Enhancing Vulnerability Detection: A Comparative Study of Change Identification Methods Across Granularity Levels

D.D.J.Z. Berendsen

Enhancing Vulnerability Detection: A Comparative Study of Change Identification Methods Across Granularity Levels

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

D.D.J.Z. Berendsen
born in Qinghai, China

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Enhancing Vulnerability Detection: A Comparative Study of Change Identification Methods Across Granularity Levels

Author: D.D.J.Z. Berendsen
Student id: 4904982

Abstract

Open source software (OSS) vulnerabilities form a real threat to the security of software that employs them. Efforts to mitigate these risks exist in the form of dependency check tools, however these often suffer from imprecise warnings due to the utilization of only metadata. This thesis investigates the use of CVEs in a more code-centric approach and the effect this has on the detection of vulnerability reachability in OSS dependencies. This paper proposes an automated approach to enrich CVEs with preciser code-level information by leveraging references such as patches, repositories, and vulnerability databases. This thesis then heads out to investigate the impact on accuracy of various (novel) approaches in terms of granularity (packages, classes, methods) and in terms of source for the patch information (link to a commit, PR of commits, or binary diff). Our experimental results show that these code-centric approaches significantly improve vulnerability detection, achieving higher precision compared to traditional dependency checkers. Additionally, we present the trade-offs between the different methods, highlighting their strengths and weaknesses. Through this work, we show how utilizing code information into dependency analysis can substantially enhance the detection of vulnerable code paths, offering more accurate risk assessments in software ecosystems.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr.ing. S. Proksch, Faculty EEMCS, TU Delft
Committee Member:	Dr. H. Griffioen, Faculty EEMCS, TU Delft

Preface

I want to thank my daily supervisor Dr. Ing. Sebastian Proksch for his constructive feedback and valuable support in our weekly meetings. Furthermore I want to thank Prof.dr. Arie van Deursen for providing me with supervision and feedback during the development of this project.

D.D.J.Z. Berendsen
Delft, the Netherlands
November 26, 2024

Contents

Chapter 1

Introduction

With the increased use of the internet and modern technologies, ensuring good security is currently of paramount importance for software developers. Various rules and tools have been developed to help with tracking and maintaining a good level of security in software. One such initiative is the Common Vulnerabilities and Exposures (CVE) system, which serves as a comprehensive dataset for tracking and documenting standardized information for known vulnerabilities within software and hardware. This dataset, maintained by MITRE [8], plays a critical role in vulnerability management.. Using vulnerable packages and thus vulnerable code remains a significant risk. For instance, *Using Components with Known Vulnerabilities* was ranked 9th within the OWASP top 10 in 2017 and in 2021 this was even moved up to 6th. Some notable examples of vulnerabilities within open-source software (OSS) projects include the HEARTBLEED [10] and the LOG4SHELL [6] vulnerabilities. These critical issues impacted millions of users due to vulnerabilities within the OPENSSL and the LOG4J open source software packages. In response to such risks, considerable efforts have been made to help mitigate these risks, particularly in detecting the use of vulnerable package. Most of this effort has come in the form of dependency checkers such as the OWASP DEPENDENCY CHECKER [19], DEPENDABOT from GITHUB [13] as well as the commercial tools like SNYK [26] and CHECKMARX [5]. However, these tools primarily focus on checking the dependencies' metadata. There is increasing interest in tools that provide more granular insights into CVE-related vulnerabilities, including the identification of fixes and improved reporting. For example CVEFIXES [2] has been developed to map code vulnerabilities to their corresponding fixes. Similarly, ECLIPSE STEADY [22] provides enhanced code-level reporting, while its sub-project, FIXFINDER [16] contained within PROJECT-KB [21], maps CVE entries to the code patches that address those vulnerabilities in OSS projects.

The key issue with the current use of CVEs in dependency tracking is that these vulnerabilities are mainly reported on the coarse-grained package level. Simply including a package with a known vulnerability within it will trigger a warning, even if the vulnerable code within the package is not actually used. This results in overly conservative CVE reporting for dependencies, often leading to false positive warnings, where safe usage of vulnerable dependencies is still flagged as risky. Consequently, developers may unnecessarily remove dependencies that are, in practice, safe to use. On the other hand, this imprecision generates

an excessive number of security alerts with low precision, leading to *alert fatigue*. Over time, developers may become desensitized to these warnings and begin ignoring them altogether, increasing the risk of overlooking genuine security threats. There is a clear need to reduce the amount of false positives and improve the precision of the vulnerability detection tools. Adopting finer-grained vulnerability detection and reporting that analyzes actual code usage, rather than simply package inclusion, could address this issue. For example, a dependency might be flagged as vulnerable under coarse-grained detection based solely on metadata and consequently rejected. However, with a finer-grained report analyzing the file, class, or even method level, developers could more accurately determine whether the vulnerability actually affects the current usage of that dependency. With this level of detail it could be justified to continue the use of an dependency, saving developers the time and effort needed for replacing the dependency. With this level of precision, developers could justify continuing to use a flagged dependency, saving the time and effort needed to replace it unnecessarily. By identifying whether a vulnerability is actively used within the codebase, finer-grained tools provide more precise indications of when mitigation is truly required. Furthermore, when a vulnerability is confirmed to be present, finer-grained reporting can better pinpoint the specific code affected. This not only increases accuracy but also accelerates the process of fixing vulnerabilities introduced by a dependency.

Mentioned earlier ECLIPSE STEADY has made progress in achieving code-centric vulnerability reporting for dependencies. However, a significant challenge lies in the way the information is actually gathered. ECLIPSE STEADY relies on PROJECT-KB which, as stated in its documentation and data, is a manually curated dataset. This manual approach is not very scalable, as it depends on volunteer effort for additional contributions. This limitation contributed to the discontinuation of PROJECT-KB, and as PROJECT-KB becomes outdated, ECLIPSE STEADY also loses becomes outdated. Another challenge is the complexity of linking CVEs to their code/commit links. ECLIPSE STEADY employs a machine learning model (FIXFINDER [16]) to help identify and rank candidate commits. Similar to other solutions for linking CVE to commits, FIXFINDER scans whole GITHUB projects for relevant commits and uses some heuristics to filter results. However, these heuristics introduce risks, such as missing fixes that do not explicitly mention a CVE in their commit messages. Additionally, scanning entire GITHUB projects increases both the computational workload and the likelihood of retrieving irrelevant commits. With the growing number of CVEs and vulnerable packages, this approach becomes even less scalable over time. The inefficiency of cloning and scanning entire repositories underscores the need for a more targeted method of enrichment. There is a clear necessity for targeted automated methods of CVE enrichment, or at the very least, bridging the gap between the initial reporting of a CVE and the complex or manual processes required for its enrichment. Such enrichment is crucial for tools aiming to provide more in-depth and precise vulnerability reporting. One potential solution could involve automatically (programmatically) enriching CVE data to link it to the actual vulnerable code. This would reduce reliance on manual curation and improve scalability while enabling more accurate and actionable insights.

CVEs contain structured information about the vulnerabilities, however the primary issue (among others) is the inconsistency of the information within CVEs. These discrepancies are partially caused by the diverse ways in which CVEs are reported. Moreover, incon-

sistencies are often actually inherent within the CVE data itself. Variations exist not only in the formats used to report CVEs but also in the quality and availability of the information. Because of these inconsistencies, automated systems often rely on ad hoc methods for processing CVEs, and achieving generalization is not without its hurdles. These challenges complicate the automated extraction of code-level information from CVEs. To address this issue, we propose a fully automated approach to enrich CVE data, as such enrichment is essential for obtaining accurate code-level details. This effort required a thorough investigation into the structure of CVEs and the practical ways in which they are reported. However, an important challenge to consider in an automated approach is that without manual curation, there is no real verification and guarantee as to the correctness and accuracy of the resulting CVE enrichments.

Obtaining code information is crucial for addressing the issue of low precision inherent in the coarse-grained dependency checkers. However, a critical issue that must be resolved is determining the appropriate level of detail for improved reporting. This includes determining the right level of code information that should be collected for meaningful vulnerability analysis. Previous work often selects commit links as the preferred level of code information. However, commits may not always provide the most suitable level of details, and the challenge of unavailable commit link information must be tackled. To address these challenges alternative approaches for gathering the CVE code information should be investigated. In addition to determining the appropriate level of detail, it is equally important to identify the granularity at which automated systems can effectively mitigate the imprecision of dependency checkers. For example, greater precision might be achieved by assessing the reachability of vulnerabilities at the method level. Conversely, it could be argued that class-level granularity strikes a better balance between precision and practical applicability. Thus, exploring different granularity levels of vulnerability analysis such as method or class/file level is essential.

In the context of security, minimizing false negatives is paramount, as overlooking a vulnerability is far more damaging than addressing false positives. Consequently, while aiming for improved precision, it is critical to consider multiple levels of information gathering and mapping strategies for vulnerable code. This involves evaluating diverse approaches, including more conservative strategies that prioritize better recall (minimizing false negatives) over maximizing precision. Such strategies could serve as viable alternatives to expected high-precision methods which focus exclusively on commit- and method-level details.

The first goal of this research is to achieve fully automated enrichment of CVE information. This results in bridging the gap between CVE reporting and the manual processes currently required for data enrichment. This paper also aims to make CVE vulnerability reporting more actionable through a finer-grained level of detail, thereby increasing the precision of vulnerability warnings.

Central to this research is the question: What level of detail is most appropriate for actionable vulnerability reporting? To address this, our investigation begins with analyzing the CVE data itself and developing an approach for automated CVE enrichment with vulnerable code information. The effectiveness of this approach will be evaluated based on its ability to identify relevant code for CVEs and, more importantly, to accurately link

the correct code to the corresponding CVE. Additionally, the implications of these results for consequently improving vulnerability reporting will be discussed. The vulnerability reporting includes exploring methods to determine where and how a vulnerability manifests within a call graph, with the focus on generating more fine-grained vulnerability reports. Evaluation of the vulnerability reporting will specifically examine the impact the different approaches have on reporting back the exposure and the reachability of a vulnerability in the context of a particular library or dependency usage. This project will focus exclusively on JAVA vulnerabilities within the CVE dataset.

In this thesis, we investigate whether it is possible to improve the precision of vulnerability reporting by obtaining more fine-grained information on CVEs in an automated manner. Specifically we explore how different mapping strategies, from vulnerabilities to affected methods or files/classes, impact the results of vulnerability reporting. This project addresses several research questions related to the key steps involved in producing a fine-grained vulnerability report. The process starts with publicly available CVE information, which serves as the foundation for further analysis. From this starting point, it is necessary to extract data about vulnerability fixes, as these fixes indicate the locations in the code responsible for the vulnerabilities. However determining the appropriate starting point is not trivial due to the variability of CVE data. Accordingly, the first research question of this thesis is:

RQ$_1$ How are CVEs reported in practice, and what useful information do they actually contain for identifying vulnerable code?

For RQ$_1$, this paper conducts an empirical study on CVEs, analyzes related work, and identifies the information needed to uncover fine-grained code details for CVEs. Through the investigation we identified that vulnerable code and corresponding patches can be identified through references within the CVE data, and with the associated vulnerable packages. Based on these findings, the approach for gathering this information is developed, enabling the investigation of the second research question. In this phase, we evaluate the effectiveness of our automated approach in identifying vulnerable code through the references.

RQ$_2$ Can we find commit links that direct us towards code patches for a vulnerability?

- RQ$_{2.1}$: Can we recover commit links for CVEs?
- RQ$_{2.2}$: What is the precision of finding the correct commit links to a fix for a CVE?

For RQ$_2$ we find that our automated approach is able to meaningfully contribute towards recovering commit links. Our results also demonstrate that our patch-finding process improves the identification of the correct commit links. However it is also evident that this approach is not infallible, consequently this paper investigates different methods and levels of mapping vulnerable code to produce finer-grained vulnerability report. This exploration is driven by the following research questions:

RQ$_3$ How do the commit-, pull- and release version patch CVE information mappings impact the exposure and reachability results of a vulnerability?

RQ$_4$ How do the method and file level granularity mappings impact the exposure and reachability results of a vulnerability?

For these research questions, the impact of different methodologies on precision is evaluated by generating vulnerability reports via call graph reachability and exposure. Our results demonstrate that, with our approach the precision of vulnerability reporting can be significantly improved when compared to current dependency checkers. Furthermore, our findings highlight the trade-off between achieving higher precision and maintaining guarantees on recall, i.e., the ability to identify utilization of vulnerabilities. As expected the finest grained information leads to the highest precision but risks lower recall when compared to more conservative methods. More conservative methods, while presenting lower precision, offer better guarantees on recall. Nevertheless, even these conservative methods still significantly outperform dependency checkers in terms of precision.

This thesis thus presents the following insights and main contributions:

- An automated approach for enriching CVEs by adding patch information. [1]
- Our empirical analysis reveals that references, such as commit links, within CVEs can be utilized to identify code (changes) related to the vulnerability. Furthermore, our findings suggest that the annotations accompanying these references should not play a significant role in the code discovery process. We also introduce the novel idea of using package version diffs in the process of identifying changed code.
- Our research into the impact of different patch-finding methods on finer-grained vulnerability reporting in OSS projects demonstrates that significant improvements in precision by utilizing finer-grained code information, can be achieved over traditional dependency checkers. Additionally, we show the trade-offs between precision and recall for different approaches, highlighting how more conservative patch-finding methods can provide stronger recall guarantees while balancing precision.
- Our investigation into the impact of different levels of granularity in vulnerable code reachability detection and subsequent vulnerability reporting indicates that, alongside method-level code mapping, reporting vulnerabilities at the class/file granularity offers a suitable and more conservative alternative.

[1] https://github.com/DanDanBro/fg-cve-aggregator

Chapter 2

Related Work

In the paper by CADARIU ET AL. [4] the authors demonstrated that the use of vulnerable components poses a significant risk within the software development industry. By employing the OWASP Dependency-Check tool for vulnerability scanning, they revealed that many projects contain dependencies with CVE vulnerabilities. However, the paper also highlighted the issue of low precision due to the prevalence of false positives. These inaccuracies underscore the risks associated with CVEs and the need for improved detection and reporting mechanisms.

Dependency check tools The first efforts towards vulnerability detection and mitigation in dependencies primarily came in the form of dependency check tools. Numerous studies have examined on automated tools to assist in this space, including tools like OWASP DEPENDENCY-CHECK, SONATYPE NEXUS, and SNYK. However, these tools typically rely on metadata for detecting vulnerable dependencies, resulting in imprecise warnings and contributing *alert fatigue*. Additionally, DONG ET AL. [9] raised concerns about inconsistencies in the vulnerable version reporting of CVEs, which often lead to both overestimation and underestimation of vulnerabilities' scope. These challenges have motivated researchers to explore more precise methods for obtaining and utilizing CVE information.

