Using Program Analysis to Identify the Use of Vulnerable Functions

Rasmus Hagberg[1,2], Martin Hell[3] and Christoph Reichenbach[1]
[1]*Department of Computer Science, Lund University, Box 118 Lund, Sweden*
[2]*Debricked AB, Malmö, Sweden*
[3]*Department of Electrical and Information Technology, Lund University, Box 118 Lund, Sweden*
rasmus.hagberg@debricked.com, martin.hell@eit.lth.se, christoph.reichenbach@cs.lth.se

Keywords: software security, vulnerabilities, call graphs, open-source software, vulnerability analysis

Abstract: Open-source software (OSS) is increasingly used by software applications. It allows for code reuse, but also comes with the problem of potentially being affected by the vulnerabilities that are found in the OSS libraries. With large numbers of OSS components and a large number of published vulnerabilities, it becomes challenging to identify and analyze which OSS components need to be patched and updated. In addition to matching vulnerable libraries to those used in software products, it is also necessary to analyze if the vulnerable functionality is actually used by the software. This process is both time-consuming and error-prone. Automating this process presents several challenges, but has the potential to significantly decrease vulnerability exposure time. In this paper, we propose a modular framework for analyzing if software code is using the vulnerable part of a library, by analyzing and matching the call graphs of the software with changes resulting from security patches. Further, we provide an implementation of the framework targeting Java and the Maven dependency management system. This allows us to identify 20% of the dependencies in our sample projects as false positives. We also identify and discuss challenges and limitations in our approach.

1 INTRODUCTION

Vulnerabilities in third-party or open-source software (OSS) is not a new problem, but as organizations, society, and even our critical infrastructure become increasingly dependent on software-based systems, handling such vulnerabilities becomes more important. In 2020, Synopsys reported that 99% of audited codebases contained open source components and on average 70% of the codebase was open-source code (Synopsys, 2020).

Better understanding and efficiently reacting to vulnerabilities is both a technical and an organizational challenge (Alomar et al., 2020; Höst et al., 2018). From a technical perspective, the increased use of OSS together with the high increase in reported vulnerabilities in the last few years (NIST, 2021) require technical tools for identifying, analyzing, and prioritizing remediation of vulnerabilities. Fully automating this process will decrease remediation time and shrink the attack surface. However, such automation is difficult due to incompatible data formats, inaccurate and incomplete data on e.g., NVD, the possibility of breaking changes when updating to new versions (Xavier et al., 2017; Anwar et al., 2020). Still, recent research has shown that both a comprehensible understanding of the problem, as well as machine learning-based algorithms can facilitate increased automation (Wåreus and Hell, 2020).

In this paper, we consider the problem of assessing whether a codebase is using vulnerable functionality. More specifically, we present a framework for automatically analyzing a codebase that depends on code with a known vulnerability and determining if the codebase can execute the vulnerable parts of that code. Though software might use a vulnerable version of a library, if the vulnerable functions are not called by the software, this should be seen as a false positive, allowing efforts to instead focus on mitigating actually exploitable vulnerabilities. Our framework consists of comparing the call graph of software to the changes made between vulnerable and non-vulnerable versions of a library. Moreover, we implement the framework for the Java programming language with the Maven dependency management system. This allows us to evaluate the performance of such an automated tool and to identify specific challenges to a successful and efficient deployment.

The paper is outlined as follows. Some background is given in Section 2. The overall framework is described in Section 3, our implementation is given in Section 4 and the evaluation results are given in Section 5. Section 6 describes limitations and scalability considerations. We discuss related work in Section 7 and conclude in Section 8.

