analysis to determine if you are.

In the vast majority of programs, external libraries are used in order to prevent duplication of work and to save time and effort. These libraries are often open source and available for free, making them broadly available and easy to incorporate into your program. This simplifies development and makes producing great software easier, faster and more accessible.

All code can contain errors however, and these libraries are of course no exception. Importing external code means importing external errors, and developers need to be aware of this risk in order to minimise the potential for harm. Not properly considering this can have grievous consequences. For example, the widely published Equifax breach was caused by a failure to install patches for a known vulnerability in an open source library.[1]

However, simply using a library with a known vulnerability doesn't mean you are at risk, you have to use the specific parts of the library actually affected.

During the course of this thesis I have written a program that analyses how a program uses a vulnerable library, in order to determine if the vulnerable parts are used.

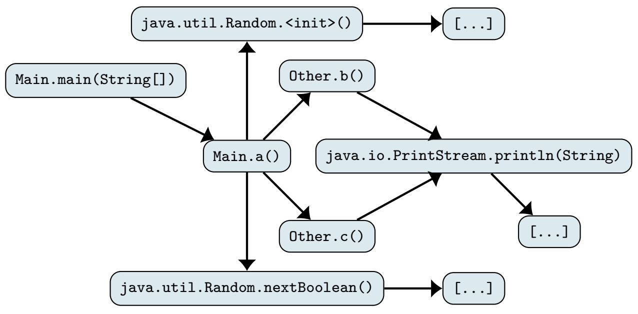

java.util.Random.<init>()
[...]
Main.main(String[])
Other.b()
Main.a()
java.io.PrintStream.println(String)
Other.c()
[...]
java.util.Random.nextBoolean()
[...]

Figure 1: Illustration of a call graph. Some methods are omitted for brevity.

This is done by analysing what parts of a library has changed between a vulnerable and a fixed version, and then building a so-called "call graph" representing the potential execution paths of the program. This graph is checked to see if it contains the vulnerable parts of the library. If it does not, there is no harm in including the library.

Using this program, I am able to show that in roughly 10-20% of cases the vulnerable part of the library was not used, making its inclusion safe. This helps the developer(s) to prioritise fixing the most important issues first, reducing the risk users are exposed to.

[1] https://web.archive.org/web/20190926085610/https://blogs.apache.org/foundation/entry/media-alert-the-apache-software