Vulnerability discovery The foundation of the CVE system lies in the discovery of vulnerabilities. For instance KLUBAN ET AL. [18] proposed a vulnerability detection framework to identify vulnerable functions in real-world JAVASCRIPT projects, enabling the discovery and reporting of new CVEs. While this work focuses on identifying vulnerabilities within individual projects, our primary interest lies in detecting vulnerabilities within dependencies, particularly determining whether vulnerabilities are actually used in a specific context. For this purpose, it is essential to understand the precise code affected by a vulnerability to assess its reachability and impact.

Vulnerability datasets Various datasets and approaches have been developed to identify which parts of code are affected by certain vulnerabilities. Some tools collect code solely utilizing the information directly provided in CVEs. For instance, REIS ET AL. [24] curated a

ground-truth dataset by collecting code exclusively from commits directly linked to the CVE data. While useful, this approach does not include additional enrichment to gather more comprehensive details about the vulnerable code beyond the explicit commit links provided in the CVE. Similarly Big-Vul [11] and VULINOSS[15] rely on CVEs providing accurate information. BIG-VUL dataset focuses on extracting commits made before and after a fix. VULINOSS introduces modifications to the references themselves, to account for issues such as missspellings, software vendor changes or name variations. Although VulinOSS attempts to address imprecise references, it remains constrained to the CVE data's limitations.

Some datasets go a step further by performing enrichment of CVE data to identify additional commit links. These approaches often rely on manual curation or complex computational models to achieve good results. For instance, VCCFINDER [20] employs a support vector machine (SVM) machine learning model to identify vulnerability-introducing commits. Similarly, V-SZZ [1] extends this idea by aiming to obtain more precise CVE information related to releases, including identifying vulnerability-introducing commits. However on the contrary most works prioritize identifying vulnerable code by locating vulnerability patching commits. Tools like CVEFIXES [2] and VULDATA7 [17] achieve the collection of fix data by scanning through GITHUB repositories, searching for patch commits. SSPCATCHER [25] introduces a machine learning model specifically designed to monitor repositories for potential vulnerability-fixing commits. The scanning of entire GITHUB repositories run a high risk of finding irrelevant commits, necessitating robust filtering mechanics to ensure meaningful results. One of the most prominent tools in the field of CVE patch information gathering is FIXFINDER (part of PROJECT-KB) [16]. FixFinder employs a machine learning model to scan through the commits, narrowing down candidate commits by filtering out those irrelevant to the vulnerability. It then ranks the remaining candidates based on their likelihood of being the fix or at least relevant. However, having a ranking means that they still require manual inspection to confirm the correct commits, which limits its scalability. Additionally, because the project was reliant on volunteer efforts for manual curation, it was officially discontinued in 2022.

Our approach Most existing works face significant challenges, either requiring extensive manual effort or relying on complex, resource-intensive processes for enriching CVE data with commit links. These limitations hinder scalability and maintainability, making such approaches less practical for long-term application. This thesis aims to provide a long term sustainable approach for obtaining finer-grained code information on CVEs. With our approach we bridge the gap between the initial reporting of CVE data and the sophisticated but high-overhead methods currently available CVE enrichment.

A comparable method to ours is proposed by XU ET AL. [27], their approach also involves scanning of entire GITHUB projects histories. Thereby inheriting some of the same limitations and pitfalls associated with repository-wide scanning as seen in other works. But similar to our work they also leverages the references inside the different advisories to locate patches. They then try to filter out commits found through the reference search based on 2 heuristics. In contrast, we argue against filtering out commits found through references. We retain all candidate commits from the reference web, even at the cost of overestimation, as these commits should be relevant to a CVE since they are found through

references. This strategy not only simplifies the enrichment process but also ensures that potentially relevant commits are not prematurely excluded.

Chapter 3

Overview

This chapter provides a comprehensive overview of this thesis. The thesis is structured into three primary aspects: *The empirical study, changed code discovery, and the vulnerability detection.* Figure 3.1 illustrates the project pipeline, highlighting the components associated with each aspect.

Empirical study The first aspect establishes the foundation for working with CVE data by investigating how CVEs are reported in practice. First in Chapter 4 we conduct an empirical study investigating the various sources and formats of CVE information. We investigated

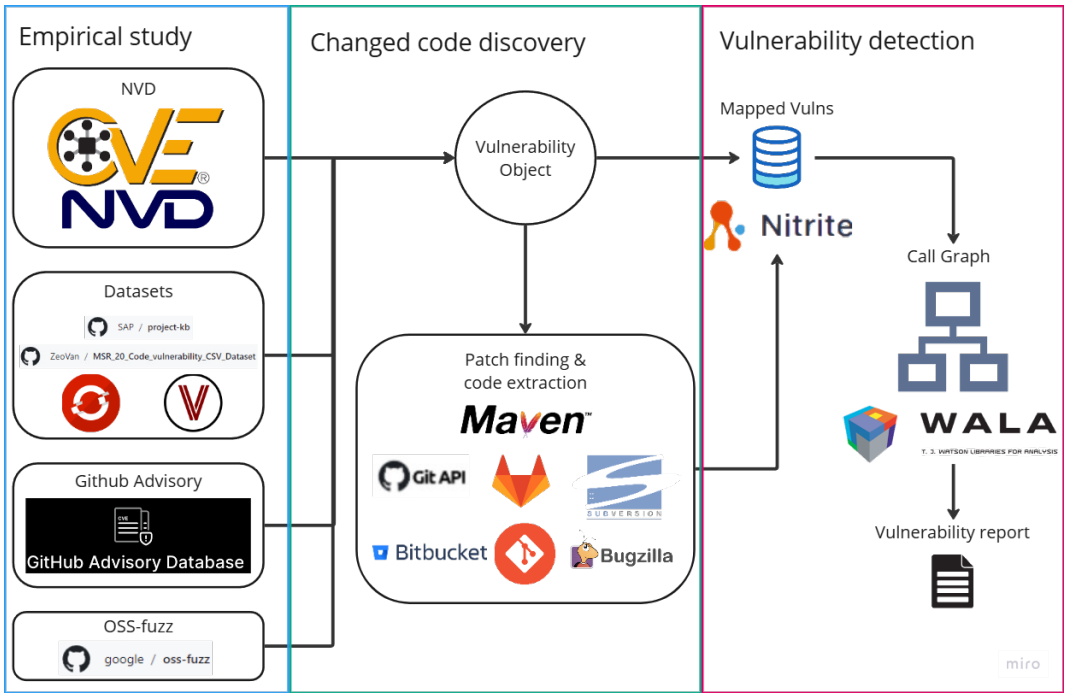

Figure 3.1: High level overview of the project

how CVEs are actually reported in practice. With this investigation we address RQ_1, and we determine the basis of the information we use. In this chapter we establish the aggregation for the starting dataset we use within this thesis. Furthermore we discuss the implications of these findings for the subsequent aspects of the project.

Changed code discovery The second aspect focuses on identifying the code changes associated with CVEs, a critical step for achieving finer-grained vulnerability reporting. Based on the findings from the empirical study we form the process of obtaining finer-grained code details for CVEs. In Chapter 5 we cover the changed code discovery aspect which starts with the process of collection and parsing the available CVE data into a unified format. Then we propose our approach for, automated CVE enrichment with changed code information, utilizing references within the CVE information. Via both commit patch finding as well as release diffs, code changes that should be associated with CVEs within our starting dataset are extracted. Chapter 5 covers the details and the evaluation of the code change discovery. In particular, the patch commit identification capability of our automated approach is evaluated in two different evaluations, covering the sub questions of RQ_2. Through answering the sub questions via the assessment of the patch finding capability, we answer the overarching RQ_2. The CVE data and the code changes are combined into a resulting dataset called `MappedVulns` where the CVEs are associated with their patch code.

Vulnerability detection The third aspect leverages the MappedVulns dataset to investigate finer-grained vulnerability detection in dependencies. The `MappedVulns` dataset provides the data used for the vulnerability detection covered in Chapter 6. We introduce various alternative approaches for the mapping of vulnerable code. In the context of vulnerability detection, our focus lies in assessing the impact of these different approaches on the vulnerability detection capabilities. Additionally, we investigate whether our methods for detecting vulnerability utilization through dependencies can yield superior results compared to existing solutions. From the `MappedVulns` dataset the affected packages are identified and call graphs are generated for finding the exposure or reachability of vulnerable code. Based on the exposure or reachability we can generate vulnerability usage reports. Based on the results of call graph searches and the vulnerability report generation we answer both RQ_3 and RQ_4.

Lastly, in Chapter 7.1, reflects on the results for each of the three aspects and their implications. Furthermore we discuss opportunities for future work and address any threats to validity encountered during the research.

Chapter 4

How are CVEs reported in practice?

To answer RQ_1: "How are CVEs reported in practice, and what useful information do they actually contain for identifying vulnerable code?", this section focuses on understanding the structure, variability, and practical reporting of CVEs. This understanding is critical for processing and enriching CVE information effectively, particularly for producing finer-grained vulnerability reports.

CVEs (Common Vulnerabilities and Exposures) often contain metadata that includes a vulnerability description, severity ratings, and information on affected software or versions. Building from the understanding of what CVE data can offer is, perhaps even more importantly, the need for understanding how CVEs are actually reported in practice. CVEs are typically reported through platforms like the MITRE CVE system, the NATIONAL VULNERABILITY DATABASE (NVD), and vendor-specific advisories. The process of CVE reporting is not standardized across these platforms, leading to inconsistent data formats, varying degrees of information granularity and quality, and discrepancies in reporting practices. Some vulnerabilities, for example, may first appear as bug reports on forums or through bug bounty programs, and these initial reports can follow different disclosure protocols, further impacting the quality and timing of the information. This variability in reporting practices makes handling CVE data challenging, as it requires navigating the inconsistencies in how CVEs are disclosed and reported. Additionally, there are often discrepancies in the level of detail provided, with some CVEs offering more comprehensive information than others. These challenges complicate the task of analyzing and integrating CVE data effectively. Thus, it is crucial to have an understanding of CVE and we conduct an empirical study into CVEs in practice. This will aid in analyzing how data from the CVEs can be processed and what sources can be reliably used for further research. Moreover, for the goal of obtaining a finer-grained vulnerability reporting, it is crucial to understand what information from CVEs can actually contribute towards this goal. As noted in the paper by PONTA ET AL. for PROJECT-KB [21], obtaining more precise vulnerability reporting requires finer-grained details of the vulnerability on a code-centric level. This thus involves identifying the vulnerable code through for example patches. To achieve this it is necessary to identify within the available CVE information if and more importantly how vulnerable code can be found. It is essential to establish what information fields within a CVE can contribute towards this goal.

To enrich CVE data through an automated process, such as associating specific code with a vulnerability, an investigation has been conducted into how CVE are reported in practice and how an automated process could leverage this practical information. The primary goal of this investigation was to determine what information contained within the CVEs can be utilized for identifying concrete code that should be linked to a specific CVE vulnerability. Given the wide range of sources for CVE data, it is important to establish a dataset we will use as a starting point. Ideally, all CVE information and sources would be considered and processed. However due the varying quality of information, both in terms of accuracy and usability, not all sources are equally valuable and some could be left out. The main consideration here is the trade-off between gaining additional actionable information and the generalization or effort needed to incorporate certain sources.

This chapter provides insights into how CVEs are reported and the types of information within them. From the CVE data it is identified what information fields can be utilized to find finer-grained code details. Furthermore, we present a small empirical study into these fields that could lead to the code, delving into some of the implications within this data. Based on the findings from our investigation, the datasets that are collected and used in this thesis are established.

4.1 CVEs in practice and identifying 'useful' information fields

There are many different ways how CVEs are first reported, for example as bugs or reports on websites and forums. These reports are subsequently collected, submitted, put together and managed in the CVE database from the CVE program [8]. They are also stored and further enriched by the NVD from NIST where their staff collects data points from 'the description, references and other data that can be publicly found' [3]. This then results in a typical NVD CVE format as seen in 4.1.

- CVE ID: A unique identifier assigned to each vulnerability entry. In the format of `CVE-{YEAR}-{IDENTIFIER}`. For example, `CVE-2024-123456`.
- Description: High level details of the vulnerability, including affected software, potential impact and hints about its causes.
- CVSS Score: severity ratings, Common Vulnerability Scoring System score, which quantifies the seriousness of the vulnerability based on factors like exploitability and impact.
- Disclosure: Dates of when the vulnerability was publicly disclosed or discovered and last updated. Also contained here is a pointer towards the original source of the disclosure.
- References: Links to external resources or documentation that provide more information about the vulnerability.
- Weakness Enumerations: Classifies the nature of the vulnerability e.g., buffer overflow, SQL injection. (Represented in NVD by CWEs)

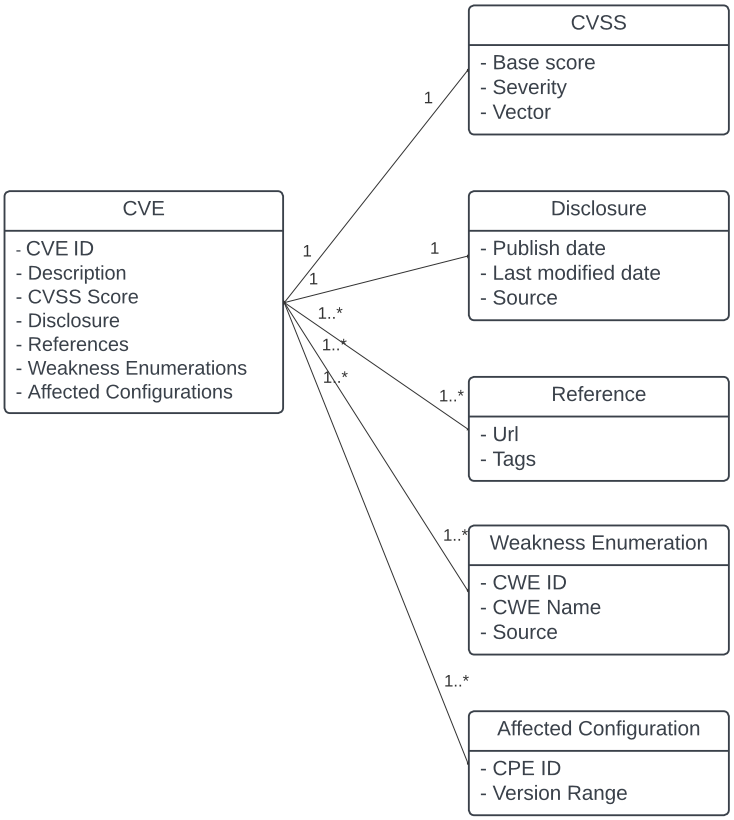

Figure 4.1: NVD CVE diagram

- Affected Configurations: Lists software, hardware, or systems affected by the vulnerability. Such as specific libraries and their vulnerable versions. (Represented in NVD by CPEs)

The NVD is the largest central place with verified information on the CVEs following the CVEs from MITRE. However, as mentioned previously there are also other websites/advisories/vendors that report CVEs such as GITHUB ADVISORIES [14], SPRING SECURITY ADVISORIES [12] and MICROSOFT SECURITY RESPONSE CENTER (MSRC) [7] among others. Some of these are actually CVE Numbering Authority (CNA) partners, which are authorized organization to assign CVE IDs and publish CVE Records [1]. As of 2024 there are over 400 CNA partners contributing to the wide variety of possible CVE data sources. Additionally, there are (manually curated) datasets that have been published, where either extra information regarding CVEs has been collected or the data within them has been more refined and verified. For example the PROJECT-KB dataset is a manually curated dataset, where specifically the patch links for a CVE have been identified and added as an extra

[1] https://www.cve.org/ResourcesSupport/AllResources/CNARules

field. Some vendors/advisories/datasets might provide extra information fields described below.