2 BACKGROUND

2.1 Software Vulnerabilities

Vulnerabilities can be found in several places, including issues, advisories, and mailing lists, but the largest public database is NVD, maintained by NIST, with a current collection of about 150k vulnerabilities. These are enumerated using the Common Vulnerabilities and Exposures (CVE) scheme (The MITRE Corporation, 2021), maintained by MITRE. NVD adds information that is useful for analyzing the vulnerabilities. This includes e.g., CPE (Common Platform Enumeration) identifiers (NIST, 2011). A CPE is a standardized representation of a software library. It is given by a string,

```
cpe:2.3:a:vendor:product:version:*:*:*...
```

In addition to specifying vendor, product, and version, as above, the string supports several additional attributes, such as update, language, target software and hardware, etc. In order to identify the use of vulnerable libraries, the library version can be matched against the CPEs for a given vulnerability. This assumes that a correct CPE for a library can be determined, e.g., from the dependency management system. The Maven dependency management system lists the dependencies of a project in a file named pom.xml. This XML file[1] defines a dependency using three main attributes.

- **groupId**. This should be unique for an organization or project.
- **artifactId**. This points out the specific project.
- **version**. This gives the version of the dependency or a valid range of versions that Maven can download and link at compilation time.

The discrepancy between CPEs, as used by e.g., NVD and the Maven dependency data causes additional challenges when identifying potentially vulnerable libraries.

[1]See https://maven.apache.org/pom.html for details.

2.2 Call Graphs

A call graph is a graph that describes which method (function, subroutine, ...) in a program can call which other methods (Ryder, 1979). The graph represents each method as a node and each potential call as an edge from a caller to a (potential) callee. When a method m_1 can reach a method m_2 by following a path (a sequence of edges) in the call graph, the call graph predicts that a call to m_1 might (indirectly) result in a call to m_2.

In software security, we are mainly interested in *sound* call graphs, i.e., call graphs that conservatively approximate all possible calls. In a sound call graph, the absence of a path from the node for method m_1 to the node for method m_2 means that a call to m_1 can *never* result in the call to method m_2.

Program analysis researchers have developed a number of techniques for automatically extracting call graphs from software. For example, dynamic call graph analysis (Xie and Notkin, 2002) computes *precise* call graphs by running a program through a test suite and observing which calls go to which methods; however, the resultant dynamic call graphs lack any edges that were not exercised in the test suite, meaning that they are not (generally) sound.

Static call graph analysis, by contrast, examines the program's structure and produces conservative approximations. Much of the work in this area has been on increasing precision, i.e., eliminating impossible call edges (Smaragdakis et al., 2015). In a recent analysis, Sui et al. provide an overview over the main static techniques (Sui et al., 2020), but also observe that the state of the art does not produce sound call graphs: They report a median recall of 88.4%, or 93.5% with more expensive modeling for dynamic language features. These algorithms can miss call edges due to "dynamic" language features such as reflective calls (e.g., Java's `Method::invoke`) or dynamic linking (e.g., POSIX' `dlopen`), but especially due to native (e.g., Java-to-C) calls. For highly dynamic languages such as JavaScript that heavily rely on dynamic code loading and code generation, static call graph construction is particularly challenging (Antal et al., 2018).

3 FRAMEWORK OVERVIEW

In this section, we describe an overall framework for determining if a piece of software is using the vulnerable function. An overview of the framework is given in Figure 1.

First, we detect which libraries, and versions, are

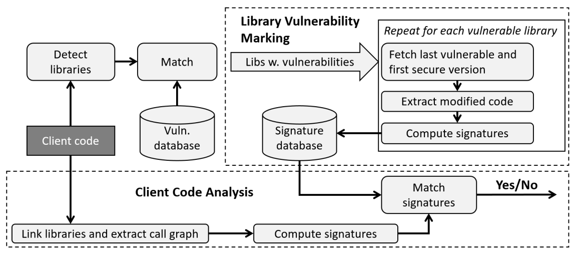

Figure 1: Vulnerable Functionality Framework

used by the software. This can be done in several ways, e.g., through a software Bill of Materials (BoM), scanning of the sources, or by analyzing the dependency file used by dependency managers. It is important to detect not only the direct dependencies used but also all transitive dependencies, as vulnerabilities can be located also in these. The next step is to detect vulnerabilities in the used libraries. Vulnerabilities can be identified e.g., using NVD, but other databases or sources can also be used. With a match between a CPE and a used library, we proceed with the library vulnerability marking and program analysis respectively in order to detect if vulnerable functionality is being used.