- Vendor Fix: Information about patches, updates, or mitigations provided by the vendor to address the vulnerability. (This is not a seperate field in NVD as it gets put inside the references, however some advisories such as Github Advisories and Spring Securities specifically offer this field)
- Acknowledgments: Credits individuals or organizations that reported or discovered the vulnerability.

From the different fields of information that are contained within a CVE it is identified, that through the references and in the case of some of the other datasets: the patch link field, patch information regarding the vulnerabilities can be found. Additionally, leveraging the vulnerable project versions (affected configurations) code changes can be discovered by comparing version releases. This leads to finding finer-grained information regarding vulnerable code. Mainly through the references and patch links field, source code level information can be found. As with these links code changes in the form of commit-, pull request- and issue links can be found. Next to this the pointers for specific (affected) packages and their versions, by NVD represented by Common Platform Enumeration (CPE) ids and in general by the affected package URL (PURL), can guide towards version release code locations.

CPEs are created by the NIST and provide a standardized format to identify software, hardware and operating systems. This allows the NIST to identify technology products across databases and reports. The format of a CPE is as follows:
`cpe:2.3:<part>:<vendor>:<product>:<version>:<update>:<edition>:<language>:<sw_edition>:<target_sw>:<target_hw>:<other>`
a typical example of this is `cpe:2.3:a:apache:log4j:2.14.1:*:*:*:*:*:*:*`. Important for CPEs to note is that via the `<part>` the type of technology is indicated with ”a” for applications, ”o” for operating systems and ”h” for hardware. Furthermore within the CPE data it is seen that the components after the `<version>` component with rare exception are represented by the `*` wildcard value. Mainly through the use of the `<vendor>:<product>:<version>` components, CPEs can be linked towards PURLs and thus in an indirect way, point towards the locations of for example the binary releases. A PURL follows the format of `scheme:type/namespace/name@version?qualifiers#subpath` [2] for example the PURL for the springframework is `pkg:maven/org.springframework/spring-framework/6.1.14`. The key difference of a PURL is the `<type>` part which indicates the ”type” or ”protocol” such as Maven, PyPi and NPM. CPEs/PURLs often have a range of the impacted versions associated. The use of these ranges implicitly enables the identification of the non vulnerable versions and their code locations.

Lastly the weakness enumeration, specifically the Common Weakness Enumeration (CWE) could hint towards vulnerable code patterns. CWE is a standardized list of software

[2] For precise details visit https://github.com/package-url/purl-spec

weaknesses maintained by MITRE. Each CWE represents a specific type of security flaw or software vulnerability pattern, allowing developers, researchers, and security professionals to identify, classify, and mitigate weaknesses in code. For example `CWE-89` indicates a vulnerability regarding SQL Injection. It would be possible to search for code corresponding with the vulnerability types and patterns within vulnerable packages. However the detection of these patterns falls mainly in the field of vulnerability discovery. In this project the focus is on identifying vulnerable code through change discovery, therefore the use of CWEs falls outside the scope of this project.

4.2 An investigation into the reporting of NVD CVE references

NVD is often used as a backbone for research into CVEs due to both the quality and quantity of the available data on CVEs. However the data format within NVD CVE data deviates from what is more commonly seen. For example the use of CPEs and CWEs is typical for NVD CVEs, another difference is that NVD adds annotation for references through *tags*. The NVD enriches references with various tags including but not limited to: ”Patch”, ”Exploit”, ”Vendor Advisory”, ”Mitigation”, ”Broken Link” tags. Given the variety of resources the references can point to, the use of these tags is an attempt from NVD to provide a clearer overview of the types of links within CVEs. However the paper by DONG ET AL. [9] also warns that CVE data and their annotation can be inconsistent. Therefore we conduct a small investigation into these tags, assessing if they could potentially be leveraged within the process of code change discovery. Another issue is that computers are limited in processing information as humans do. Given the wide and unknown variety of information that could potentially be presented through the links, our investigation also aims to determine the types of references that should be processed by our methods.

Methodology For the research in this paper the ”Patch” tag was of most interest, as references with this tag should point towards code patches. For the empirical study into the NVD CVE references some statistics on both the type of links and the ”Patch” tag have been collected. Within a collection of 257.468 CVE entries obtained from the NVD api it was identified that there are 33.576 CVEs which only contained CPEs pointing towards Hardware or where their Affected Configuration could not be found/processed. These CVE fall outside the scope of this project, and were thus left out in the further investigation. For the references in the remaining CVE the *base URL* was extracted. Here the base URL is represented by the domain of the URL and added is the type of URL if within the full URL one of the following keywords is found: ”commit”/”revision”, ”pull”, ”issue”, ”release” and ”GHSA” (Specific to distinguish GITHUB ADVISORIES links). The base URL for a GITHUB commit link is represented as `github.com/commit`.

Results All the remaining 223.892 CVEs contained a total of 848.590 references out of which 98.948 (11.66%) were patch-tagged references and 749.642 (88.34%) non patch-tagged references. This results in a total of 67.123 (29.98%) CVEs that have at least one ”Patch” tagged reference. In Table 4.1, we present the distribution of the `baseURL + "/" +`

Table 4.1: patch-tagged links

Repository	Count	Percentage
github.com/commit	12647	12,78
securityfocus.com	6313	6,38
portal.msrc.microsoft.com	4682	4,73
secunia.com	4173	4,22
oracle.com	6453	6,52
github.com/pull	3025	3,06
helpx.adobe.com	2446	2,47
github.com/issue	2297	2,32
bugzilla.redhat.com	2105	2,13
ibm.com	1767	1,79
openwall.com	1752	1,77
source.android.com	1596	1,61
plugins.trac.wordpress.org	1445	1,46
git.kernel.org	1390	1,40
msrc.microsoft.com	1244	1,26
github.com/GHSA	1129	1,14
huntr.dev	1124	1,14

Table 4.2: Non-patch-tagged links

Repository	Count
github.com/issue	8272
github.com/GHSA	4542
github.com/release	2951
chromereleases.googleblog.com/release	1982
lists.apache.org/issue	1468
lists.apache.org/commit	1364
code.google.com/issue	1328
googlechromereleases.blogspot.com/release	1313
github.com/commit	1267
git.kernel.org/commit	972
gitlab.com/issue	935
github.com/pull	835
code.google.com/issue	623
issues.rpath.com/issue	612
about.gitlab.com/release	533

`type` of the URLs in these 2 sets. In Table 4.2, we further investigate the existence of links towards version control platforms within the non patch-tagged links set.

Table 4.1, shows that GITHUB commits are the largest section within the ”Patch” tagged references. However, it also reveals that a significant portion of the ”Patch” tagged references do not point towards a version control platform. For instance the second most prevalent ”Patch” tagged baseURL is for `securityfocus.com/`, however this website does

not exist anymore. Furthermore, the references to these non version-control resources like MSRC and ORACLE mainly lead towards vulnerability reports only suitable for human reading. This highlights that even within the ”Patch” tagged references a wide variety of links are present. Emphasizing that there is no one-size-fits-all method to process all CVE data, indicating that specialized methods are often required. For this reason the main focus of the change discovery is on version control platform references. As for these types of references only slight specialization is needed within an overarching, generalizable approach to process the data.

In Table 4.2 presents that even within the 749642 non patch-tagged references, URLs to version control platforms can be found. We show that 9.11% of all GITHUB commit links are not ”Patch” tagged. For GITHUB pull request links even 21.63% is not ”Patch” tagged. These references might be related to the exploit or introduction of a vulnerability. However some of these references are missing the ”Patch” tag due to inaccurate annotation. In any case, these reference are found through the CVE data which implies that any code changes that can be discovered through these references should still be linked to the CVE.

Conclusion Via our empirical study we find similar to the findings of the empirical study in the paper by XU ET AL. [27] that GITHUB links form the largest section for the patch link references. Interestingly we observe that within the version control platform references a significant part of the references are of higher level code change aggregation types such as pull request and release URLs. It is shown that even within references that are grouped by a tag there is a high variability of types, furthermore between references of the same type there is variability in how they are annotated. Already within the NVD data inconsistencies are present, which illustrates one of the main challenges within the whole field of CVE data. CVE data is very diverse and may contain inconsistencies, which is made even more challenging by the wide variety of sources and different formats that have to be considered. The findings support the message of the DONG ET AL. paper [9] that CVE information can be inconsistent and even may be inaccurate, which should be kept in mind for further research. The tags used by the NVD for references provide a decent indication, however even in the absence of an explicit tag a reference may still fall within that category. It is concluded that no filtering based on tags should be applied, and that tagged and non tagged references should still be considered within the process of enrichment. Although GITHUB is the most prevalent version control platform in the references the decision was made to also include some of the other major version control platforms such as GITLAB and BITBUCKET, but also more generally the URLs with `git.` or `svn` within them. Lastly we also determined that the discovery of changed code should not be limited to only the commit/revision link types, pull request and issue references are also considered

This chapter has provided a brief insight into how CVEs are reported in practice. Additionally, it establishes that *the useful information fields within CVEs for identifying vulnerable code* are primarily the references and affected configurations, as these fields lend themselves well to discovering code changes.

Table 4.3: Sources used for CVE data

Data source	License	**Update frequency**
NVD	Public domain	2 hours
Github Advisory	Public domain	Daily
MSR 2019	Public Domain	n/a
MSR 2020	Public Domain	n/a
cvedb (by fabric8-analytics)	n/a	n/a
victims-cve-db	CC BY-SA 4.0	n/a
SAP project-kb	Public Domain	n/a

4.3 Data points used for the starting dataset

In this project, as a starting point for obtaining finer-grained vulnerability details an aggregation of CVE datasets has to be collected. This aggregation determines which CVEs are processed, enriched and evaluated to obtain finer-grained vulnerability reporting. With the different sources of CVE data it is essential to establish a set of sources from where the CVE data should be retrieved from. Given that many different sources provide CVE data such as the over 400 CNA partners, and the existence of many other datasets on CVEs, it is infeasible to collect everything. Therefore, a selection must be made, where various aspects have to be considered during the selection. These aspects include the usability of the data, the quality and also the added value that a source could provide. As a result, some CVE data sources are left out due to a shortcoming in any of those aspects.

CVE data source selection Table 4.3 provides an overview of all the data sources.

For this dataset, it was decided to use the NVD as the basis. NVD contains the most CVE entries as it is directly derived from the Mitre CVE dataset. Furthermore, NVD is already somewhat enriched by workers at NIST where the time difference between a CVE entering into Mitre and getting the CVE into NVD, is usually within a few weeks. Github Advisories is also chosen for similar reasons. Additionally since the data is already hosted on Github these entries often contain precise/extra links pointing inside the actual Github projects. Additionally there have been various datasets collected by other people that have enriched CVEs with more precise data such as vulnerability patch links, which are also collected for the starting dataset. Since these dataset are manually procured the amount of data they bring contribute might be limited, however they guarantee high quality data with precise, verified information about the CVEs.

It was decided not to use Spring Advisories as a data source. In general the information field ”Mitigation” inside the Spring advisories would be the main focus for finding additional information regarding patches. However in practice it was observed that this field is mainly intended for human consumption. Moreover, the actual mitigations that are suggested are mostly recommendations for upgrading to non-vulnerable package versions. We also chose not to use the Debian tracker as this primarily contains information con-

cerning vulnerabilities which impact projects that utilize programming languages other than Java.

Chapter 5

Automated Patch-Commit Recovery

With the goal of obtaining finer granularity reporting, it is essential to find more detailed information about a CVE and its vulnerability. Existing *package level* metadata granularity reporting tools often produce false positives, as they raise warnings as soon as there is an inclusion of a vulnerable package as a dependency. For a more precise approach to vulnerability reporting, details beyond package vulnerability alone must be identified. Drawing inspiration from the ideas described by ECLIPSE STEADY [22], our approach aims to find information on the vulnerability of a CVE at the code level. In an ideal world the specific lines of code that are vulnerable would be identified. However, locating this code is not trivial, and furthermore obtaining information about vulnerable code on the most precise granularity is not always feasible. Thus a methodology for identifying vulnerable code needs to be formulated and evaluated. This goal leads us to explore RQ_2: Can we find commit links that direct us towards code patches for a vulnerability? More specifically we break this down into 2 aspects, patch link recovery and patch link accuracy. We thus answer $RQ_{2.1}$: Can we recover commit links for CVEs? and $RQ_{2.2}$: What is the precision of finding the correct commit links to a fix for a CVE?

Having identified the ability to locate actual code, that should be associated with a vulnerability through the use of *patch links*, one can proceed with collecting the CVE data and work towards identifying these relevant *patch links*. Given the variety of data sources and different formats, an approach to collect these different kind of data points and unify them into a workable format is necessary to be established. Each source requires a tailored approach for data retrieval and parsing of the data to ensure further utilization. Importantly within the aggregation process of the data there should retain the available information, minimizing data loss. Drawing inspiration from the FASTEN PROJECT VULNERABILITY PRODUCER [23] the methodology for the aggregation of the data and the unified format has been developed. This unified format, referred to in this paper as a `Vulnerability Object`, is designed to accommodate the storage of almost all the types of data fields seen within the different sources. The `Vulnerability Object` contains next to the CVE data collected, additional fields to accommodate the data enrichments which are added in the later stages of the pipeline within the *patch finding* and extraction steps. The used format retains all the fields from the NVD CVE format (see figure 4.1) with some slight naming differences. The most notable difference however, is the inclusions of two additional fields: the set of

`PatchLinks` field and the set of `Patches` field. The `PatchLinks` field comprises a set of identified commit or revision links pointing towards the patch code. The `Patches` field is for storing the actual code extracted from the `PatchLinks` links.