3.1 Library Vulnerability Marking

When a piece of software calls a library with a known vulnerability, the software *may* be vulnerable. Whether it *is* vulnerable depends on whether the software calls the vulnerable portion of the library's code in a way that exercises the vulnerability.

In general, this property is not decidable, but we can approximate it e.g. by checking if the client software can ever call the vulnerable library.

In our framework, we use a more precise technique, in which we mark library vulnerabilities at the *method level*; i.e., we identify which methods may trigger the vulnerability. Our technique relies on two versions of the same library: one (referenced by the client software) with a known vulnerability, and another, later version in which the developers have fixed that vulnerability. These versions may be different releases of the same library or different commits to the library's revision control system. We extract the modified code by computing the difference between these two versions and conservatively mark all methods that have been changed as vulnerable. The result is a set of potentially vulnerable methods. We then compute uniquely identifying signatures for these methods and store these signatures in a database. It can be noted that this information is independent of the client code and can be re-used for any number of clients.

Table 1: Language features that interfere with sound call graph construction for a selection of popular languages. The below is approximate, due to language extensions (e.g., inline assembly for C/C++ adds "Native Calls"-like complexity) and special run-time features (e.g., the POSIX signal handling facility adds "External Callbacks"-like challenges).

	C/C++	Java	Python	JavaScript
Not Memory Safe	X			
`eval()`			X	X
Dynamic Code Loading	X	X	X	X
Native Calls		X	X	
External Callbacks		VM		Browser
Reflection		X	X	X
Prototype Based				X

3.2 Client Code Analysis

To check if a piece of client code may exercise a known vulnerability, we link the code against its libraries and compute the code's call graph. We then extract all methods that are reachable from the client code's entry points, compute the corresponding signatures, and compare them against our signature database with library vulnerabilities. If there is a match, we report that the client code is affected by a vulnerability.

Our database can also track which vulnerabilities are related to which methods, and how precise we believe our vulnerability markings to be for this library (e.g., whether we were able to compute method-level information at all), to improve reporting.

4 IMPLEMENTATION

Instantiating the framework requires choices and solutions to a set of challenges. In this section, we describe our implementation and design decisions.

4.1 Language Support

Our techniques depend intrinsically on each programming language's semantics. Many languages have features for indirect calls, including calls to dynamically loaded or synthesized code. For example, C/C++ support function pointers, and the lack of type safety in C/C++ allows programmers to hide function pointers in arbitrarily complex encodings. Dynamic languages (JavaScript, Python etc.) provide `eval()` functions (McCarthy, 1960) that execute string values as code. All these features complicate analysis.

Table 1 provides an overview of likely challenges in static call graph generation for several popular languages. "Dynamic Code Loading", "Native Calls" and "External Callbacks" allow calls to or from code that the static analysis may not be able to see. "Reflection" is a weakened form of `eval()` that allows e.g. instantiating a class from a string that provides the class name. "Prototype Based" refers to a dynamic form of inheritance that is incompatible with most static call graph analysis techniques.

While specialized techniques can improve call graph precision to some degree (Ko et al., 2015), sound call graph construction may still identify a very high number of false positives, and practical call graph construction algorithms may choose to relax soundness in favor of higher precision. If the extent of this relaxation is clearly specified, we refer to the relaxed analysis as "soundy" (Livshits et al., 2015). A productionized version of our approach could combine soundy call graph analysis with a sound detector for all sources of unsoundness, for manual inspection.

Our implementation targets Java, due to the maturity of Java tooling. Java is memory safe and has no built-in `eval()`, but supports dynamic loading and reflection (Li et al., 2019).

4.2 Library and Vulnerability Detection

To favor simplicity and ease of use, as much of the process as possible should be automated. Thus, an implementation should be integrated with a developer's existing toolchain. In modern development, delivering and deploying new software versions and services is automated through Continuous Integration and Continuous Deployment (CI/CD). The CI/CD flow automates the software development flow, from design to testing and deployment, enabling incremental roll-out of new software. This allows e.g., vulnerable libraries to be updated without waiting for other code to finish. Assuming such a modern, and agile, development process, we use information that is already being submitted to the CI/CD pipeline and integrate our analysis framework with the version management system. This way, we parse the same files that the version management system does, ensuring that the user only needs to supply files readily available and that are anyway uploaded to version management and CI/CD systems. This has the additional advantage of ensuring that the vulnerability filter fetches exactly the same libraries that the code normally uses, eliminating a potential source of error. In particular, we implement support for the Maven dependency management system. Maven is a popular dependency management system for Java, used for dependency resolution, including transitive dependencies, as well as some version management.

We use NVD for finding vulnerabilities that affect the libraries in use. If the CPE information for a given vulnerability matches the library version given by the Maven dependency management system, then this library will be subject to our library analysis.

4.3 Implementing Library Analysis

With a known vulnerability, we also know the latest vulnerable version from the CPE list on NVD. Using Maven, we fetch the first non-vulnerable version and compare these two. The comparison is done in two steps. First, we use the `java-diff-utils`[2] library to produce the diff between the versions, and the list of line numbers that are changed, added or removed, as well as if any new files are added to the library. Then we use `ExtendJ` (Ekman and Hedin, 2007) to translate this list of files and lines into a list of Java classes and methods. Then, these classes and methods are converted into unique signatures. These signatures uniquely identify a method or class, and so we are able to know which methods and classes have changed between the vulnerable and fixed versions. This list is stored in a database for later use.

This process of library analysis is fully automated, from determining the non-vulnerable version to producing the output with signatures. However, automation here comes with a cost. Versions susceptible to a vulnerability (and by the method of elimination which versions are not) are available from the vulnerability database. Which specific commit includes the fix is however not available. Because of this, we limit ourselves to analyzing released versions, possibly introducing some false positives when the releases include more updates than just the vulnerability patch.

4.4 Implementing Client Code Analysis

For client code analysis, we construct a call graph of the client code. As we detail in Section 5, test coverage is very low for many projects. Thus, we choose the static approach over a dynamic one.

We create the call graph using the Soot (Vallée-Rai et al., 1999) static analyzer, using the Class Hierarchy Analysis algorithm (Dean et al., 1995). Soot is a framework for Java code analysis, transformation, and optimization, but we use it only for its ability to generate call graphs of Java programs.

Once we have constructed a call graph over the client code (with the included library code) we can check this to see if it contains any of the vulnerable signatures that we found in the library analysis phase. Here, the Soot output is used to create signatures that can be matched with those computed in the library

[2]https://java-diff-utils.github.io/java-diff-utils/

analysis. If there is a match, we have shown that the client is using the part of a library that changed between a vulnerable and a fixed version. Thus, they are not only using a vulnerable library but likely the specific vulnerable part of that library. We also present the execution path to the code to inform the user which part or parts of their codebase that is affected. This is useful in case the library is not updated, but the vulnerability is mitigated in another way, e.g., if the update introduces breaking changes.

5 RESULTS

Projects using Maven were selected by searching GitHub 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 reasonably random result and we chose 200 programs to analyze.

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, either directly or transitively.

We first remove projects that are not interesting or even possible to analyze. Out of the 200 projects, 50 failed to compile, 9 had no code to analyze, and an additional 11 had to be excluded due to having erroneous or no longer working Maven configurations. This leaves us with a total of 130 projects that can potentially be analyzed.

5.1 Analyzable Projects

The version of Soot we use can only analyze Java version 8 or lower code. Thus, 15 projects had to be excluded due to Soot not being able to analyze them.

One program was manually excluded since the repository supposedly containing many of the dependencies of the said program no longer did so. This left us with a total of 114 projects that compiled, had code to analyze, and that Soot was capable of analyzing.