Once the data has been collected and parsed it into a workable format the so called *patch finding* (actually code change discovery) can begin. *Patch finding* is a crucial step in the process of mapping CVEs to their associated vulnerable code. Ideally, the CVE data would directly include the fine grained details on the actual precise vulnerable code/method(s) already associated with a CVE, however in practice it is seen that obtaining this information requires some additional intermediate steps to be taken. Most sources provide CVE data which does not contain the actual changed code, but instead have references that point towards commits/revisions in version control platforms/systems such as GITHUB and GIT where the changed code can be found. This changed code can then be used to identify the vulnerable code, as logically, when a code change introduces or patches a vulnerability it would follow that the changed code contains or at least reveals the vulnerable code causing the vulnerability. *Patch links* are in the context of this paper thus actually referring to commit links that point towards changed code that should be associated with a CVE vulnerability. Therefore for finding the actual code it is essential to identify these commit links.

This chapter provides a high level overview of the first part of the full project pipeline covering the collection, parsing and patch finding processes for CVEs. Subsequently, the first part of the pipeline and mainly the *patch finding* can be evaluated and RQ_2 is addressed. The evaluation proceeds in two stages, first evaluating the ability of finding back commit links and secondly if the correct commit links are found.

5.1 Limitations of current work

In Chapter 4 we identified that through both the references/patch links as well as the PURLs, code of packages associated with a CVE can be found. These information fields offer a basis for locating the actual vulnerable code of these packages. Prior research has made some strides towards identifying code or commit links for CVEs. Most of this work focuses primarily on identifying the code that patches a vulnerability, the search processes involve for example scanning repositories for commits that reference the related CVE. However, scanning repositories for commits, results in the need for manual verification of the outcomes. Manual verification is required because the results of these works may miss commits and critically may capture commits that are entirely unrelated.

For example ECLIPSE STEADY uses a prospector that scans whole GITHUB projects, with the help of a machine learning algorithm, trying to find candidate commit links. ECLIPSE STEADY requires manual work then to pick out the relevant commits out of these candidate links. Furthermore, some solutions employ the use of various heuristic while scanning commits to find the candidate commit links, for example scanning if commit messages contain keywords such as ”patch”, ”secur”, ”vuln” or the specific CVE ID. However, relying on these heuristics introduces risks, as their effectiveness depends on the quality of commit messages. Although, fix commits for vulnerabilities often do mention CVEs or relevant

keywords about a vulnerability this is not always the case. It could happen that vital commits are overlooked simply because their commit message might fall outside of the expected patterns. On the other hand, if heuristic that are too broad are employed, e.g. only scan for key words such as ”patch” or ”fix”, there is a high chance that commits that are totally irrelevant to a CVE will be linked.

Our approach To address some of the limitations of existing work, this project focuses on performing this *patch finding* solely through the available CVE data itself. By employing only the use of CVE data we are mimicking more closely the behavior of an human investigator and we keep this project better scalable. For the approach in this paper we opted to perform the *patch finding* using mainly the references, particularly searching for references towards version control platforms. Where more targeted heuristics can be applied, for instance the heuristic used for processing GITHUB issue references is to scan for commit references within them. Since our approach is based on the CVE data references and does not simply derive from larger aggregations, all commit references we identify should be relevant to the CVE. We want our approach to be fully automated to ensure that our is also more sustainable for the growing number of CVEs. Our approach is somewhat similar to the approach in the paper by XU ET EL. [27]. However, in contrast to the approach of XU ET AL. we do not build a reference network nor do we filter out candidate commit our approach finds.

Another method for finding vulnerable code, that we introduce, utilizes the information on the metadata of vulnerable PURLs and implicitly the non-vulnerable PURLs. By examining the differences between the vulnerable and non-vulnerable package versions one can come to an (over) estimation of the vulnerable code. This method will almost certainly capture irrelevant code changes, but we obtain the guarantee that we capture any changed code related to a vulnerability.

5.2 Methodology for the CVE data collection and patch finding

Starting from a CVE to get more fine-grained details about the vulnerable code and the actual impact the vulnerability has 2 things have to be known. First the actual vulnerable package has to be known, in our approach represented by PURLs, to get the vulnerable metadata. For these packages to get more fine grained reporting, on a finer level within these package the vulnerability has to be identified. Thus the actual vulnerable code has to be identified, e.g. in the form of method signatures or classes. As mentioned this can be achieved with the code extracted from the patches identified during the patch finding which uses the CVE data.

This section discusses the methodology of the first part of pipeline for getting from CVE data towards the actual vulnerable code mapped towards a CVE. From the starting dataset mentioned in Chapter 4 the data gets processed and patch finding is done.

Data collection First the starting dataset of CVEs as described in Chapter 4 is collected.

The data from the different data sources is collected through downloading/cloning or

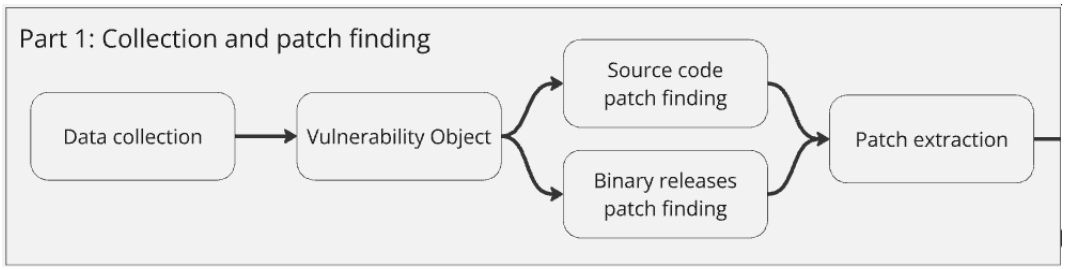

Figure 5.1: Part 1: Collection and patch finding

requesting the data, with the methods being dependent on the way the sources made the CVE data available.

Parsing of the data into a `Vulnerability Object` The collected data is parsed into a `Vulnerability Object` so that there is a uniform format the pipeline can work with. If a CVE is present within multiple sources the different `Vulnerability Objects` with the same CVE ID are merged without data loss within this process. During the parsing there is one important difference compared to the NVD format. The vulnerable packages in NVD CVEs are represented by Common Platform Enumeration (CPE) ids. CPEs are mainly used by the NIST and provide a standardized format to identify software, hardware and operating systems. However CPE don't necessarily point towards for example the distributor or code location of the package which is what we are mainly interested in for getting finer grained details, this information however is included in the PURLs. Additionally with a PURL it is possible to distinguish more directly the package manager and thus also the programming language. Therefore it was opted to save all vulnerable packages and versions as PURLs instead of CPEs. In this process the use of the SCANOSS/PURL2CPE project[1] was employed for getting the mappings of CPE to their PURL. Another slight difference is that we elected to translate the versions ranges to all versions within them, so we store all the individual version entries as their own PURLs.

Source code Patch finding From the Vulnerability objects source code patch finding is started. Patch finding looks to find code patches URLs by going through the references of the CVE. The references are matched with regular expressions to identify the version control platform references. Furthermore these references are matched for their respective types of links corresponding to the same types as the GITHUB commit, pull request and issue link types, i.e. merge requests are the same type as pull requests. Each type has its own targeted heuristics, such as taking the closing commit reference, to find further references towards the commit links.

Binary Releases Change finding Another change discovery method is the binary releases change finding. Using the PURLs and the vulnerable ranges the vulnerable - non

[1] https://github.com/scanoss/purl2cpe

vulnerable version pairs are identified per CVE. With this project being Java focused these pairs consist of MAVEN PURLs. The choice was made in the binary release change finding to focus on identifying vulnerable - fixed versions, as in practice it is seen for CVE vulnerabilities that it is way less consistent and reliable to find non-vulnerable - vulnerability introducing version pairs. For example it is not necessarily the case that vulnerabilities are introduced in a single version upgrade, as sometimes vulnerabilities are a result of a compound of changes within development. Meaning that the actual introduction of vulnerability is likely to be way less clear cut between versions. Also considered is that the vulnerability ranges seen in the data for example indicate 'all versions until a certain version' thus the occurrence of CVEs where there are no non-vulnerable - vulnerable introducing version pairs. Furthermore the availability of older versions for packages is less guaranteed. As older versions for example get archived or are no longer hosted even. Using the PURLs and vulnerable version ranges, vulnerable - patched version pairs are collected which consist of a last in the range vulnerable version and a following non-vulnerable (patched) version.

Patch Extraction In patch extraction the changed code is retrieved from the patch commit links by requesting the data from the commit links via for example the GITHUB API. This *Patch* information is added to the `Vulnerability Object`.

These steps result in an intermediate dataset (from here on referenced as the `ParsedVulns` dataset) where all code patches associated with a CVE are collected and linked to the CVE.

5.3 The reliability of commit links

Within the ParsedVulns dataset the patch links have been identified through the patch finding. It is essential to find the right patch links, as this leads to collecting and associating the right code to a CVE. However it can occur that CVEs don't contain commit references themselves, or on the other hand the identified commit links are not actually the correct patch link. For achieving finer-grained vulnerability reporting we need to be able to find encapsulating code mappings. Which depend on if we can even find commits for the code. Moreover it is important that within the patch finding at least the correct commit links are actually found. Therefore, it is important to evaluate both aspects: recovering commit links and identifying the correct ones within them. A crucial part for these evaluations is establishing a ground truth on which these abilities can be evaluated on. For the ground truth on recovering commits the NVD can be used. Furthermore given that PROJECT-KB is a manually curated dataset, we can use it as a ground truth for evaluation the discovery of the correct commit links.

The identified patch links are evaluated to measure the success of the patch-finding process. This evaluation was carried out in two parts. In the first evaluation, a ground truth was established using CVE patch link information from the NVD to assess how effectively the patch-finding method recovers commit links. In the second evaluation, the collected patch links were compared against the ECLIPSE STEADY (PROJECT-KB) dataset, providing insight into how accurately the patch-finding process identifies the correct commit links.

5.3.1 Can we recover commit links?

Since NVD is seen as a basis for most datasets it is the perfect candidate to evaluate the recovery of commit links. By examining cases where there would be no direct information about patch links from NVD itself the ability of recovering patch links can be assessed. The NVD uses the "Patch" tag specifically for references that should point towards a patch. From this it can reasonably be assumed that all the "Patch" tagged commit links are the actual patch commit links or that the code within them, at the very least is highly connected to the actual vulnerability. There it is important to find back as many of these references as one can via the *patch finding*. Hence an evaluation for finding back the patch links inside NVD is done.

Methodology For the evaluation initially to establish a ground truth all "Patch" tagged references received from NVD were taken and used for a ground truth to check if the commit finding methodology works. This ground truth was extended with the non "Patch" tagged commit link, considering the results of the investigation that has been done for RQ_1 where it was found that the tag annotation might be inconsistent. As it was found that 90% of all GITHUB commit links in NVD are tagged with the "Patch" tag. And 78% of all GITHUB Pull requests are tagged with the "Patch" tag. Therefore a brief investigation into the non "Patch" tagged commit links was performed. By manual checking 200 randomly selected GITHUB commit links out of all 1267 non "Patch" tagged GITHUB commit links, it was found that 186 (93%) out of these 200 actually are patch commit links or at least highly related to the patch. Thus the choice was made to perform the evaluation with a ground truth that contains all commit links instead of only the "Patch" tagged ones. The full list of links manually inspected out of all non "Patch" tagged GITHUB commit links can be found at doi: 10.5281/zenodo.14219216.

To evaluate Part 1 of the pipeline on the ability of its patch finding, an evaluation parsing run has been performed. The evaluation testing the ability to find commit links if these would not have been present in a CVE. The first part of the pipeline has thus been run for all CVEs from 1999-2023 for a total of 219489 CVEs. To be able to evaluate our solution, a modified parsing was done for evaluation where all commit links and all "Patch" tagged pull request references from NVD were ignored for the test set. Simulating the situation as if they were not present in NVD. It was opted to also remove these patch tagged pull requests as in principal these are an 1 step away pointer towards the commit links. The patch finding then proceeded as normal with the modified parsed CVEs, these modified patch finding done CVEs were then evaluated for how well inside the patch finding the patch links from the NVD ground truth were found back.

Starting with the Evaluation set of 19535 CVEs from NVD that contain a "Patch" tagged link or commit. Intersecting with the set of CVEs obtained from the evaluation parsing run and filtering for commit or revision links we get a Comparion set of 14850 CVEs that contain a patch link. These CVEs have a total of 17978 patch link that we are trying to recover.

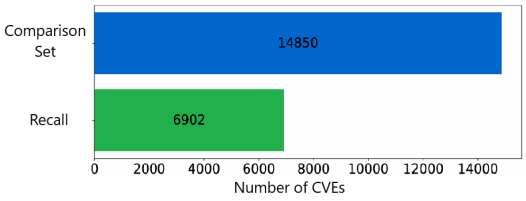

Figure 5.2: NVD CVE recall: 46.5%

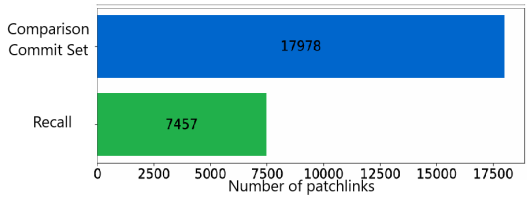

Figure 5.3: NVD patchLinks recall: 41.5%

Figure 5.4: NVD ground truth recall evaluation results

Results Thus there is a NVD Evaluation set of CVEs which contains all CVE from NVD that contain commit links and/or "Patch" tagged pull links. These links from here on called the NVD ground truth patchLinks. And there is a Parsed Set which consists of the CVEs that were parsed and patch found without the NVD ground truth patch links.

The intersection of the CVE entries of this Parsed set with CVE entries of the NVD Evaluation set results in a Comparison CVE set. From this the NVD ground truth patchLinks have been filtered for only commits/revision links into a Comparison commit set. As after the parsing and patch finding only commit links are saved into the patchLinks field (For pull requests the merge commit or a list of all the commits within the pull request are stored).

In Figure 5.2 and Figure 5.3 the recall of finding back a commit from the NVD ground truth commit links from the Comparison CVE set within the Parsed commit set can be seen. With Figure 5.2 displaying the amount of CVE entries and recall for how many CVE entries at least one NVD ground truth patchLink has been found back. In Figure 5.3 the recall for all the patchLinks inside the Comparison commit set is seen.

We see that in the end for 46.5% of the CVE entries within the comparison set we have found back at least one commit patch link. For the commit patch links themself we see a recall of 41.5%. This shows that it is possible to find back some commit links through the patch finding even with the absence of those in NVD. However we also see that both results are less than 50% successful showing that our approach for patch link recovery misses the majority of patchLinks.

Furthermore an investigation towards the precision has been done shown in Figure 5.5.

In Figure 5.5 we see the precision score of the commit links of the Parsed set. With a precision of 62.7% and recovering an over estimation of other commit link amounting to 37.3%. Furthermore we achieve a recall 41.5% giving us a F1-score of 49.9%.