5.2 Dependencies with Matching Vulnerabilities

Of the 114 projects that we could analyze, 111 had third-party dependencies, while 3 did not. Further, 102 projects were analyzed to have at least one dependency with a CVE listed against the version in use. In other words, 89% of the projects used dependencies with known vulnerabilities, while 11% (12 out of 114) did not have any dependencies with known vulnerabilities (CVEs).

Table 2: Test coverage for the sampled projects.

Category	Projects	Can calc. coverage	Test coverage
Total projects	200	23	26.8%
Static analysis	114	20	27.2%
Vuln. dependency	76	14	16.4%

Looking at the 111 projects with dependencies, there were in total 8701 usages of 3117 unique dependencies and versions. Of the 8701 usages, 2741 were of a dependency with a CVE listed against it, meaning that about 32% of the time a dependency was included that dependency had a CVE listed against it. These usages of vulnerable dependencies represented in total 299 unique dependencies and versions.

5.3 Analyzable Dependencies

Of these 2741 usages of dependencies with CVEs listed against them, we were not able to analyze 514 of them (19%). Out of the 2227 that we were able to analyze, 1682 used the dependency in such a way as to expose the vulnerable parts of the dependency. The other 545 uses were analyzed not to use the vulnerable functionality. This means that about 20% (545 out of 2741) of dependencies with a matching vulnerability (CVE) were not vulnerable since the vulnerable functionality was not used. If we exclude dependencies that we could not analyze, 24% (545 out of 2227) did not use the vulnerable functionality.

5.4 Test Coverage

Our choice of static analysis for creating the call graph is motivated by the test coverage for the analyzed projects. To measure the test coverage, the JaCoCo Maven plugin was used. Of the 200 projects, only 32 had tests that passed and 23 of these produced valid test coverage with JaCoCo, with an average of 26.8% reported coverage. For projects with vulnerable dependencies, we found that the test coverage was only 16.4% on average, see Table 2. This shows the importance of using static call graph generation as opposed to a dynamic approach.

6 LIMITATIONS AND SCALABILITY

In this section, we identify why some (514 out of 2741) of the dependency usages with a listed CVE could not be analyzed. Then, we also analyze the scalability of our approach.

Table 3: Dependency usages that could not be analyzed for some packages.

Project	Usages that could not be analyzed
postgresql	190
jetty	75
javax mail	50
dom4j	25
apache poi	20
other	154
Total	514

Table 4: Most common reasons for why a vulnerable dependency could not be analyzed.

Reason	Vuln. that could not be analyzed
Version not available	82
Versioning Scheme Mismatch	81
Dependency moved	73
Program limitations	67
No end version	26
Modifier changed	12
Other	11
Total	352

6.1 Dependencies that Could not be Analyzed

We see that a small number of dependencies that are listed in many CVEs are responsible for a large share of these. For example, PostgreSQL alone is responsible for over a third of the vulnerability usages we are unable to analyze, 190 of the 514. Table 3 summarizes the dependencies that caused most problems.

Many of these 514 dependency usages refer to the same 299 vulnerable dependencies, but possibly to different vulnerabilities. We therefore combine references to the same vulnerabilities in our summary in Table 4, for a total of 352 references to different vulnerabilities that we could not analyze.

In 82 cases the specific version that we need for the analysis is not available in the Maven repository, i.e., a version referenced in NVD is not listed by Maven at all. Similarly, we found 81 cases where there was a versioning scheme mismatch between NVD and Maven. This case was specific to postgresql. In some cases, we noticed that the groupId or artifactId changed between two software versions. Such changes in the dependency naming occurred in 73 cases. We identified 67 cases that we attribute to limitations in the software, e.g., handling of additional qualifiers and build dates in version information. In 26 cases, NVD does not have any information about in which version a vulnerability is fixed, i.e., no end version CPE is provided for the vulnerable software. Finally, in 12 cases we found that a modifier changed between versions. A modifier is the part of the version string that is not dot-separated numbers. For example, we want to compare 24.2-jre with 24.3-jre, not with 24.3-android.

6.2 General Limitations

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.

Sometimes, there is a discrepancy between the naming schemes in the vulnerability database and the dependency management system. This can lead to both false negatives and false positives when names are interpreted and matched. An example of a potential error is the spring framework. In Maven, Spring implementations are distributed as separate packages with the `org.springframework` group ID and with artifact IDs like `spring-web`, `spring-orm` etc. In NVD, however, no such distinction is made and CVEs are just filed against "`spring_framework`".

Any call graph of reasonable precision for a language such as Java will not be sound in the general case. In other words, errors can be introduced when creating the call graph. The exact error depends on many design choices in the call graph analysis.We therefore opted for a highly conservative call graph algorithm. Still, some of the challenges identified in this study, such as call graph algorithms failing to fully model callbacks from native code may also affect the recall of our approach.

6.3 Efficiency and Scalability

To be viable for larger projects, the approach needs to scale well with project size. Large projects are here seen as either projects with a large number of classes and methods, or as projects with a large number of dependencies. Due to dependencies possibly being transitive, importing one library could result in several hundred dependencies, where some of them can be matched against a CVE. Thus, when looking at scaling, we analyzed two factors, namely (a) the number of vulnerable dependencies (before analysis), and (b) the size of the call graph.

We ran these experiments on a Dell XPS 9350 laptop. Considering the running time as a function of the

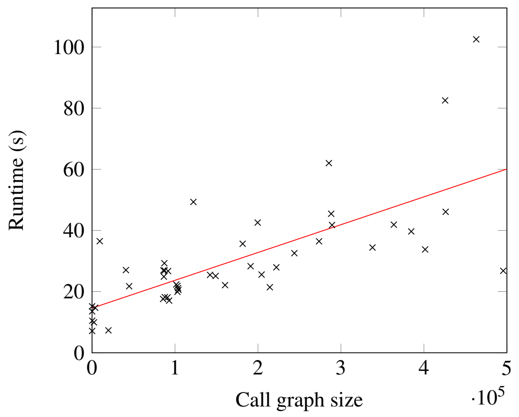

Figure 2: Call graph size vs. runtime, zoomed in on projects with at most 500,000 call graph edges.

number of vulnerable dependencies, the correlation coefficient is 0.11, indicating a very weak correlation.

If we instead consider the number of edges in the call graph, we see a much stronger correlation to the running time. The correlation coefficient is here 0.96. For the vast majority of projects, the call graph has fewer than 500,000 edges, with only a few outliers in our sample. Looking at only projects with up to 500,000 call graph edges, the correlation coefficient is 0.71, still a rather strong correlation. This is shown in Figure 2, together with linear regression. The lower correlation is due to the library analysis not being included in the call graph size, and for large projects this is negligible. For smaller projects, the library analysis will affect the running time, but will still not be visible when looking at the program call graph.

7 RELATED WORK

In our work, we focus on utilizing static call graphs to improve the precision of vulnerability discovery. Our static approach is similar to Plate et al.'s dynamic approach (Plate et al., 2015), which collects call graphs while running test. Their approach therefore requires a high-quality test suite. Existing techniques for automatic test generation can aid such approaches, but cannot supplant hand-written tests (Sui et al., 2020).

Ponta et al. expand on Plate et al.'s approach, combining static and dynamic analysis (Ponta et al., 2018) to address this limitation in a productionized tool used internally at SAP and exploring mitigation strategies. Their findings show that on SAP code, 7.9% of the vulnerabilities that they detect require both static and dynamic analysis. It is not clear how their findings translate to software developed outside of this specific industrial environment; for instance, the low degree of unit test coverage that we found (Section 5.4) would severely limit their dynamic techniques. They do not describe their call graph construction algorithm, so we cannot make a more detailed comparison.