Both results for the recall and the precision indicate that the current implementation does rely quite a bit on the property that there is an accurate starting link within the data, from which the search can be started. It should be considered for the results that removal of references towards patch links causes some of them to become impossible to recover without manual effort. The absence of the patch links in our results can be attributed to a combination of possible shortcomings in our approach and patch links becoming unrecoverable. The results do show however that we can still recover a significant portion of patch links through our approach. Answering $RQ_{2.1}$ we have shown that we indeed can recover commit links for CVEs.

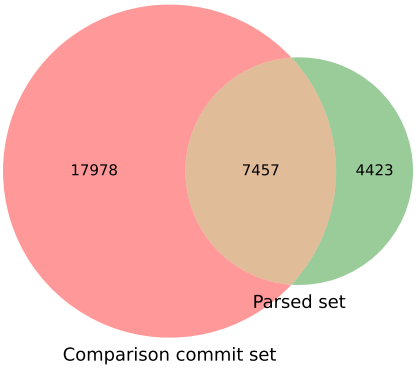

Figure 5.5: patchLinks recovery precision of the Parsed CVE patchLink set: 62.7%

EvaluationCVE	PLFoundCVE	TotalPLs	FoundPLs	additionalCLinks
1297	1195	1824	1824	1224

Figure 5.6: Eclipse steady evaluation

5.3.2 Can we find the right commit links?

Comparing against the ground truth established from NVD mainly investigates the capabilities of the pipeline to find patch links through information that is not the patch links themselves without knowing them already directly from NVD. With the results seen in Figure 5.4 we see that for some CVE we can still in that situation can find commit links. However more importantly to know is if within these commit links actually the right commit links can be found. Furthermore in the NVD ground truth evaluation any direct patch link information was ignored and thus this evaluation does not give an full overview of the actual patch finding capabilities of the patch finding. Since PROJECT-KB is a manually curated dataset with information on the correct commit links for a CVE vulnerability it can be used to evaluate if the patch finding actually finds the right commit links. Thus another evaluation against the manually curated PROJECT-KB dataset is done. Which thus also shows the performance of our fully automated collection approach vs a manually curated set.

Interestingly running an evaluation against the PROJECT-KB dataset where the PROJECT-KB dataset was included in the parsing showed that the PROJECT-KB dataset itself was not entirely consistent. All PROJECT-KB CVEs were compared to the resulting CVEs of the pipeline without any filtering. It was evaluated how many CVEs had a patch link found back for them (`PLFoundCVE`), how many patchlinks were found (`FoundPLs`, this is expected to be 100% given that the PROJECT-KB is also included as a source) and how many other over estimated patchlinks were found (`additionalCLinks`). The result of this investigation is shown in Table 5.6

The main point of interest to take from Table 5.6 is that for not every CVE a patch link was found back, however 100% of all patchLinks were found. This happened because 101 CVE entries from the PROJECT-KB dataset do not actually containing any information re-

garding patch links. E.g. CVE-2020-24750 only contains artifacts information. Interestingly this thus shows that even within a largely manually curated dataset such as PROJECT-KB the data format itself is not 100% consistent.

Methodology For the evaluation against ECLIPSE STEADY, the full dataset of PROJECT-KB is seen as ground truth. The parsing and patch finding pipeline is run with the exclusion of the parsing of the PROJECT-KB data. Our project pipeline retrieves CVE data from the different sources like NVD and the GITHUB ADVISORIES. However the CVE entries in PROJECT-KB were published/last updated anywhere in the range of september 30th 2020 - july 24th 2023 with the bulk being released in 2020/2021. Since this time the corresponding CVE entries in NVD and GITHUB ADVISORIES have been open to updates of their information. To ensure a fair comparison, any discrepancies in starting/available data at the time were addressed by aligning the datasets. For the comparison with ECLIPSE STEADY, we adjusted our solution's CVE dataset to reflect its contents at the time of PROJECT-KB'S release. This has been done with the help of the NVD CVE CHANGE HISTORY API for the entries received from NVD. For GITHUB ADVISORIES this has been done by accessing the change history of the json statement files themselves.

For each entry in PROJECT-KB it is evaluated whether the patch finding was able to find back the same commits that are in the PROJECT-KB ground truth. First the CVE set after only parsing the collected CVE data is evaluated against the PROJECT-KB ground truth. Any patch links that are found here were thus already available from the data collected itself directly. Some patch links are for example already available from the NVD data. Secondly the CVE set after patch finding has been evaluated against the Project-kb ground truth. By looking at the difference we obtain an indication as to how our patch finding approach adds value by finding additional correct patch links. Some pre-processing and filtering has been done for the PROJECT-KB dataset and our evaluation set. Considering the the whole of the PROJECT-KB dataset as an evaluation set, the CVEs that were not parsed in our approach were filtered out. For example CVE-2018-7574 which is a rejected CVE and was thus not parsed. Furthermore as seen in Figure 5.6 some entries from PROJECT-KB themselves actually had no commit link data. These have thus also been removed, this filtering resulted in a *Valid Set* with which the evaluation has been done. An evaluation for the recall of CVEs where at least a patch link has been found back after both of the 2 stages is done. And an evaluation on the patch links themselves after the 2 stages is done.

Results In Figure 5.9 the results are shown for the recall of CVEs where we found back at least one patch link for them. Figure 5.12 shows the results on the evaluation of the patch links showing both the recall and the precision. Figure 5.7 and Figure 5.10 show the respective results after parsing but before the patch finding. Figure 5.8 and Figure 5.11 show the results after the patch finding has been performed.

As shown in Figure 5.9 for the pre-patch finding we achieve a precision of 69.5% and a recall of 30.5% we obtain an F1 score of 42.4% For the post patch finding we achieve a precision of 53.4% and a recall of 47.0% and we obtain an F1 score of 49.9% We show that in the patch finding of patch links we achieve an increase of 16.5% for the recall of the

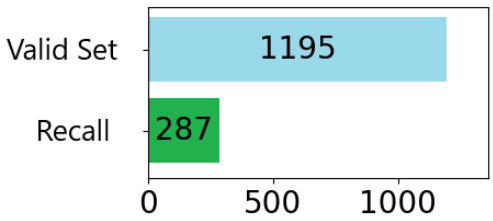

Figure 5.7: pre-patch finding CVE recall Eclipse steady: %

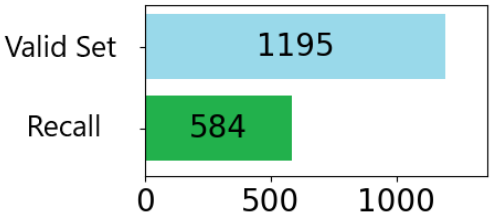

Figure 5.8: Patch found CVE recall Eclipse steady: %

Figure 5.9: CVEs with a patch link recall

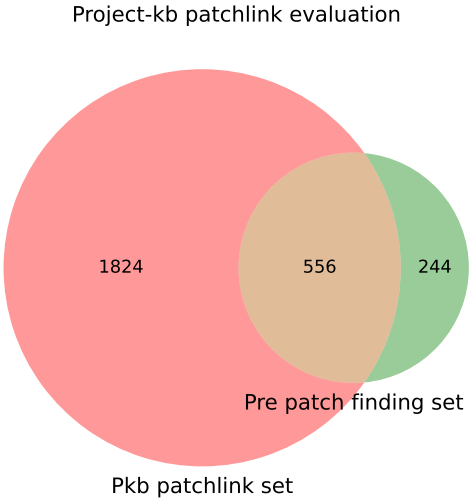

Figure 5.10: Patch links evaluation of the parsed pkb excluded CVEs set with recall: 30.5%

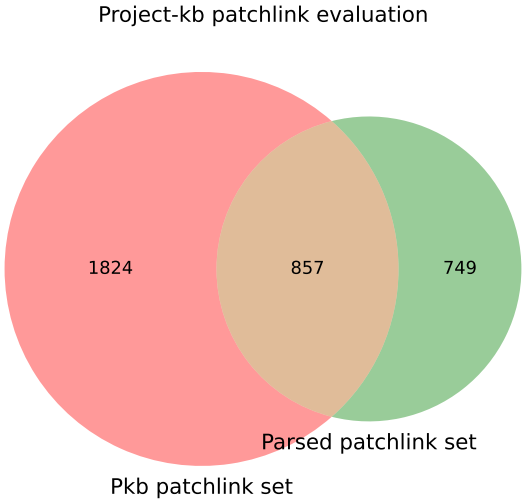

Figure 5.11: Patch links evaluation of the mapped pkb excluded CVEs set with recall: 47.0%

Figure 5.12: Correct patch link discovery evaluation

PROJECT-KB patch links. Interesting to note is the increase of finding at least one patch link for an additional 297 cves, given that in total an additional 301 patch links are found. Indicating that most of these extra patch links were found for CVEs that not yet had any patch links found and that the patch finding by finding these patch links thus mainly improved the coverage as opposed to the 4 patch links that would only mainly impact precision. However also here still less than 50% of the patch links is found back showing that the manual approach is still very necessary for finding the right information.

Since our solution only goes off information that can be directly found from the CVE information finding the correct commit highly depends on the references that are actually being associated with a CVE. For example in cases where a CVE is published but fixed in the future, finding the right commit link highly depends on if the CVE gets correctly updated according to the fix being published. Furthermore our solution might under perform on the recall metric when compared to solutions that actually scan and try to find right commit links of whole GITHUB projects. But our approach does show that even with less computation effort, being more scalable and maintainable, an increase of patch link finding can be achieved. We have answered $RQ_{2.2}$ by showing that we are able to obtain an F1-score of 49.9% for finding the correct patch links. However more importantly to point out is that our approach meaningfully was able to improve on the current directly available data. Thus our approach is shown to be a good candidate for bridging the CVE reporting and performing manual data enrichment.

5.4 Conclusion for finding commit links to a fix for a CVE

The results in Section 5.3 show that our approach for parsing and subsequent patch finding is able to find commit links, even without their immediate presence within the NVD data. Furthermore we have shown that the patch finding actually contributes a significant amount towards finding back correct commit links. From both of the results it can be concluded that the parsing and patch finding of our automated approach is able to meaningfully contribute towards finding patch code that should be associated with a CVE vulnerability. However considering that the results we achieve are around 50% for the F1-score, our approach is not a replacement for doing manual work. Despite this the results are promising, with room for improvement, showing that this approach can aid in the process of patch finding. Even though the enrichment still requires manual efforts to ensure the best results, our approach can support this immensely by providing some of the first steps for the enrichment. Coincidentally both F1 scores for the NVD ground truth evaluation and for the ECLIPSE STEADY evaluation on the patch links end up the same as 49.9%. Although both evaluations and their results are expected to be related to each other and similar results are expected, from the difference in precision and recall of the results it can be concluded that they are not actually directly related to each other.

In this chapter we answered RQ_2 by demonstrating that, finding commit links pointing us towards code patches for vulnerabilities, can indeed be accomplished. We have shown that for both aspects of identifying commit links our approach can offer valuable contributions. Given that this approach is able to recover commit links and actually con-

tribute towards finding the correct commit links it is suggested that efforts should be made for incorporating the concepts of our approach within the current processes for enriching/updating CVEs. The implementation of our automated approach can be found on https://github.com/DanDanBro/fg-cve-aggregator.

Chapter 6

How does code-mapping granularity impact vulnerability detection?

To achieve finer-grained, code-centric vulnerability reporting, it is essential to investigate the impact of vulnerabilities. This is accomplished by addressing two research questions related to the exposure and reachability of vulnerabilities: RQ_3 How do commit, pull request, and release version patch mappings of CVE information affect the exposure and reachability results of a vulnerability? RQ_4 How do method and file-level granularity mappings influence the exposure and reachability results of a vulnerability?

Working towards the goal of finer-grained vulnerability reporting, we obtained the parsedVuln dataset, as described in Chapter 5. At this stage, the code associated with a CVE is known. However, for finer-grained reporting, it is crucial to determine whether this code is actually exposed or reachable by a package. To do this, we need mappings of the vulnerable code to the corresponding methods and classes/files within the vulnerable packages.

Extracting exact lines of code for each package from the patch links is impractical. A more feasible approach is to map the vulnerable code to the methods and/or classes/files affected by the patch. This mapping enables analysis of the call graph to assess reachability and exposure, allowing for static analysis of packages and facilitating finer-grained vulnerability reporting.

The actual impact of vulnerable code on a package can be assessed through *exposure*, which determines whether the vulnerability is reachable from public or protected methods or classes. If no exposure is detected, it may indicate that a package is not vulnerable, and, when used as a dependency, it could eliminate warnings from a normal package-level reporter. However, it is important to consider that the absence of exposure could be due to factors like code refactoring, which may cause false negatives where methods are no longer detected.

Another aspect of vulnerability impact is reachability — how the vulnerable code is accessed by other projects, providing insight into the vulnerability's effect on package users. For example, if vulnerable methods have limited exposure but are found to be frequently reached by other projects, it suggests that the vulnerability resides in critical code within the package.

This project aims to improve the precision of vulnerability reporting, ensuring that all warnings are actionable. However, we recognize that the mappings of vulnerable code to CVEs are still estimates, which implies varying guarantees of recall and precision. Striving for higher precision could result in a loss of recall. This trade-off must be examined, and different approaches should be evaluated for their trade-offs between precision and recall. From this, the idea emerged to investigate different levels of aggregation for the code mappings, as well as the appropriate level of granularity for vulnerability reporting.

This chapter discusses the second part of the project pipeline, where the parsedVuln dataset from the first part is used to map the vulnerable methods/files/packages. Using these mappings, the impact of vulnerabilities is evaluated through static analysis of the program to determine the exposure and reachability of vulnerable code. This results in the final `MappedVulns` dataset, which supports call graph generation, call detection, and vulnerability usage/exposure detection. Additionally, this chapter addresses RQ_3 and RQ_4, exploring how mappings at different levels of granularity affect vulnerability reporting and their impact.

6.1 Aggregation Layers of Change Discovery

Improving precision through the use of fine-grained information, obtained from patch commit links, may lead to a loss of recall regarding vulnerabilities. The challenge with mapping code from commits is that they may not exclusively contain the changes relevant to the vulnerable code, or they might only include part of the patch for a given vulnerability. As a result, using the patch commit data often provides an estimation of the vulnerable code rather than an exact mapping. The primary concern with relying solely on patch commit code is the potential for false negatives, where some vulnerabilities may not be detected. This is emphasized in Section 5.3.2 where our approach demonstrated that is it not always successful in identifying the correct commit.

Furthermore it was shown in 5.3.1 that the Parsed Set contains 37.2% of (over estimated) commit links outside of the ground truth and for 5.3.2 this even is 46.6%. This led to the idea of obtaining a more conservative *overestimation* of vulnerability. The most extreme form of overestimation is already seen in dependency checkers that use package-level warning systems. In these systems, if the code within a package—or, more specifically, for Java packages, the combination of groupId, artifactId, and version (GAV)—is vulnerable, the GAV as a dependency automatically raises a vulnerability flag to ensure no false negatives. However, as discussed earlier, this approach results in very low precision because it overestimates the vulnerability by flagging all non-vulnerable code within the package as reachable. This concept of *overestimating* vulnerable code leads to new levels of CVE information that should be considered for mapping code. These levels include the package metadata level and the commit information level, with commits being more precise but risking lower recall of the actual vulnerability. Between these levels, one could also examine the pull request (PR) level, which aggregates several commits and may introduce additional information or unrelated noise, and the binary releases information level, which captures all relevant changes describing a fix, but also contains significant noise. Each of these levels involves trade-offs in terms of precision, recall, and the guarantees regarding the correctness

of code associations. In the following section, we will discuss these different levels of information gathering and evaluate their respective benefits and challenges in finding vulnerable code.

CVEs usually refer to vulnerable packages through package names and versions, yet, these pointers could be resolved on multiple levels to find the associated code. a commit is the most fine-grained information available, zooming out would give you a PR, which might already aggregate several commits (which might include additional information about a fix, but also introduce unrelated noise), and ultimately, the release level, which definitely contains all relevant changes that describe a fix, but also a lot of noise. In this section, we will discuss for the different levels of information gathering towards vulnerable code. Each level has its own trade offs in terms of precision and recall but also on the guarantees regarding the correctness of the code association.

Patch commit links information The finest granularity, the commit level where only the changed code within the commits is marked vulnerable. Here all the patch links extracted code is used to form the vulnerability code estimation. This level should provide the most precise results given as the estimation is purely based of the associated commits. With commits in common practice being focused on specific tasks for changing the code.

Pull requests information The pull request (PR) level uses all the commits within the same pull request as the patch links collected to estimate the vulnerable code. This approach builds on the patch links identified within the ParsedVulns set. For each patch link, the goal is to identify and retrieve all other commits that are part of the same pull request, if applicable. The idea behind using the pull request level is to overestimate the possible vulnerable code, increasing the likelihood of associating the vulnerability with related code. This improves the chances of flagging the vulnerability but results in lower precision regarding the exact vulnerable method(s). Pull requests often introduce changes in code related to fixing an issue, meaning commits within the same pull request are typically connected. Therefore, capturing all commits in a pull request has a higher likelihood of capturing the complete fix.

Additionally, vulnerability fixes are sometimes spread across multiple commits, so using pull requests helps ensure that all relevant changes are captured. Thus, the expectation is that using the pull request information level offers improved recall at the cost of reduced precision.

A potential downside of the pull request level is the low availability of data. Since not all commits are part of a pull request, this approach may not be applicable for many CVEs, limiting its overall usefulness.

Binary releases A higher level of overestimation can be achieved at the release/version level. In package-level detection, warnings are typically issued based on the package and its version (GAV). However, by specifically analyzing the code changes between a vulnerable and a non-vulnerable version (preferably consecutive versions), one can gain insights into the actual changes that address the vulnerability. This

release-level information, like package-level detection, guarantees the absence of false negatives. A vulnerability would have to be caused by a change between the two versions, and capturing all changes ensures that the vulnerability is identified. At the release/version level, the overestimation of vulnerable code is expected to be significant. However, not all code within a package is necessarily changed between versions, so this approach has the potential to increase detection precision while maintaining a very high recall.

Additionally, code change discovery via (binary) releases/versions can be facilitated using the Affected Configurations (PURLs) field. This is particularly useful for CVEs that lack patch links, providing a solid alternative for identifying changes.

Package GAV level The highest CVE information level is the package version level, where a vulnerable PURL indicates that a particular GAV is affected. At this level, all code associated with the vulnerable GAV is considered vulnerable. Consequently, any usage of the package would be flagged for the vulnerability. This approach mirrors what is commonly seen in dependency checkers, where the inclusion of a vulnerable dependency results in all code within that dependency being flagged as vulnerable. While this level guarantees perfect prevention of false negatives within a GAV, it also leads to a significant reduction in precision.

The four levels differ in sizes of vulnerable code estimation. Where commit data is a subset of the pull request data, which in turn would be a subset of a binary release patch. It is therefore interesting to understand how they affect the tracability of vulnerabilities in a code base.

6.2 Associating CVEs with code on different granularities

For the exposure and the reach ability of course ideally only the usage of vulnerable methods should be flagged. However arguments can be made for not even allowing files/classes that have a vulnerability inside them to be used. As already using a vulnerable class indicates a higher potential risk of eventually using the vulnerability later in the development. Furthermore using a coarser granularity would mean you let less pass by the checker which should thus give a higher degree of confidence of avoiding false negatives.

Another angle for mapping code would be the granularity level. As code specifically is mapped to methods to obtain precise results. However in programming languages such as Java, methods are part of classes/types. Classes/types are the middle level between packages and methods. Mapping CVEs to types/classes is already suspected to improve the precision with regards to package level. The main motivation for flagging classes/types would be to enable the option of over estimating additional relevant vulnerable code, beyond what can be determined from the CVE information. Furthermore, it could be argued that one should not only avoid using vulnerable methods but should also refrain from using any vulnerable classes. Using a vulnerable class, even without directly calling a vulnerable method, increases the likelihood that future development could inadvertently make use of the vulnerability. In the same vein this could be argued for the whole package level of

course. However with the package level as mentioned the precision is worse, mapping towards the class granularity level could however provide a middle ground for improving precision but having a coarser grained reporting than method level granularity. Thus an investigation into the exposure and reach ability of vulnerable classes is warranted. Where files/classes that have any vulnerable code are marked whole as vulnerable.

6.2.1 Granularity levels

We distinguish three different granularity levels, the method level, the class level and the package level. Each of these levels are marked vulnerable based on a subpart being identified as vulnerable. Moving up from the method level, larger aggregations of code are marked as vulnerable according to each level.

Method level The finest granularity level, the method call level, directly examines the method calls within a project and flags only the usage of vulnerable methods. This level provides the highest precision for detecting both the presence and location of vulnerabilities, apart from knowing the exact lines of vulnerable code. By performing a method-level search within call graphs, it is possible to achieve the most accurate results.

Class level The middle level of granularity focuses on the types/classes used in projects. At this level, all types/classes that encapsulate vulnerable methods are marked as vulnerable, and any usage of a vulnerable type/class triggers a vulnerability flag. This approach increases coverage compared to the commit level while maintaining better precision than the package level. An additional advantage of using class-level information is its ability to capture vulnerabilities that are not tied to specific methods. By monitoring file changes within class configurations, it is possible to detect static code changes that may impact the entire class. While configuration vulnerabilities are outside the scope of this project, the class level is well-suited for capturing such static code changes. An alternative would be to scan code line by line to detect specific usages, but this approach would require significantly more processing. Therefore, the class-level granularity strikes a balance by estimating vulnerabilities efficiently.

Vulnerable-package level At the coarsest level, which mirrors many current CVE detection systems, is the package level. In this approach, if any vulnerable code is identified within a package, the entire GAV (groupId, artifactId, version) is marked as vulnerable, and all methods and classes within the package are flagged as vulnerable. A distinction is made between two related concepts: the package GAV CVE information level and the vulnerable-package granularity level. The package GAV CVE information level relies on CVE data that indicates a specific GAV is vulnerable. At this level, the assumption is that all code within the package is vulnerable, and therefore, any usage of the GAV should be flagged as vulnerable. On the other hand, the vulnerable-package granularity level is based on the identification of vulnerable code (methods or classes) within the package itself. If such vulnerable code is found, the entire GAV is then marked as vulnerable, reflecting the granular analysis of the package's code.

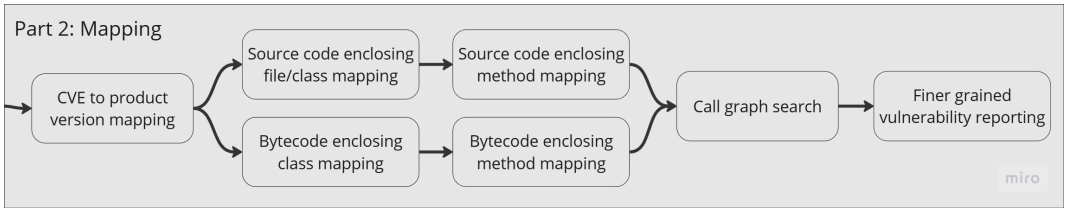

Figure 6.1: Part 2: Mapping

6.3 Methodology for mapping the encapsulating method/types

In the `ParsedVulns` dataset, we have access to the locations of the changed code. To analyze the vulnerability exposure and reachability, it's essential to map this changed code to specific methods and classes. To achieve this mapping, two different approaches are used, each corresponding to different CVE information levels. The first approach relies on the source code extracted from the patch commit links. This method allows us to identify methods and classes directly within the context of the source code changes. The result is a set of methods, each associated with key attributes such as class name or filename, access modifiers, return type, method name, parameter types, and any exceptions. In Java, a method is uniquely identified by the combination of the class, return type, method name, and parameter types. While access modifiers and exceptions are not strictly necessary for identifying unique methods, they are also recorded as they can contribute to refining the method signature and increasing precision. The second approach uses binary releases and works with bytecode. This method involves analyzing compiled bytecode, which enables us to map the changed code to methods and classes in a similar way to the patch commit links approach, but with a focus on the compiled output rather than source code. For class identification, a broader approach is taken: any file that has changed, whether due to modifications in methods or static fields, is recorded. Since patch diffs typically provide filenames rather than class names, filenames are used for this mapping. While it is common in Java for filenames to match the names of the classes they contain, using filenames could lead to mismatches if there are multiple top-level classes within a single file.

This *vulnerable* code that is mapped to the CVE has to be mapped to within the actual code of the vulnerable packages. As based on this mapping of vulnerable code within the packages the finer-grained vulnerability reports can be generated. Figure 6.1 shows the steps within the second part of the pipeline to obtain a finer grained vulnerability report based on the `ParsedVulns` dataset presented in Chapter 5.

CVE to product version mapping CVE to product mapping builts on the work done within the data parsing of part 1, where the affected product and its versions are identified through the purls. In this step here the binary releases (jars) are collected via the Purls for later use. The Purls are filtered on containing ”maven” and are then fed into a maven resolver for getting the jars.

Source code enclosing methods and files/classes mapping Enclosing file mapping takes

the patches from patch extraction and from the source code of these patches maps the CVE to the affected methods and files/classes. Within the mapping of the code there are some important factors to consider for the different levels of mapping code. For the commit, pull and PROJECT-KB commit information levels all the code within the diffs from the commits is considered. Within this the method signatures of encapsulating methods are determined. However static changes within code such as modifications within class fields are not necessarily bound within encapsulating methods. This code cannot be mapped towards methods, however to preserve the knowledge about these code changes they are stored within $< clinit >$ placeholder methods. This thus means that for files where only static information is modified the knowledge is preserved that the file was modified. This ensures that within the mapping for classes these files and thus classes are still marked vulnerable. Within the diffs extracted from the patch links for the file/class granularity mapping all file names within the diffs are stored. The methods signatures are identified and extracted with the use of regular expressions for Java method signatures.

Bytecode enclosing methods and classes mapping Bytecode analysis between the vulnerable - patched releases pairs is done obtaining the changed methods and classes between the versions. Discovering changed code between versions is possible in multiple ways. Our method utilizes binary releases which are discovered through the use of the PURLs. However another possible solution would be the use of the github releases. As it is possible on GITHUB to obtain the source code diff between release tags. These release tags could be identified with the help of the version parts of the PURLs. It was decided to use the binary releases mainly because of the capabilities of the PURLs. PURLs themselves indicate, through the `<type>` part, the protocol for the package from this potentially both the programming language as well as the distributor protocol can be recognized. Via the package manager distributors obtaining the binary release is fairly straightforward. Whereas for GITHUB projects some effort has to be made to find the releases. For example the owner of the project might not correspond with the `<namespace>` of the PURL. The use of a *release* reference is also not very viable as the empirical study in Chapter 4 shows that a minority of CVE have such a reference.

For the release bytecode analysis ASM[1] was used. For all the vulnerable - patched releases pairs the bytecode of the 2 versions in a pair were compared. From here all the methods that were either removed or modified were extracted to flag them as vulnerable. It was opted to not actually store and flag any methods that were added in the *patched* release as although this could be part of a patch the method itself would never be found back anyway in the vulnerable release. Thus forming the enclosing methods and classes estimation of the vulnerability.

Call graph search Using the binary releases from the CVE to product mapping the call graphs can be generated. For the call graph generation the WALA FRAMEWORK[2]

[1] https://asm.ow2.io/

[2] https://github.com/wala/WALA

is used. Using the in built `AllApplicationEntrypoints` class of WALA a `0-CFA callgraph` is generated (reflection options set to `ONE_FLOW_TO_CASTS_NO_METHOD_INVOKE/NONE` for performance reasons). It was opted to use a `0-CFA call graph` as it allows a conservative approach for dealing with overrides and extendings for methods/classes. Using these call graphs combined with the method and classes identified in the enclosing methods and classes mapping the call graphs can be searched for the reachability and exposure of these mappings.

Finer grained vulnerability reporting Based on the findings within the call graph search we are able to generate a finer grained vulnerability reporting.

The dataset that results from these mappings steps is the referenced within this paper by the `MappedVulns` dataset. With both the final dataset and the call graph search formulated, both the set and the vulnerability reporting can be evaluated.

6.4 Vulnerability (over) estimation comparison to Eclipse steady

As PROJECT-KB is a manually curated dataset comparison against eclipse steady is done. The idea behind this comparison is to evaluate the over estimation of vulnerable methods. So how much more often does the full pipeline with the latest update flag a project vulnerable in comparison to eclipse steady. As comparing the amount of methods that are associated with a CVE should already give an indication/explanation towards the differences later for the exposure and reachability evaluations the exposure and reachability of the vulnerability in the packages.

From the programmers point of view, directly comparing the actual method sets that are associated with a CVE. Thus the F1-score for the datasets of vulnerable methods from eclipse steady and our dataset has been calculated.

For the methods sets evaluation the sizes of the eclipse steady methods vs the `MappedVulns` methods sets (This thus also includes all the eclipse steady methods, having 100% recall) are compared. All eclipse steady method sets are compared to their counterparts in the `MappedVulns` set. We see the average difference of the `MappedVulns` method sets sizes with respect to the eclipse steady method sets sizes. From this evaluating 551 sets we get an average size difference of 32.07%, giving a precision of 75.71% and a F1-score of 0.8616. However it was noticed that there was an extreme outlier case within these method sets. The method set for `CVE-2020-13921` showed an increased size of 7560% which is over 30 times the average increase. Furthermore this increase is 9 times as large as the second largest increase. Therefore it was decided to evaluate the dataset again with the exclusion of `CVE-2020-13921`. This resulted in a evaluation of 550 method sets getting an average size difference of 18.38% giving a precision of 84.51% and a F1-score of 0.91.