Our approach relies on CPE information to identify vulnerable versions of libraries. However, that information may be incomplete or inaccurate. If the CPE does not specify the earliest vulnerable version of a library, we currently make the conservative assumption that all versions prior to the reported fix (if any) are vulnerable. Dashevskyi et al. introduce a technique for tracing a vulnerability backward in time through an affected piece of software to find out at what point that vulnerability was first introduced (Dashevskyi et al., 2019). This approach could complement our technique and reduce the false positive rate.

8 CONCLUSIONS

We have described a framework for analyzing the use of vulnerable functionality in libraries containing vulnerable code. We instantiated the framework to analyzing Java libraries, using data from the Maven dependency management system and evaluated the on randomly chosen GitHub projects. Of 111 analyzable projects with dependencies, we identified 2741 usages of dependencies with known vulnerabilities and successfully analyzed 2227 of these usages, of which 545 uses did not use the vulnerable functionality. Though our implementation has limitations, it provides a promising step towards fully automating vulnerability identification, removing false positives stemming from a pure CPE match.

Acknowledgements

This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation, and partially by the Swedish Foundation for Strategic Research, grant RIT17-0035.

REFERENCES

Alomar, N., Wijesekera, P., Qiu, E., and Egelman, S. (2020). "You've got your nice list of bugs, now what?" vulnerability discovery and management processes in the wild. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS) 2020*, pages 319–339.

Antal, G., Hegedűs, P., Tóth, Z., Ferenc, R., and Gyimóthy, T. (2018). Static JavaScript Call Graphs: a Comparative Study. In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE.

Anwar, A., Abusnaina, A., Chen, S., Li, F., and Mohaisen, D. (2020). Cleaning the NVD: Comprehensive quality assessment, improvements, and analyses. arXiv:2006.15074.

Dashevskyi, S., Brucker, A. D., and Massacci, F. (2019). A screening test for disclosed vulnerabilities in foss components. *IEEE Transactions on Software Engineering*, 45(10):945–966.

Dean, J., Grove, D., and Chambers, C. (1995). Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer.

Ekman, T. and Hedin, G. (2007). The JastAdd extensible java compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–18.

Höst, M., Sönnerup, J., Hell, M., and Olsson, T. (2018). Industrial practices in security vulnerability management for iot systems–an interview study. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, pages 61–67.

Ko, Y., Lee, H., Dolby, J., and Ryu, S. (2015). Practically tunable static analysis framework for large-scale javascript applications (T). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 541–551. IEEE.

Li, Y., Tan, T., and Xue, J. (2019). Understanding and analyzing java reflection. *ACM Trans. Softw. Eng. Methodol.*, 28(2):7:1–7:50.

Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B.-Y. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. (2015). In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195.

NIST (2011). Common Platform Enumeration: Naming Specification, Version 2.3, NIST Interagency Report 7695.

NIST (2021). National vulnerability database. *https://nvd.nist.gov/.*

Plate, H., Ponta, S. E., and Sabetta, A. (2015). Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–420.

Ponta, S. E., Plate, H., and Sabetta, A. (2018). 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)*.

Ryder, B. G. (1979). Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226.

Smaragdakis, Y., Balatsouras, G., Kastrinis, G., and Bravenboer, M. (2015). More sound static handling of java reflection. In Feng, X. and Park, S., editors, *Programming Languages and Systems*, pages 485–503, Cham. Springer International Publishing.

Sui, L., Dietrich, J., Tahir, A., and Fourtounis, G. (2020). On the recall of static call graph construction in practice. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1049–1060. IEEE.

Synopsys (2020). Open source security and risk analysis report. Online.

The MITRE Corporation (2021). Common Vulnerabilities and Exposures.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L. J., Lam, P., and Sundaresan, V. (1999). Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '99*.

Wåreus, E. and Hell, M. (2020). Automated cpe labeling of cve summaries with machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–22. Springer.

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

Xie, T. and Notkin, D. (2002). An empirical study of java dynamic call graph extractors. *University of Washington CSE Technical Report 02-12*, 3.