As earlier presented, 1224 extra over estimated patch links have been found, however as is also shown in 6.2 this results in 2000 extra over estimated methods that would be flagged vulnerable. This gives us a recall of 0.991, precision of 0.858, and a F1-score of 0.919. From this we show that although we found a substantial amount of patch links outside of

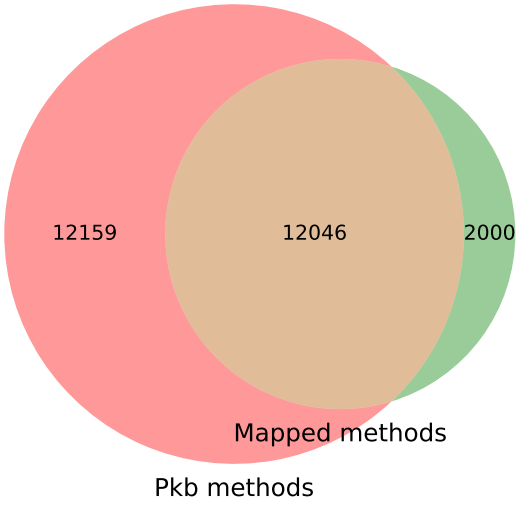

Figure 6.2: Venn diagram for the method in Project-kb and mapped method

the ground truth, these extra patch links result in a relatively smaller impact on the actual code (over) estimations we obtain. This over estimation is even less noticed in the viewpoint of the user where in over 95% of the cases the reachability analysis for `MappedVulns` and eclipse steady report the same results.

6.5 How do (different) CVE impact code mappings?

The CVE to code mapping levels can be expressed in 2 dimensions. The first dimension is the CVE information level, described in section 6.1 where this paper investigates the level of patch information that is collected. The second dimension is the granularity level from section 6.2 which inspects the level on which projects are using vulnerable packages. Investigating the granularity level in terms of method, file/class and package level, the vulnerable package is thus marked vulnerable on.

It is important to evaluate what the actual impact of our work/ the mappings of CVE to code is. As stated in introduction the aim is to reduce the amount of false positive warnings or in other words improve the precision of vulnerability reports. For the impact of vulnerabilities on packages it is important to consider 2 sides of the CVE mappings to code.

First is the view of vulnerability exposure which for Java code means what public and protected code has reachability to the vulnerability. This would thus indicate how much of a dependency is actually vulnerable or not. The other side to consider for the impact of mappings is the practical impact on actual usage of a package. As a vulnerability could for example be small only affecting a small set of methods/classes in a package. However if the vulnerability is inside important code, inside features/the core of what people mainly use from that dependency, the impact of the vulnerability would be different than if for example a whole feature would be vulnerable but one which is not often actually used. For example take the WALA framework used in this project, a vulnerability within the callgraph with for

example the intermediate representations (IR) would have a different (larger) impact than a vulnerability that is inside code to turn a CGNode into json. Thus it is important to look at the impact for users.

Since this paper focuses on Java packages the CVEs were filtered for indicating a vulnerable maven PURL. This resulted in a set of 2176 CVE which have a maven vulnerable purl. For both evaluations the CVEs evaluated were randomly picked out of this set.

6.5.1 Vulnerability exposure

To quantify the extent to which vulnerabilities impact their affected package, we want to quantify how much of them is exposed through transitive method calls. This will provide a better understanding of the likelihood that dependents are actually affected by the vulnerability. To measure this, we conduct a analysis with the MappedVulns dataset, in this analysis we establish the exposure of vulnerable code through public and protected methods. Where we show that our approaches using more fine grained code change mappings can achieve better precision than dependency checkers.

Methodology For this evaluation, we compare the exposure of public and protected methods across different levels of granularity in the vulnerability mapping. To do this, for each CVE, all vulnerable packages associated with that CVE are collected. The vulnerable packages are identified through the vulnerable PURLs (Package URLs) present in the CVE data. Since this project focuses on Java code, the Maven PURLs are resolved via Maven Central, and if available, the corresponding JAR files of the vulnerable PURLs are retrieved. Once the JAR files are obtained, the call graphs for these projects are generated. The next step is to search for vulnerable methods associated with the CVE across these call graphs. The methods flagged as vulnerable are determined based on the different CVE information levels. After identifying the vulnerable methods within a package, a backward search is conducted to find all the callers of these vulnerable methods. This backward search should result in identifying all public and protected methods within the package that could potentially call a vulnerable method, either directly or indirectly. The focus of this analysis is to determine the exposure of the vulnerability within the package. Public methods are exposed to external access, so they are naturally of concern. Protected methods are also included in the analysis since they may be called through inheritance in subclassed code, making them potentially accessible and vulnerable in a different context. This approach allows for a comprehensive assessment of the vulnerability's exposure in the package, both from directly accessible methods and those that might be accessible through inheritance or other means.

For the exposure evaluation for every CVE entry for every (maven) vulnerable purl in that CVE the jar of the release was retrieved and a WALA framework callgraph was generated. Using this callgraph a search for matching identified vulnerable methods/classes in the mapping to actual methods/classes in the callgraph is done. From any found vulnerable method/class in the callgraph a backwards search through all callers of the vulnerable method/class is done. Along this backwards search the set of all public/protected meth-

	Granularity Level %				
CVE Information	JAR File	Class	Method	**#Cases**	**#CVEs**
GAV + Project kb commit links	64.9	13.7	6.4	7365	127
GAV + commit link	69.0	15.1	6.4	7465	130
GAV + commit's pull requests	75.3	38.8	28.4	124	2
GAV + binary releases	95.0	87.9	56.9	7234	116

Figure 6.3: Exposure of Vulnerabilities

ods/classes that could (indirectly) call a vulnerable method is extracted. This thus gives the set of exposed vulnerable methods.

As mentioned for the class granularity level all methods within a class with vulnerable code are mapped and flagged as vulnerable. For the exposure, if another method from a non vulnerable class would call a method from a vulnerable class the class of that method stays marked as non vulnerable. As marking that class as vulnerable would give an avalanche effect which would give an over inflated picture of the vulnerability exposure.

Due to the nature of jars and thus inclusion of code that might not actually be part of a package but of one of its dependencies some filtering was done for the exposed methods. This was done by checking parts of the class paths for corresponding with part of the purl.

Results In Table 6.3 the results for the exposure of vulnerable methods/classes are presented. Here we present the exposure of vulnerable public/protected methods as a percentage of all public/protected methods. The results are obtained by taking a double average over the exposure of the GAVs and CVEs. First within a single CVE the exposure of all GAVs within that CVE is averaged. Then our results are obtained by averaging over the CVE percentages. This is expressed as number of found public or protected methods that can reach a vulnerable method as a ratio of the total number of public or protected methods in the *vulnerable* packages.

The package granularity level is the ratio of packages where vulnerable code has been found. For the type and method levels the double average of the methods is taken. So over the ratio of methods within a packages the average is taken. This ensures that each package is as important as other packages. Given that some packages have more or less public or protected methods than others.

To note here is also that for some versions no vulnerable exposed methods or classes were actually found. This is mainly due to no actual vulnerable method being found at all inside those versions call graphs. This on the one hand could indicate as also seen in Eclipse steady that some versions in the vulnerable versions range are not actually vulnerable. Or on the other hand this could be due to other factors such as the possibility of code refactoring between versions. Where thus some code and classes have been identified within the mapping that are not present in that package version. Thus leading that some vulnerabilities might actually be present in the code, however due to refactoring the vulnerable methods found in the patches are not reflected in these 'older non refactored' different versions.

For example `CVE-2018-1257` within NVD points towards Spring framework version. Evaluating the spring-core versions no exposure is found of vulnerable methods. However when evaluating the spring-messaging versions the vulnerable methods are found and exposure is found.

Furthermore when comparing the different levels of CVE information it is noticed that the cases where commit's pull request information can be used is very low. Using commit's pull request information might provide a broader over estimation the low number of cases it can actually be used makes this a nice extra conservative approach. Although very little extra computation is needed for the commit's pull level it would not be very viable to rely on this solely for getting more conservative reporting.

6.5.2 Vulnerability reachability

To better understand the impact of vulnerabilities on dependent projects, we quantify how many vulnerabilities are exposed through transitive method calls. This helps us assess the actual exposure of the vulnerability within the dependent projects, providing a clearer picture of the vulnerability's real-world impact. The study utilizes the `MappedVulns` dataset, which contains the mappings of vulnerable code to methods and classes. In this evaluation, we focus on reachability, specifically investigating how dependent projects interact with and use vulnerable packages. While exposure alone gives insight into how accessible a vulnerability is within a package, it does not fully capture the impact on dependents. For instance, even if a class granularity level shows more exposure, it might not necessarily affect the practical risk if a dependent has already been flagged vulnerable due to a method granularity-level flag. Therefore, this evaluation takes reachability into account—determining if and how dependent projects call the vulnerable code. This step evaluates the practical implications of CVEs, i.e., whether the vulnerable code is actually being used by projects that rely on the package, and how that usage affects the dependents' vulnerability status. The results of this experiment are influenced by the dataset of dependents chosen for evaluation. To ensure a diverse representation, we combined both random and targeted selection methods for the dependents. This approach ensures a broad evaluation of different vulnerabilities and how they propagate through a variety of dependent packages, ultimately providing a more accurate reflection of the real-world impact of the vulnerabilities.

Methodology For the evaluation of the mapping a comparison between the different levels of granularity is done. For this evaluation for each CVE various projects that use a vulnerable package associated with this CVE have been collected. These projects have their call graphs generated and searched for the usage/reachability of the vulnerable marked method/file/class with regards to the different levels of granularity. The selection of packages and related cve and vulnerable package to be evaluated has been done by picking a random pkgVulnerablePurl of the CVE Vulnerability, and then collecting a random dependent of this purl found in the list of the Maven central repository[3].

[3]https://central.sonatype.com/

Results In 6.4 the results of vulnerability reach-ability within various projects is shown. Expressed as a percentage of projects being flagged in the experiment by this tool for each of the different levels of mapping. On the x-axis mapping of the granularity levels are compared against each other for the number of projects represented by the #cases. This is done for each of the CVE information levels mapping displayed on the y-axis.

	Granularity Level %		
CVE Information	Class	Method	**#Cases**
group artifact version (GAV)	1	1	360
GAV + Project kb commit links	90.0	66.1	340
GAV + commit link	90.0	66.7	360
GAV + pull request link	91.8	89.8	49
GAV + binary releases	72.3	70.6	300

Figure 6.4: Reachability of Vulnerabilities (From client A to library L)

The package version row and also the package granularity level column are to show that the actual package within the evaluations are actually used by the to be evaluated purl. This shows the baseline and shows the same results as other package warning tools.

The commit information level shows that based on the methods flagging a significant portion of warnings could be reduced. However with the knowledge that the patch finding is quite lacking a portion of this reduction could be due to poor patch finding. Another interesting point however is the large difference between the method and the type information level. Where the type level indicates that in a high percentage of cases the classes usage is actually flagged meaning also at least some methods within those classes were flagged which would point to having at least found methods for a large number of cases in the right direction.

The pull information level shows a high flagging ratio which as thus far seen in the data is likely due to a large number of the cases where the package was already flagged on the commit information level also being the CVEs where pull evaluation could be done for. A correlation between being flagged on commit level and the commits of those CVEs being in pull requests would have to be researched. But that is out of the scope of this project.

Finally in the releases information level both the type and method level show the same reachability analysis results. However of interest the rates are lower than the commit information level type level. And also for a reduced number of CVE actually release methods could be collected. With the way random purls were collected and the whole CPE - PURL conversion and perhaps imprecise ranges some release information could have been lost. Furthermore some releases versions are so outdated they are no longer hosted in maven. Thus performing release method extraction would require ad hoc solutions searching through archives to find a vulnerable - patched release pair.

The results compared between the different CVE information levels are out of line with what would be expected. This is likely due some bias in the selection of the evaluated

dependent packages, this has less effect on the results between the class and method level. The results do provide insights into the trade-offs between the class and method granularity levels. We show that as expected the class level raises warnings more conservatively than the method level.

6.6 Conclusion

We have answered RQ_3 and RQ_4 in this chapter. We show that the method granularity and commit information levels achieve the most precise results for the vulnerability reporting. However these levels have weaker guarantees on finding the vulnerability, resulting in them being prone to losing recall. Moving to higher levels of aggregation, the pull request information and class granularity levels have shown to be more conservative alternatives. Both the pull request information level and the class granularity level show a trade-off against the most precise levels. The precision is lower but this is traded off for better guarantees on higher recall of a vulnerability. Lastly the release information level shows that the precision can be improved without notably compromising on recall. Overall it is shown that each level can positively impact the vulnerability reporting by obtaining more precise results than dependency checkers.

Chapter 7

Conclusion

This paper has presented an empirical study into CVEs in practice, where we determined that references and the affected configurations could be used for code change discovery. An investigation into the ability of our approach to find patch links has been conducted. Chapter 5 showed that we are able to, first off, recover a substantial portion of patch links. Furthermore, we showed that we are able to provide significant contribution towards finding the correct patch links. Lastly, the impact of different code mapping levels on the vulnerability reporting has been outlined, where we show that different levels can provide alternatives, trading off precision and recall. More importantly, we demonstrated that each approach for the different levels can provide a positive impact on the precision versus dependency checkers.

7.1 Discussion

In this paper an empirical study into the CVEs in practice and the identification of useful information fields for getting finer grained details on a vulnerability was done. We formulated and evaluated an automated approach for enriching CVEs with code change information. Lastly we evaluated the impact of different CVE change discovery aggregations layers and different code mapping granularity levels on the vulnerability reporting. Mainly focusing on the impact on the precision and the recall of these different methods.

Other useful information fields In our empirical study the references and PURLs have been identified as the main fields through which patches in the form of commit links and changed code could be found. However, an idea that could be worth investigating is the possibility of combining our approach, which uses the code obtained from the references and purls, with efforts for vulnerability detection through CWEs. This could prove to further enhance the precision or on the other hand provide better guarantees for finding vulnerabilities within OSS packages. The precision could potentially be improved by for example utilizing the vulnerability patterns from the associated CWEs to scan through the vulnerable code that has been identified. This could be used to filter out unrelated code, or it could find that the vulnerability pattern is not detected within the code, indicating that a more conservative

approach might be more appropriate. The use of CWE code mappings is however expected to require a significant bigger effort to allow combining with the patch code approach. Future research should be conducted to investigate if the results of such a combination are significant enough to justify the extra effort.

A complimentary automated approach In Chapter 5, we presented our automated approach with an F1-score of 49.9% for finding commit links for NVD CVEs. Furthermore, we have shown an increase of 16.5% in recall for finding the correct commit links for ECLIPSE STEADY. With the results presented in Chapter 5, we concluded that the automated approach of this project can actually help in finding the patch link. Although the results of the patch finding are still far from perfect, they show that meaningful enrichment of CVEs can be made with our automated approach. Our approach trades off the guarantees on correctness for less effort needed compared to some of the other solutions, such as ECLIPSE STEADY. For ECLIPSE STEADY, first of all, an entire model had to be trained; furthermore, entire GITHUB repositories are crawled by the prospector to find candidate commits. Since this project opted for an automated approach, our approach is more suitable and scalable for integration within the processes of updating CVEs. Additionally, our approach only performs the search for patches based on the CVE information itself. This would imply that the patch links found by the patch finding that are not actually in the ground truth should still be related to the CVE. Our patch finding is, however, still somewhat limited. Given the many different websites and resources to which the references could point, it is not feasible to develop an approach that would be able to perfectly handle all kinds of references. Therefore, our approach mainly tries to find references to version control platforms. However, future research could look into improving the reference scanning. In many of the other solutions for finding code, they use heuristics, and in some cases, even machine learning models are used for finding/filtering commits within the GITHUB repositories of the vulnerable packages. The concepts behind these methods could actually prove to be very useful to apply to, for example, forum references. Future work for this could thus look into ways to add heuristics or even use machine learning models to allow the processing of some of the other types of references encountered within CVE data. For example, if some heuristics or an NLP model could be used/developed to allow for scanning through a forum post that reports a CVE and scanning for references towards code. By applying the concepts of the heuristics/ machine learning models for the references, the patch finding could be improved by having it mimic more closely what manual work would be for the references.

Using (binary) releases One benefit of relying on releases for change-based code discovery, rather than patch links, is that almost all CVEs contain information on the vulnerable versions. Whereas, only a subset of CVEs have a patch link that could be associated with the CVE. Furthermore, we obtain a very high guarantee that the patch was captured within the vulnerable-fixed pairs, so the number of false negatives should be very low at this level.

However, a key concern when using mapped code from any of the approaches across the different levels is the risk of false negatives occurring due to code refactoring across different releases. If vulnerable code is within a method that was refactored between older versions (e.g., method signature change or class refactoring), the mapped code may fail to

identify the vulnerable methods within these older versions. To mitigate this threat, future work should look into the possibility of tracking refactoring of the vulnerable code.

Finer-grained levels increase precision Chapter 6 demonstrated that our approach yielded results very similar to those when only using the patch link information from PROJECT-KB. We showed that the commit method level provided the most precise results. Additionally, we demonstrated that with each increase in either the CVE information level or the granularity level, there was a corresponding increase in vulnerability warnings and a decrease in precision. From the results shown in chapter 6, several conclusions can be drawn. First, from the results shown in 6.2 and 6.3, we observed that although the vulnerable method set sizes for our approach include an additional 16.4% of methods that fall outside the ECLIPSE STEADY ground truth, these additional methods cause less than a 6% increase in the exposure findings. This indicates that the overestimation caused by having more commit links contributes less substantially to an overestimation of vulnerable methods. These results demonstrate that, despite a significant overestimation of vulnerable code, our vulnerability reporting aligns closely—within 6%—with the results from ECLIPSE STEADY. Given that ECLIPSE STEADY relies on a manually curated, high-quality dataset, the similarity in results suggests that our approach can also achieve high-quality outcomes.

Even when using more conservative CVE information level approaches, we can improve precision over dependency checkers. We demonstrated that using binary release code changes for mapping vulnerable code, as expected, raises more (imprecise) warnings than the precise commit information level. However, we also showed that the binary release information level still offers a considerable increase in precision compared to the package level. Given these characteristics of the binary release information level, the approach with this level would be the prime candidate as the next step after dependency checkers in the fields of vulnerability detection and prevention.

Regarding the different granularity levels, we have shown that the method granularity level provides the most precise results. Using the method granularity level would positively impact vulnerability reporting by providing more precise results. The class granularity produces more warnings than the method level. However, while the method granularity is considered the most precise level, the class granularity offers a compromise between precision and coverage. Even though the class level sacrifices some precision compared to the method level, it still shows considerable improvements in vulnerability reporting precision when compared to package dependency checkers. The class granularity approach thus offers a more conservative alternative to approaches that focus on achieving the most precise reporting at the method level. Furthermore, the class level could provide a more precise approach for future vulnerability prevention. As class granularity does not necessarily indicate that a vulnerability is actively used, it can still provide valuable insight. In cases where a warning would be raised at the class granularity but not at the method level, it indicates that code usage is near a vulnerability. With this knowledge, a better-informed decision can be made for further development involving vulnerable classes, proceeding with extra caution if at all. For the reasons mentioned above, we recommend that class granularity be incorporated or, at the very least, considered in any future work related to finer-grained vulnerability reporting.

Overall, we have shown that by examining more fine-grained details of the CVE and the associated vulnerable code, better precision in warning generation can be achieved compared to dependency checkers. Most importantly, we showed that it is possible to significantly reduce false positive warnings. For each level, further investigation into improving and refining the approach is worthwhile.

Finding non-vulnerable package versions in the affected configurations data Lastly, we presented findings showing that for some indicated vulnerable packages, we actually found no exposure of a vulnerability. As mentioned earlier, this could be due to code refactoring between versions. However, this could also indicate that certain GAVs are not actually vulnerable. This supports the findings of DONG ET AL. [9], who suggested that CVEs sometimes overstate the ranges of vulnerable versions. Therefore, more fine-grained exposure reporting could be used to filter out non-vulnerable versions, resulting in a more precise identification of vulnerable versions. A similar observation is made within the PROJECT-KB dataset, where some versions are marked as non-vulnerable based on analysis.

False negatives As discussed throughout the project, our approach provides an estimation of the vulnerability. At best, it maps the precise vulnerable code, but it is also possible that we either overestimate or underestimate the vulnerability. Since we have demonstrated that there is still room for improvement in precision, overestimation of a vulnerability is not a significant concern. However, underestimation poses a major risk, which motivated the introduction of different levels of granularity, though even these levels do not guarantee 100% accuracy. Additionally, there is currently no reliable method to validate the impact of false negatives other than manually inspecting entire packages, which is not a feasible solution. Therefore, the results should be interpreted with the understanding that there is no 100% guarantee of perfect recall at this stage. This limitation could affect the actionability of the results. A warning from our approach is always actionable, as all warnings correspond to code related to the vulnerability. However, when no warning is raised—particularly with regard to reachability—the results can only be trusted to a certain degree, with confidence based on whether our approach at least identified some exposures within the package.

For improved actionability, we recommend that future work integrate dynamic analysis alongside static analysis. One potential idea for handling warnings would be to incorporate an agent within the dynamic analysis process that triggers an error if a vulnerable method is called, ensuring that malicious code does not execute.

Reproducibility The CVE initiative is an ongoing process, with new CVEs being added or updated daily. Additionally, the methods for retrieving CVE data may evolve over time. For example, during the development of this project, NIST released version 4.0 of the CVSS. Due to the dynamic nature of CVE data, collecting it independently and processing it through our pipeline could yield different results. To improve reproducibility, we have made a snapshot of our starting dataset available at DOI: 10.5281/zenodo.14219216.

7.2 Threats to Validity

Due to the inconsistent nature of the data and the fact that our automated approach is not infallible, there are potential threats to validity that must be considered.

Java focus This project focused exclusively on investigating Java packages. Java was chosen due to its widespread use and the vast number of open-source packages available for it. Additionally, Java is a well-defined and structured programming language, which facilitates the clear extraction and separation of different levels in the analysis. However, restricting the study to Java introduces a threat to validity in terms of the generalizability of our approach. By concentrating solely on Java, more specifically, Maven packages, the findings are inherently tied to vulnerabilities present in Java-based systems. That said, the approach could be extended to other programming languages. For instance, ECLIPSE STEADY already supports Python, and interestingly, the class/file granularity might be more suitable for Python, as its code is not necessarily confined to defined methods. In principle, the concepts used in this project can be adapted to other programming languages, provided that some form of encapsulation of vulnerable code exists and its usage can be tracked (e.g., through a call graph). Despite the focus on Java in this thesis, we believe the trade-offs presented are still valid, given the language's prominence and the diverse set of CVEs and packages analyzed.

Unreliable version information in CVEs Previous research has shown that the versions associated with CVEs may not always accurately reflect the actual vulnerable versions [9]. In our own study, we encountered inconsistencies in the CPE and PURL mappings, as well as references to packages no longer hosted on the Maven Central repository. We found that the CPE-to-PURL conversion is not flawless; for instance, the CPE for `Spring-framework:*` incorrectly points to the PURL for Spring-core. Additionally, the PURL-to-Maven JAR conversion can be imprecise, leading to version mismatches where older, unmaintained versions of projects are no longer available on Maven.

Currently, there is no definitive method—aside from manual inspection—to verify the accuracy of the version ranges specified in all CVEs. Nonetheless, for the purposes of this project, we treated the indicated versions as accurate. However, during the exposure evaluation, we observed instances where no actual vulnerable code could be traced back, raising concerns about version accuracy.

Another challenge we encountered involved the use of JAR files, which may include code not strictly belonging to the package in question. For instance, JAR files might contain code from other dependencies. To address this, we performed filtering to exclude non-relevant code. Although this filtering is not perfect, it is not typically an issue if too little code is filtered out. The greater concern arises when too much code is filtered out, potentially causing warnings to be missed. One example is the case of `pkg:maven/org.apache.flex.blazeds/flex-messagingcore@4.7.2`, where no methods were found in the exposure due to class names being unidentifiable with standard PURL parts. This was an outlier case in the project, and therefore, we assumed that if some methods

could still be identified after filtering, the filtering process and the resulting findings were valid.

7.3 Summary

Current work using CVEs for detecting the use of vulnerable dependencies is often very imprecise raising many false positive warnings. Various challenges are faced in achieving more precise finer-grained vulnerability reporting. The first hurdle is the variability of CVE data which makes working with CVEs complex and often introduces the need for manual work. This means that work for finer-grained vulnerability information is often not easily scalable and maintaining the work is difficult. This creates the need for establishing a robust approach for processing and enriching this data. This (automated) approach should bridge the gap between the reporting and complex often manual enrichment of CVE data. However with an automated approach potential risk of recall loss arises, thus different methods have to be considered which balance a trade off between precision and recall guarantees.

This paper has addressed these challenges by showing that an automated approach for collecting patches can meaningfully contribute in the processes of enriching CVE data. Furthermore the concept of collecting different level of CVE code information and the idea of mapping vulnerabilities to different granularity levels has been introduced. The methods for these different levels have been evaluated, showing that through each of the different CVE code information collection levels a greater precision than dependency checkers can be achieved. Additionally it is demonstrated that mapping vulnerabilities towards the class-level granularity provides an viable solution for compromising between the most precise method granularity and the imprecise marking of a full package as vulnerable. For each level across the two dimensions of CVE code information and granularity, the trade off between coverage and precision is shown. Where it is demonstrated that each approach still provides improved precision over vulnerability reporting that relies solely on dependency metadata.

Bibliography

[1] Lingfeng Bao, Xin Xia, Ahmed E Hassan, and Xiaohu Yang. V-szz: automatic identification of version ranges affected by cve vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2352–2364, 2022.

[2] Guru Bhandari, Amara Naseer, and Leon Moonen. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*, page 10. ACM, 2021. ISBN 978-1-4503-8680-7. doi: 10.1145/3475960.3475985.

[3] Robert Byers, Chris Turner, and Tanya Brewer. National vulnerability database. `https://doi.org/10.18434/M3436`, 2022. Accessed: 2023-2024.

[4] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. Tracking known security vulnerabilities in proprietary software systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 516–519. IEEE, 2015.

[5] Checkmarx. Checkmarx: Application security testing solutions, 2023. URL `https://www.checkmarx.com/`. Accessed: 2023-2024.

[6] Cloudflare. Inside the log4j2 vulnerability (cve-2021-44228), 2021. URL `https://blog.cloudflare.com/inside-the-log4j2-vulnerability-cve-2021-44228/`. Accessed: 2023-11-23.

[7] Microsoft Corporation. Microsoft security update guide, 2024. URL `https://msrc.microsoft.com/update-guide`. Accessed: 2023-2024.

[8] Mitre Corporation. Cve, 1999. URL `https://cve.mitre.org/`.

[9] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX security symposium (USENIX Security 19)*, pages 869–885, 2019.

[10] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[11] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512, 2020.

[12] Spring Framework. Spring security, 2024. URL `https://spring.io/security`. Accessed: 2023-2024.

[13] GitHub. Github dependabot, 2023. URL `https://github.com/dependabot/dependabot-core`. Accessed: 2023-09-23.

[14] Inc. GitHub. Github security advisories, 2024. URL `https://github.com/advisories`. Accessed: 2023-2024.

[15] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. Vulinoss: a dataset of security vulnerabilities in open-source systems. In *Proceedings of the 15th International conference on mining software repositories*, pages 18–21, 2018.

[16] Daan Hommersom, Antonino Sabetta, Bonaventura Coppola, Dario Di Nucci, and Damian A Tamburri. Automated mapping of vulnerability advisories onto their fix commits in open source repositories. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–28, 2024.

[17] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. [engineering paper] enabling the continuous analysis of security vulnerabilities with vuldata7. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 56–61. IEEE, 2018.

[18] Maryna Kluban, Mohammad Mannan, and Amr Youssef. On detecting and measuring exploitable javascript functions in real-world applications. *ACM Transactions on Privacy and Security*, 27(1):1–37, 2024.

[19] OWASP. Owasp dependency-check, 2023. URL `https://owasp.org/www-project-dependency-check/`. Accessed: 2023-2024.

[20] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 426–437, 2015.

[21] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and C'edric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories*, May 2019.

[22] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020.

[23] FASTEN Project. Vulnerability producer, 2024. URL `https://github.com/fasten-project/vulnerability-producer`. Accessed: 2023-2024.

[24] Sofia Reis and Rui Abreu. A ground-truth dataset of real security patches. *arXiv preprint arXiv:2110.09635*, 2021.

[25] Arthur D Sawadogo, Tegawendé F Bissyandé, Naouel Moha, Kevin Allix, Jacques Klein, Li Li, and Yves Le Traon. Sspcatcher: Learning to catch security patches. *Empirical Software Engineering*, 27(6):151, 2022.

[26] Snyk. Snyk: Security for open source and proprietary code, 2023. URL `https://snyk.io/`. Accessed: 2023-2024.

[27] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. Tracking patches for open source software vulnerabilities. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 860–871, 2022.